Contents

Creational Patterns

Introduction

Almost all programming languages with object oriented support provides you rich features in creating instances of classes using constructor methods. Inside the constructors you can add business logic to initialize objects and make them ready for use. Some programming languages (like Java or C++) even have the capability to have more than one constructor method so that instances can be shipped with different states depending on the chosen constructor.

Unfortunately native constructors capabilities are not powerful enough for our standards of elegance. Some systems have complicated object production mechanisms that require extra capabilities. Sometimes classes have too many attributes for a simple constructors. Sometimes object creation require polymorphic support and decoupling against the exact subtypes or realizations. Sometimes you need to ensure that certain classes have exactly one instance throughout the lifetime of your system.

Learning Outcomes

1. Design systems that apply the factory method design pattern
2. Design systems that apply the abstract factory pattern

Factory Method Pattern

Problem

The exact type of the dependency (a product) created and used by some client (a factory) is decided by a client of that factory. Somewhere, inside this factory class, a specific product is being instantiated and maybe used (this instantiation happens maybe more than once). But, as it turns out, there are different types of products, (there's also the possibility of more product types in the future).

You can change the code of the factory class to accommodate multiple product types. For every product, you modify the factory and add some if else clause to produce the correct product type.

As you see this process is quite tedious. For every new product type that is added to your system, you perform surgery to the factory class. This process will end up forcing you to create smelly if-else checks to switch to the correct product type.

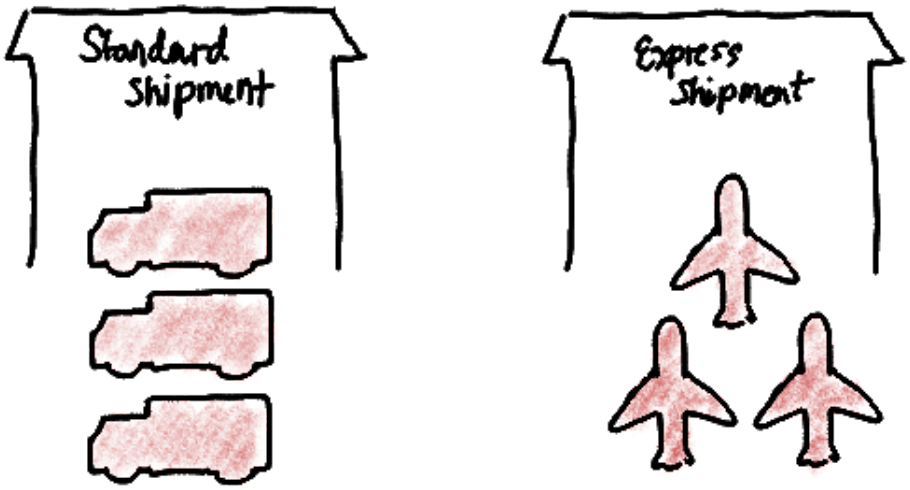

Figure 1: Factory Method

Solution

You encapsulate the creation of a class inside a **factory method** that is specified to return an abstraction of the product. If there are other real product types that have to be produced, you create a specialized factory which overrides the factory method.

> Somewhere inside factory you have one or more instances of creating or using the product.

If you choose to build `Factory` as a concrete superclass, the factory method inside the `Factory` should return some realization of `Product`. This is the default product returned by any `Factory`. If you need to return a different `Product` realization, you override the factory method to return that particular `Product` realization.

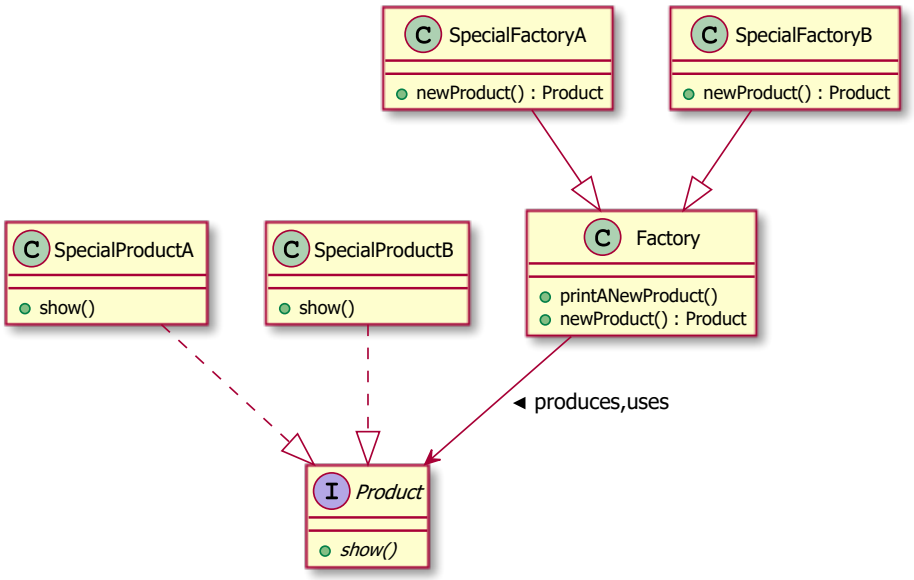

Figure 2: factory method class diagram

Example

Online Marketplace Delivery Consider you're developing the product delivery side of an online marketplace app (think Amazon/Lazada). Your app is on its early stage so their is only one delivery option, standard nationwide delivery that takes a minimum of 7 days.

What you have is `Shipment` class that contains a `StandardDelivery` class. Inside the shipment class is the `shipmentDetails()` builder which builds a string representing the details of the shipment, this includes the delivery details (which requires access to the composed `StandardDelivery` instance). Inside the constructor of `Shipment` an instance of `StandardDelivery` is created so that every `Shipment` is set to be delivered using standard delivery.

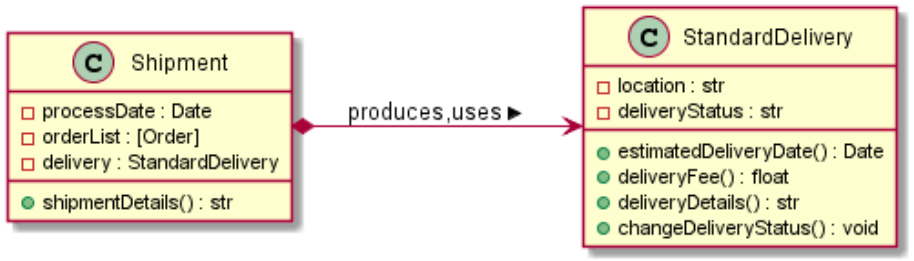

Figure 3: online marketplace

This system does work. It works but it is still inelegant. As soon as your

app grows, you will incorporate new delivery options like express delivery, or pickups or whatever. Every time you need to add a new delivery method you will need to perform surgery in `Shipment` since the `StandardDelivery` instance is created inside the constructor of `Shipment`. `Shipment`'s code is too coupled with `StandardDelivery`.

To solve this you need to implement the factory method pattern. Right now shipment is a factory since it constructs its own instance of `StandardDelivery`. To refactor this into elegant code, you need to so create an abstraction called`Delivery` first to support polymorphism. Inside `Shipment` instead of creating instances of `Delivery`'s using a constructor, you invoke a factory method that encapsulates the instantiation of `Delivery`. In this case we name this method `newDelivery()`. All it does is return an instance of `StandardDelivery` using its constructor.

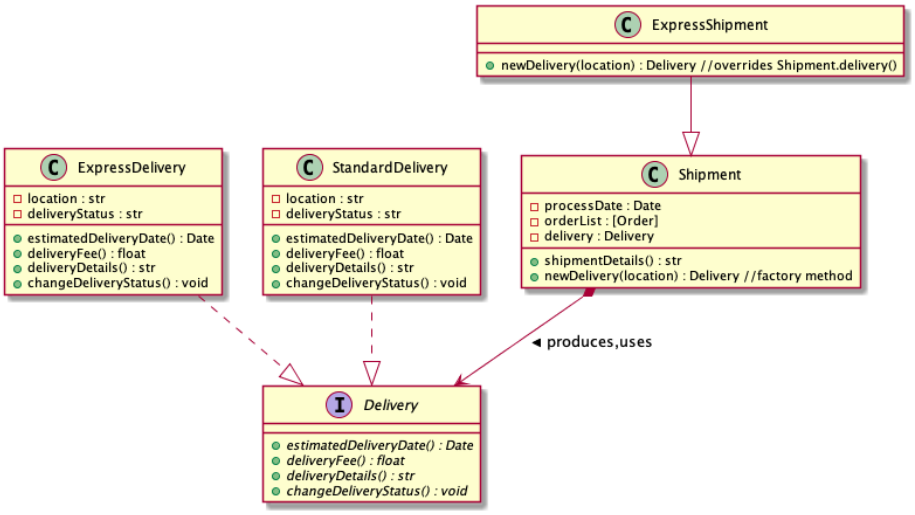

Figure 4: online marketplace

In this new architecture, whenever there are new delivery methods a shipment could have, all you have to do is to create a realization of that delivery method. In this case the new delivery method is `ExpressDelivery` which delivers for two days but is twice as expensive. And instead of changing `Shipment` (violates Open/Closed Principle), you make an extension to `Shipment`. This extension is the specialization to shipment called `ExpressShipment` (a shipment that uses express delivery). In this specialization, you only need to override the factory method delivery, so that every instance of delivery construction creates `ExpressDelivery`. The difference between `ExpressDelivery` and `Delivery` is that `ExpressDelivery` has a delivery fee of 1000 and the estimated delivery

date is 1 day after the processing date.

Why this is elegant

- **Single Responsibility Principle** - the extra level of encapsulation on the construction of the product (factory method), allows the factory to be responsible of creating the exact product type it needs.
- **Open/Closed Principle** - instead of modifying the factory to incorporate the creation of different product realizations, you instead create an extension of the factory. No need for introspective checks since the factory method supports polymorphism of the product it creates.
- *Encapsulate what varies* - This pattern upholds one of OOP paradigms most important principles. Since the construction of product varies from product type to product type, it is encapsulated into the factory method.
- This avoids tight coupling between the factory and the product

  Coupled classes are classes which are very dependent on each other. Changing the code of one will most likely affect the other

How to implement it:

1. Create an abstraction for all product types (`Product`).
2. Inside the base factory create the the factory method function `newProduct():Product`. Make sure it is specified to return abstraction `Product`.

3. For every new product type that is added to the system, create 2 new classes: the new product type as a realization of `Product` and, and the factory for the new product type as a realization of `Factory`.
4. Inside each factory specialization override `newProduct()` to return the correct realization of `Product`.
5. Replace every instance of constructor calls inside `Factory` with a call to the factory method `newProduct()`.

Abstract Factory

Problem

Your system consists of a family of related products. These products also have different variants. You need a way to create these products so that the products match the the same variant. The exact variants of the family of products are decided during runtime, somewhere else in the code (similar to product creation in a factory method)

Figure 5: Abstract Factory

Solution

You create different kinds of factories that realize under the same abstract factory. The exact type of factory will decide the variant of the family of products that are created. To do this you need to create different factory methods for each product. These factory methods must be abstract methods in the abstract factory so that every factory realization can create all members of the product family.

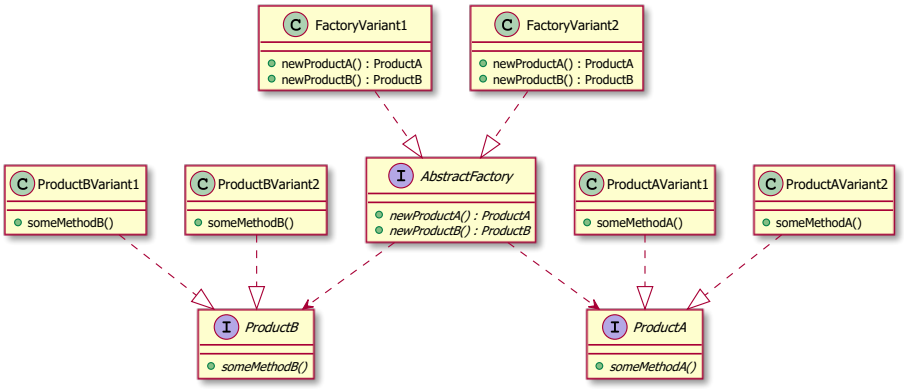

Figure 6: abstract factory

The family of products, are `ProductA` and `ProductB`, These products come in two variants, variant 1 and 2. `FactoryVariant1` is a realization of `Factory` which creates all of the product in variant 1 while `FactoryVariant2` creates all the products in variant 2.

> If it makes sense for the system you can make an abstract `Product` class for all the types of products.

When the client of an abstract factory produces its products, it doesn't need to know what kind of factory is producing the products. This means that the

concrete type of a product (its variant) is not decided during compile time but instead it depends on the concrete type of the factory that is creating it.

Example

Bootleg Text-based Zelda Game You're creating the dungeon encounter mechanics of some bootleg text-based zelda game. In this game,every time you enter a dungeon, you encounter 0-8 monsters (the exact number is randomly determined). There are 3 types of monsters, bokoblins, moblins, and lizalflos (different types have different moves). The exact type of monster is randomly decided as well.

Right now the game works like this:

As soon as you enter the dungeon, all the enemies are announced:

```
5 monsters appeared
A lizalflos appeared
A lizalflos appeared
A lizalflos appeared
A moblin appeared
A moblin appeared
```

After this, each enemy in the encounter attacks. They randomly pick an attack from their moveset.

```
Lizalflos thorws its lizal boomerang at you for 2 damage
Lizalflos thorws its lizal boomerang at you for 2 damage
Lizalflos camouflages itself
Moblin stabs you with a spear for 3 damage
Moblin stabs you with a spear for 3 damage
```

The encounter ends with Link dying since you haven't coded anything past this part.

You decide to make things exciting for your game by adding harder dungeons, medium dungeon and hard dungeon.

Medium dungeon Instead of encountering, normal monsters you encounter stronger versions of the monsters, these monsters are blue colored:

- **Blue Bokoblin**
  - equipped with a spiked boko club and a spiked boko shield
  - bludgeon deals 2 damage
- **Blue Moblin**

  - equipped with rusty halberd
  - stab deals 5 damage
  - kick deals 2 damage
- **Blue Lizalflos**
  - equipped with a forked boomerang
  - throw boomerang deals 3 damage

Hard dungeon These monsters are silver colored extra stronger versions of the monsters

- **Silver Bokoblin**
  - equipped with a dragonbone boko club and a dragonbone boko shield
  - bludgeon deals 5 damage
- **Silver Moblin**
  - equipped with knight's halberd
  - stab deals 10 damage
  - kick deals 3 damage
- **Silver Lizalflos**
  - equipped with a tri-boomerang
  - throw boomerang deals 7 damage

To seamlessly incorporate these harder monsters in your system, you need to create an abstract factory for each dungeon difficulty. There are now three variants for each monster. For every variant, there is a factory that spawns new instances of each monster.

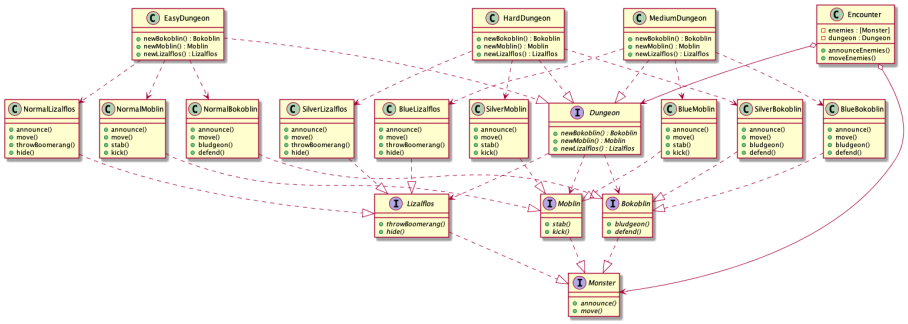

Figure 7: abstract factory example

Why this is elegant

- **Open/Closed Principle** - This solution is easier to maintain since you can add more variants of `Product` without touching any existing code. All you have to do is to add new realization for `Product` and a new realization

`AbstractFactory`

- Changing the form and behavior of specific variants are isolated since its creation is abstracted.
- You can easily switch between variants by swapping out the factory.

How to implement it:

- For every product in the family of products, create an abstraction of it (`ProductA`, `ProductB`).
- For every variant of the products, create a factory, (`FactoryVariant1`, `FactoryVariant2`). These factories must realize under an abstract `Factory`. The factory should contain abstract factory methods for each product,
- Inside every factory implement all factory methods.

Singleton (Optional Read)

Problem

Sometimes it wouldn't make sense for a class to have more than one instance in the lifetime of the application. These things are called singletons.

Solution

Inside the singleton. Create a static attribute that represents the singleton. Since it is static all instances of the class will share this value. Create a builder to lazily instantiate the value of the singleton and expose the value of the instance.

Disallow the usage of the normal constructor as much as possible. To access the shared static instance, use the builder.

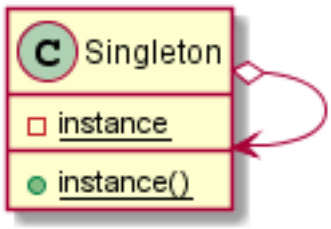

Figure 8: singleton

> Disallowing the creation of a singleton depends on the language you use, you can set the constructor to private, or you can raise an error if you try to use the constructor outside the instance builder.

You can also choose to not disallow the use of the constructor, as long as you trust the users to always use the instance builder instead.

Example

A Catalog of Globals It doesn't make sense for you to keep multiple copies of global variables in your application, so you decide to place them in a singleton class.

Why this is NOT is elegant A singleton pattern is actually hated by most developers. Yes you can ensure that there is exactly one instance of a class, but its advantages come with a lot of drawbacks.

- You can ensure single instance classes, just by being vigilant.
- Singletons require the use of static attributes and methods. Statics are anti-pattern because they are global variables and manipulators that can have invisible changes to state.
- Singletons are usually symptoms of bad design.

Optional Reading

Shvets A. (2018) Creational Patterns Accessed August 31, 2020

Behavioral Patterns

Introduction

Some systems require complex and extremely decoupled relationships. Behavioral patterns are used on these tightly interconnected systems so that they are easier to maintain. These patterns separate behavioral responsibilities among the classes in your system in such a way that volatile behaviors are encapsulated deep into your object structure.

Learning Outcomes

1. Design systems that apply the strategy pattern
2. Design systems that apply the state pattern
3. Design systems that apply the command pattern
4. Design systems that apply the observer pattern
5. Design systems that apply the template pattern
6. Design systems that apply the iterator pattern

Strategy pattern

Problem

Some systems require behavior that have to be parametrized for other behavior. This is easily done in a functional programming environment since higher order functions are used to represent these. In programming languages that don't support these features, the strategy pattern is used.

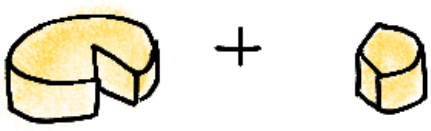

Figure 9: strategy

Solution

Functions that are not first class citizens are encapsulated inside a `Strategy` class. A strategy class simply contains the method `execute(params)`, which represents the behavior that should be passed into a higher order function. Any method that can be passed into the higher order function should realize `Strategy`.

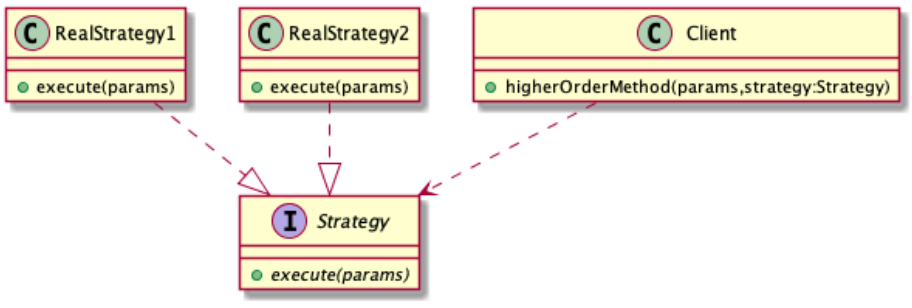

Figure 10: Strategy pattern

The object `params` represent the data that you need to pass into the correct class. In this pattern you pass the the whole `Strategy` realization so that

`strategy.execute(params)` perform the desired behavior. You can add other methods in the `Strategy` abstraction, if it makes sense for the system.

Example

Fraction Calculations You're creating a less sophisticated version of a fraction calculator. This calculator only has arithmetic operations inside it, addition, subtraction, division, and multiplication. Inside this calculator, a calculation is represented in a `Calculation` instance. Every calculation has four parts:

- `__left` - represents the left operand fraction
- `__right` - represents the right operand fraction
- `__operation` - represents the operation ($+$,$-$,$\times$,$\div$)
- `__answer` - represents the solution of the operation

Python does indeed support higher order functions but your boss is anti-functional programming so he forbids the use these features. Because of this you decide to implement the strategy pattern.

To do this, you need to create an abstraction called `Operation` to represent the different operations. For each operation, you create a class that realizes `Operation`.

> `execute()` should have been named like a builder method (something like `solution()`), I'm keeping the name `execute()` since this is how Strategy patterns usually names this particular method.

Why this is elegant

- **Open/Closed Principle** - If you want to add new strategies, you wouldn't need to touch any existing code.
- The implementation of a strategy is deeply tucked inside multiple layers of encapsulation. Changing these implementations is very easy.
- You can swap strategies during runtime in the same way you do in functional programming.

How to implement it

1. Create an abstract `Strategy` that contains an abstract method called `execute`. This method should be specified to accept all the necessary parameters needed by your parameterized function.
2. For every strategy, the higher order method can accept, you create a realization for `Strategy` and implement the correct behavior in `execute`.

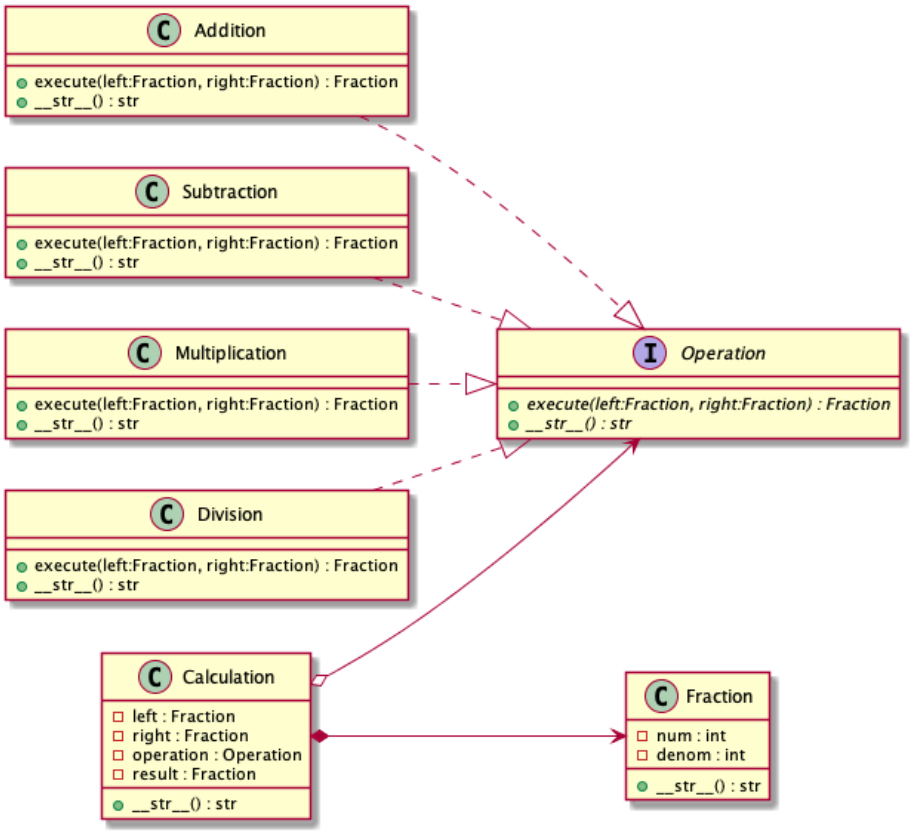

Figure 11: strategy pattern example

3. The higher order function should now be specified to accept a `strategy` of type `Strategy`.
4. Inside the higher order function, whenever it wants to perform the strategies embedded behavior, call `strategy.execute(...)`.

State Pattern

Problem

Some objects can change into many different states. If different objects behavior is dependent on its current state, it would require, bulky and annoying if-else blocks to handle its dynamic behavior.

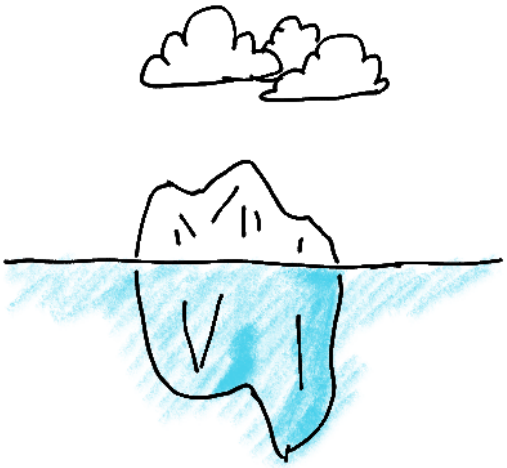

Figure 12: State

Solution

An object that can have many states should contain an attribute representing its state. Instead of performing, state dependent behavior directly inside the object, you delegate this responsibility to its embedded state instead. In this way the object will behave according to its current state.

The state may be required to contain backreference to the context object that owns it. This is only required if state methods requires to access/control the context that owns it.

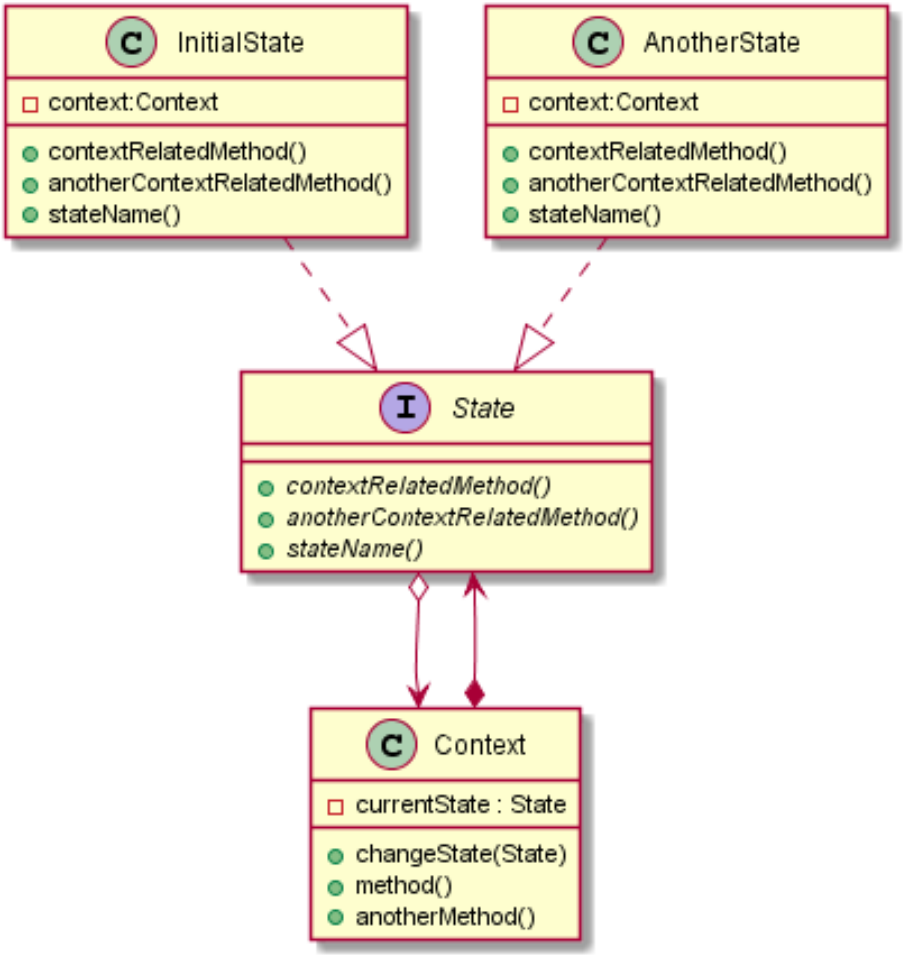

Figure 13: state pattern

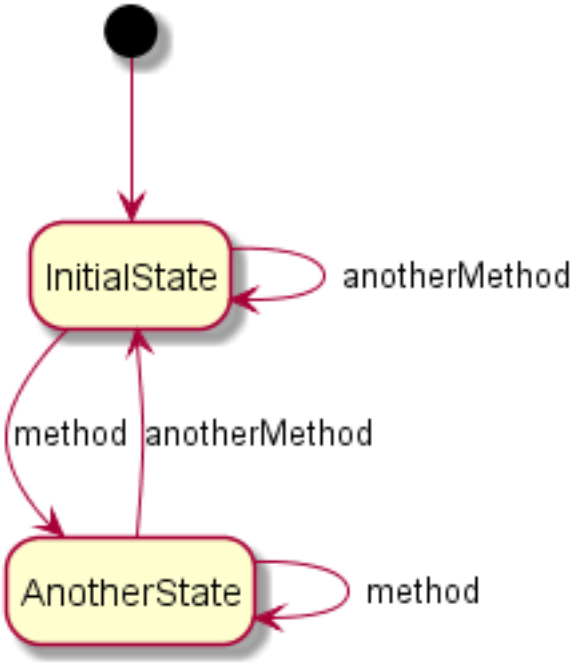

Figure 14: state diagram

Example

States of matter The state of any given matter is dependent on the pressure and temperature of its environment. If you heat up some liquid enough it will turn to gas, if you compress it enough it will become solid.

You are to build a less sophisticated version of this model in code. Matter comes in three states, solid, liquid, and gas. The state of the matter may change if you put/remove pressure on it or heat/cool it.

The state diagram would look something like this:

To implement something like this, you would need to create matter which owns an attribute called `state` which represents the matter's current state. Since there are three states, you create three realizations to a common abstraction to state.

When you compress/release/cool/heat the matter, you delegate the appropriate behavior and state change inside `state`'s version of that. Each `State` realization will need a backreference to the `Matter` that owns it so that it can change it's state.

Delegating behavior to the composed state means that, when the

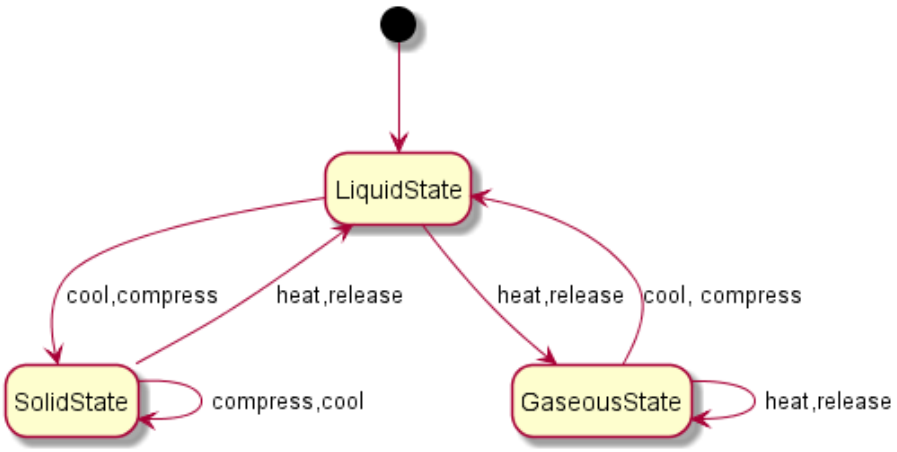

Figure 15: state diagram example

`Matter` instances invoke, `compress()`, `relaease()` `heat()`, and `cool()`, the composed `State` owned by the matter calls its own version of `compress()`, `relaease()` `heat()`, and `cool()`.

Matter owns an instance of `State`, and that instance has an attribute called `matter`. The attribute `matter` is the reference to the instance of `Matter` that owns it. The `State` instance needs this reference so that it can change the matter's state when it is compressed, released, heated, or cooled.

Why this is elegant

- **Single Responsibility Principle** - behavior related to state is delegated to the state itself.
- **Open/Closed Principle** - You can incorporate new states to the system without touching any existing client code
- Implementing this pattern will remove bulky and annoying state conditionals

How to implement it

1. Create an abstract `State` that contains abstract methods for all state dependent behavior (context related behaviors that are dependent on context's state).
2. For every state the `Context` can have, create a realization of `State`.
3. `Context` owns an attribute that represents the current state (`currentState`) that it owns.

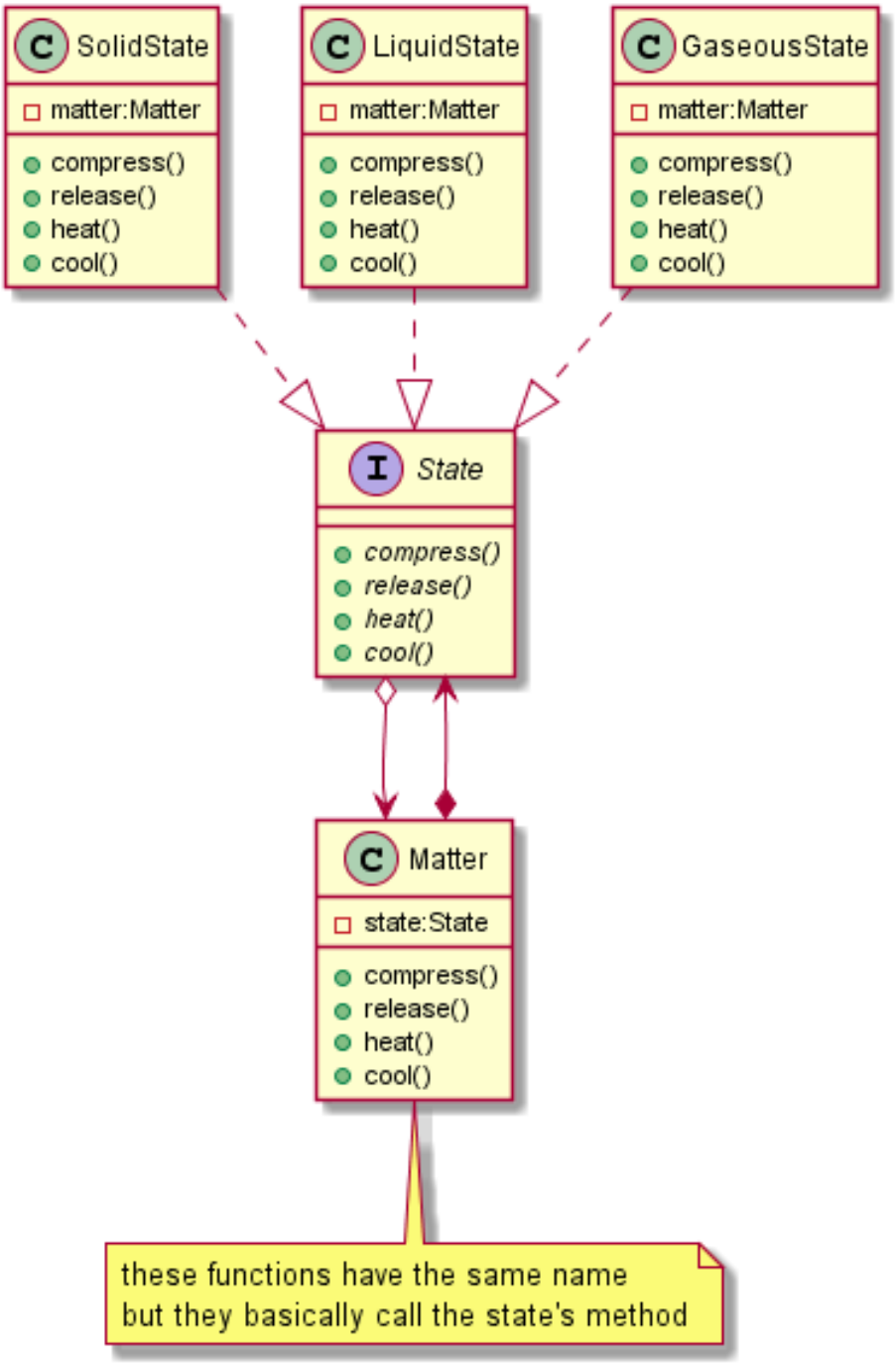

Figure 16: state example

4. If the state needs to control the `Context` instance that owns it, add a backreference to `Context` inside state.
5. Whenever a `Context` instance performs state dependent methods, it calls `currentState.contextRelatedMethod()` instead so that its behavior is dependent on its current state.

Command Pattern

Problem

Sometimes, object behavior contain complicated constraints. Sometimes, the system require the behavior to be invoked by an object but performed by another (this is common in presentation layer/domain layer separation in MVC enterprise systems). Sometimes, the system requires behavior to be undone. Sometimes the system requires a history of the behaviors that were being performed.

Figure 17: Command (What does this strange picture mean?)

Solution

These problems have a common solution, the **command pattern**. A `Command` is a more powerful version of a `Strategy`. While both of them encapsulates behavior, a `Strategy` is just that, a function wrapper. A `Command` on the other hand contains which object performs the behavior, which parameters are needed to perform the behavior, and how to undo the command (if needed).

Creating `Command`s, allow for more flexible behavior responsibility assignments. A separate `Invoker` object triggers the behavior by creating a command. This `Invoker` prepares the command with the appropriate `Reciever` (the object performing the command), and the appropriate parameters. The invoker then executes the prepared command. The command doesn't actually do anything, it just tells the `Reciever` to call the appropriate method.

This separation of responsibility allows for the creation of extra features that may be required for your system:

- If you want to keep a history of the performed commands, the `Invoker` may keep a list of `Commands`. This way the data stored in the list history, is a perfect representation of the previous commands.
- If you want the `Commands` to be undoable, you can store a backup of the receiver (and other affected objects) by the `Command` inside each instance of `Command` . Undoing a command will be as simple as restoring the receiver to its backup.

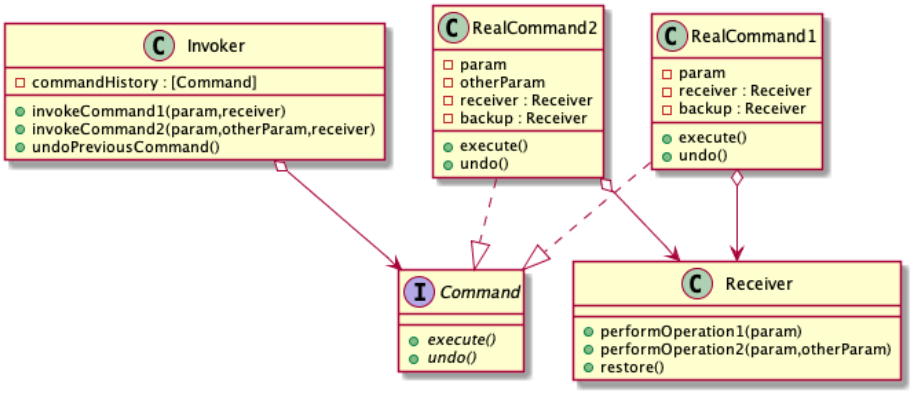

Figure 18: command pattern

Instead of passing the receiver in the `Invoker` methods, you can create an attribute called receiver inside `Invoker`. But doing this will make it so there is one `Receiver` instance for every `Invoker` instance.

The commands should only affect the receiver. If the behavior that is performed changes a lot of objects, then make a `Receiver` class that encapsulates all of the affected objects. Doing this will make the implementation of undo easier since the backup inside of the command will simply be an older version of `Receiver`.

Different command realizations are not necessarily of the same`Strategy`. That's why the parameters of the behavior are stored as attributes of the command, not passed in the `execute()` function. This is so that no matter what the command is, all `execute()` functions will have the same type signatures.

Example

Zooming through a maze You're creating a maze navigation game thing. This is what the application currently has right now:

- `Board` - this represents the layout of the maze. The layout is loaded from a file. It has these attributes:
  - `__isSolid` - this is a 2 dimensional grid encoded as a nested list of booleans which represents the solid boundaries of the maze. For example if `__isSolid[row][col]` is true then it means that that cell on (row,col) is a boundary
  - `__start` - a tuple of two integers that represent where the character starts
  - `__end` - tuple of two integers that represent the position of the end of the maze
  - `__cLoc` - tuple of two integers that represents the current location of the character
  - `moveUp()`, `moveDown()`, `moveLeft()`, `moveRight()` - moves the character one space, in the respective direction. The character cannot move to a boundary cell, it will raise an error instead.
  - `canMoveUp()`, `canMoveDown()`, `canMoveLeft()`, `canMoveRight()` - returns true if the cell in the respective direction is not solid.
  - `__str()__` string representation of the board. It shows which are the boundaries and the character location

What's missing right now is controller support. This is how a player controls the character on the maze:

- `dpad_up()`, `dpad_down()`, `dpad_left()`, `dpad_right()` - The character dashes through the maze in the specified direction until it hits a boundary.
- `a_button()` - The character undoes the previous action it did.

To implement controller support you need to create a `Command` abstraction which is realized by all controller commands. The `Controller` (which represents the controller) is the invoker for the commands. Since commands are undoable, this controller needs to keep a command history, represented as a list. Every time a controller button is pressed, it creates the appropriate `Command`, executes it and appends it to the command history. Every time the `a_button()` is pressed to undo, the controller pops the last command from the command history and undoes it.

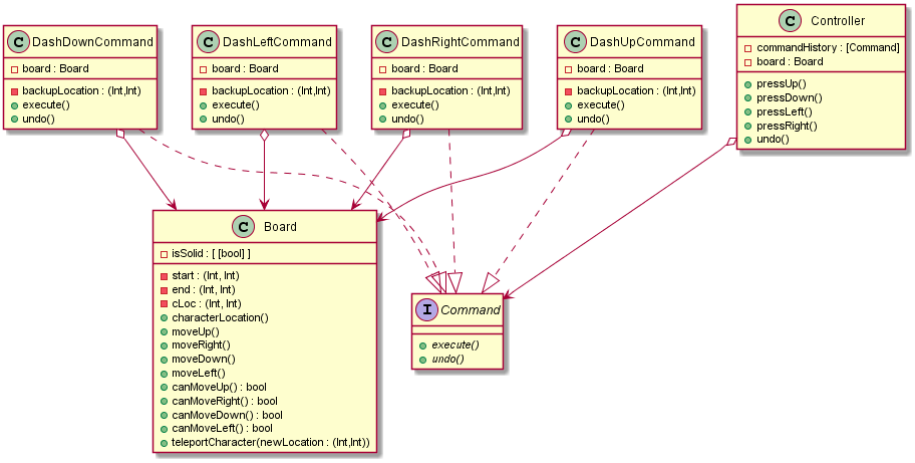

Figure 19: command example

Why this is elegant

- **Single Responsibility Principle** - The behavioral responsibilities in the system are thoroughly separated. One invokes the command, and the other performs the behavior associated with the command.
- **Open/Closed Principle** - If there are more commands you want to add, you don't have to touch any existing code.
- Switching between invokers and receivers is easily done
- You can implement undo (and redo)
- You can defer the execution of behavior
- A command may be made of smaller simpler commands

How to implement it

1. Create an abstraction `Command` that contains abstract method `execute()`, and other command related methods like `undo()`.
2. For every command, create a realization to `Command`. These commands uses a reference to a `Reciever` instance. This instance represents the instance/s that are affected whenever `Command` realizations are executed.
3. Create an `Invoker` class that will be responsible for instantiating, preparing, and executing commands. Inside these class are methods for invoking each commands. When these methods are called, the invoker does the following:
    1. Instantiate an instance of `Command` called `c` with the correct realization.
    2. select the receiver of the `Command`, including the related parameters.

3. invoke `c.execute()`.

4. If the system supports undoable commands, the `Invoker` should keep a list of commands called `commandHistory` and each command instance should keep a reference called `backup` to enable restoration of `Reciever` instances.

Observer Pattern

Problem

What if you need to inform a lot of objects about the changes to some interesting data? If you globalize the data and let your client objects poll for changes all the time, this will affect the security and safety of your interesting data. Plus, global data is something that should be avoided as much as possible. Also, forcing your objects poll for changes all the time will be inefficient if your interesting data has not changed.

Figure 20: observer

Solution

The responsibility of sharing information about the changes to interesting data should not be placed in the clients of the data. You should create a notifier class that encapsulates the interesting data. This class should be responsible of notifying interested clients about changes on the data.

To do this you need to encapsulate the interesting data (from now on lets call it the `subject`), into a `Publisher` class. An instance of this class will be responsible of notifying the observers for any change in the `subject`. Whenever there

are changes to the subject, the `Publisher` instance calls `notifyObservers()` so that all interested, observers will be informed of the change. Any class that is potentially interested in the `subject` should realize an `Observer` abstraction, which in the bare minimum contains, the `update(updatedSubject)` function. Inside `Publisher`'s `notifyObservers()` method, every subscriber (an interested observer) is updated (`subscriber.update()`).

Any instance of an `Observer` should be subscribed to the change notifications using `Publisher`'s `subscribe()` function. They can also be unsubscribed using the`unsubscribe()` function.

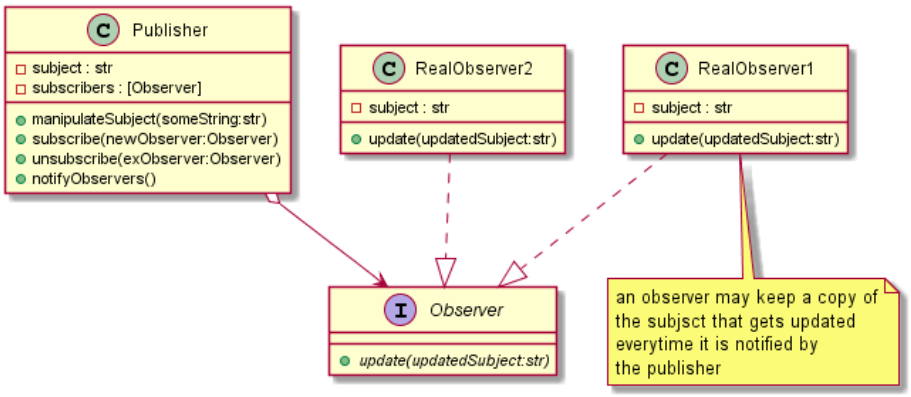

Figure 21: observer

> Whenever an observer has updated, the publisher needs to pass all the necessary details in the notification. This is generally done by passing the updated subject in the `update(updatedSubject)` method.
>
> In some cases, the observer needs to keep a copy of the subject as an attribute. Make sure to change the value of this attribute during updates.
>
> Make sure that changes to the subject are only done using the `Publisher` class (`manipulateSubject()`). If you change the subject without using `Publisher`'s methods, your subscribers won't be notified.

Example

Push Notifier for Weather and Headlines You are creating a push notification system that works for multiple platforms. You want to distribute information about the current weather and news headlines. This system will be

potentially used on many platforms so you have to think about the maintainability issues for adding new platform support.

To implement this, you have to apply the observer pattern. Your subject would be `Weather` data and `Headline` data (which are their own classes). These subjects should be encapsulated into a single publisher class (which will be called `PushNotifier`).

Any platform, that is interested in the changes to the subject should realize a `Subscriber` abstraction (Observer), which contains the abstract method update().

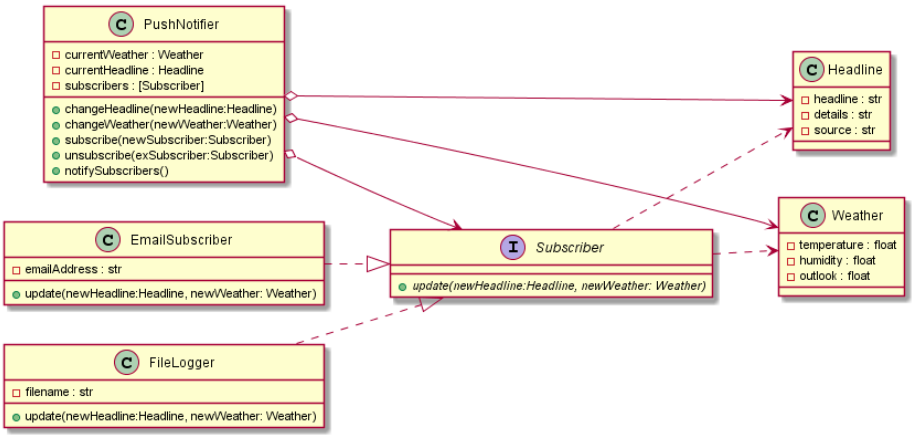

Figure 22: observer example

Why this is elegant

- **Open/Closed Principle** - You can add new `Observer` realizations seamlessly every time there are new objects that are interested in the subject.
- A observer can be subscribed/unsubscribed during runtime

How to implement it

1. Create an `Observer` abstraction that represents all classes that can potentially observe changes to the `Publisher`. `Observer` will contain the abstract method `update()`.
2. All classes that want to be notified about changes to the `subject` should realize `Observer`.
3. `Publisher` will either own/use an instance of the `subject` of interest. It will also use an attribute which is stores the list of `Observers` that are interested in the subject. To attach or detach `Observers`, `Publisher` contains the methods `subscribe()` and `unsubscribe()`.

4. Every time `subject` is manipulated, it should be done through `Publisher` , because `Publisher` needs to notify all `Observers` in its observer list attribute after every manipulation. This notification is done through `notifyObservers()` after the end of every subject manipulation.
5. Inside the `Publishers` `notifyObservers` method, every `Observer` in its list of observers invoke their `update()` method.

Template Method Pattern

Problem

Say you have two or more *almost* identical behaviors from different classes. Rewriting these object behaviors as separate methods for each class duplicates many parts of the code (especially if the behavior has a lot of lines of code).

Figure 23: template

Solution

To avoid code duplication, you break down your code into individual steps. By doing this you can create a superclass that contains the implementation for all common steps. This superclass will also contain the common implementation for the **template method**, the method that combines all steps into the original object behavior. differences between steps will be resolved under different specializations of this superclass.

If the exact instance of the class is a `Specialization1`, it performs the template method with special versions of `step1()` and `step4()` (since `Specialization1` overrides them) but the other parts are inherited from the `Template`.

The steps in the superclass can be a mix of abstract methods and

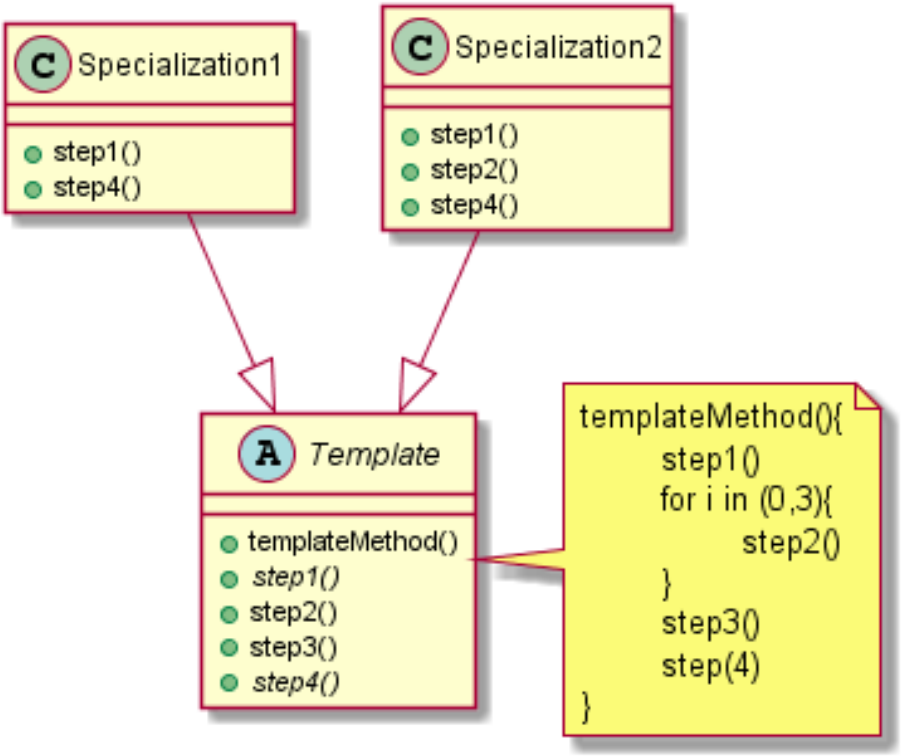

Figure 24: template method

concrete methods. Make a method abstract if you want to force all specializations to override these steps. You'll want to do these if some of the steps in your template doesn't have a default implementation.

Example

Brute Force Recipe If you write brute force algorithms as search problems, they will have a common recipe. This is the reason why it is called the exhaustive search algorithm. It will traverse all of the elements in the search space, trying to check the validity of each element, until it completes the solution

Equality Search

Search for integers equal to the target

```
#searchSpace = [2,3,1,0,6,2,4]
#target = 2

i = 0
solutions = []
candidate = searchSpace[0]
while(i<len(searchSpace)):
    if candidate == target:
        solutions.append(candidate)
    candidate = searchSpace[++i]

#solution = [2,2]
```

Divisibility Search

Search for integers divisible by the target

```
#searchSpace = [2,3,1,0,6,2,4]
#target = 2

i = 0
solutions = []
candidate = searchSpace[0]
while(i<len(searchSpace)):
    if candidate % target == 0:
        solution.append(candidate)
    candidate = searchSpace[++i]
```

```
#solution = [2,0,6,2,4]
```

Minimum Search

No target, searches for the smallest integer

```
#searchSpace = [2,3,1,0,6,2,4]
#target = None

i = 1
solutions = [searchSpace[0]]
candidate = searchSpace[1]
while(i<len(searchSpace)):
    if candidate <= solutions[0]
        solutions[0] = candidate
    candidate = searchSpace[++i]

#solution = [0]
```

Common Recipe

```
i = 0
solutions = []
candidate = first()
while(isStillSearching()):
    if valid(candidate):
        updateSolution(candidate)
    candidate = next()
```

Because of this we can write a general brute force template method that would return the solution to brute force problems. To do this you create a superclass `SearchAlgorithm()` that contains the template method for brute force algorithms. If you want to customize this algorithm for special problems, all you have to do is to inherit from `SearchAlgorithm` and override only the necessary steps.

> is `isValid()` and `updateSolution(candidate)` is different for each algorithm so it doesn't have a default implementation. It would be best to make these steps abstract.

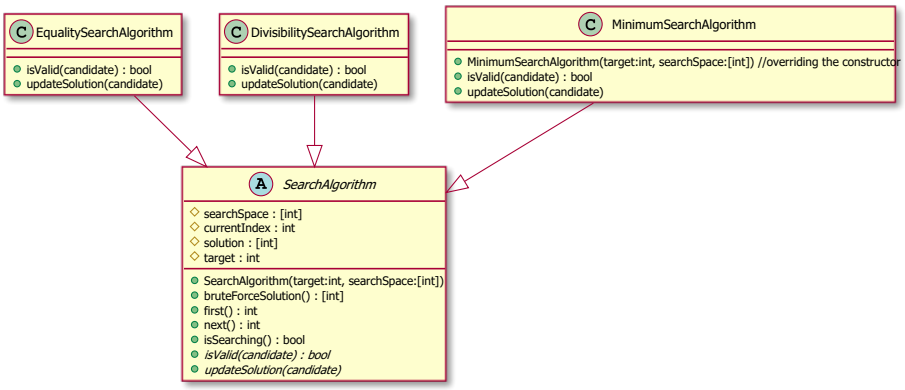

Figure 25: template example

Why this is elegant

- **Open/Closed Principle** - The `Template` is open for extension but closed for modification
- *Encapsulate what varies* - steps can vary from specialization to specialization, therefore they are encapsulated into step methods.
- Implementing this pattern will remove duplicate code in the common parts of the algorithm.
- Clients may override only certain steps in a large algorithm, making it easier to create specializations

How to implement it

1. Create an abstract class called `Template`. It contains the method `templateMethod()` broken down into separate steps through separate `step1()`, `step2()`, … and etc. methods. When invoked `templateMethod()` will just call these step methods.
2. Each step method inside `Template` will contain the default implementation of that step. If there is no default implementation, the method should be abstract.
3. For every similar behavior to `templateMethod()` a specialization of `Template` is created. These methods will implement all abstract methods and override all step methods that vary for its behavior.

Iterator

Problem

One of the most common iteration recipes that you'll likely implement is the **for-each** loop. This loop traverses a collection, and performing some kind of operation along the way. Most programming languages implement for each loops on built in collections like arrays, sets, and trees. But what about non-built in collections?

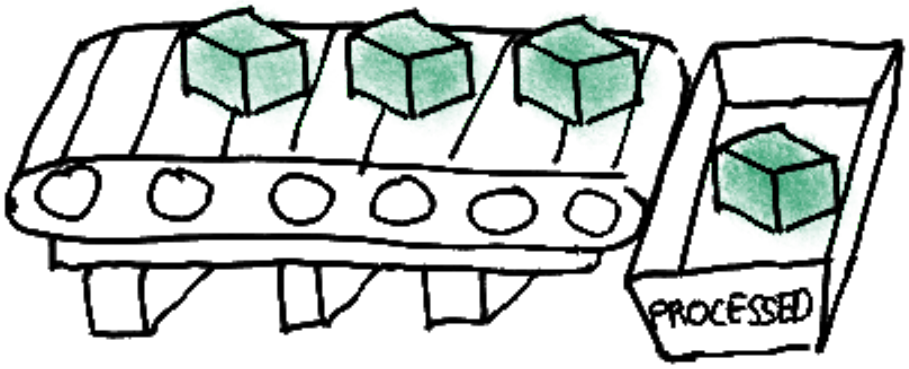

Figure 26: iterator

Solution

For non built-in collections, you can create an iterator that does the traversal for you. On the bare minimum these iterators will realize some `Iterator` abstraction that contains the methods, `next()`, and `hasNext()`. From these methods alone you can easily perform complete traversals without knowing the exact type of the collection:

```
i = collection.newIterator()
while i.hasNext():
    print(i.next())
```

The `hasNext()` method, returns a boolean value that indicates whether or not there are more elements to be traversed. The `next()` method, returns the next element in the traversal.

A collection can have more than one `Iterators`, if it makes sense for the collection to be travsersed in more than one way. Despite this possibility, a collection must have a default iterator which will be the type of the new instance returned in the factory method, `newIterator()`

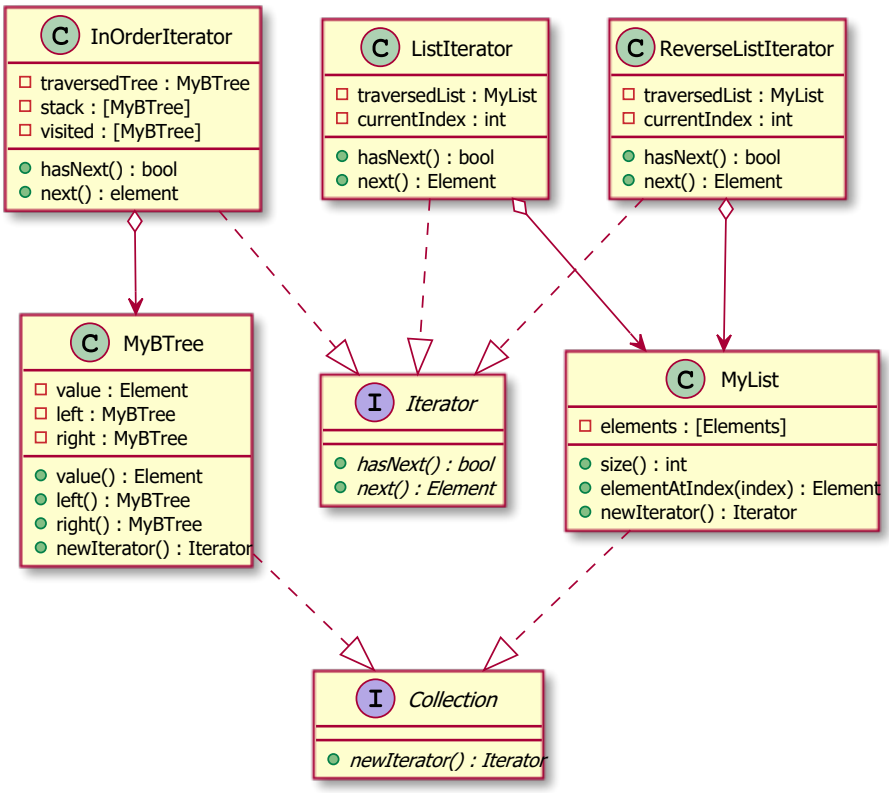

Figure 27: iterator

Why this is elegant

- **Single Responsibility Principle** - Traversal algorithms can now be placed into separate classes that interact with an iterator instead of the collection itself.
- **Open/Closed Principle** - You can implement new types of collections and iterators without touching any existing code.
- You can traverse all the elements in a collection, even if you don't know the exact type of the said collection.
- Two iterators, can iterate over the same collection without problem as long as the iterators are of different instances.

How to implement it

1. Create an abstraction called `Iterator` which contains the abstract methods `next()` and `hasNext()`.
2. Create an abstraction called `Collection` which contains the abstract method `newIterator()`.
3. For very collection that can be iterated through create a realization to `Collection`. Inside these `Collection` realizations, the `newIterator()` method must be implemented which simply returns a new instance of the default `Iterator`. (for collections that can be iterated through in more than one way, create different methods for creating other iterators as well but always keep `newIterator()` as the one that returns a new instance of the default iteration).
4. For every different way of iterating through a `Collection` realization, a realization to `Iterator` must be created as well.
5. `Iterator` realizations should contain an attribute that refers to the collection instance it is iterating through.

Optional Reading

Shvets A. (2018) Behavioral Patterns Accessed August 31, 2020

Structural Patterns

Introduction

As your system evolves, the structure of your classes could get complicated. As you introduce more features, your classes become bigger and harder to maintain.

To alleviate these issues, you can assemble objects into maintainable structural patterns.

Learning Objectives

1. Design systems that apply the decorator pattern
2. Design systems that apply the adapter pattern

Decorator Pattern

Problem

Some of your classes require extra features that can be added and removed during runtime. Sometimes you even need to support a set of extra features that can be arbitrarily combined with each other. You need to do this without breaking how these classes are being used by their clients.

Figure 28: decorator

Solution

To solve this issue, all you have to do is to apply the open/closed principle. For every feature that can be arbitrarily added to some simple class, you need to create a `Decorator` that extends the features of classes using inheritance and composition at the same time. The neat thing about this pattern is that the `Decorators` will have polymorphically the same type as the simple class due to inheritance. `Decorators` will also be able to control instances of the simple class because of composition.

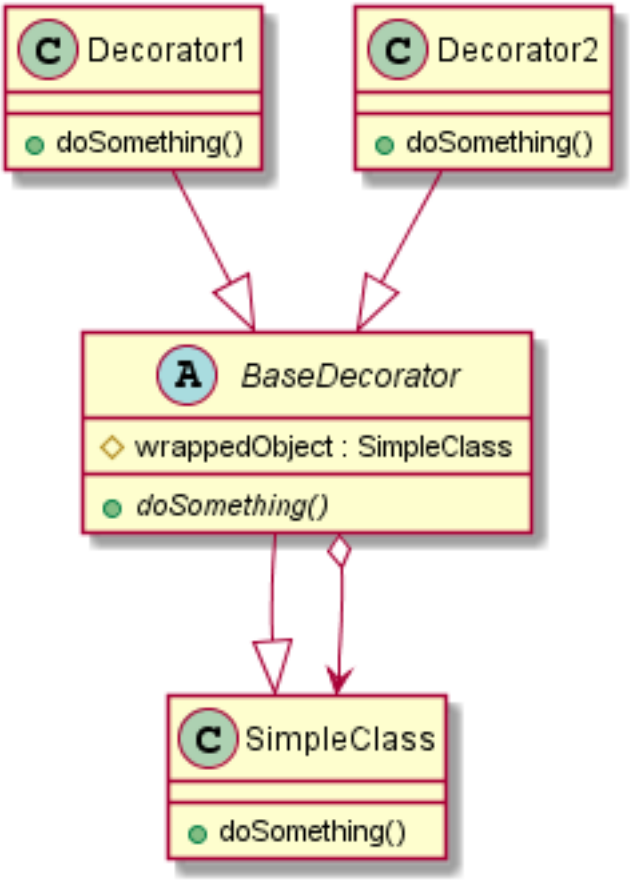

Figure 29: decorator

To create an instance of a `SimpleClass` decorated by `Decorator1`, all you need to do is to wrap the `SimpleClass` instance with an instance of `Decorator1`. When this `Decorator1` instance, calls `doSomething()` it calls the wrapped `SimpleClass` instance's `doSomething()` and do some extra behavior.

```
#Decorator1's implementation of doSomething():

def doSomething(self):
    self._wrappedObject.doSomething()
    doSomthingExtra()
```

It would be handy to create a `BaseDecorator` abstract class that is inherited by all decorators. It's not required but this class will form a class hierarchy for all decorators. Plus, you can write all of the common behavior and data into this class. It would be better for this class's `doSomething()` to be abstract, since it doesn't make sense for you to create instances of `BaseDecorator`.

Example

Decorating Sentences A sentence can be defined as a list of words (words are strings). The string representation of a sentence is the concatenation of all of the words in the list, separated by a space.

Instances of sentences can be printed with formatting:

- **bordered** - Given the sentence, `["hey","there"]` it prints:

  ```
  -----------
  |hey there|
  -----------
  ```

- **fancy** - Given the sentence, `["hey","there"]` it prints:

  ```
  -+hey there+-
  ```

- **uppercase** - Given the sentence, `["hey","there"]` it prints:

  ```
  HEY THERE
  ```

The formatting of a sentence is decided during runtime. These formats should also allow for combinations with other formats:

- **bordered fancy** - Given the sentence, `["hey","there"]` it prints:

  ```
  ---------------
  |-+hey there+-|
  ---------------
  ```

- **fancy uppercase** - Given the sentence, `["hey","there"]` it prints:

  `-+HEY THERE+-`

To accomplish these features, you need to implement the decorator pattern. Each formatting will be a decorator for `Sentence` objects. These formats need to inherit from some abstract `FormattedSentence` class. This abstract class is specified to compose and inherit from sentence. The behavior that needs to be decorated is the `__str__()` function since you need to change how sentence is printed for every format.

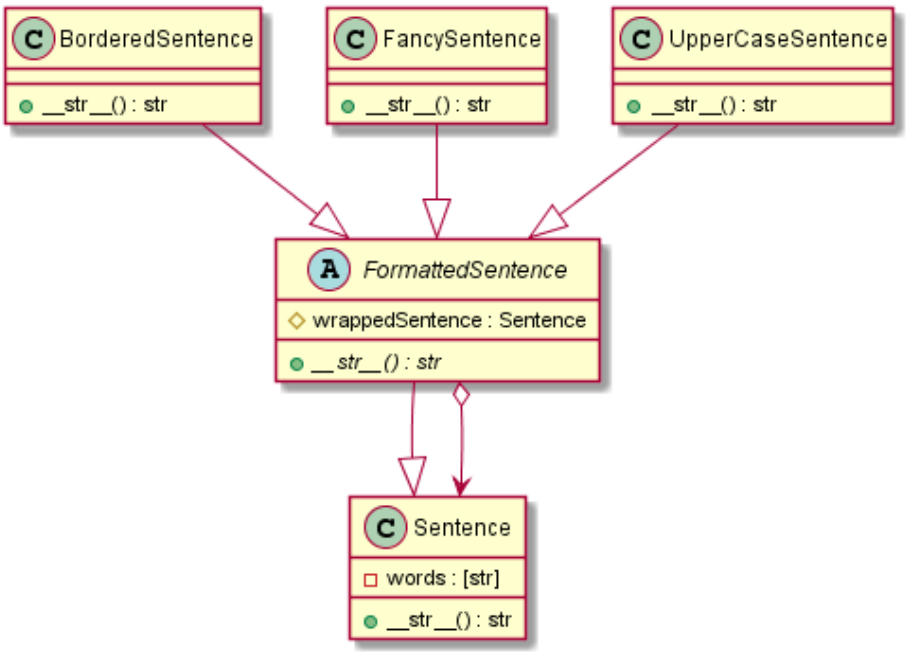

Figure 30: decorator example

Why this is elegant

- **Open/Closed Principle** - Decorators extend classes via inheritance. It is easier to add new decorators without touching any exiting code.
- A class which can have many variants because of diverse combinations of behaviors can be cleanly implemented using this pattern.
- You can arbitrarily mix and match decorators without the worry of polymorphic incompatibility during runtime

How to implement it

1. Create an abstract class `BaseDecorator` that specializes some `SimpleClass`. This `BaseDecorator` will also have the attribute `wrappedObject` which is a reference to an instance of the decorated `SimpleClass`. This attribute is set as protected so that it may be inherited by `BaseDecorator`s specializations. `BaseDecorator` also contains the abstract method `doSomething()`. This method's behavior, when invoked by `BaseDecorator`s specializations, changes depending on the decorations attached to `SimpleClass`
2. For every decoration, that can decorate `SimpleClass`, a specialization for `BaseClass` is created. These specializations implement `doSomething()` in a manner that augments/modifies `SimpleClass`'s own `doSomething()`

Adapter Pattern

Problem

As the system evolves, you'll likely encounter interfaces of instances that are incompatible with their intended clients. These interfaces do perform the necessary behavior, but maybe the method names are just different. This happens quite a lot since the interface of the dependency may be originally built for different reason. The interface may be an external module imported on existing client code. You can just change the incompatible interface to support the functionality you need but this is not always possible and may introduce code duplication.

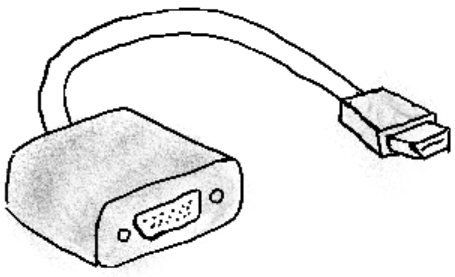

Figure 31: adapter

Solution

In the same way a usb-c interface is usable on a usb 2.0 using an adapter, you can use an incompatible service on a client as a compatible instance using the adapter pattern.

Say you have an instance of `AbstractService` (it could be any realization of `AbstractService`), that needs to be used like an instance of `RequiredInterface` by some client. What you need to do is to create an adapter to `AbstractService` called `ServiceAdapter` which realizes `RequiredInterface`. To adapt the instance of `AbstractService`, you have to compose it inside the `ServiceAdapter`. So that `serviceMethod1()` is adapted to `method1()`.

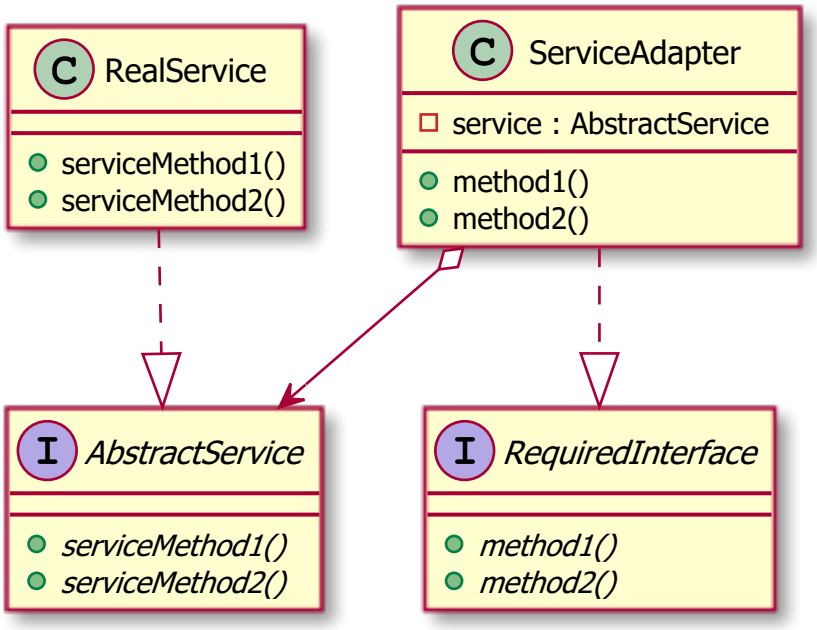

Figure 32: adapter

Whenever, a `ServiceAdapter` calls `method1()` it instead delegates the behavior to the embedded service, which instead calls `serviceMethod1()`

Example

Printable Shipments Looking back at our previous lab exercises, some of the example classes contain string representation but do not implement the `__str__()` function. An example of this is `Shipment` back from the factory method example. It does contain a string representation builder

called `shipmentDetails()`, but printing a shipment is quite tedious since you have to print, `s.shipmentDetails()`. You can replace the name of `shipmentDetails()` to `__str__()` but this will potentially affect other clients of shipment. You can add the `__str__()` function which does exactly the same but this may introduce unwanted code duplication.

The best solution for this problem is to create an adapter for shipment called `PrintableShipment`. This adapter will realize some `Printable` abstraction, which only contains the abstract method `__str__()`.

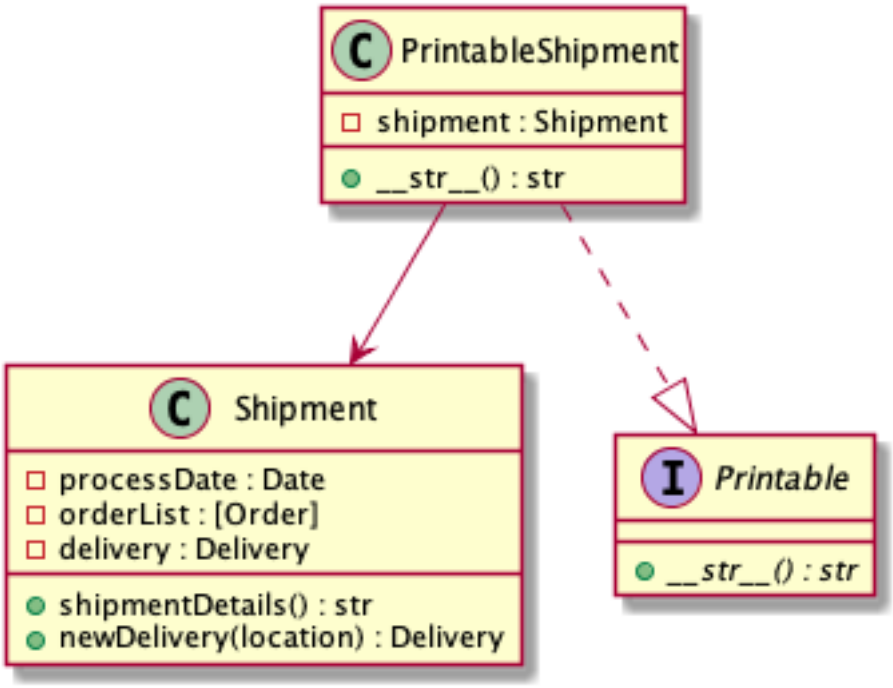

Figure 33: adapter example

Why this is elegant

- **Open/Closed Principle** - Instead of changing existing incompatible interfaces, you can extend them by creating adapters.
- Instead of cluttering up your code with duplicate functions and unused interface methods, you instead create adapters only when it is needed.

How to implement it

1. If you want an `AbstractService` to be used like a `RequiredInterface`, Create a `ServiceAdapter` that realizes `RequiredInterface` and

contains an attribute `service` that is a reference to an instance of `AbstractService`.

2. Inside `ServiceAdapter`, the implementations of `RequiredInterface`'s abstract methods are merely calls to the methods of the reference `service`.

Composite (Optional Read)

Problem

When entities in your system needs to be represented like trees, then you represent them like trees.

Solution

The composite pattern describes a tree structure described polymorphically. A tree node can either be a general tree or a leaf. in the composite pattern, a `Component` (tree node) can either be `Composites` (general tree), or a `Leaf`. Leaves and Composites are realizations of `Component`.

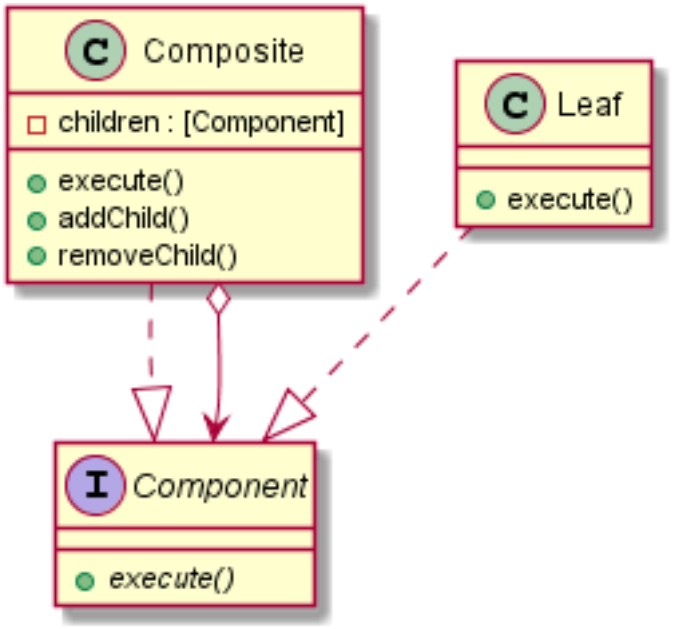

Figure 34: composite

Example

File System The file system in your computers are described using a tree structure. The entities in your file system are either directories or files.

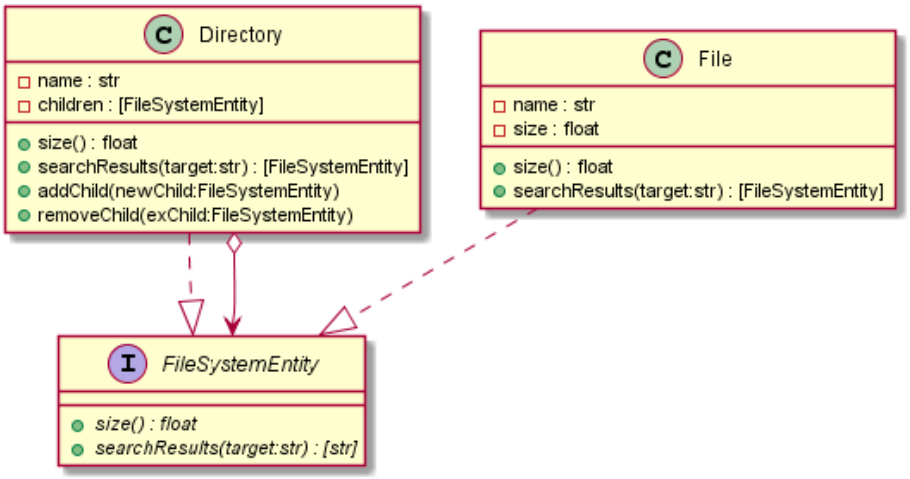

Figure 35: composite pattern

What you need to do is to implement a simulation of a file system. Each node of the file system should be able to call the following methods:

- `size()` - the size of a `File` is based on the attribute size which is set during the initialization of the `File` instance. The size of a `Directory` is the size of all the `FileSystemEntities` inside it.
- `searchResults(target)` - if used by a `File`, if the name of the file matches the `target` it returns a list containing the `File`, if not it returns an empty list. If used by a `Directory` returns a list containing all the of the instances of `FileSystemEntity` (including itself) that matches the target inside the `Directory`.

Why this is elegant

- **Open/Closed Principle** - You can introduce new component realizations in the system without touching any existing code.
- Working with composite trees are easier because of the polymorphism in the pattern

How to implement it

1. Create an abstraction called `Component` that contains the abstract methods that are supposed to be executed across all components.
2. The `Component` has two realizations, `Composites` and `Leafs`.
3. `Component` contains an attribute `children` which is a list of `Composite` instance references and the methods `addChild()` and `removeChild()` to

attach/detach `Composites`. It also has `execute()` which an implemented method from `Component`.

4. `Leaf` contains the method `execute()` as well.
5. When a `Composite`'s execute is invoked, it calls invokes all of its children's`execute()`. When `Leaf`'s execute is invoked it performs, leaf related behavior.

Facade (Optional Read)

Problem

When your system becomes large enough, parts of the system which is responsible for a single operation may involve interactions between multiple classes. Creating flexible and maintainable systems tend to look like this.

Looking from the outside, simple functionality (like borrowing a book or depositing money to an account) will appear complex since it involves multiple lines of object interaction.

Solution

To solve this issue, you create a straightforward interface, that contains methods to encapsulate complicated functionality in your subsystem. Instead of using the internal classes to perform some functionality, you call the facade interface's method instead.

Why this is elegant

- Implementing this pattern encapsulates complicated subsystem behavior into simple straightforward functions.

Optional Readings

Shvets A. (2018) Structural Patterns Accessed August 31, 2020

Lab Exercise 1 (Structuring the Document) (Optional)

> This is meant as review of imperative programming languages like C. This Lab Exercise is optional.

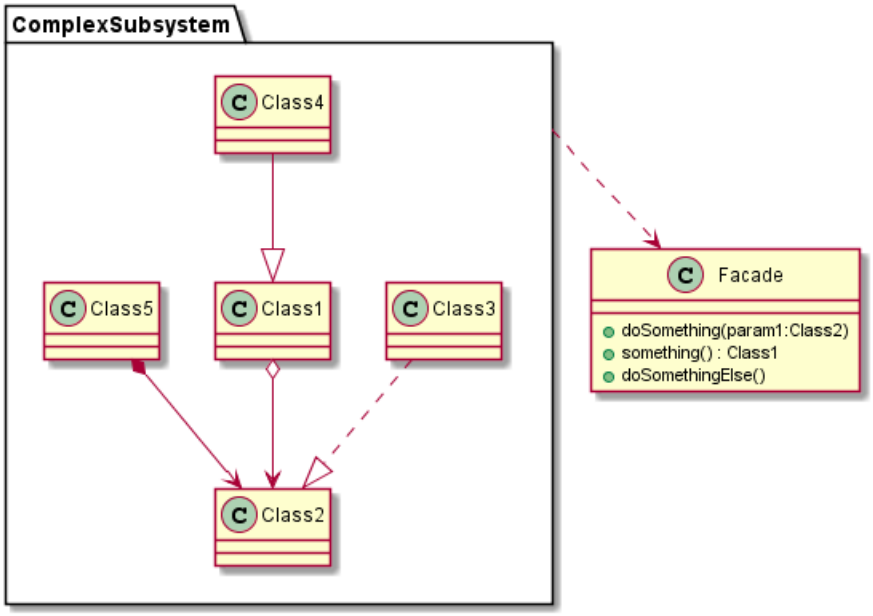

Figure 36: facade

Task

A document is represented as a collection paragraphs, a paragraph is represented as a collection of sentences, a sentence is represented as a collection of words and a word is represented as a collection of lower-case ([a-z]) and upper-case ([A-Z]) English characters. **Create a C program that will convert a raw text document into its component paragraphs, sentences and words**. To test your results, queries will ask you to return a specific paragraph, sentence or word as described below.

Words, sentences and paragraphs are represented using C structures

```
struct word {
    char* data;
};
```

words

```
struct sentence {
    struct word* data;
```

```
    int word_count;//the number of words in a sentence
};
```

sentences The words in a sentence are separated by one space. The last word does not end in space (" ").

```
struct paragraph {
    struct sentence* data  ;
    int sentence_count;//the number of sentences in a paragraph
};
```

paragraphs The sentences in the paragraph are separated by one period (".."). There are no spaces after the period.

```
struct document {
    struct paragraph* data;
    int paragraph_count;//the number of paragraphs in document
};
```

document The paragraphs in the document are separated by one newline (\n). The last paragraph does not end with a newline.

The paragraphs in the document are separated by one newline(\n). The last paragraph does not end with a newline.

For example:

```
Learning C is fun.
Learning pointers is more fun.It is good to have pointers.
```

The only sentence in the first paragraph could be represented as:

```
struct sentence first_sentence_in_first_paragraph;

first_sentence_in_first_paragraph.data = {"Learning", "C", "is",
 ↪  "fun"};
```

The first paragraph itself could be represented as:

```
struct paragraph first_paragraph;
```

```
first_paragraph.data = {{"Learning", "C", "is", "fun"}};
```

The first sentence in the second paragraph could be represented as:

```
struct sentence first_sentence_in_second_paragraph;

first_sentence_in_second_paragraph.data = {"Learning",
 ↪  "pointers", "is", "more", "fun"};
```

The second sentence in the second paragraph could be represented as:

```
struct sentence second_sentence_in_second_paragraph;

second_sentence_in_second_paragraph.data = {"It", "is", "good",
 ↪  "to", "have", "pointers"};
```

The second paragraph could be represented as:

```
struct paragraph second_paragraph;

second_paragraph.data = {{"Learning", "pointers", "is", "more",
 ↪  "fun"}, {"It", "is", "good", "to", "have", "pointers"}};
```

Finally, the document could be represented as:

```
struct document Doc;

Doc.data = {{{"Learning", "C", "is", "fun"}}, {{"Learning",
 ↪  "pointers", "is", "more", "fun"}, {"It", "is", "good", "to",
 ↪  "have", "pointers"}}};
```

Alicia has sent a document to her friend Teodora as a string of characters, i.e. represented by char* not a struct document. Help her convert the document to struct document form by completing the following functions:

- `void initialize_document(char *text)` - to intialise the document. to return the paragraph in the document.
- `struct paragraph kth_paragraph(int k)` - return the kth paragraph in the document
- `struct sentence kth_sentence_in_mth_paragraph(int k, int m)` - to return the kth sentence in the mth paragraph.
- `struct word kth_word_in_mth_sentence_in_nth_paragraph(int k,`

`int m, int n)` - to return the kth word in the mth sentence of the nth paragraph.

Input Format:

The first line contains the integer `paragraph_count`. Each of the next `paragraph_count` lines contains a paragraph as a single string. The next line contains the integer `q`, the number of queries. Each of the next `q` lines contains a query in one of the following formats:

- `1 k`: This corresponds to calling the function`kth_paragraph`.
- `2 k m`: This corresponds to calling the function `kth_sentence_in_mth_paragraph`.
- `3 k m n`: This corresponds to calling the function `kth_word_in_mth_sentence_in_nth_paragraph`.

Constraints

- The text which is passed to get_document has words separated by a space, sentences separated by a period and paragraphs separated by a newline.
- The last word in a sentence does not end with a space.
- The last paragraph does not end with a newline.
- The words contain only upper-case and lower-case English letters.
- 1 <= number of characters in the entire document <= 1000 .
- 1 <= number of paragraphs in the entire document <= 5.

Output Format

Print the paragraph, sentence or the word corresponding to the query to check the logic of your code.

Sample Input 0

```
2
Learning C is fun.
Learning pointers is more fun.It is good to have pointers.
3
1 2
2 1 1
3 1 1 1
```

Sample Output 0

```
Learning pointers is more fun.It is good to have pointers.
Learning C is fun
Learning
```

Explanation 0

- The first query returns the second paragraph.

- The second query returns the first sentence of the first paragraph.
- The third query returns the first word of the first sentence of the first paragraph

Assessment Criteria

This exercise is optional

Lab Exercise 2

Task

We've discussed functional programming paradigms using the language haskell as a representative. For this exercise, you'll familiarize yourselves on how to write pure functions in haskell. **Create a haskell file (".hs") containing the following functions below.**

For those of you using REPL's haskell compiler add the following snippet of code the the bottom of your function definitions. For those of you using ghc in your computers ignore this.

```
main :: IO ()
main = return ()
```

Easy functions

- `cube :: Int -> Int` - Consumes an integer and produces the cube of that integer
- `double :: Int -> Int` - Consumes an integer and produces the 2 times that integer

Recursive Functions

- `modulus :: Int -> Int -> Int` - Consumes two integers x and m and produces $x \mod m$
- `factorial :: Int -> Int` - Consumes an integer and produces the factorial of the integer
- `summation :: Int -> Int` - Consumes a natural number and produces the summation of numbers from 1 to n. $\sum_{i=1}^{n} i$.

```
summation :: Int -> Int
summation n = if (n <= 1) then n else (n + (summation
 ↪  (n-1)))
```

Higher order function

- `compose :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)` - Consumes two functions $f : \mathbb{Z} \to \mathbb{Z}$, and $g : \mathbb{Z} \to \mathbb{Z}$ and produces the function $f \circ g$.
- `subtractMaker :: Int -> (Int -> Int)` - Consumes an integer x and produces a function that consumes an integer y and produces $x - y$
- `applyNTimes :: (Int -> Int) -> Int -> Int -> Int` - Consumes a function $f : \mathbb{Z} \to \mathbb{Z}$ and and two integers n and x. `applyNTimes` produces an integer which is the result of the function applied to x, n-times. If n is less than or equal to 0 it must produce zero applications of f therefore it produces x.

Assessment Criteria

- Completeness of haskell functions - 40

Deadline Jan 11, 2022

Lab Exercise 3 (Higher Order Functions for List Comprehension)

Task

After familiarizing yourselves with haskell functions, lets move on to list comprehension functions. These functions are probably functional programming's most well-known contribution to other paradigms. Although these functions are already built in inside haskell, you are going to make your own versions of it. **Create a haskell file (".hs") containing the following functions below.** One of them (`my_filter`) is already written for you but please still include this function in your haskell file.

Lists in Haskell

Before you start implementing the functions in this exercise, you need to understand how lists work in Haskell. Haskell lists are written like a list in math. The Haskell list:

```
[1,2,3,4,5]
```

is basically equivalent to the list:

$$[1, 2, 3, 4, 5]$$

One of the most important things about lists is that you can access the list as a whole and you can access the elements inside the list. One thing you can do is you can concatenate lists using the **++** function:

```
ghci> [1,2,3,4,5] ++ [6,7,8]
[1,2,3,4,5,6,7,8]
```

You can find the length of the list using the **length** function:

```
ghci> length [1,2,3,4,5]
5
```

These are the different ways you can access the elements of the list and the sublists in the list:

head takes a list and returns the first element of the list

```
ghci> head [1,2,3,4,5]
1
```

tail takes a list and returns the same list except the first element of the list

```
ghci> tail [1,2,3,4,5]
[2,3,4,5]
```

init takes a list and returns the same list except the last element of the list

```
ghci> init [1,2,3,4,5]
[1,2,3,4]
```

last takes a list and returns the last element in the list

```
ghci> last [1,2,3,4,5]
5
```

Using these functions you can traverse a list using the head-tail recipe. For example, if you want to add 1 to each of the elements of the array:

```
addone :: [Int] -> [Int]
addone l =
  if length l == 0 then []
  else [(head l) + 1] ++ addone (tail l)
```

Let's dissect this function one by one, for the first line you can see the type signature `[Int]->[Int]` meaning `addone` accepts a list of `Int`s and returns a list of `Int`s. Any type `a` surrounded by brackets is a list of `a`s (`[a]` is a list of `a`'s, `[Char]` is a list of `Char`s, `[[Int]]` is a list of `[Int]`s or a list of list of `Int`s).

In the second line were binding the list we are passing to `l`.

The third line refers to the base case, this happens when the list is empty. When writing the base case, think about the most simple possible list the function may be applied to. The simplest case would be an empty list. When the list is empty we return `[]` which refers to an empty list.

And finally the last line refers to the general case. Here we are see the subexpression `[(head l) + 1]` which is a list containing one element namely, the first element of the list plus 1. And then we are concatenating this one element list to the result of the call `addone (tail l)` which is a recursive call to the rest of add one to the `tail` of l (or the rest of `l`). Assuming `addone` works perfectly, the recursive call `addone (tail l)` will return the tail of the `l` but the elements are added with one. By concatenating `[(head l) + 1]` with the result of this recursive call, we complete the desired result.

- Implement the higher order functions, `my_map`,`my_filter`, and `my_foldl` and `my_foldr`, `my_zip`
  - `my_map :: (a -> b) -> [a] -> [b]` - The `map` function accepts a function

    $$f$$

    and a list (with elements of type `A`)

    $$l = [l_1, l_2, l_3, ..., l_n]$$

    . It returns the list (with elements of type `B`):

    $$l' = [f(l_1), f(l_2), f(l_3), ..., f(l_4)]$$

    . The new list `map` produces is a list which is the image of `l` from the function `f`.
  - `my_filter :: (a -> Bool) -> [a] -> [a]` - The `filter` function

accepts a predicate

$$f$$

and a list

$$l = [l_1, l_2, l_3, \cdots, l_n]$$

. `filter` returns a new list

$$l'$$

such that the contents satisfy

$$f(l_i)$$

is true, retaining the order it appears in

$$l$$

.

BONUS (here's `my_filter` solved for you, use this as a guide):

```
my_filter :: (a -> Bool) -> [a] -> [a]
my_filter p l =
    if length l == 0 then []
    else (if p (head l) then [head l] else []) ++
     ↪  my_filter p (tail l)
```

The notable part of this `my_filter`'s body is the last. The non-base case clause evaluates the following line

```
(if p (head l) then [head l] else []) ++ my_filter p
 ↪  (tail l)
```

The first part is the if-then-else clause `(if p (head l) then [head l] else [])` which evaluates to either the list containing the first element of `l` (`[head l]`) or an empty list (`[]`). If the first element (`head l`) satisfies the predicate `p` (therefore the if clause contains the expression `p (head l)`), then the if-then-else clause evaluates to `[head l]` otherwise it evaluates to `[]`. Whatever, the `if-then-else` clause evaluates to is then concatenated to the result of the recursive call to the tail of `l` (`my_filter p (tail l)`). Every time the function recurses, the first element is either concatenated or not concatenated to the rest of the list, this filtering out all elements that do not satisfy the predicate.

– `my_foldl :: (a -> a -> a) -> a -> [a] -> a` - The `foldl` func-

tion accepts a function

$$f$$

, a list

$$l = [l_1, l_2, l_3, ..., l_n]$$

and an initial value

$$u$$

. The `foldl` function returns the value

$$f(f(f(f(u, l_1), l_2), l_3), l_n)$$

.

- `my_foldr :: (a -> a -> a) -> a -> [a] -> a` - The `foldr` function accepts a function

  $$f$$

  , a list

  $$l = [l_1, l_2, l_3, ..., l_n]$$

  and an initial value

  $$u$$

  . The `foldr` function returns the value

  $$f(l_1, f(l_2, f(l_3, f(l_n, u))))$$

- `my_zip :: (a -> b -> c) -> [a] -> [b] -> [c]` - THe `zip` function accepts a function

  $$f$$

  and two lists

  $$l = [l_1, l_2, l_3, ..., l_n]$$

  ,

  $$m = [m_1, m_2, m_3, ..., m_n]$$

  and returns a new list,

  $$k = [f(l_1, m_1), f(l_2, m_2), f(l_3, m_3), \cdots, f(l_n, m_n)]$$

- Without using loops (use the functions above instead), write the following functions.
  - Given a list of numbers, return the sum of the squares of the numbers

- Given three lists, a list of first names, A, a list of middle names B, and a list of surnames C. Return a list of whole name strings (list of chars) (`[firstname] [middle initial]. [lastname]`) where the length of the string (including spaces and period) is an even number. Example

```
ghci> wholeName ["Foo", "Bar", "Foo"] ["Middle",
 ↪  "Center", "Name"] ["Lastn", "Surname", "Abcd"]
["Foo M. Lastn", "Bar C. Surname"]
```

  ("Foo N. Abcd" is filtered out because it has 11 characters)

Assessment Criteria

- Completeness of haskell functions - 35

Deadline January 11, 2022

Lab Exercise 4 (Drama in the Clue Mansion)

Task

You can use a knowledge base to represent human relationship networks. This is what you will be doing for this exercise. I've written a few example facts, and rules as your guide below. **Create a prolog knowledge base (".pl") containing the following facts. Also, create a text file containing the answers to to queries you can below.**

- Create a knowledge base and place them all inside a file with a ".pl" extension
  1. *Miss Scarlet, Mrs. White, Mrs. Peacock, Dr. Orchid are female*
  2. Prof. Plum, Colonel Mustard, and Rev. Green are all male
  3. *Miss Scarlet hates Rev. Green.*
  4. Rev. Green hates Miss Scarlet
  5. Prof. Plum and Mrs. White hate each other.
  6. Col. Mustard hates all females and Prof. Plum.
  7. Miss Scarlet and Mrs. Peacock both like Dr. Orchid.
  8. Dr. Orchid likes Mrs. Peacock
  9. Miss Scarlet likes Mrs. White
  10. Miss Scarlet and Prof. Plum like each other.
  11. Prof. Plum likes everyone Col. Mustard hates.
  12. *People who hate each other are enemies*
  13. People who like each other are friends

14. The enemies of someone's enemies is his/her friend.

- Based on the knowledge base you created, ask it the following queries by running `swipl labExer4.pl`. Write the solutions to each query into a text file.
  1. *Which pairs are enemies?*
  2. Which pairs are friends?
  3. Which people are liked by Prof. Plum.
  4. Which people like themselves?
  5. Which males are liked by females? (this query must be written as a conjunction)
  6. Which people are hated by the one they like? (this query must be written as a conjunction)

Some example facts and rules as guide

(don't skip writing these facts and rules in your knowledge base so that it works)

```
%Propositions in item 1
female(scarlet).
female(peacock).
female(orchid).

%Proposition in item 2 (Miss Scarlet hates Rev. Green)
hates(scarlet, green).

%Rule in item 12 (People who hate each other are enemies)
enemies(X,Y) :- hates(X,Y), hates(Y,X).
```

Some example queries

(Although the answers are already provided here, still, copy them on the text file containing the answers from the other queries).

You should get similar answers to the following queries

Which pairs are enemies?

```
?- enemies(A,B).

A = scarlet,
B = green ;
A = green,
B = scarlet ;
A = plum,
```

```
B = white ;
A = white,
B = plum ;
false.
```

Which males are liked by females?

```
?- likes(A,B), female(A), male(B).
```

```
A = scarlet,
B = plum ;
false.
```

Assessment Criteria

- Completeness of knowledge base - 20
- Accuracy of query results - 20

Deadline January 11, 2022

Lab Exercise 5 (Snakes)

Task

To familiarize yourselves with how python syntax works, create the python library (".py") and **write the following functions inside it:**

```
def doubledInt(x:int) -> int:
    #accepts an in and return the double of that int
```

double

```
def largest(x:float,y:float) -> float:
    #accepts two floats and returns the larger value
```

largest

```
def isVertical(a:(float,float),b:(float,float)) -> bool:
    #accepts two (float,float) tuples which represent two points
    ↪ in a cartesian plane and returns true if the points
    ↪ describe a vertical line and false otherwise
```

vertical line

```
def primes(n:int) -> [int]:
    #accepts an integer n and returns the first n primes
```

primes

```
def fibonacciSequence(n:int) -> [int]:
    #accepts an integer n and returns a list containing the first
     ↪  n elements of fibonacci sequence (starting with 0 and 1)
```

fibonacci

```
def sortedIntegers(l:[int]) -> [int]:
    #accepts a list of integers and returns a list with the same
     ↪  integers sorted from smallest to highest
```

sorting

```
def sublists(l:[int]) -> [[int]]:
    #accepts a list of integers and returns all the sublists of
     ↪  the list. Sublists are contiguous chunks of a list
     ↪  (including an empty list and the list itself). [1,2],
     ↪  [2], [], [2,3,4], and [1,2,3,4,5] are sublists of
     ↪  [1,2,3,4,5] but [3,5] and [1,2,3,4,6] are not.
```

sublists

fast modular exponentiation (optional) this should work for large values of b and p. To do this implement fast modular exponentiation from CMSC 56

```
def fme(b:int,p:int,m:int) -> int:
    #accepts 3 ints, b,p and m and computes b^p mod m
```

Assessment Criteria

- Completeness of python functions - 35
- Correctness of function names and parameter names - 5

Deadline January 11, 2022

Lab Exercise 6 (Borrowing from the Library)

Task

You are to implement the following system. This system represents the borrowing and returning functions of a library. Here's the class diagram. **Edit the python file that came with this document.** The edited file is what you are going to submit.

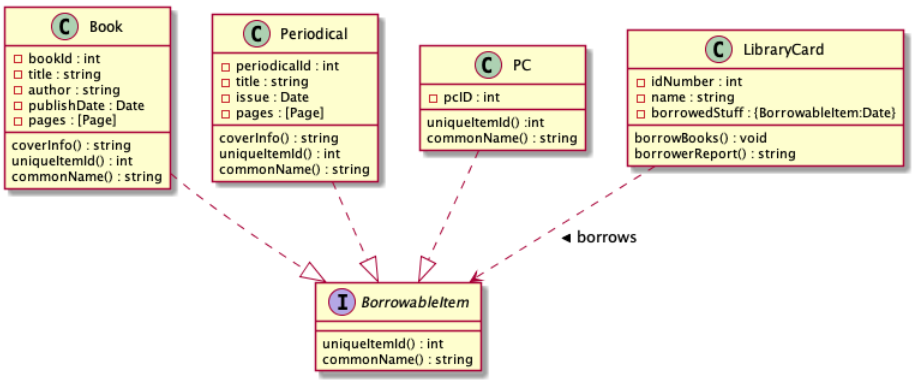

Figure 37: UML

There is some code already written for you here:

```
from abc import ABC, abstractmethod

class Date:
    def __init__(self, month:int, day:int, year:int):
        self.__month = month
        self.__day = day
        self.__year = year
    def mdyFormat(self) -> str:
        return str(self.__month) + "/" + str(self.__day) + "/" +
         ↪  str(self.__year)

class Page:
    def __init__(self, sectionHeader:str, body: str):
        self.__sectionHeader = sectionHeader
        self.__body = body
```

```
class BorrowableItem(ABC):
    @abstractmethod
    def uniqueItemId(self) -> int:
        pass
    @abstractmethod
    def commonName(self) -> str:
        pass



class Book(BorrowableItem):
    def __init__(self, bookId:int, title:str, author:str,
     ↪  publishDate:Date, pages: [Page]):
        self.__bookId = bookId
        self.__title = title
        self.__publishDate = publishDate
        self.__author = author
        self.__pages = pages
    def coverInfo(self) -> str:
        return "Title: " + self.__title + "\nAuthor: " +
         ↪  self.__author
    def uniqueItemId(self) -> int:
        return self.__bookId
    def commonName(self) -> str:
        return "Borrowed Item:" + self.__title + " by " +
         ↪  self.__author


class LibraryCard:
    def __init__(self, idNumber: int, name: str, borrowedItems:
     ↪  {BorrowableItem:Date}):
        self.__idNumber = idNumber
        self.__name = name
        self.__borrowedItems = borrowedItems
    def borrowItem(self,book:BorrowableItem, date:Date):
        self.__borrowedItems[book] = date
    def borrowerReport(self) -> str:
        r:str = self.__name + "\n"
        for borrowedItem in self.__borrowedItems:
```

```
            r = r + borrowedItem.commonName() + ", borrow date:"
 ↪  + self.__borrowedItems[borrowedItem].mdyFormat() + "\n"
        return r
```

Creating an instance of a `BorrowableItem` (in this case an instance of the particular realization, `Book`) is done using the following code.

```
b:BorrowableItem = Book(10991,"Corpus Hermeticum", "Hermes
 ↪  Trismegistus", Date(9,1,1991), [])
print(b.commonName()) #commonName() returns the string
 ↪  representation of a borrowable item
```

Creating an instance of a `LibraryCard` is done using the following.

```
l:LibraryCard = LibraryCard(9982,"Rubelito Abella",{})
```

A library card borrows something using the `borrowItem(BorrowableItem)` method. And the `borrowerReport()` prints the library card owners name and the items he/she has borrowed.

```
l.borrowItem(b,Date(9,25,2019))
print(l.borrowerReport())
```

What you should do:

The class definitions above are still missing `Periodical` and `PC`.

- a **`Periodical`** represents a periodical (newspaper, magazines, etc). It is a realization of a `BorrowableItem`. It contains the following methods and attributes:
  - `__init__`: initializes a periodical instance with the attributes `__periodicalID:int`, `__title:str`, `__issue:Date`, `__pages:[Page]`
  - `__periodicalID:int`: unique id for a periodical
  - `__title:str`: The title of the periodical ("National Geographic", "New York Times")
  - `__issue:Date`: The date when the issue was published
  - `__pages:[Page]`: A list of `Page`s that represent the contents
  - `uniqueItemId()`: Returns `periodicalID`
  - `commonName():str`: (Implementation of the abstract method from `BorrowableItem`). It returns the title and the issue date in month/date/year format as a string for example "National Geographic issue: 4/6/2001")

- a `PC` represents a library PC. It is a realization of a `BorrowableItem`. It contains the following methods and attributes:
  - `__init__`: initializes a PC instance with the attribute `__pcID:int`.
  - `__pcID:int` : unique id for a PC
  - `uniqueItemId():int`: Returns `__pcID`
  - `commonName():str`: (Implementation of the abstract method from `BorrowableItem`). It returns "PC<___pcID>" (the string "PC" followed by the value attribute `__pcID`, for example "PC1342")

Add the following methods to `LibraryCard` and implement them

- **`returnItem(b:BorrowableItem):`** : returns nothing, it removes the `BorrowableItem`, `b` from the `__borrowedItems` dictionary.
- **`penalty(b:BorrowableItem,today:Date):float`** : returns a float which is the calculated penalty for `BorrowableItem`, `b` when returned today. Every day after the due date the penalty increases by 3.5. An item which is overdue for 4 days has a penalty of 14.
- **`itemsDue(today:Date):[BorrowableItem]`** : returns a list of `BorrowableItems` which are on or past the due date. The due date for a `Book` is 7 days, a `Periodical` is 1 day, and a `PC` is 0 days.
- **`totalPenalty(today:Date):float`** : returns a float which is the total penalty for all the overdue items when calculated today

  The parameter `today:Date` represents the date today. For example `itemsDue(today:Date)` will return a list of `BorrowableItems` past the due date if if the date was `today`.

Feel free (in fact you are encouraged) to add extra methods to any of the classes above that will help you in implementing the whole system. Just make sure the extra methods don't unnecessarily expose hidden attributes.

Assessment Criteria

- Completeness of `BorrowableItems` attributes and methods - 20
- Completeness of `LibraryCard` methods - 20
- Correctness of attribute and methods names - 10

Deadline January 11, 2022

Lab Exercise 7 (Designing an OOP System)

Task

Banking System

You are to design a banking system that supports the following features:

- 3 kinds of bank accounts:
  - payroll - this account can withdraw funds. This account can receive funds from transfers but it cannot transfer.
  - debit - this account can withdraw and deposit. This account can transfer funds to any account. Each month the balance compoundingly increases based on an interest rate. This account has a required balance that must be kept every month. If this balance is not kept the account becomes inactive (It's up to you to set that required balance amount).
  - credit - this account can withdraw. Withdrawing increases the credit balance (the amount owed). This account has a credit limit. The credit balance cannot exceed this credit limit. It can deposit which basically deducts the deposited amount from the current credit balance. This account can also transfer funds to any account which also increases the credit balance. It also has an interest rate where the credit balance increases compundingly each month. (it's also up to you to set the credit limit)
- applying all account changes - every month this is invoked, this changes the debit balances and credit balances based on interest. It also changes account status (deactivating/activating) once this triggers. (Note: you do not actually need to implement this in such a way that the changes happens automatically every month. Assume that this method is invoked every month)
- fund transfer from one account to another (all kinds of accounts can be transferred to, Note: do not allow a debit accounts to transfer an amount greater than the balance or a credit account to transfer an amount that will make the credit balance exceed the credit limit)
- deactivate and activate accounts
- account withdrawal (do not allow withdrawals that exceed the balance and the credit limit as well)

- account deposit (payroll accounts can't deposit)
- show the balance report of an account
- show account information (name of the account owner, type of the account and active/inactive status of the account)

You can place all of the classes into a single python file and you can also separate them into their own python files (just make sure you're importing correctly). If you put them all into a single python file, submit the python file. If you separate them into multiple python files, package them all into a zipped folder and submit the zipped folder.

Feel free to add your own extra methods and extra features to this system. This is an open ended design exercise, go ahead and be creative. For example, I have not specified how the system might react if there are invalid withdrawals or transfers, it's up to you to implement those (do you print a message? or raise an error?)There is no single correct class architecture for this system.

I'm not actually expecting you to build perfectly a elegant architecture for this system, this exercise's purpose is to get you used to building OOP systems from scratch.

Deadline January 11, 2022

Lab Exercise 8 (Shipment)

Task

Online Marketplace Delivery Consider you're developing the product delivery side of an online marketplace app (think Amazon/Lazada). Your app is on its early stage so their is only one delivery option, standard nationwide delivery that takes a minimum of 7 days.

What you have is `Shipment` class that contains a `StandardDelivery` class. Inside the shipment class is the `shipmentDetails()` builder which builds a string representing the details of the shipment, this includes the delivery details (which requires access to the composed `StandardDelivery` instance). Inside the constructor of `Shipment` an instance of `StandardDelivery` is created so that every `Shipment` is set to be delivered using standard delivery.

```
from datetime import date,timedelta

class Order:
```

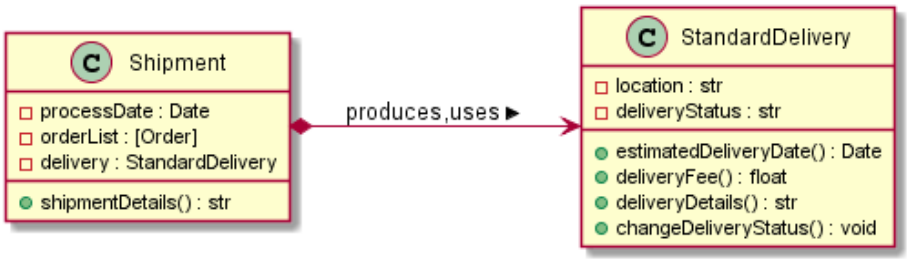

Figure 38: online marketplace

```
    def __init__(self,productName:str, productPrice:float):
        self.__productName = productName
        self.__productPrice = productPrice
    def orderString(self) -> str:
        return "%s P%.2f" %
         ↪  (self.__productName,self.__productPrice)
    def price(self) -> float:
        return self.__productPrice

class StandardDelivery:
    def __init__(self,location:str):
        self.__location = location
        self.__deliveryStatus = "Processing"
    def deliveryDetails(self) -> str:
        r = "STANDARD DELIVERY\nDELIVER TO:%s\nDELIVERY STATUS:
↪  %s\nDELIVERY FEE: P%.2f" %
↪  (self.__location,self.__deliveryStatus,self.deliveryFee())
        return r
    def deliveryFee(self) -> float:
        return 500
    def estimatedDeliveryDate(self,processDate:date) -> float:
        return processDate + timedelta(days = 7)
    def changeDeliveryStatus(self,newStatus:str):
        self.__deliveryStatus = newStatus

class Shipment:
    def __init__(self, orderList:[Order], processDate: date,
     ↪  location):
        self._orderList = orderList
```

```
        self._processDate = processDate
        self._delivery = StandardDelivery(location)

    def totalPrice(self) -> str:
        t = 0.0
        for order in self._orderList:
            t+=order.price()
        return t

    def shipmentDetails(self) -> str:
        r = "ORDERS:" + str(self._processDate) + "\n"
        for order in self._orderList:
            r += order.orderString() + "\n"
        r += "\n"
        r += "TOTAL PRICE OF ORDERS: P"  + str(self.totalPrice())
↪   + "\n"
        r += self._delivery.deliveryDetails() + "\n\n"
        r += "PRICE WITH DELIVERY FEE : P" +
↪   str(self.totalPrice()+self._delivery.deliveryFee()) + "\n"
        r += "ESTIMATED DELIVERY DATE: " +
↪   str(self._delivery.estimatedDeliveryDate(self._processDate))
        return r

o = [Order("Surface Pro 7",40000),Order("Zzzquil",900)]
s = Shipment(o,date(2019,11,1),"Cebu City")
print(s.shipmentDetails())
```

This system does work. It works but it is still inelegant. As soon as your app grows, you will incorporate new delivery options like express delivery, or pickups or whatever. Every time you need to add a new delivery method you will need to perform surgery in `Shipment` since the `StandardDelivery` instance is created inside the constructor of `Shipment`. `Shipment`'s code is too coupled with `StandardDelivery`.

To solve this you need to implement the factory method pattern. Right now shipment is a factory since it constructs its own instance of `StandardDelivery`. To refactor this into elegant code, you need to so create an abstraction called`Delivery` first to support polymorphism. Inside `Shipment` instead of creating instances of `Delivery`'s using a constructor, you invoke a factory method that encapsulates the instantiation of `Delivery`. In this case we

name this method `newDelivery()`. All it does is return an instance of `StandardDelivery` using its constructor.

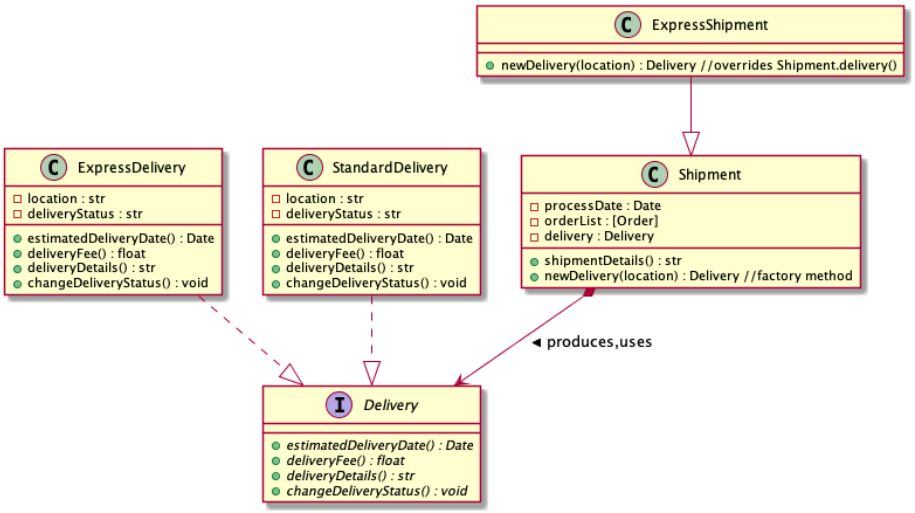

Figure 39: online marketplace

In this new architecture, whenever there are new delivery methods a shipment could have, all you have to do is to create a realization of that delivery method. In this case the new delivery method is `ExpressDelivery` which delivers the next day but is twice as expensive. And instead of changing `Shipment` (violates Open/Closed Principle), you make an extension to `Shipment`. This extension is the specialization to shipment called `ExpressShipment` (a shipment that uses express delivery). In this specialization, you only need to override the factory method delivery, so that every instance of delivery construction creates `ExpressDelivery`. The difference between `ExpressDelivery` and `Delivery` is that `ExpressDelivery` has a delivery fee of 1000 and the estimated delivery date is one day after the processing date.

Complete the system using the factory method pattern.

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2021

Lab Exercise 9 (Bootleg Text-based Zelda Game)

Task

You're creating the dungeon encounter mechanics of some bootleg text-based zelda game. In this game,every time you enter a dungeon, you encounter 0-8 monsters (the exact number is randomly determined). There are 3 types of monsters, bokoblins, moblins, and lizalflos (different types have different moves). The exact type of monster is randomly decided as well.

```
from random import randint

class NormalBokoblin:
    def bludgeon(self):
        print("Bokoblin bludgeons you with a boko club for 1
        ↪  damage")
    def defend(self):
        print("Bokoblin defends itself with a boko shield")
    def announce(self):
        print("A bokoblin appeared")
    def move(self):
        if randint(1,3) > 1:
            self.bludgeon()
        else:
            self.defend()

class NormalMoblin:
    def stab(self):
        print("Moblin stabs you with a spear for 3 damage")
    def kick(self):
        print("Moblin kicks you for 1 damage")
    def announce(self):
        print("A moblin appeared")
    def move(self):
        if randint(1,3) > 1:
            self.stab()
        else:
            self.kick()

class NormalLizalflos:
    def throwBoomerang(self):
```

```
        print("Lizalflos throws its lizal boomerang at you for 2
        ↪  damage")
    def hide(self):
        print("Lizalflos camouflages itself")
    def announce(self):
        print("A lizalflos appeared")
    def move(self):
        if randint(1,3) > 1:
            self.throwBoomerang()
        else:
            self.hide()



class Encounter:
    def __init__(self):
        self.__enemies = []
        for i in range(randint(0,8)):
            r = randint(1,3)
            if r == 1:
                self.__enemies.append(NormalBokoblin())
            elif r==2:
                self.__enemies.append(NormalMoblin())
            else:
                self.__enemies.append(NormalLizalflos())

    def announceEnemies(self):
        print("%d monsters appeared" % len(self.__enemies))
        for enemy in self.__enemies:
            enemy.announce()

    def moveEnemies(self):
        for enemy in self.__enemies:
            enemy.move()



encounter = Encounter()
encounter.announceEnemies()
```

```
print()
encounter.moveEnemies()
```

Right now the game works like this:

As soon as you enter the dungeon, all the enemies are announced:

```
5 monsters appeared
A lizalflos appeared
A lizalflos appeared
A lizalflos appeared
A moblin appeared
A moblin appeared
```

After this, each enemy in the encounter attacks. They randomly pick an attack from their moveset.

```
Lizalflos thorws its lizal boomerang at you for 2 damage
Lizalflos thorws its lizal boomerang at you for 2 damage
Lizalflos camouflages itself
Moblin stabs you with a spear for 3 damage
Moblin stabs you with a spear for 3 damage
```

The encounter ends with Link dying since you haven't coded anything past this part.

You decide to make things exciting for your game by adding harder dungeons, medium dungeon and hard dungeon.

Medium dungeon Instead of encountering, normal monsters you encounter stronger versions of the monsters, these monsters are blue colored:

- **Blue Bokoblin**
  - equipped with a spiked boko club and a spiked boko shield
  - bludgeon deals 2 damage
- **Blue Moblin**
  - equipped with rusty halberd
  - stab deals 5 damage
  - kick deals 2 damage
- **Blue Lizalflos**
  - equipped with a forked boomerang
  - throw boomerang deals 3 damage

Hard dungeon These monsters are silver colored extra stronger versions of the monsters

- **Silver Bokoblin**
  - equipped with a dragonbone boko club and a dragonbone boko shield
  - bludgeon deals 5 damage
- **Silver Moblin**
  - equipped with knight's halberd
  - stab deals 10 damage
  - kick deals 3 damage
- **Silver Lizalflos**
  - equipped with a tri-boomerang
  - throw boomerang deals 7 damage

To seamlessly incorporate these harder monsters in your system, you need to create an abstract factory for each dungeon difficulty. There are now three variants for each monster. For every variant, there is a factory that spawns new instances of each monster. **Complete the system using the abstract factory pattern.**

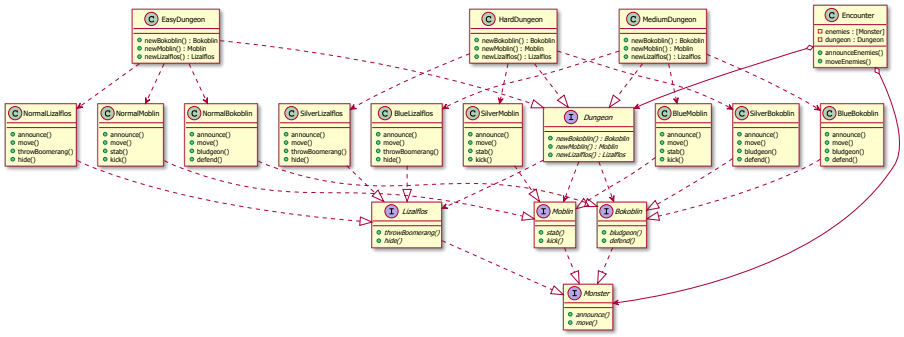

Figure 40: abstract factory example

Assessment Criteria

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 21, 2022

Lab Exercise 10 (Fraction Calculator)

Task

You're creating a less sophisticated version of a fraction calculator. This calculator only has arithmetic operations inside it, addition, subtraction, division, and multiplication. Inside this calculator, a calculation is represented in a `Calculation` instance. Every calculation has four parts:

- `__left` - represents the left operand fraction
- `__right` - represents the right operand fraction
- `__operation` - represents the operation $(+,-,\times,\div)$
- `__answer` - represents the solution of the operation

```
class Fraction:
    def __init__(self,num:int,denom:int):
        self.__num = num
        self.__denom = denom
    def num(self):
        return self.__num
    def denom(self):
        return self.__denom
    def __str__(self) -> str:
        return str(self.__num) + "/" + str(self.__denom)

class Calculation:
    def
     ↪  __init__(self,left:Fraction,right:Fraction,operation:Operation):
     ↪  #will cause an error when ran since Operation does not
     ↪  exist yet
        self.__left = left
        self.__right = right
        self.__operation = operation #the parameter that
         ↪  represents the operation
        self.__answer = None #the answer should be calculated
         ↪  here

    def __str__(self):
        return str(self.__left) + " " + str(self.__operation) + "
         ↪  " + str(self.__right) + " = " + str(self.__answer)
```

```
f:Fraction = Fraction(1,4)
print(f)
```

Python does indeed support higher order functions but your boss is anti-functional programming so he forbids the use these features. Because of this you decide to implement the strategy pattern.

To do this, you need to create an abstraction called `Operation` to represent the different operations. For each operation, you create a class that realizes `Operation`.

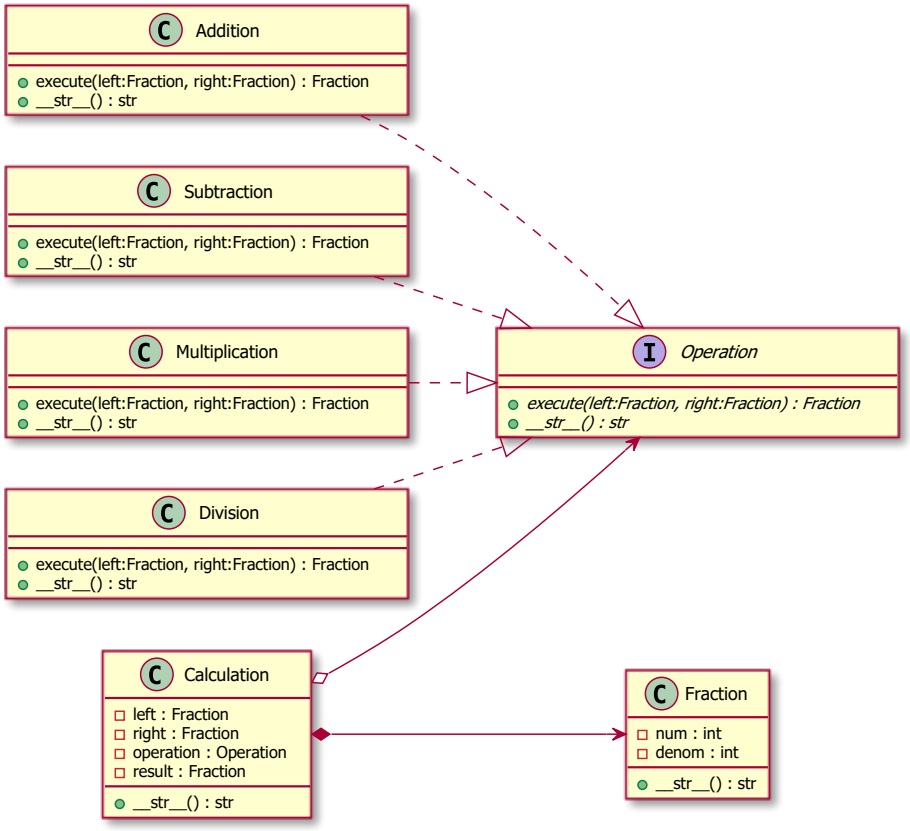

Figure 41: strategy pattern example

> `execute()` should have been named like a builder method (something like `solution()`), I'm keeping the name `execute()` since this is how Strategy patterns usually names this particular method.

Complete the system using the strategy pattern.

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

Lab Exercise 11 (States of Matter)

Task

The state of any given matter is dependent on the pressure and temperature of its environment. If you heat up some liquid enough it will turn to gas, if you compress it enough it will become solid.

```
class Matter:
    def __init__(self,name:str):
        self.__name = name
        self.__state = None #change this to the appropriate
        ↪  initial state (liquid)
    def changeState(self,newState):
        pass
    def compress(self):
        pass
    def release(self):
        pass
    def cool(self):
        pass
    def heat(self):
        pass
    def __str__(self):
        return "%s is currently a %s" %
        ↪  (self.__name,self.__state) #formatting strings just
        ↪  like you format strings in C
```

You are to build a less sophisticated version of this model in code. Matter comes in three states, solid, liquid, and gas. The state of the matter may change if you put/remove pressure on it or heat/cool it.

The state diagram would look something like this:

To implement something like this, you would need to create matter which owns an attribute called `state` which represents the matter's current state. Since

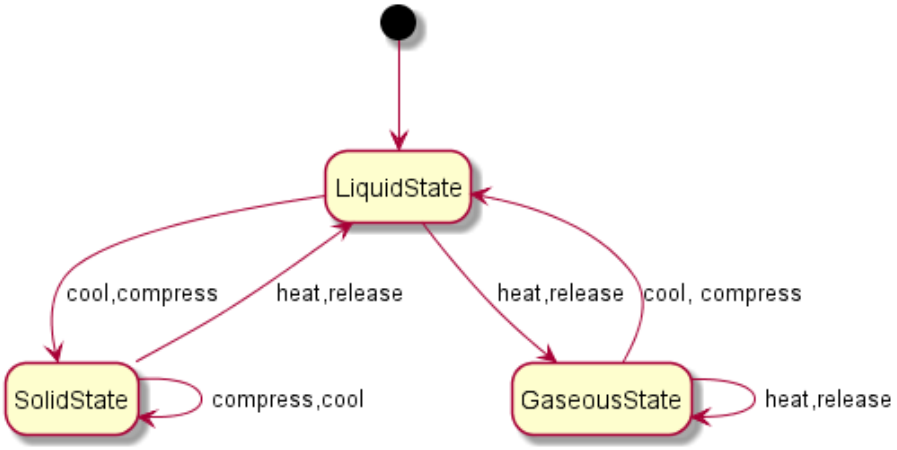

Figure 42: state diagram example

there are three states, you create three realizations to a common abstraction to state.

When you compress/release/cool/heat the matter, you delegate the appropriate behavior and state change inside `state`'s version of that behavior. Each `State` realization will need a backreference to the `Matter` that owns it so that it can change its state ("it" referring to the `Matter` instance that owns this `State` instance).

> Delegating behavior to the composed state means that, when the `Matter` instances invoke, `compress()`, `relaease()` `heat()`, and `cool()`, the composed `State` owned by the matter calls its own version of `compress()`, `relaease()` `heat()`, and `cool()`.
>
> Matter owns an instance of `State`, and that instance has an attribute called `matter`. The attribute `matter` is the reference to the instance of `Matter` that owns it. The `State` instance needs this reference so that it can change the matter's state when it is compressed, released, heated, or cooled.

Complete the system using the state pattern.

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 21, 2022

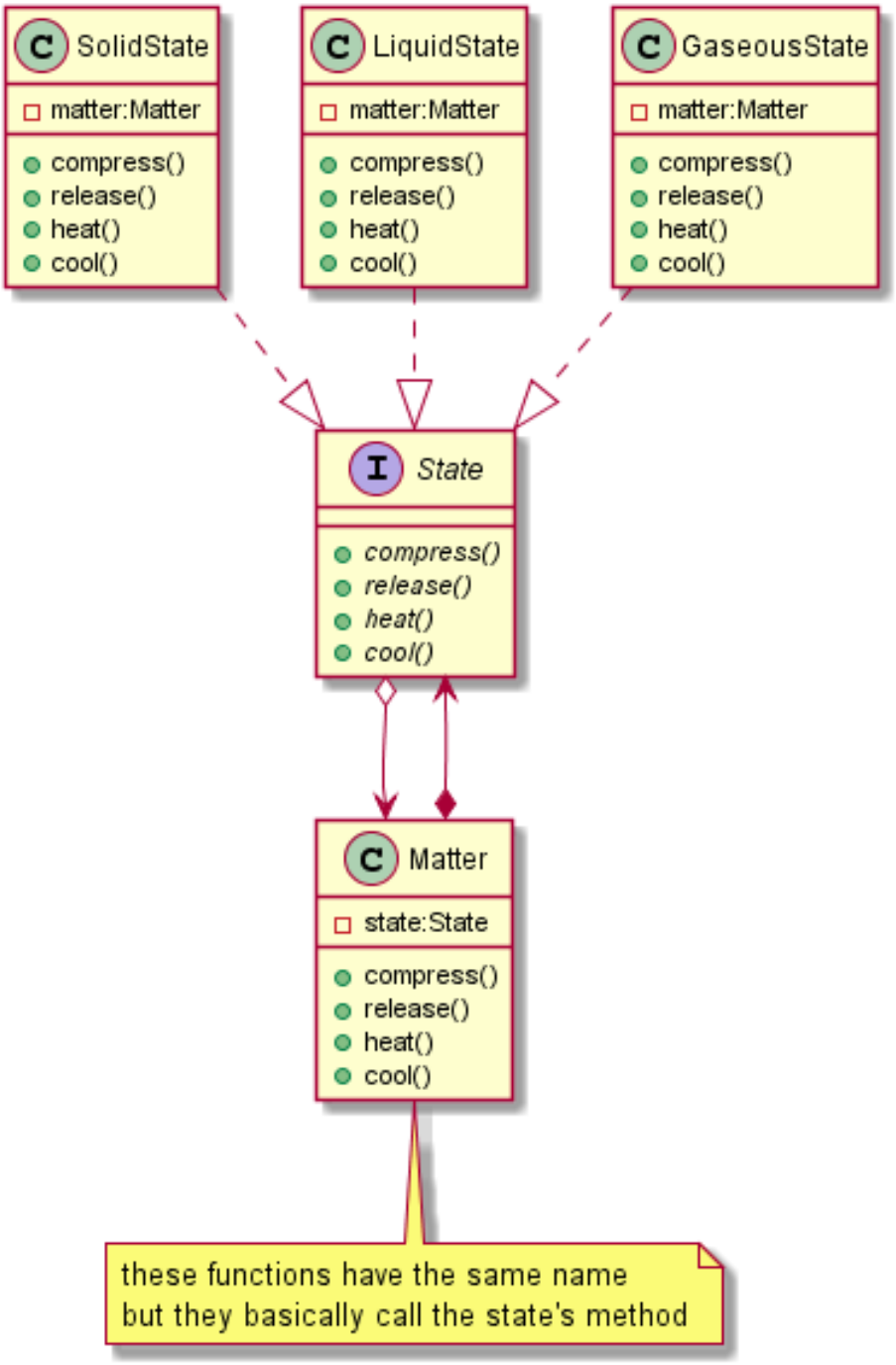

Figure 43: state example

Lab Exercise 12 (Zooming through a Maze)

Task

You're creating a maze navigation game thing. This is what the application currently has right now:

- `Board` - this represents the layout of the maze. The layout is loaded from a file. It has these attributes:
  - `__isSolid` - this is a 2 dimensional grid encoded as a nested list of booleans which represents the solid boundaries of the maze. For example if `__isSolid[row][col]` is true then it means that that cell on (row,col) is a boundary
  - `__start` - a tuple of two integers that represent where the character starts
  - `__end` - tuple of two integers that represent the position of the end of the maze
  - `__cLoc` - tuple of two integers that represents the current location of the character
  - `moveUp()`, `moveDown()`, `moveLeft()`, `moveRight()` - moves the character one space, in the respective direction. The character cannot move to a boundary cell, it will raise an error instead.
  - `canMoveUp()`, `canMoveDown()`, `canMoveLeft()`, `canMoveRight()` - returns true if the cell in the respective direction is not solid.
  - `__str()__` string representation of the board. It shows which are the boundaries and the character location

Board

```
class BoundaryCollisionError(Exception):
    def __init__(self,point):
        self.collidingBoundary = point

class Board:
    def __init__(self,filename:str="boardFile.py"):
        self.__isSolid = []
        with open(filename,"r") as f:
            self.__start = tuple(map(int,f.readline().split()))
            self.__end = tuple(map(int,f.readline().split()))
            rawcontents = f.readlines()

            for line in rawcontents:
```

```
            self.__isSolid.append(list(map((lambda x:
             ↪  x=="#"), line[:-1])))

    self.__cLoc = self.__start



def characterLocation(self):
    return self.__cLoc

def moveRight(self):
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row][col+1]:
            raise BoundaryCollisionError((row,col+1))
        else:
            self.__cLoc = (row,col+1)
    except IndexError:
        raise BoundaryCollisionError((row,col+1))

def moveDown(self):
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row+1][col]:
            raise BoundaryCollisionError((row+1,col))
        else:
            self.__cLoc = (row+1,col)
    except IndexError:
        raise BoundaryCollisionError((row+1,col))

def moveUp(self):
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row-1][col]:
            raise BoundaryCollisionError((row-1,col))
        else:
            self.__cLoc = (row-1,col)
    except IndexError:
        raise BoundaryCollisionError((row-1,col))
```

```
def moveLeft(self):
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row][col-1]:
            raise BoundaryCollisionError((row,col-1))
        else:
            self.__cLoc = (row,col-1)
    except IndexError:
        raise BoundaryCollisionError((row,col-1))

def canMoveUp(self) -> bool:
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row-1][col]:
            return False
        else:
            return True
    except IndexError:
        return False

def canMoveDown(self) -> bool:
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row+1][col]:
            return False
        else:
            return True
    except IndexError:
        return False

def canMoveRight(self) -> bool:
    (row,col) = self.__cLoc
    try:
        if self.__isSolid[row][col+1]:
            return False
        else:
            return True
    except IndexError:
```

```
            return False

    def canMoveLeft(self) -> bool:
        (row,col) = self.__cLoc
        try:
            if self.__isSolid[row][col-1]:
                return False
            else:
                return True
        except IndexError:
            return False


    def __str__(self):
        mapString = ""
        for row in range(len(self.__isSolid)):
            for col in range(len(self.__isSolid[0])):
                if ((row,col) == self.__start or (row,col) ==
                 ↪  self.__end) and (row,col) != self.__cLoc:
                    mapString += "o"
                elif (row,col) == self.__cLoc:
                    mapString += "+"
                elif self.__isSolid[row][col]:
                    mapString += "#"
                else:
                    mapString += "."
            mapString += "\n"
        return mapString

    def teleportCharacter(self,newLocation):
        (row,col) = newLocation
        if self.__isSolid[row][col]:
            raise BoundaryCollisionError((row,col))
        else:
            self.__cLoc = newLocation



#to test if board works, uncomment the following
```

```
#b = Board("boardfile.in")
#print(b)
#b.moveRight()
#print(b)
```

Controller

```
from board import Board
from commands import
 ↪  DashUpCommand,DashLeftCommand,DashDownCommand,DashRightCommand

class Controller:
    def __init__(self,board:Board):
        self.__board = board
        self.__commandHistory = []

    def pressUp(self):
        command = DashUpCommand(self.__board)
        self.__commandHistory.append(command)
        command.execute()

    def pressDown(self):
        command = DashDownCommand(self.__board)
        self.__commandHistory.append(command)
        command.execute()

    def pressLeft(self):
        command = DashLeftCommand(self.__board)
        self.__commandHistory.append(command)
        command.execute()

    def pressRight(self):
        command = DashRightCommand(self.__board)
        self.__commandHistory.append(command)
        command.execute()

    def undo(self):
        undoneCommand = self.__commandHistory.pop()
        undoneCommand.undo()
```

```
b = Board("boardFile.in")
c = Controller(b)
print(b)
c.pressRight()
print(b)
c.pressDown()
print(b)
```

What's missing right now is controller support. This is how a player controls the character on the maze:

- `dpad_up()`, `dpad_down()`, `dpad_left()`, `dpad_right()` - The character dashes through the maze in the specified direction until it hits a boundary. (these methods are equivalent to `pressUp()`, `pressDown()`, and etc on the `Controller`)
- `a_button()` - The character undoes the previous action it did. (this is equivalent to `undo()` on `Controller`)

To implement controller support you need to create a `Command` abstraction which is realized by all controller commands. The `Controller` (which represents the controller) is the invoker for the commands. Since commands are undoable, this controller needs to keep a command history, represented as a list. Every time a controller button is pressed, it creates the appropriate `Command`, executes it and appends it to the command history. Every time the `a_button()` is pressed to undo, the controller pops the last command from the command history and undoes it.

> `backupLocation : (Int,Int)` is an attribute but the diagram shows it as a method. It's supposed to be a private attribute of type `(Int,Int)` tuple but planUML (the uml renderer im using) reads it as a tuple because of the tuple parentheses

Complete the system using the command pattern

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

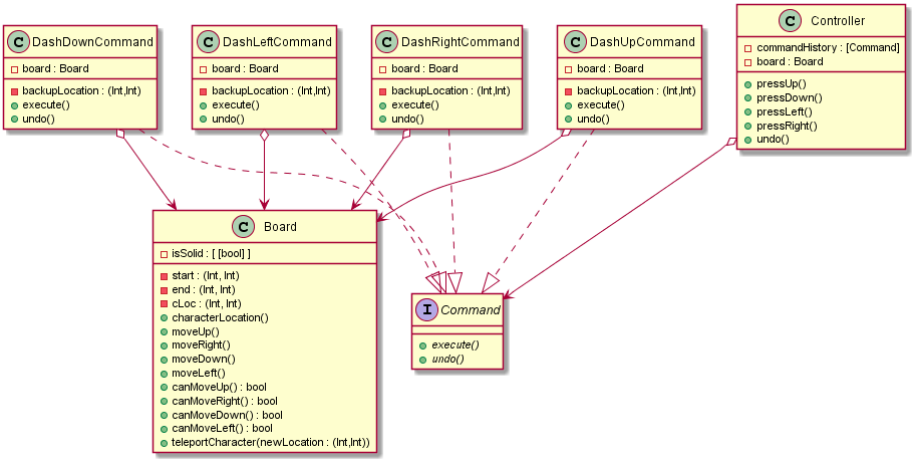

Figure 44: command example

Lab Exercise 13 (Weather Notifier)

Task

You are creating a push notification system that works for multiple platforms. You want to distribute information about the current weather and news headlines. This system will be potentially used on many platforms so you have to think about the maintainability issues for adding new platform support.

```
class Headline:
    def __init__(self, headline:str, details:str, source:str):
        self.__headline = headline
        self.__details = details
        self.__source = source

    def __str__(self) -> str:
        return "%s(%s)\n%s" % (self.__headline, self.__source,
        ↪  self.__details)

class Weather:
    def __init__(self, temp:float, humidity:float, outlook:str):
        self.__temp = temp
        self.__humidity = humidity
        self.__outlook = outlook

    def __str__(self) -> str:
```

```
        return "%s: %.1fC %.1f" % (self.__outlook, self.__temp,
         ↪  self.__humidity)


h = Headline("Dalai Lama Triumphantly Names Successor After
 ↪  Discovering Woman With 'The Purpose Of Our Lives Is To Be
 ↪  Happy' Twitter Bio","Details","The Onion")
w = Weather(25.0,0.7,"Cloudy")
print(h)
print(w)
```

To implement this, you have to apply the observer pattern. Your subject would be `Weather` data and `Headline` data (which are their own classes). These subjects should be encapsulated into a single publisher class (which will be called `PushNotifier`).

Any platform, that is interested in the changes to the subject should realize a `Subscriber` abstraction (Observer), which contains the abstract method update().

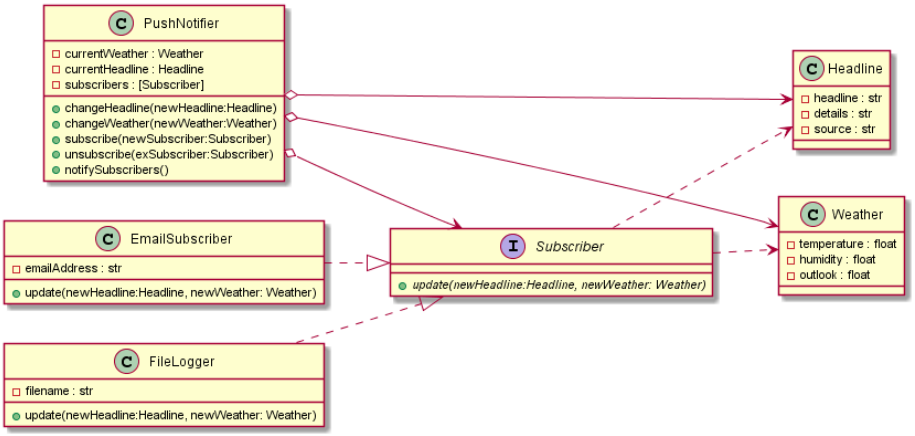

Figure 45: observer example

Complete the system using the observer pattern.

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2021

Lab Exercise 14 (Brute Force Search)

Task

If you write brute force algorithms as search problems, they will have a common recipe. This is the reason why it is called the exhaustive search algorithm. It will traverse all of the elements in the search space, trying to check the validity of each element, until it completes the solution

Equality Search

Search for integers equal to the target

```
#searchSpace = [2,3,1,0,6,2,4]
#target = 2

i = 0
solutions = []
candidate = searchSpace[0]
while(i<len(searchSpace)):
    if candidate == target:
        solutions.append(candidate)
    candidate = searchSpace[++i]

#solution = [2,2]
```

Divisibility Search

Search for integers divisible by the target

```
#searchSpace = [2,3,1,0,6,2,4]
#target = 2

i = 0
solutions = []
candidate = searchSpace[0]
while(i<len(searchSpace)):
    if candidate % target == 0:
        solution.append(candidate)
    candidate = searchSpace[++i]

#solution = [2,0,6,2,4]
```

Minimum Search

No target, searches for the smallest integer

```
#searchSpace = [2,3,1,0,6,2,4]
#target = None

i = 1
solutions = [searchSpace[0]]
candidate = searchSpace[1]
while(i<len(searchSpace)):
    if candidate <= solutions[0]
        solutions[0] = candidate
    candidate = searchSpace[++i]

#solution = [0]
```

Common Recipe

```
i = 0
solutions = []
candidate = first()
while(isStillSearching()):
    if valid(candidate):
        updateSolution(candidate)
    candidate = next()
```

Because of this we can write a general brute force template method that would return the solution to brute force problems. To do this you create a superclass `SearchAlgorithm()` that contains the template method for brute force algorithms. If you want to customize this algorithm for special problems, all you have to do is to inherit from `SearchAlgorithm` and override only the necessary steps.

```
class SearchAlgorithm(ABC):
    def __init__(self, target:int, searchSpace:[int]):
        self._searchSpace = searchSpace
        self._currentIndex = 0
        self._solutions = []
        self._target = target

    def bruteForceSolution(self):
```

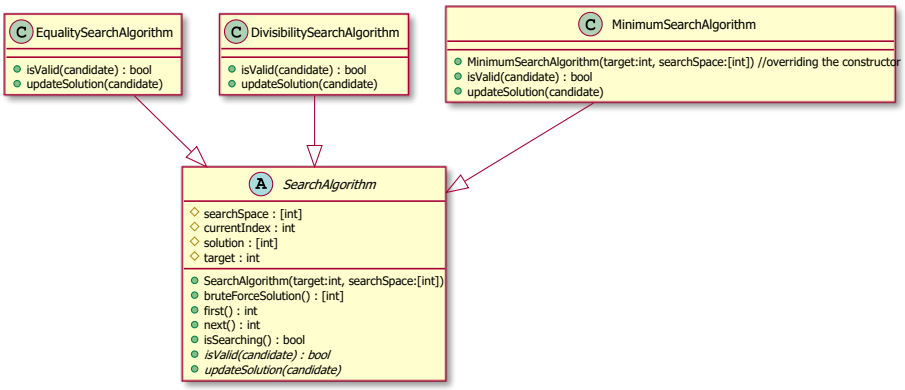

Figure 46: template example

```
        candidate = self.first()
        while(self.isSearching()):
            if self.isValid(candidate):
                self.updateSolution(candidate)
            candidate = self.next()
        return self._solutions

    def first(self) -> int:
        return self._searchSpace[0]

    def next(self) -> int:
        self._currentIndex += 1
        if self.isSearching():
            return self._searchSpace[self._currentIndex]

    def isSearching(self) -> bool:
        return self._currentIndex < len(self._searchSpace)

    @abstractmethod
    def isValid(self, candidate) -> bool:
        pass

    @abstractmethod
    def updateSolution(self, candidate):
        pass
```

is `isValid()` and `updateSolution(candidate)` is different for each algorithm so it doesn't have a default implementation. It would be best to make these steps abstract.

Complete this system using the template pattern

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

Lab Exercise 15 (Iterator Pattern)

Task

For non built-in collections, you can create an iterator that does the traversal for you. On the bare minimum these iterators will realize some `Iterator` abstraction that contains the methods, `next()`, and `hasNext()`. From these methods alone you can easily perform complete traversals without knowing the exact type of the collection:

```
i = collection.newIterator()
while i.hasNext():
    print(i.next())
```

```
from abc import ABC,abstractmethod

class Iterator(ABC):
    @abstractmethod
    def next(self):
        pass

    @abstractmethod
    def hasNext(self):
        pass

class Collection(ABC):
    @abstractmethod
    def newIterator(self):
        pass
```

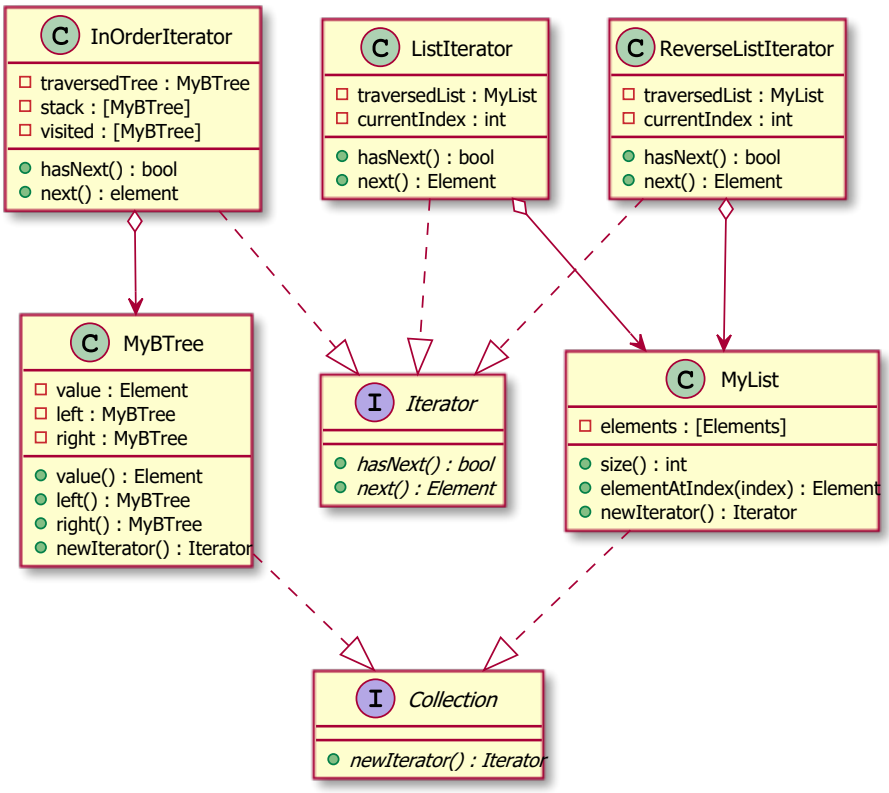

Figure 47: iterator

```
class MyList(Collection):
    def __init__(self, elements):
        self.__elements = elements

    def size(self):
        return len(self.__elements)

    def elementAtIndex(self,index):
        return self.__elements[index]

    def newIterator(self):
        return ListIterator(self)

class ListIterator(Iterator):
    def __init__(self, list):
        self.__traversedList = list
        self.__currentIndex = 0

    def next(self):
        self.__currentIndex += 1
        return
         ↪  self.__traversedList.elementAtIndex(self.__currentIndex-1)

    def hasNext(self):
        return self.__currentIndex < self.__traversedList.size()

#OPTIONAL
"""
class MyBTree(Collection):
    def __init__(self,value:int,left:'MyBTree' =
 ↪  None,right:'MyBTree' = None):
        self.__value = value
        self.__left = left
        self.__right = right

    def left(self):
        return self.__left
```

```
    def right(self):
        return self.__right

    def value(self):
        return self.__value

    def newIterator(self):
        return InOrderIterator(self)
"""


c:Collection = MyList([1,2,3,4])
iter:Iterator = c.newIterator()

while(iter.hasNext()):
    print(iter.next())
```

The `hasNext()` method, returns a boolean value that indicates whether or not there are more elements to be traversed. The `next()` method, returns the next element in the traversal.

A collection can have more than one `Iterator`s, if it makes sense for the collection to be travsersed in more than one way. Despite this possibility, a collection must have a default iterator which will be the type of the new instance returned in the factory method, `newIterator()`

> Note: `MyBTree` and its iterator is optional

Complete the system using the iterator pattern

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

Lab Exercise 16 (Formatted Sentence)

Task

A sentence can be defined as a list of words (words are strings). The string representation of a sentence is the concatenation of all of the words in the list,

separated by a space.

```
from abc import ABC,abstractmethod

class Sentence:
    def __init__(self,words:[str]):
        self.__words = words

    def __str__(self) -> str:
        sentenceString = ""
        for word in self.__words:
            sentenceString += word + " "
        return sentenceString[:-1]
```

Instances of sentences can be printed with formatting:

- **bordered** - Given the sentence, `["hey","there"]` it prints:

  ```
  -----------
  |hey there|
  -----------
  ```

- **fancy** - Given the sentence, `["hey","there"]` it prints:

  ```
  -+hey there+-
  ```

- **uppercase** - Given the sentence, `["hey","there"]` it prints:

  ```
  HEY THERE
  ```

The formatting of a sentence is decided during runtime. These formats should also allow for combinations with other formats:

- **bordered fancy** - Given the sentence, `["hey","there"]` it prints:

  ```
  ---------------
  |-+hey there+-|
  ---------------
  ```

- **fancy uppercase** - Given the sentence, `["hey","there"]` it prints:

  ```
  -+HEY THERE+-
  ```

To accomplish these features, you need to implement the decorator pattern. Each formatting will be a decorator for `Sentence` objects. These formats need to inherit from some abstract `FormattedSentence` class. This abstract class is specified to compose and inherit from sentence. The behavior that needs to be

decorated is the `__str__()` function since you need to change how sentence is printed for every format.

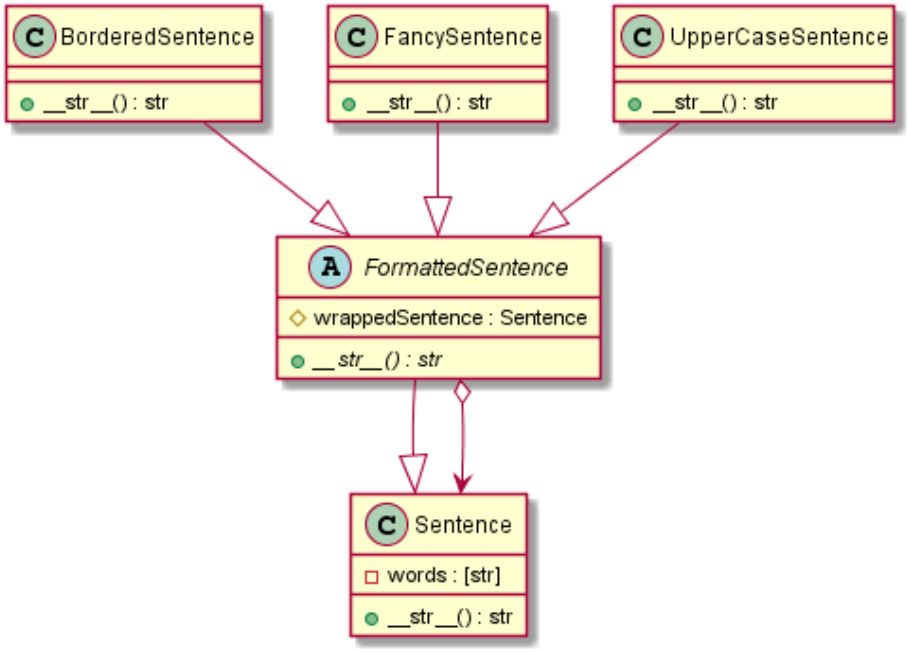

Figure 48: decorator example

Complete the system using the decorator pattern

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

Lab Exercise 17 (Printable Shipment)

Task

Looking back at our previous lab exercises, some of the example classes contain string representation but do not implement the `__str__()` function. An example of this is `Shipment` back from the factory method example. It does contain a string representation builder called `shipmentDetails()`, but printing a shipment is quite tedious since you have to print, `s.shipmentDetails()`. You can replace the name of `shipmentDetails()` to `__str__()` but this will potentially

affect other clients of shipment. You can add the `__str__()` function which does exactly the same but this may introduce unwanted code duplication.

```
from abc import ABC,abstractmethod
from datetime import date,timedelta

class Delivery(ABC):
    @abstractmethod
    def deliveryDetails(self) -> str:
        pass
    @abstractmethod
    def deliveryFee(self) -> float:
        pass
    @abstractmethod
    def estimatedDeliveryDate(self,processDate:date) -> float:
        pass
    @abstractmethod
    def changeDeliveryStatus(self,newStatus:str):
        pass


class Order:
    def __init__(self,productName:str, productPrice:float):
        self.__productName = productName
        self.__productPrice = productPrice
    def orderString(self) -> str:
        return "%s P%.2f" %
         ↪  (self.__productName,self.__productPrice)
    def price(self) -> float:
        return self.__productPrice
class StandardDelivery(Delivery):
    def __init__(self,location:str):
        self.__location = location
        self.__deliveryStatus = "Processing"
    def deliveryDetails(self) -> str:
        r = "STANDARD DELIVERY\nDELIVER TO:%s\nDELIVERY STATUS:
↪  %s\nDELIVERY FEE: P%.2f" %
↪  (self.__location,self.__deliveryStatus,self.deliveryFee())
        return r
    def deliveryFee(self) -> float:
```

```
        return 500
    def estimatedDeliveryDate(self,processDate:date) -> float:
        return processDate + timedelta(days = 7)
    def changeDeliveryStatus(self,newStatus:str):
        self.__deliveryStatus = newStatus

class Shipment:
    def __init__(self, orderList:[Order], processDate: date,
     ↪  location):
        self._orderList = orderList
        self._processDate = processDate
        self._delivery = self.delivery(location)

    def delivery(self,location:str) -> Delivery:
        return StandardDelivery(location)

    def totalPrice(self) -> str:
        t = 0.0
        for order in self._orderList:
            t+=order.price()
        return t

    def shipmentDetails(self) -> str:
        r = "ORDERS:" + str(self._processDate) + "\n"
        for order in self._orderList:
            r += order.orderString() + "\n"
        r += "\n"
        r += "TOTAL PRICE OF ORDERS: P"  + str(self.totalPrice())
↪   + "\n"
        r += self._delivery.deliveryDetails() + "\n\n"
        r += "PRICE WITH DELIVERY FEE : P" +
↪   str(self.totalPrice()+self._delivery.deliveryFee()) + "\n"
        r += "ESTIMATED DELIVERY DATE: " +
↪   str(self._delivery.estimatedDeliveryDate(self._processDate))
        return r


class ExpressDelivery(Delivery):
    def __init__(self,location:str):
```

```
        self.__location = location
        self.__deliveryStatus = "Processing"
    def deliveryDetails(self) -> str:
        r = "EXPRESS DELIVERY\nDELIVER TO:%s\nDELIVERY STATUS:
↪  %s\nDELIVERY FEE: P%.2f" %
↪  (self.__location,self.__deliveryStatus,self.deliveryFee())
        return r
    def deliveryFee(self) -> float:
        return 1000
    def estimatedDeliveryDate(self,processDate:date) -> float:
        return processDate + timedelta(days = 2)
    def changeDeliveryStatus(self,newStatus):
        self.__deliveryStatus = newStatus

class ExpressShipment(Shipment):
    def delivery(self,location) -> Delivery:
        return ExpressDelivery(location)
```

The best solution for this problem is to create an adapter for shipment called `PrintableShipment`. This adapter will realize some `Printable` abstraction, which only contains the abstract method `__str__()`.

Complete the system using the adapter pattern

Assessment Criteria

- Completeness of the pattern - 40
- Elegance of method and attribute naming - 10

Deadline January 11, 2022

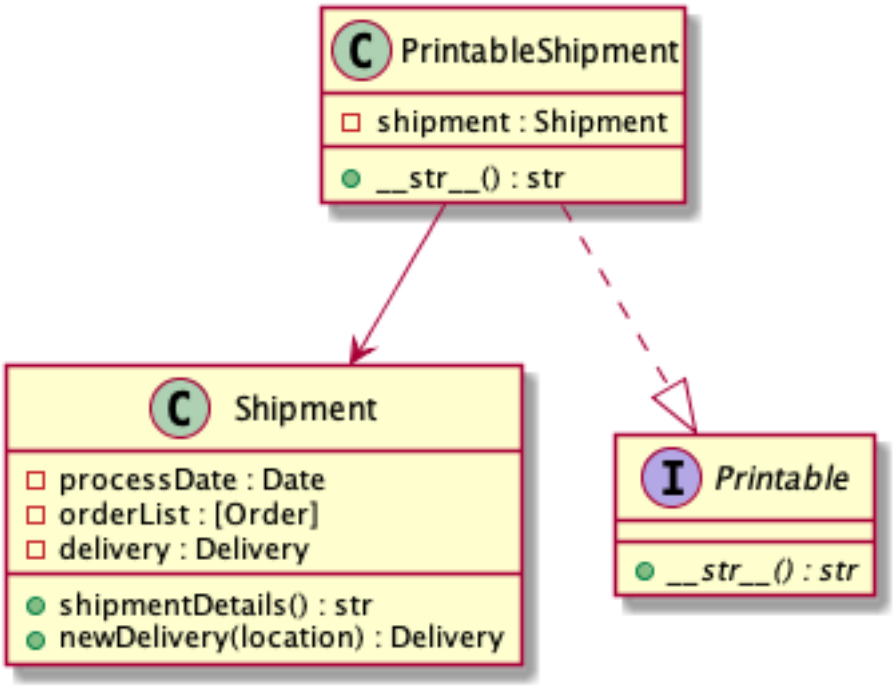

Figure 49: adapter example