Class Diagram

Class diagram is provided with attachment class_diagram.png

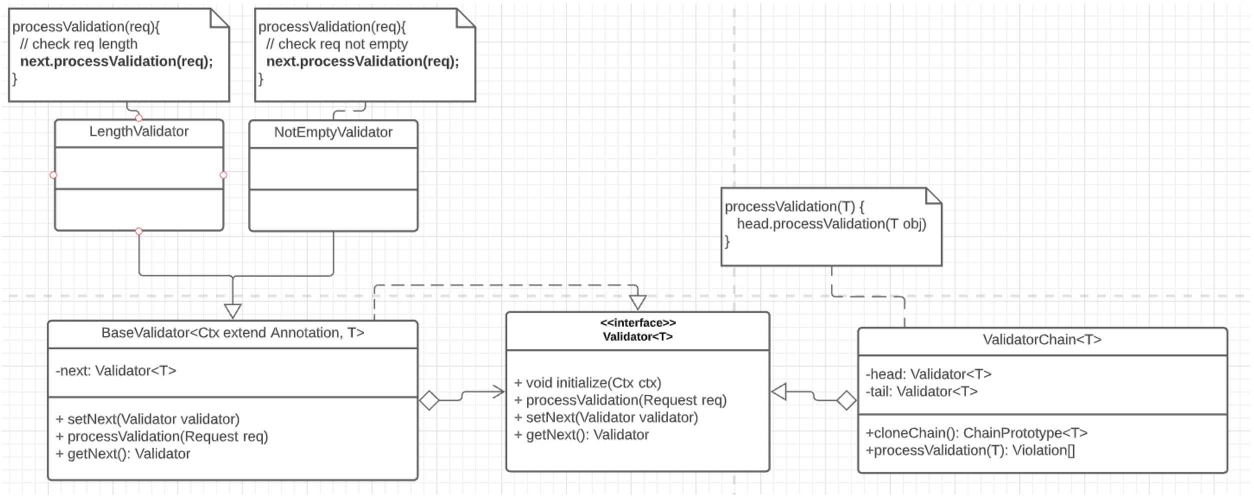
Used patterns

1. **Chain of responsibility**

The reason why we use this pattern is because constraints is applied to the item can be apply as a chain:

- For example, @NotNull @NotEmpty @Length(max = 10) String a; (a is not null, not empty and has length <= 10),
- We could apply the constraint as a chain to handle validation so that NotNull.validate(a), NotEmpty.validate(a), Length.validate(a) consecutively without knowing each other

In our design, we mainly apply it to our validator:



```
processValidation(req){
  // check req length
  next.processValidation(req);
}
```

```
processValidation(req){
  // check req not empty
  next.processValidation(req);
}
```

```
processValidation(T) {
   head.processValidation(T obj)
}
```

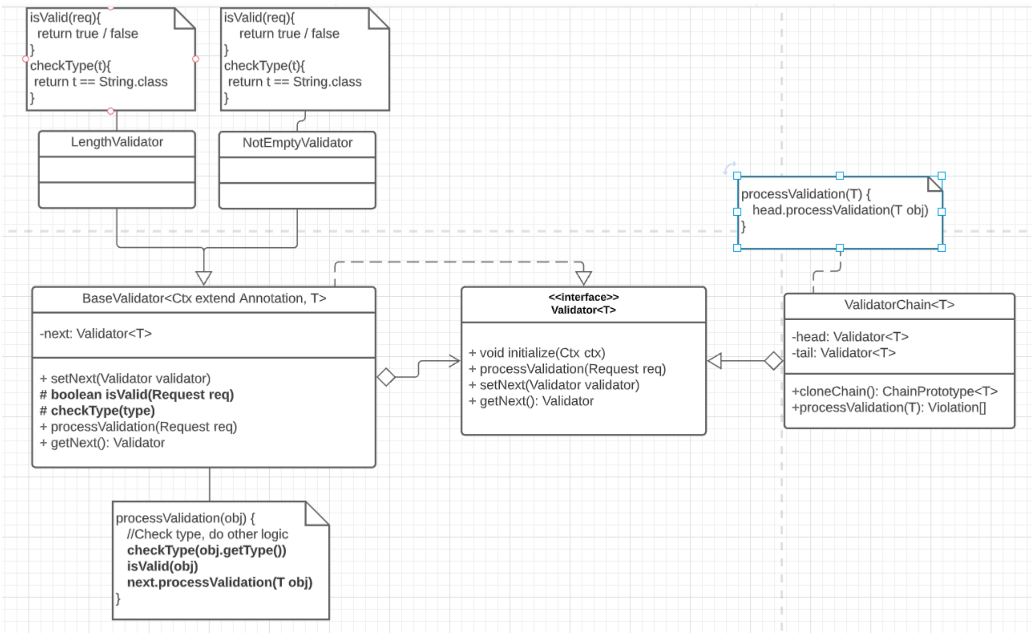
LengthValidator
NotEmptyValidator
BaseValidator<Ctx extend Annotation, T>
-next: Validator<T>
+ setNext(Validator validator)
+ processValidation(Request req)
+ getNext(): Validator
<<interface>>
Validator<T>
+ void initialize(Ctx ctx)
+ processValidation(Request req)
+ setNext(Validator validator)
+ getNext(): Validator
ValidatorChain<T>
-head: Validator<T>
-tail: Validator<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]

- ValidatorChain will hold the head, and when client want to validate, they could call *validatorChain.processValidation(obj)* and it will call the chain consequentially to pass the request to every handler in the chain
- Without coupling between *Concrete Validators* and *ValidatorChain*, we could easily introduce new type of validators and customize it without breaking our current code.
- Take a look at *Validator<T>* interface design, we could see that we can apply **template method** here to abstract the handler fro m calling next

2. **Template method**

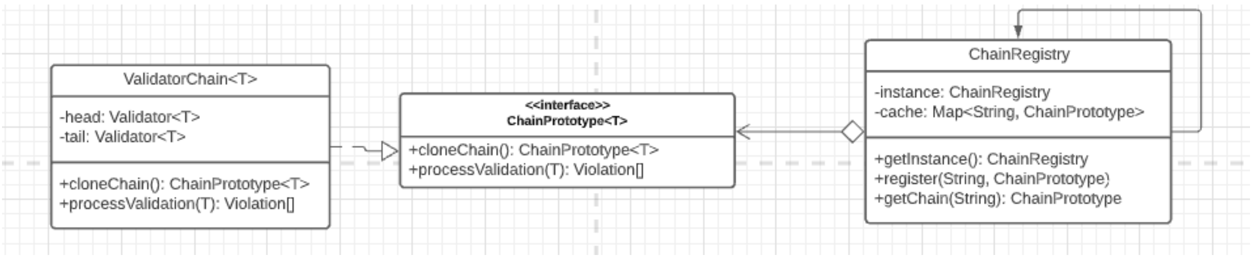
In our current design, we could see that the redundancy of calling "next.processValidation(req)" in every single concrete class – which is a bad idea since our project allows client to create their own validators (the logic will break if they 'forget' to call next.processValidation(req) in their validator implement).

To abstract it from the concrete class, we introduce an abstract method (Template method) for other concrete validators to override. In our case, we introduce *isValid()* and *checkType()* as a validate methods for specific validators:



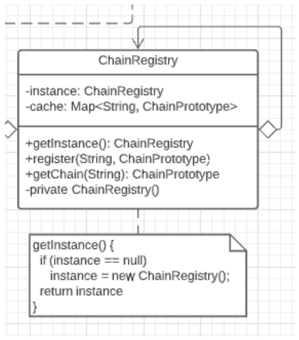
3. **Prototype:**

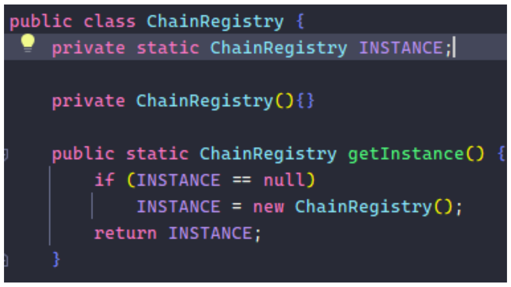
In our design, we aim to provide a specific type of chain for each Object in class. Since every instance of a class may have the same chain (We usually register constraint as annotation in our class). And the process of resolve validator chain for each class is complex, so this is the ideal case for Prototype, which prefer cloning to instantiating. And with the help of Prototype Registry, we could register the chain for other usage in our app without redundant initialization.


ValidatorChain<T>
-head: Validator<T>
-tail: Validator<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
<<interface>>
ChainPrototype<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
ChainRegistry
-instance: ChainRegistry
-cache: Map<String, ChainPrototype>
+getInstance(): ChainRegistry
+register(String, ChainPrototype)
+getChain(String): ChainPrototype

4. **Singleton**

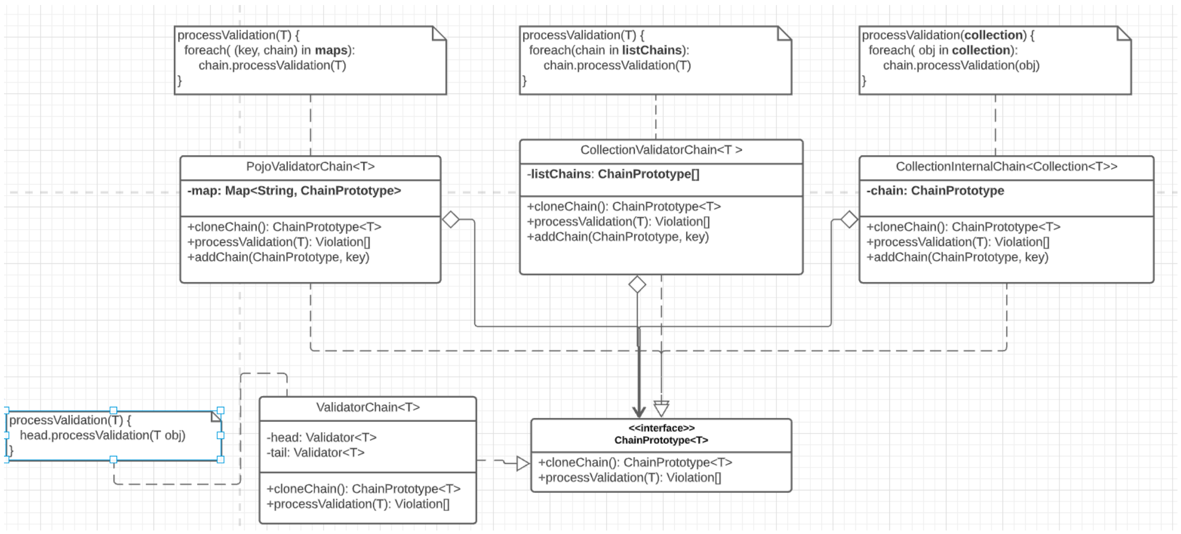
Of course, if we want to make our Registry unique, Singleton is a must!


ChainRegistry
-instance: ChainRegistry
-cache: Map<String, ChainPrototype>
+getInstance(): ChainRegistry
+register(String, ChainPrototype)
+getChain(String): ChainPrototype
-private ChainRegistry()
getInstance() {
if (instance == null)
instance = new ChainRegistry();
return instance
}



```
public class ChainRegistry {
    private static ChainRegistry INSTANCE;

    private ChainRegistry(){}

    public static ChainRegistry getInstance() {
        if (INSTANCE == null)
            INSTANCE = new ChainRegistry();
        return INSTANCE;
    }
```

5. **Composite:**


processValidation(T) {
foreach((key, chain) in maps):
chain.processValidation(T)
}
processValidation(T) {
foreach(chain in listChains):
chain.processValidation(T)
}
processValidation(collection) {
foreach(obj in collection):
chain.processValidation(obj)
}
PojoValidatorChain<T>
-map: Map<String, ChainPrototype>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
+addChain(ChainPrototype, key)
CollectionValidatorChain<T >
-listChains: ChainPrototype[]
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
+addChain(ChainPrototype, key)
CollectionInternalChain<Collection<T>>
-chain: ChainPrototype
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
+addChain(ChainPrototype, key)
processValidation(T) {
head.processValidation(T obj)
}
ValidatorChain<T>
-head: Validator<T>
-tail: Validator<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
<<interface>>
ChainPrototype<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]

The chain is good to go if we want to validate a simple variables. (i.e: ValidatorChain<String>)

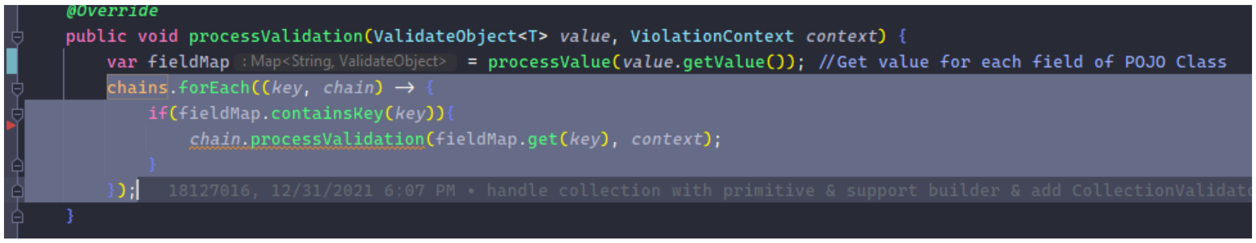
But that is is not the case for complex or nested object:

- i.e: *Student {String name; int age;}*.
  Student {String name; int age; Classroom classroom}.
  Student {String name; int age; Collection<Classroom> classroom}.
  Student {String name; int age; Collection<@Length(max=4) String> stamps}.

Therefore, in order to handle these cases, we create composites chains that contains complex data structure to handle such situations:

- ValidatorChain: Leaf node in the composite tree, handle the '*processValidation*'
- PojoValidationChain (Plain-old-java-object validation chain): A chain that suitable for validating an object:
  - i.e: *Student {@Length(max=4) String name; @Positive @Max(100) int age; Classroom classroom }.Classroom{@Positive int students}*
  - The chain will store the validators by each field that object has:
    - "name" => ValidatorChain(LengthValidator)

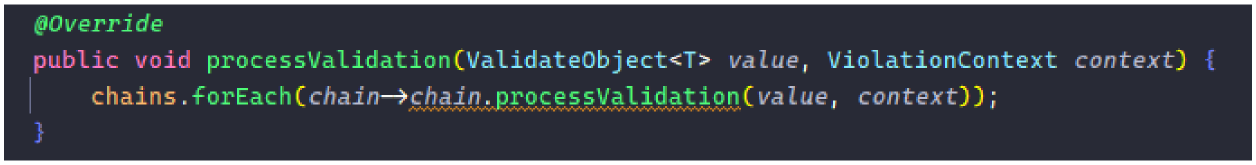
        - “age” => ValidatorChain(PositiveValidator -> MaxValidator)
        - “classroom” => PojoValidationChain:
            - “students” => ValidatorChain(PositiveValidator)
    - processValidation will pass the validating object to these validators to handle!



```
@Override
public void processValidation(ValidateObject<T> value, ViolationContext context) {
    var fieldMap : Map<String, ValidateObject> = processValue(value.getValue()); //Get value for each field of POJO Class
    chains.forEach((key, chain) → {
        if(fieldMap.containsKey(key)){
            chain.processValidation(fieldMap.get(key), context);
        }
    });|  18127016, 12/31/2021 6:07 PM • handle collection with primitive & support builder & add CollectionValidat
}
```

Figure: PojoValidationChain `processValidation()` implementation in source code

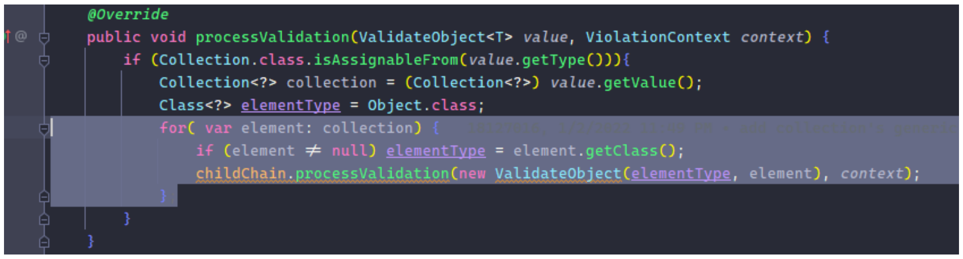
- CollectionValidationChain: A chain that contains multiple chain. This chain is usually used by PojoValidationChain to store multiple chain for 1 field.
    - i.e: *`Student { @NotNull Classroom classroom }`*.
    - This will be stored as:
        - “classroom” => CollectionValidationChain contains
            - ValidatorChain(NotNullValidator)
            - PojoValidationChain:
                - “students” => ValidatorChain(PositiveValidator)



```
@Override
public void processValidation(ValidateObject<T> value, ViolationContext context) {
    chains.forEach(chain→chain.processValidation(value, context));
}
```

Figure: CollectionValidationChain `processValidation()` implementation in source code

- CollectionInternalValidationChain: A chain for data which has Collection type. Since Collection<T> is handled totally different from plain T, we introduce this composite class to handle this specific case:
    - I.e: *`{String name; @NotEmpty Collection<@NotNull Classroom> classrooms}`*
    - This could be resolved as:
        - “name” => ValidatorChain(LengthValidator)
        - “classrooms” => CollectionValidationChain contains
            - PojoValidationChain:
                - “students” => ValidatorChain(PositiveValidator)
            - CollectionInternalValidationChain(chain = NotNullValidator): *for every* `object` *in classrooms, run* `chain.processValidation(object)`

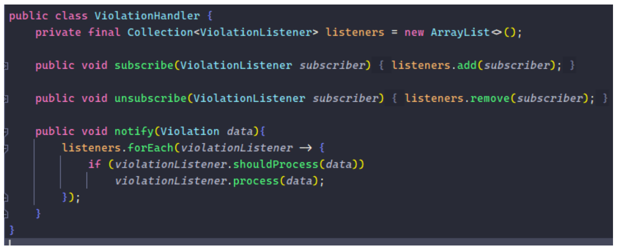


```java
@Override
public void processValidation(ValidateObject<T> value, ViolationContext context) {
    if (Collection.class.isAssignableFrom(value.getType())){
        Collection<?> collection = (Collection<?>) value.getValue();
        Class<?> elementType = Object.class;
        for( var element: collection) {
            if (element ≠ null) elementType = element.getClass();
            childChain.processValidation(new ValidateObject(elementType, element), context);
        }
    }
}
```

CollectionInternalValidationChain processValidation() implementation in source code

6. **Observer pattern:**

In order to catch the event where the violation occurs, we introduce *ViolationHandler()* that implement Observer pattern to listen to violation events:



```java
public class ViolationHandler {
    private final Collection<ViolationListener> listeners = new ArrayList<>();

    public void subscribe(ViolationListener subscriber) { listeners.add(subscriber); }

    public void unsubscribe(ViolationListener subscriber) { listeners.remove(subscriber); }

    public void notify(Violation data){
        listeners.forEach(violationListener → {
            if (violationListener.shouldProcess(data))
                violationListener.process(data);
        });
    }
}
```

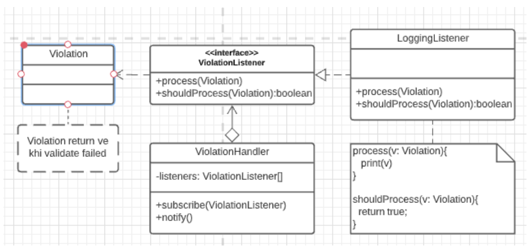

Violation
Violation return ve khi validate failed
<<interface>> ViolationListener
+process(Violation)
+shouldProcess(Violation):boolean
LoggingListener
+process(Violation)
+shouldProcess(Violation):boolean
ViolationHandler
-listeners: ViolationListener[]
+subscribe(ViolationListener)
+notify()
process(v: Violation){
print(v)
}
shouldProcess(v: Violation){
return true;
}

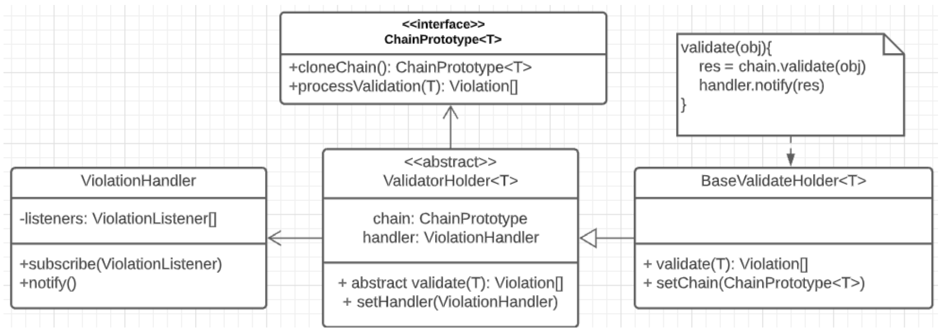
Figure: ViolationHandler implementation in source code (left) – Class diagram (right)

The idea here is to subscribe to a specific handler, whenever the violation occurs, the VioliationHandler will notify all the subscriber (all ViolationListener concrete classes that call handler.subscribe()) and invoke *process()* function. Note that the event can be filtered by shouldProcess() function.

The main reason why we use the observer is to reduce the coupling between subscribers and notifiers by abstracting subscribers. It means that clients can customize and introduce their own Listeners for their specific usage without breaking the existing code.

So how does it fit into our current system? In order to combine the handler to trigger when the chain validate, we introduce ValidatorHolder – a class that has **Bridges** to abstract chain implementation

7. **Bridge pattern**

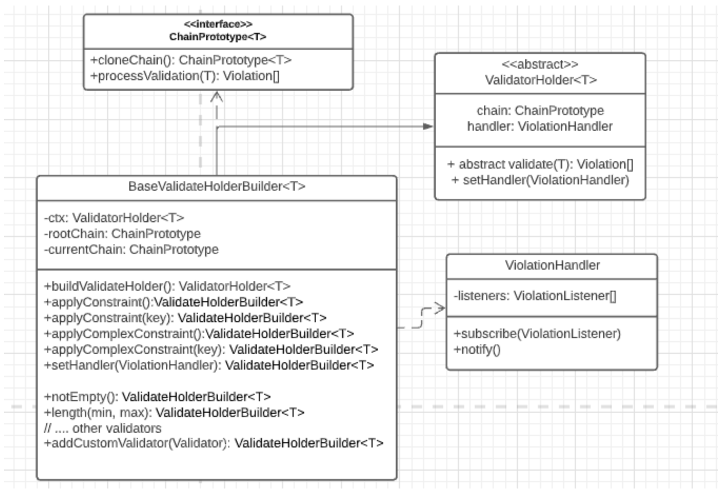

<<interface>> ChainPrototype<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
validate(obj){
res = chain.validate(obj)
handler.notify(res)
}
ViolationHandler
-listeners: ViolationListener[]
+subscribe(ViolationListener)
+notify()
<<abstract>> ValidatorHolder<T>
chain: ChainPrototype
handler: ViolationHandler
+ abstract validate(T): Violation[]
+ setHandler(ViolationHandler)
BaseValidateHolder<T>
+ validate(T): Violation[]
+ setChain(ChainPrototype<T>)

We introduce a class from which the implementation of Validation chains (which has a set of composites chains) is separated from the interface. And adding some additional logics besides validation logics such as notifying subscriber.

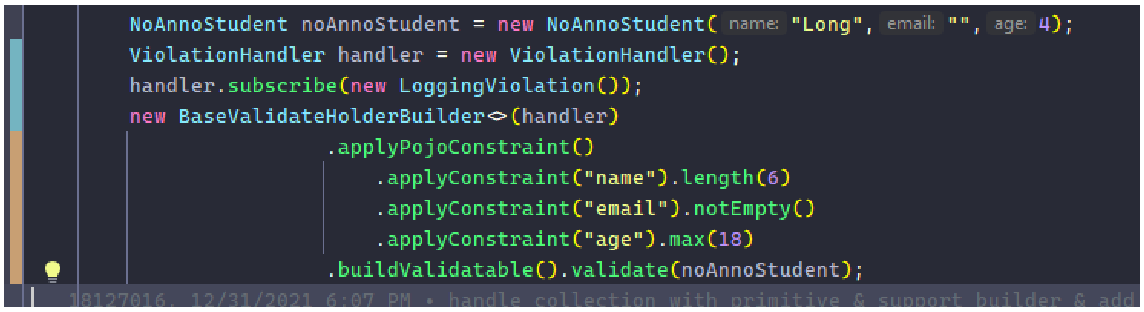
Next, what we need is a way to create our chains from client code. User can create chains either by using **Builder** or by resolving annotation by IoC Provider

8. **Composite Builder**

To create such complex Composite-ChainPrototype, we create a builder to divide the constructing process into smaller steps



<<interface>>
ChainPrototype<T>
+cloneChain(): ChainPrototype<T>
+processValidation(T): Violation[]
<<abstract>>
ValidatorHolder<T>
chain: ChainPrototype
handler: ViolationHandler
+ abstract validate(T): Violation[]
+ setHandler(ViolationHandler)
BaseValidateHolderBuilder<T>
-ctx: ValidatorHolder<T>
-rootChain: ChainPrototype
-currentChain: ChainPrototype
+buildValidateHolder(): ValidatorHolder<T>
+applyConstraint():ValidateHolderBuilder<T>
+applyConstraint(key): ValidateHolderBuilder<T>
+applyComplexConstraint():ValidateHolderBuilder<T>
+applyComplexConstraint(key): ValidateHolderBuilder<T>
+setHandler(ViolationHandler): ValidateHolderBuilder<T>
+notEmpty(): ValidateHolderBuilder<T>
+length(min, max): ValidateHolderBuilder<T>
// .... other validators
+addCustomValidator(Validator): ValidateHolderBuilder<T>
ViolationHandler
-listeners: ViolationListener[]
+subscribe(ViolationListener)
+notify()



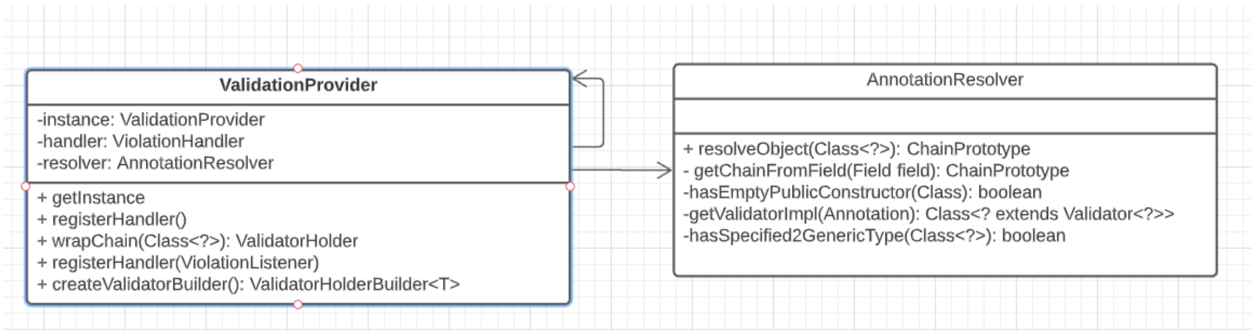
```
NoAnnoStudent noAnnoStudent = new NoAnnoStudent( name: "Long", email: "", age: 4);
ViolationHandler handler = new ViolationHandler();
handler.subscribe(new LoggingViolation());
new BaseValidateHolderBuilder<>(handler)
                .applyPojoConstraint()
                    .applyConstraint("name").length(6)
                    .applyConstraint("email").notEmpty()
                    .applyConstraint("age").max(18)
                .buildValidatable().validate(noAnnoStudent);
18127016, 12/31/2021 6:07 PM • handle collection with primitive & support builder & add
```

Figure: How the builder is used to build validators for an POJO object

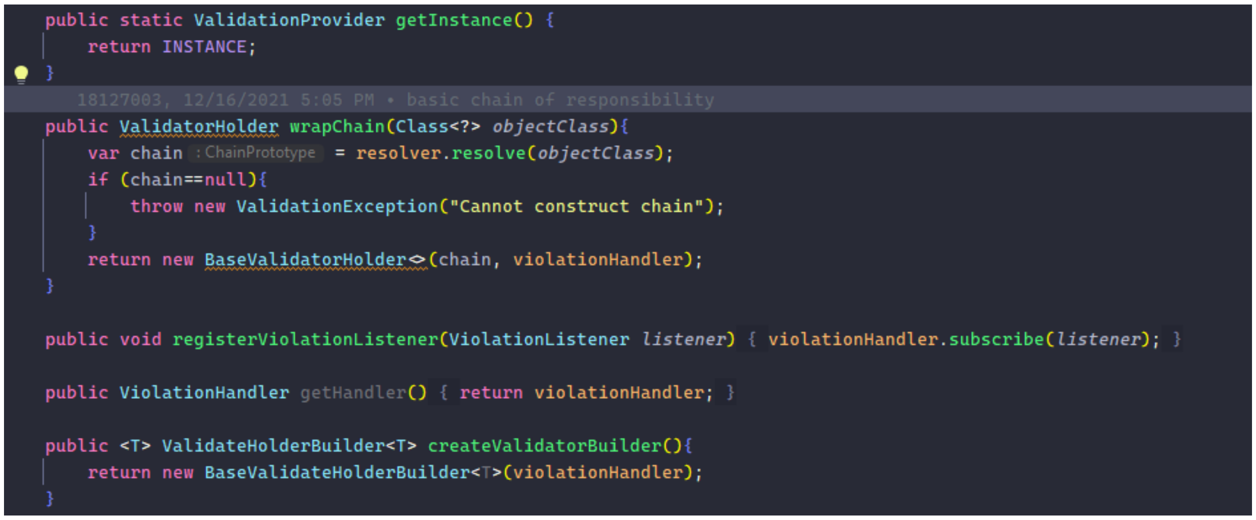
Client does not have to know the complexity of the algorithm; he could construct the validator chain steps by steps. This is the main reason why we decided to use this pattern, since it provides us an easy way to create such complex object.

Class components

1. **Validation Provider**


ValidationProvider
-instance: ValidationProvider
-handler: ViolationHandler
-resolver: AnnotationResolver
+ getInstance
+ registerHandler()
+ wrapChain(Class<?>): ValidatorHolder
+ registerHandler(ViolationListener)
+ createValidatorBuilder(): ValidatorHolderBuilder<T>
AnnotationResolver
+ resolveObject(Class<?>): ChainPrototype
- getChainFromField(Field field): ChainPrototype
-hasEmptyPublicConstructor(Class): boolean
-getValidatorImpl(Annotation): Class<? extends Validator<?>>
-hasSpecified2GenericType(Class<?>): boolean

Provider is a singleton that would handle complex logics and provide a container for some component in our app to work globally. In this case, the handler which we want to config globally should be registered here. We also provide a Factory method to create a builder with the global context data



```
public static ValidationProvider getInstance() {
    return INSTANCE;
}
  18127003, 12/16/2021 5:05 PM • basic chain of responsibility
public ValidatorHolder wrapChain(Class<?> objectClass){
    var chain :ChainPrototype = resolver.resolve(objectClass);
    if (chain==null){
        throw new ValidationException("Cannot construct chain");
    }
    return new BaseValidatorHolder<>(chain, violationHandler);
}

public void registerViolationListener(ViolationListener listener) { violationHandler.subscribe(listener); }

public ViolationHandler getHandler() { return violationHandler; }

public <T> ValidateHolderBuilder<T> createValidatorBuilder(){
    return new BaseValidateHolderBuilder<T>(violationHandler);
}
```

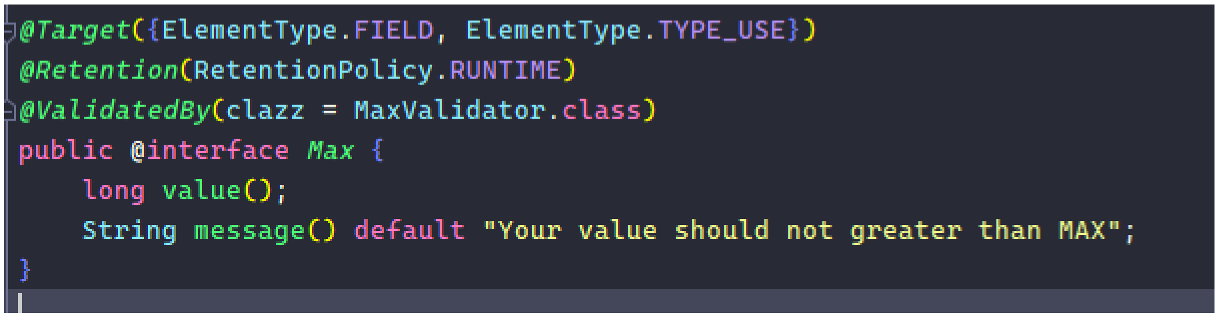
Figure: ValidationProvider implementation in code

Provider also provide an important feature for our framework: resolve annotation, which is implemented via a helper class – the Annotation Resolver. The section below will discuss more about this process

2. Annotation & annotation resolver

Annotation here is a piece of metadata that is used to label our declared variables such as class attributes, class functions, class, etc. It allows developers to label some pieces of code by some constant values in build time and retrieve them easily in runtime.

In our framework, we use Annotation to label class attributes with constraints for the validation. Here is an example for our constraint annotation:



```java
@Target({ElementType.FIELD, ElementType.TYPE_USE})
@Retention(RetentionPolicy.RUNTIME)
@ValidatedBy(clazz = MaxValidator.class)
public @interface Max {
    long value();
    String message() default "Your value should not greater than MAX";
}
```

Which includes 3 important pieces of information

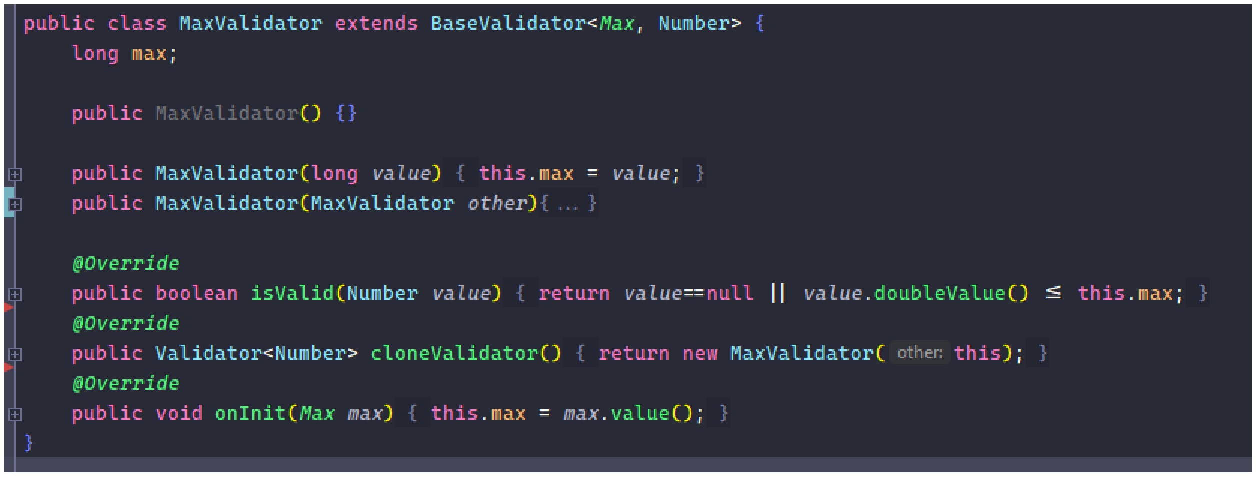
- The annotation targets only FIELDs and for TYPE_USE checking purpose only
- message() function with default value serve as an message for this constraint violation
- @ValidatedBy(class = MaxValidator.class): this should be the most important item in order to make this annotation work in our framework. This server as a mapping from this annotation interface to a **validator** implementation.

The annotation resolving process can be simply explain in 3 steps:

- Looping through all fields in an object and read all of its annotations
- Front these annotations, fetching @ValidatedBy mapping, and create a **Validator** for that annotation
- Generate a chain of validators and return the validator holder to validate the object.

3. **Validator**
   Validator interface provides a signature for validating a value via isValid method. Validator follow a generic type which mean a Validator should only validate 1 type of value.
   A class follow this interface is meant to handle validation of a value/ object.
   Validators are configured and reusable in many cases so Validator support clone method, which is Prototype pattern core method. It come with the cloneValidator method, delegate the clone to concrete classes.
   Also, Validator in a component in the validation chain so it provides signature for processing in the chain (Chain of Responsibility) which is processValidate. setNext and getNext are used for chain construction.
   a. **BaseValidator**

   Concrete validator inherits from BaseValidator<A extends Annotation, T> validator that provides abstract logic to the components. The A, T generic type provides an appropriate data type for our validator. If the type check is not passed when the processValidation() executes, the program will throw InvalidTypeException,



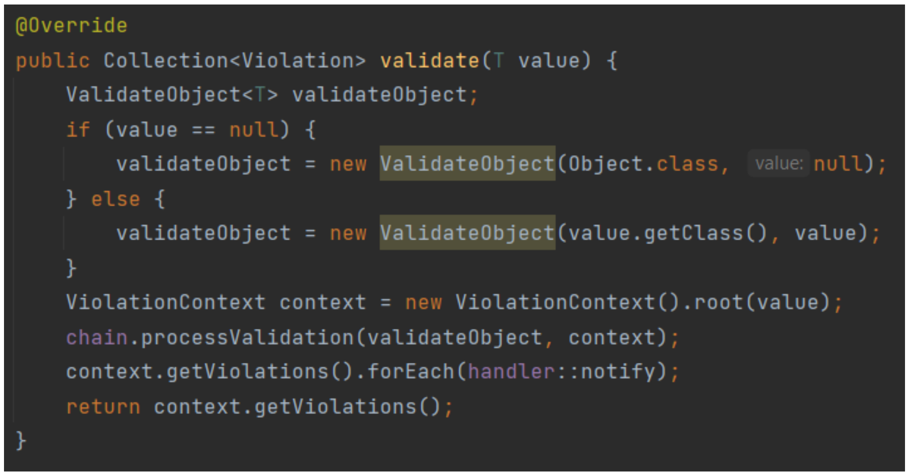
```java
public class MaxValidator extends BaseValidator<Max, Number> {
    long max;

    public MaxValidator() {}

    public MaxValidator(long value) { this.max = value; }
    public MaxValidator(MaxValidator other){ ... }

    @Override
    public boolean isValid(Number value) { return value==null || value.doubleValue() ≤ this.max; }
    @Override
    public Validator<Number> cloneValidator() { return new MaxValidator( other: this); }
    @Override
    public void onInit(Max max) { this.max = max.value(); }
}
```

4. **ChainPrototype**
   ChainPrototype interface introduce the interface for all type of chain. It provides a method processValidation for starting of validate the object through the chain. A chain contains a collection of validators which are responsible for validating a value/ object passed to the chain. ChainPrototype also follow Prototype pattern and therefore provide cloneChain method signature. The clone of a chain including cloning all its validators which also support cloning. For the need of complex validator structure, such as validate a POJO class which include validating separate fields, each field is an independent chain. So, we adapt Composite Pattern to simulate such cases, this **will help achieve one chain for each object only**. The appendValidator method is for a leaf chain (chain contains only validators), and the addChain method is for composite chains (contain chains). They are shifted to the interface which violate interface segregation but brings transparent to the ChainPrototype interface. These methods are default to throw an error in case that implementor don't need it and don't need to override, while in usage it will also not give unpredicted behavior.
5. **ValidatorChain**
   Concrete leaf chain which contains a head of the chain, also the tail for easier manage (add). The chain just needs to tell the head node to validate, and it will pass the value to the whole chain as if possible.
6. **PojoValidatorChain**
   Concrete composite chain simulates chains of fields of a Java object. Store in a Map of ChainPrototype with key is the field that the chain validate.
7. **CollectionValidatorChain**
   Concrete composite chain holding list of chains for an object. Mostly when one need a chain to validate a collection and a chain to validate its internal data, it will construct a list of chains for that collection.
8. **CollectionInternalChain**
   Concrete composite chain holds a chain for validating collection's typed value inside the collection. It will apply the chain for all collection's elements.
9. **ChainRegistry**
   A pool to manage Prototype chains, store a whole chain for an object with the key is the object's Class simple name. It will return a clone chain of the selected Prototype.
10. **ValidatorHolder**

ValidatorHolder hold a Chain for an object (generic type) together with ViolationHandler for violation notifying. It is considered as a validating context for an object. Usually, the ViolationHandler is injected into ValidatorHolder when it is created by ValidatorProvider. Provide an abstract validate method for concrete classes to handle the validate of T value object and the notification on Violations.

a. **BaseValidatorHolder**

Concrete ValidatorHolder class which take in T value, construct ValidateObject<T> to hold T with its context. Then construct ViolationContext and validate the object. After that, notify the violations through ViolationHandler.



```
@Override
public Collection<Violation> validate(T value) {
    ValidateObject<T> validateObject;
    if (value == null) {
        validateObject = new ValidateObject(Object.class,  value: null);
    } else {
        validateObject = new ValidateObject(value.getClass(), value);
    }
    ViolationContext context = new ViolationContext().root(value);
    chain.processValidation(validateObject, context);
    context.getViolations().forEach(handler::notify);
    return context.getViolations();
}
```

11. **ViolationHandler**

    This is the violation notifier of the Observer pattern. It provides subscribe, unsubscribe, notify method. It contains references to its listeners. On notify, it will call process function of its listeners.

12. **ViolationListener**

    Interface for violation listeners provide process method which take in a Violation object. The concrete listener will decide what to do with the violation.

    a. **LoggingViolation**

       Concrete violation listener that writes the violation to log.

13. **Violation**

    An object to store data about a violation that occurs during validation, constructed, and added to ViolationContext in a Validator processing function.

14. **Validatable**

    An interface for POJO objects to follow which provide a default validate method which construct a chain of that object structure (based on annotations) with ValidatorProvider and validate the object.

15. **BaseValidateHolderBuilder**

    The concrete builder to build a ChainPrototype. It supports the build step of ValidatorChain, PojoChain, CollectionChain, CollectionInternalChain and use Stack to manage them and put them together in a root chain. Because we support transparency, this will throw ChainBuilderException on any situation that the user builds the chain incorrectly. Later in build method, it wraps the root chain inside a ValidatorHolder and inject the ViolationHandler into it and return to user. ViolationHandler is passed to the Builder when created by ValidatorProvider.