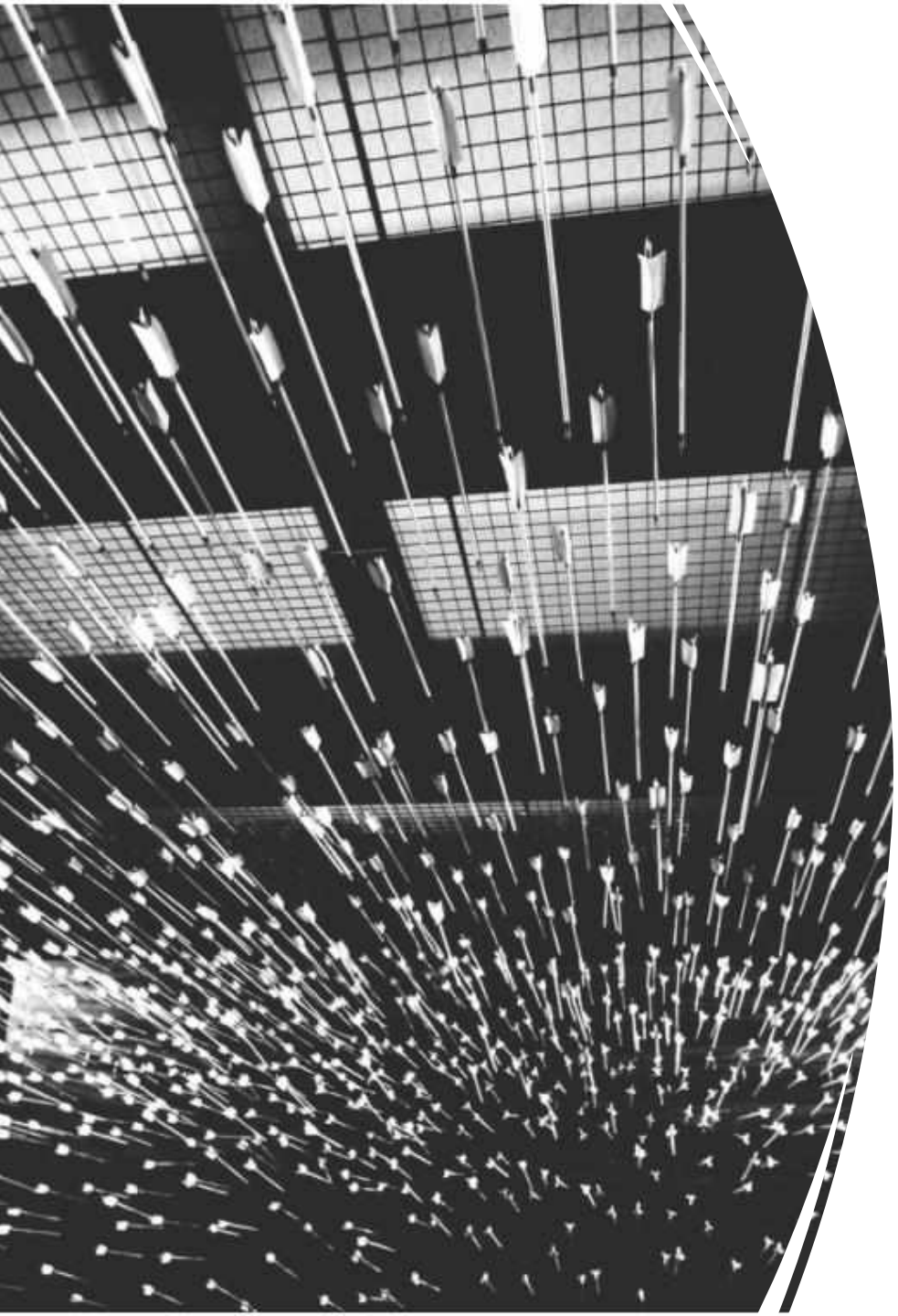
About refactoring a real world problem

Some Experiments with the open/closed principle

New Experience



Goals

- Simplyfy the printing framework by applyingopen/close principle
- Get rid of class registrations
- Get rid of long switch-statements

Approach

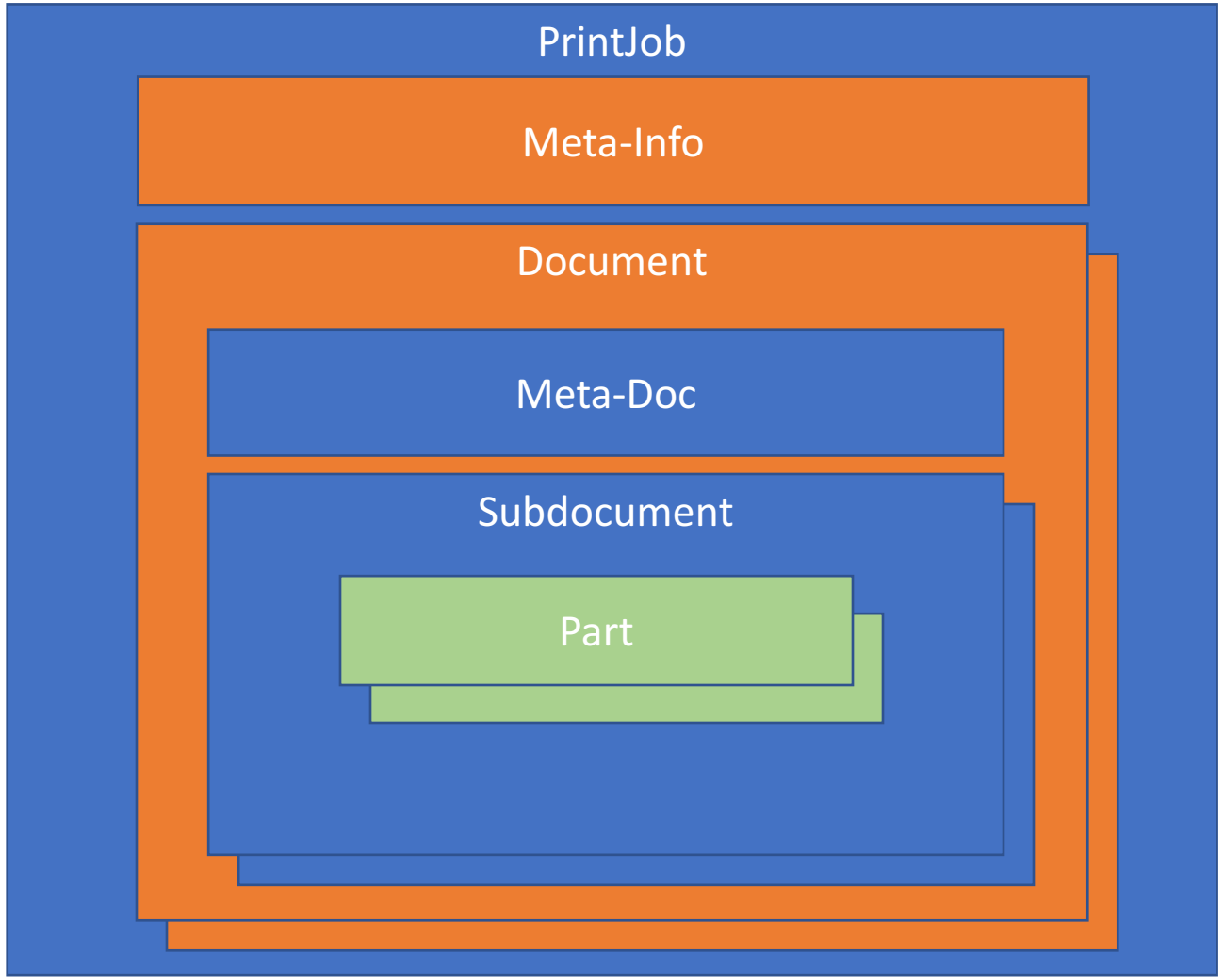
- Printing framework is aged.

⇒I wanted to play a little bit with source code in this context.

⇒I wanted to base the Framework on cdi-based discovery of extensions

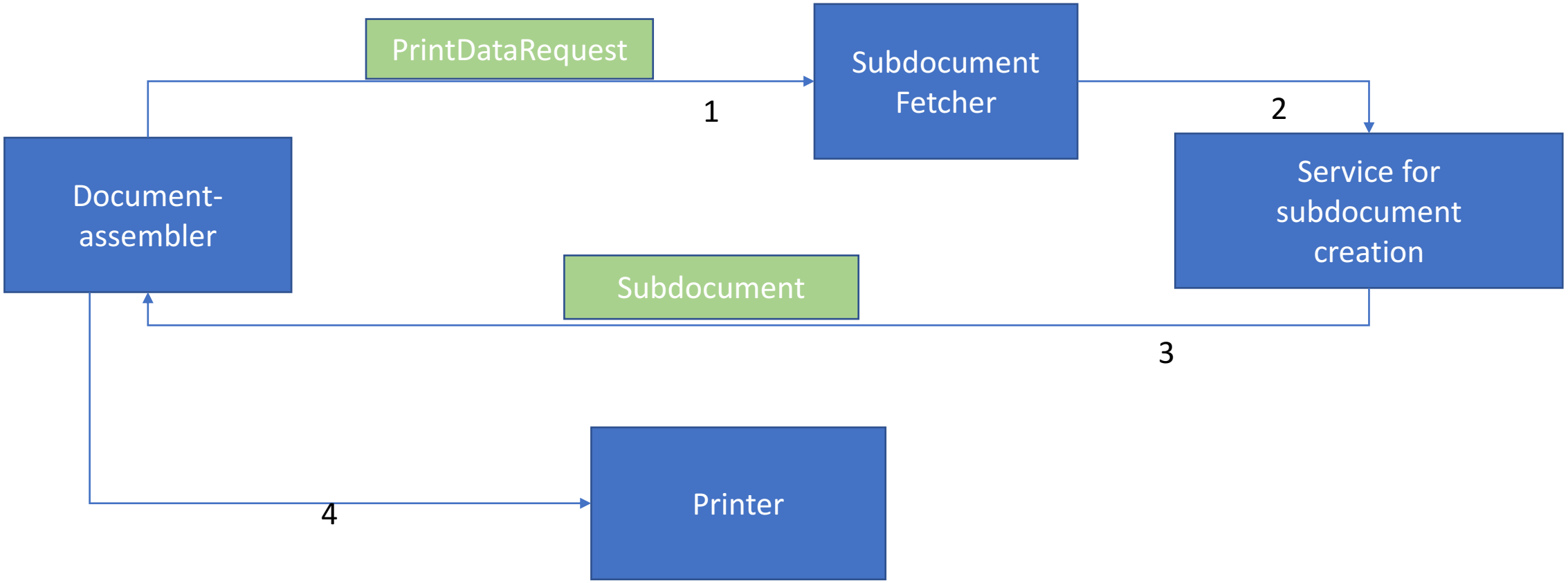
⇒I wanted to see if some ideas for optimization do realy simplify the framework.

Basic Document Structure



Current Framework basics

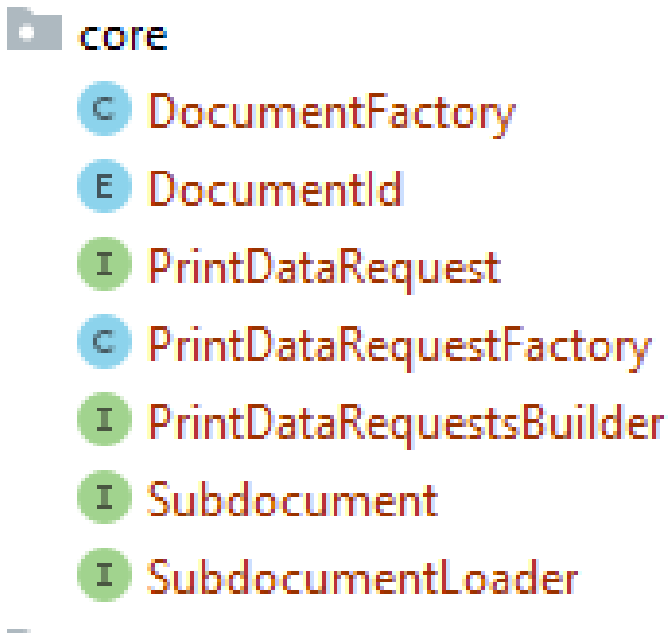
- For our considerations i’m focussing on the document and it’s subdocuments!



A simple Test Case

```
@Test
public void printDoctorList() {
   PrintDataRequestFactory printDataRequestFactory = container.instance().select(PrintDataRequestFactory.class).get();
   List<PrintDataRequest> printDataRequests = printDataRequestFactory.createPrintDataRequests(DocumentId.DOCTORSLIST);
   DocumentFactory documentFactory = container.instance().select(DocumentFactory.class).get();

   List<Subdocument> subdocuments = documentFactory.createDocument(printDataRequests);

   assertEquals("PartySubdocument,PartySubdocument", subdocuments.stream()
         .map(subdocument -> subdocument.getText())
         .sorted()
         .collect(Collectors.joining(","))
   );
}
```

Framework Core



creates the subdocuments for a document

a rather stable list of document ids

the base interface for all printdatarequests

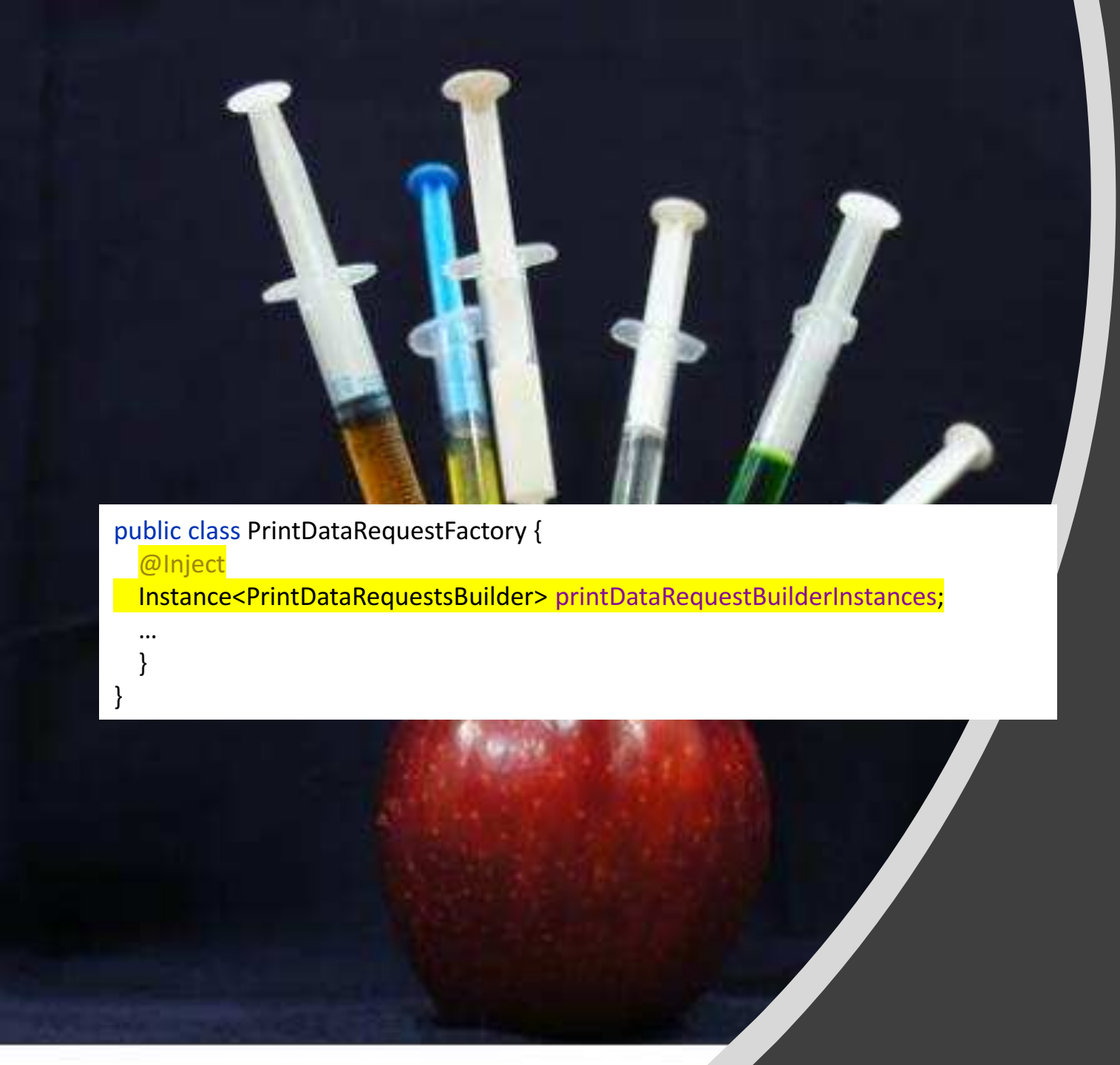
Factory that creates the printdatarequests for a document

a base interface for all printdatarequestbuilders

a base interface for all subdocuments

a base interface for all subdocumentloaders

Printing of new Documents can be added to the Framework just by providing new implementations of the interfaces.
The Factories discover these newly provides implementations automatically.



```
public class PrintDataRequestFactory {
   @Inject
   Instance<PrintDataRequestsBuilder> printDataRequestBuilderInstances;
   …
   }
}
```

Key Design Aspects

Inject all Implementations of an Interface with CDI (Weld)

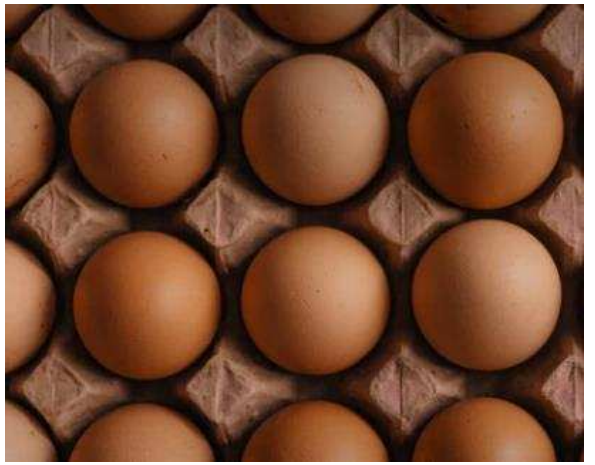
Key Design Aspects

->let the printDataRequestBuilder decide if it wants to participate for a document

```
public class PrintDataRequestFactory {
                …

   public List<PrintDataRequest> createPrintDataRequests(DocumentId documentId) {
      List<PrintDataRequestsBuilder> printDataRequestBuilders = new ArrayList<>();
      List<PrintDataRequest> printDataRequests = new ArrayList<>();
      printDataRequestBuilderInstances.forEach(
                  printDataRequestBuilder -> printDataRequestBuilders.add(printDataRequestBuilder)
       );

      printDataRequestBuilders.stream()
            .filter(printDataRequestBuilder -> printDataRequestBuilder.paticipatesInDocument(documentId))
            .forEach(printDataRequestBuilder -> printDataRequests.addAll(printDataRequestBuilder.buildPrintDataRequests()));
      return printDataRequests;
   }
}
```

DocumentFactory can be done the same way

```
@Inject
Instance<SubdocumentLoader> subDocumentLoadersInstances;

public List<Subdocument> createDocument(List<PrintDataRequest> printDataRequests) {
   List<SubdocumentLoader> subdocumentLoaders = new ArrayList<>();
   subDocumentLoadersInstances.forEach(subdocumentLoader ->
subdocumentLoaders.add(subdocumentLoader));

   List<Subdocument> subdocuments = new ArrayList<>();
   subdocumentLoaders.stream().forEach(loader -> {
      subdocuments.addAll(printDataRequests.stream()
            .filter(printDataRequest -> loader.paticipatesInPrintDataRequest(printDataRequest))
            .map(printDataRequest -> loader.loadSubocument(printDataRequest))
            .collect(Collectors.toList())
      );
   });
   return  subdocuments;
}
```



Open Issue

```
@Override
public Subdocument loadSubocument(PrintDataRequest printDataRequest) {
    if (printDataRequest instanceof ReceiverPartyPrintDataRequest) {
        return new PartySubdocument();
    }
    if (printDataRequest instanceof DoctorListPartyPrintDataRequest) {
        return new PartySubdocument();
    }
    throw new IllegalStateException("bla");
}
```

Conclusions about redesigning complex code

easy things => no need for deep methodology. But most often things got complex over the years.

1. Do some of the refactorings learned in the course for a better understand of the existing implementation
2. Reason about how to make things better
3. **Build a prototype for your ideas** and see how they look like in the code
4. Refactor your prototype until it solves the problem
5. Refactor your real code to the concepts in the prototype

- It will nevertheless be challenging to get it done right
- It is to expensive to do an iterative design refactoring directly on the real code base without verifying it upfront in a prototype.

- **Do you remember the key points about the feedback loop?**