Converter and Populator Pattern

Intention

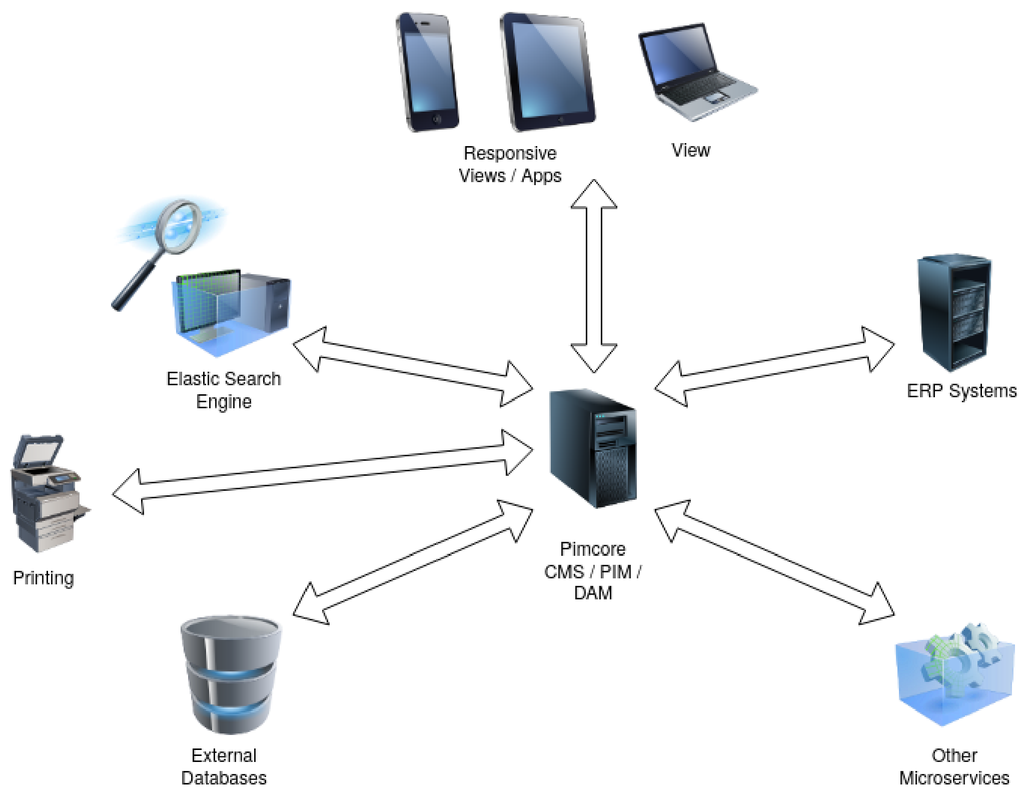
It's important to understand the intention of a pattern to avoid misuse and confusion.

The Converter and Populator pattern should be used whenever you have to transform objects of a certain type (system) into objects of a different type (system).

Let me go on speaking about source and target types.

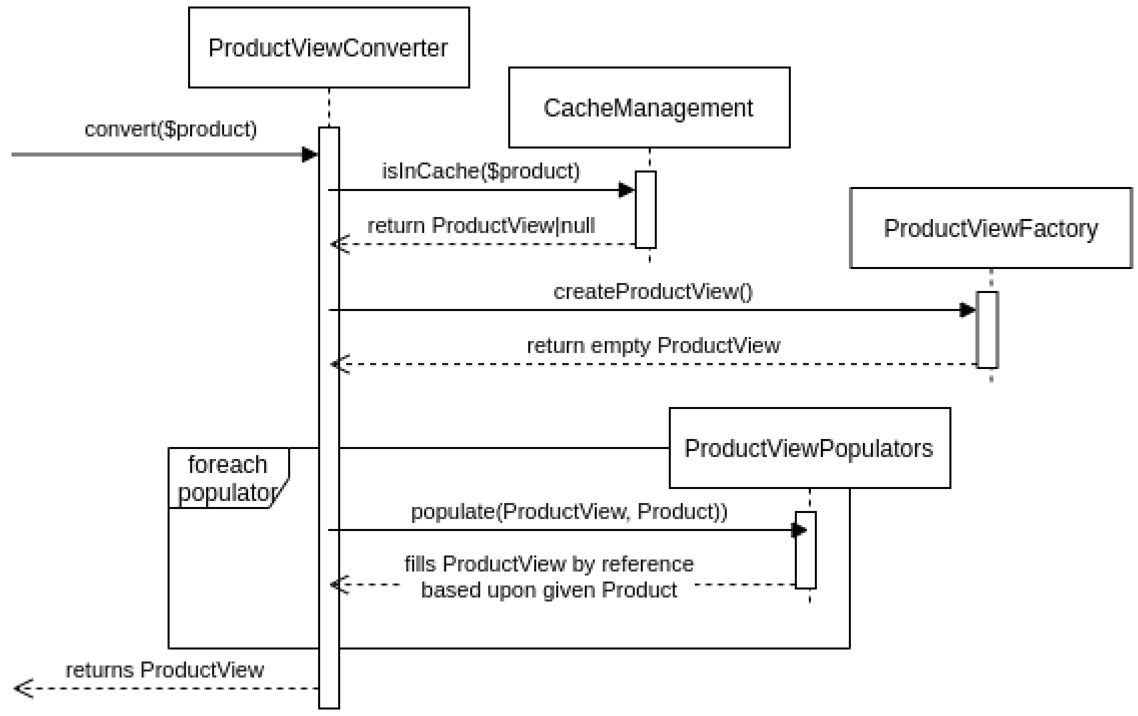
The Converter and Populator pattern is an architectural framework which guides you through the transformation (aka mapping) of objects from one universe to another.

Typically we use it for creating View objects or Elasticsearch objects based upon the business entities of our base system (e.g. Pimcore, Hybris, ...)



So many different systems using their own type system or data objects.

Pattern Structure



As you can see in this sequence diagram the conversion of objects will be done by the main component of our pattern called **Converter**.

But the Converter is more or less only the encapsulation of the steps which are necessary to "create a target object based upon the source object".

1. Look up if already converted in the Converter cache.
2. Potentially create a new target object (e.g. done here by ProductViewFactory)
3. Fit the target object with new attribute values based upon the source object (e.g. done by several different ProductViewPopulators)
4. Write back in the Cache.

Motivation

Business entity type systems are mostly created by the idea to model the real world into your digital system. Relations between entities very often will be implemented by foreign keys or other technical instructions.

A common example of the difference between the data which should be stored in the system and the data which will be shown on the screen is for example the age of a person.

Obviously it makes no sense to save the age of a person otherwise you have to think about a daily cron job which has to recalculate the age based upon the current date.

But in your view you may only want to show the age of a person and not the birthdate. So transformation is necessary.

This should give you just an idea why two different type systems based upon the same entities could arise.

Very often I've seen implementations (sometimes even inside the Controllers) done by a simple (private) method inside the view action.

I'll promise you this will grow and grow and after a while you can not handle changes without producing bugs or implement complexe acceptance tests checking that your "internal" transformation works.

So it's much better to accept the separation of the two type systems from the very beginning and start implementing this transformation by the Converter and Populator pattern.

The main reasons are the following:

1. All transformation steps will be implemented in testable small units called populators.
2. The already existing transformation will not be changed or touched by further transformations (Open-Closed-principle).

Applicability

We are using the Converter and Populator pattern in all situations where we come in touch with different type systems:

- View Type system
  As I already mentioned above in the small birthday-age-example I would recommend to separate the view to our entities always from the business type system.
  Frontend developers will love you if you give them the chance to implement their frontends based upon their "own" type system (e.g. created by 90% string attributes, 5% integer values and 5% rest).
- Elastic Search Engine
  The Elastic based description of our objects differ in very many points from our base implementation of business entities.
- Imports from external system (e.g. ERP systems like SAP)
  With the fine-grain Populator mechanism is much easier to implement different interpretations of values coming from the external system.

Participants

Target Type Converter

The Target Type Converter provides the implementation of the convert method.

If your programming language allows it it is possible to implement one single generic implementation of a converter which can be staffed with different implementations of factories and populators based upon target and source type.

Target Type

The Target Type is the type of objects which will be results of the conversion. By example it could be the ProductView class or a ProductElasticSearchDTO.

Source Type

The Source Type is the type of objects which will be converted. By example it could be the domain entity type Product.

Target Type Factory

The Target Type Factory creates an "empty" instance of the target object which should be fitted by the given source object.

Very often the implementation is just a simple new operator call.

Target Type Populator

The Target Type Populator is one component which provides a generic implementation of a populate method which fits at least one detailed attribute of our target type based upon the source typed object.

This could be a simple combination of a target - type - setter - source type - getter call. But of course Populators can in a more complex implementation use other Converters for converter aggregated objects or use different services to get additional information which are necessary for the population.

A few time ago I have always compared Populators to assembly-line workers doing only one step of the complete construction but a colleague of mine pointed out that the difference between workers and populators is that the last group should work independent to each other and potentially in parallel.

But the basic idea is the same: each Populator is doing one step of the whole conversion.

Collaboration

The Target Type Converter will be fitted with the associated Target Type Factory (mostly 1 single instance) and the Target Type Populators (mostly several instances).

The TargetType Populators can work with a well-defined target object and do what they have to do:

1. Call setters of the target object parameterized with getter calls from the source object
2. Call service methods to determine special calculated values and set the associated attribute(s) from the target object.
3. Call itself a target type converter for an aggregated partial object inside the target object.

Consequences

A positive consequence of this pattern is that you can reuse certain populations which come up several times.

A typical example are aggregated partial objects which could become a Converter-Populator construction itself.

Some developers were complaining about the complexity of this pattern but most of the artefacts you need are generic and you can reuse the code again and again.

On the other side 90% of the code needed in this pattern could be generated so the Interface and method signatures are clear and well-defined.
Last but not least only the population itself differs depending on your transformation logic.

Implementation

We use this pattern very often for transforming Pimcore Data Objects into View objects for the Frontend (Twig) Developer.

Although there are a lot of useful Twig functions one can use to show values of the Pimcore object it is much easier and more comfortable to define together with both sides of the development - Backend and Frontend - which values have to be shown and implement certain Data Transfer objects or View classes and covert and populate them based upon the Pimcore DataObjects.

Sample Code

Assume we have a facade which will be called by our Controller to get a view object based upon a Pimcore DataObject of type Firma, which is the german word for enterprises used by our customer (ubiquitous language):

```
class FirmaFacade {

   private FirmaRepository $firmaRepository;

   private ViewConverter   $firmaViewConverter;



   public function __construct(FirmaRepository $firmaRepository,
ViewConverter $firmaViewConverter)

   {
       $this->firmaRepository = $firmaRepository;

       $this->firmaViewConverter = $firmaViewConverter;

   }

   public function getFirmaView(int $firmenId): ?FirmaView

   {

       $firma = $this->firmaRepository->getById($firmenId);

       return $firma ? $this->firmaViewConverter->convert(

           $firma

       ) : null;

   }

}
```

The firmaViewConverter is the generic target type converter for Pimcore Data Objects:

```
class ViewConverter implements ViewInterface {

    private ViewFactoryInterface $viewFactory;

    private CacheManagement $cacheManagement;

    /** @var ViewPopulatorInterface[] */

    private array $populators;


    /**

     * @param ViewFactoryInterface $viewFactory

     * @param CacheManagement $cacheManagement

     * @param ViewPopulatorInterface[] $populators

     */

    public function __construct(

        ViewFactoryInterface $viewFactory,
        CacheManagement $cacheManagement,
        array $populators
    )
    {
        $this->viewFactory = $viewFactory;
        $this->cacheManagement = $cacheManagement;
        $this->populators = $populators;
    }

    public function convert(AbstractObject $source): ViewInterface
    {
        if ($this->cacheManagement->isInCache($source)) {
            return $this->cacheManagement->get($source);
        }

        $target = $this->viewFactory->create();

        /** @var ViewPopulatorInterface $populator */
        foreach ($this->populators as $populator) {
            $populator->populate($target, $source);
        }
        $this->cacheManagement->writeInCache($target, $source);
        return $target;
    }
}
```

One of the several populators we are injecting is:

```
class FirmaPopulator implements ViewPopulatorInterface
{
    /**
     * @param FirmaView $target
     * @param Firma $source
     */
    public function populate(ViewInterface $target, AbstractObject
$source): void
    {
        $target->setName($source->getName());
    }


}
```

Firma is our Pimcore DataObject with generated getters and setters and FirmaView is our target type for the Frontend development which is as well a simple data transfer object with getters and setters.