UDC 004:37

Alexander Spivakovsky, Sergey Tityenok, Dmitry Berezovsky, Yana Storozhuk, Alexander Litvinenko, Nataliia Klymenko
Kherson State University

THE PROBLEM OF ARCHITECTURE DESIGN IN A CONTEXT OF PARTIALLY KNOWN REQUIREMENTS OF COMPLEX WEB BASED APPLICATION "KSU FEEDBACK"

The problem of flexible architecture design for critical parts of "KSU Feedback" application which do not have full requirements or clearly defined scope. Investigated recommended practices for solving such type of tasks and shown how they are applied in "KSU Feedback" architecture.

Keywords: *KSU Feedback Service, 360 degree feedback, survey, target groups, software development, architecture, poll.*

Introduction

In 2009 Research IT department of Kherson State University (KSU) started to implement the prototypes of product that could be used for collecting opinions on arbitrary events and persons. The project got name "KSU Feedback". The first real use of service had a goal to collect feedback from KSU students to estimate quality of education process. The service provided an ability to create polls online, fully automated process of gathering results, granted the anonymity of the respondents, but at the same time allowed to strictly delineate the set of participants, produced the real-time aggregated results visualized on the chart. The obtained data could be stored and then used for comparison with the results collected using the same poll, but for different time interval.

Since that time it was decided to extend the sphere of usage of "KSU Feedback" and, based on experience we had, to create a platform that could cover most needs appearing during the organization of the feedback process in various cases. In other words, we were going to write a complete platform for automation of the process of building the so-called "360 degree feedback".

We defined several categories which the end product should be applicable to:

– Estimation of human resources management in corporations;
– Creation of personal CVs;
– Online fast feedback from the third-party applications;
– Marketing research.

Of course, at the moment of writing the specification we were aware of basic requirements to the target system but anyway we could not define the exact set of functionality that would be required in particular usage. So even final version of the project vision still contains unclear places which potentially can be extended or changed in future when we will have refined the requirements.

From the point of view of our experience we understood that the system would be complex and require a lot of development work. We planned to split the project into a common functionality part (a core component) and concrete implementations (sub-projects) which would use the core.

Summarizing all above, we can say that we came across a problem of developing complex IT system with partially defined requirements. As a solution for this problem we considered the possibility of creating such application architecture that would help us to implement the part of functionality which is clearly specified and then complete it with functionality for discovered ongoing needs. We have found it useful to make a research of the optimal architecture that might be applied in such situation and did not even exist.

Thus, the complex system "KSU Feedback" with partially-known requirements naturally became the object of our research and thereafter its architecture was accepted as a subject of investigation.

In our opinion, the obtained results can be treated as successful if they solve the problems listed below:

1. Architecture specifies the way of adding completely new modules which reuse the existing code base;
2. Architecture allows extending already existing modules by adding only task-specific code;
3. Architecture helps avoiding massive code refactoring after having made changes of requirements to core part;
4. Architecture defines a strict way of extending the existing type of functionality with additional features;
5. Architecture allows splitting the project into independent units of work.

In the best case we expect to design an architecture that should help the "KSU Feedback" project achieve the following goals:

1. Quickly react on functionality changes;
2. Easily extend the service with the existing kind of functionality (for example, adding new types of question, voting process, data export format, etc.);
3. Always have independent pieces of functionality responsible for their own single task;
4. Always have a working, runnable version of the application including at least a limited set of functionality.

In our research we consider the best practices of architecture design, try to find concepts which can help bringing the flexibility into the project and make an attempt to apply this concept in architecture of "KSU Feedback" service.

Investigation

In the introduction we described goals and tasks the end architecture should be able to solve, but it produces a logical question: is it ever possible?

During the initial gathering of the requirements we used Domain Driven Design (DDD) methodology and, as a result, we got an extensive set of objects which take a part in service working cycle. Thus, we decided to use object oriented paradigm in our architecture.

Object oriented programming (OOP) already has known list of recommendations and best practices to go through and even set of typical mistakes which is better to avoid (anti-patterns). But it is important to understand that none of them is kind of "panacea". Each solution solves specific problem and has its motivation to use. For this reason we made a research of recommendations which, on our opinion, can help to reach declared tasks. We made main focus on solutions which usually used to make objects independent, interchangeable and have uniform processing.

But big variety of options to go has raised a problem of estimation each of the solution. In other words we wanted to know based on what criteria we can conclude the solution is good or bad. We investigated known scales which usually applied to the architecture and among not big set of alternatives we stopped on SOLID concept identified by R.C. Martin.

SOLID is design principle and is acronym of five concepts which must be complied:

1. Single responsibility principle – an object should have only a single responsibility [1]
2. Open/closed principle – software entities should be open for extension, but closed for modification [2]
3. Liskov substitution principle – objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
4. Interface segregation principle – many client-specific interfaces are better than one general-purpose interface [1]
5. Dependency inversion principle – "depend upon Abstractions. do not depend upon concretions" [3]

At this point when we can estimate the solutions we are ready to search for the best patterns which can be applied to cope with the concrete technical problems listed in the introduction. We also kept in mind that it is not necessary to have a complete flexibility for totally all parts of architecture. The important points are only:

– General application infrastructure

– User interface changeability
– Communications between the components and database

Below we listed only those solutions which were the nearest candidates to be involved in final architecture design.

Common Application Infrastructure

Since the main accent in a context of partly-known requirements is system extensibility we need to organize linking between different modules (in terms of our application). Thus ideal architecture will allow us to add an instance of particular helper object into pool of available objects just by declaring this class guided by some known conventions.

There is special programming technique, designed to solve this architecture problem called *Inversion of Control* (IoC), which opposite to canonical programming flow when objects are statically assigned to one another, allows to assemble object graph in runtime by defined object interactions through abstractions.

The basic idea there is to have separate object, an assembler, typically implemented in some framework\library (IoC container), that populates fields of the object with concrete appropriate implementations [4]. It will be known which implementation should be used only in runtime, for example after classpath scanning. The author of this concept, Martin Fowler [4], suggests the following example for explanation of the concept: Let’s assume that we’re writing a component that provides a list of movies directed by a particular director. Let’s call this component MovieLister. It asks a finder object (which we'll get to in a moment) to return every film it knows about. Then it just hunts through this list to return those directed by a particular director. UML diagram for this is shown on Figure 1.

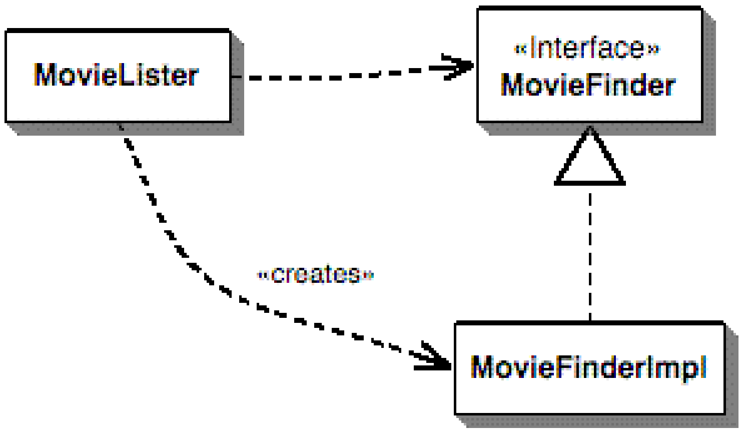

Fig. 1. *The dependencies using a simple creating in the listener class*

The question there is how to connect *MovieFinder* object to MovieLister and how minimize development in case when we need to replace implementation or if we need more than one *MovieFinder* implementation, and even don’t know who and when will extend our project with this new implementations? Traditional approach is to instantiate necessary implementation somewhere in MovieLister, most likely in constructor. But this will require from developer modification of the code of MovieLister just for replacing implementation. And that is the main issue there especially if MovieLister is distributed as compiled assembly and developer hasn’t ability to change it. To avoid such situation IoC container provides assembler which is responsible for instantiation and linking objects to each one another. Thus, using IoC we will get the following:

Definitely the example, described above is a bit synthetic but in real application with huge amount of objects and dependencies benefits of using IoC are more obvious.

User Interface Layer

This is another important part of application design, since due to the nature of the project it will provide complex UI controls (e.g. poll editor). Also we need to keep in mind that the service will be Web-based, so the communication between users and logic that responds on their request will be done through stateless HTTP protocol.

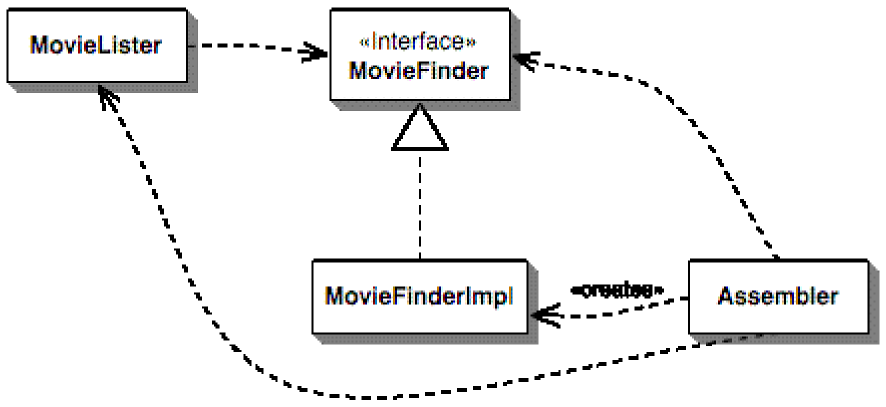

Fig. 2. *The dependencies for a Dependency Injector*

But the main goal here is to achieve full independence between complex business logic and fancy interface that would help us to have strictly separated pieces of work.

After investigation of UI patterns we found the Model-View-Presenter (MVP) pattern as the one that best fits our needs. It deserves the role of main principle of interaction between "KSU Feedback" application and the users.

The concept behind the MVP pattern is that an implementing application should be splitted into three core components; Model, View and Presenter:

– The Model component encapsulates all Business Logic and Data in the application. This may be a database transaction or a call to a web service, etc.
– The View component represents the application's Presentation layer (User Interface); this may be a standard Win Forms client, an ASP.NET Web part or Mobile client. In the MVP pattern, the View should be simplistic and responsible for rendering and accepting user input only.
– The Presenter component is responsible for orchestrating all the application's use cases. For example a sample operation would involve; taking user input from the View, invoking operations on the Model and if needed, setting data in the View to indicate the operation's result. [5]

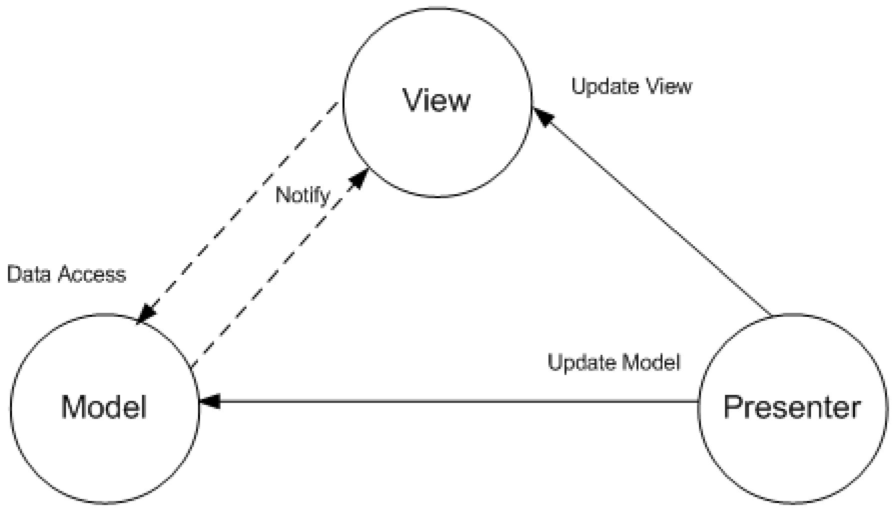

Fig. 3. *MVP components*

On a Figure 3 we can see, objects which are invoked into process of handling user's actions exists outside of each other and communicate through some defined interface.

Transaction Management

Since the most of DB related operations should be transactional, in other words if to perform any action on DB complex entity which causes execution of a set of atomic DB operations, we need to have an ability to rollback already executed of atomic operations in case of fail one of this operations. According to the widely used concept when data manipulation methods are implemented in special objects, designed for managing specific type of persistence entity. Such objects are called Data Access Objects (DAO). Thus typical method of the DAO class should look like the following:

Typical DAO method structure

```
class CompanyDao implements ICompanyDao {
.......................
@Override
public Company lockCompany(Company company, User locker, String reason) {
getCurrentSession().beginTransaction();
try {
// Do some DB operations (inserts, updates, selects)
getCurrentSession().commitTransaction(); // Commit on success
} catch (RuntimeException e) {
// Rollback if failed somewhere
getCurrentSession().rollbackTransaction();
}
}
.......................
}
```

It is easy to see that in each method we need to open transaction on the begining, close it at the end, rollback in case of runtime exception. To avoid duplication of this code, and make it more readable it is possible to use *Proxy* objects which will do transaction related logic before and after original method call. Proxy is well-known design pattern, Figure 4 illustrates appropriate UML.

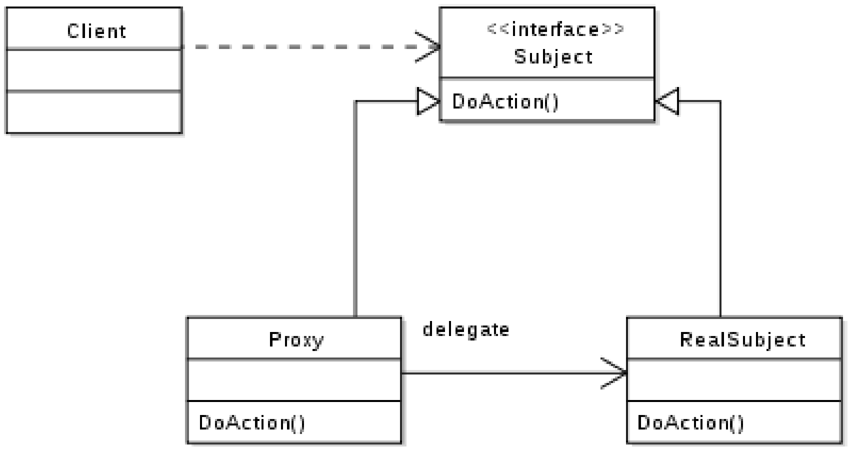

Fig. 4. *Proxy pattern in UML diagram*

The same fragment using such approach will look like the following:

DAO method using proxy for transaction management

```
class CompanyDao implements ICompanyDao {
  .......................
  @Override
```

```
    public Company lockCompany(Company company, User locker, String reason) {
      // Do some DB operations (inserts, updates,
    }
    .......................
  }
```

Event Driven Internal API

To make possible development of both extension modules without modifying main code and web services with instant event notification we need to provide an ability to hook a moment when some event occurs, e.g. poll has been created; respondent committed his vote, etc. Since it is well-known task there is design pattern, called "*Observer*" which helps to implement transmitting event information to the client once event has been fired.

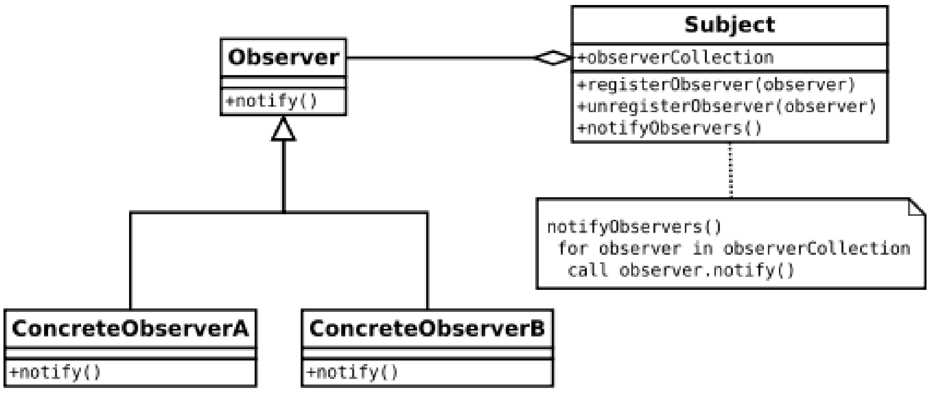

Fig. 5. *Observer pattern in UML*

The intent is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [6]. Thus, each module can subscribe on events which it is interested in by implementing special observer interface and registering it in observer repository. The core module should call *notifyObservers()* method of the repository object once it is time to notify clients that some event has been occurred. Observer repository will call *notify()* method for each of registered observers.

Implementation

All the problems we were looking solutions for, have their reflection in concrete scope of functionality that needs to be done in "KSU Feedback". Thus, let us bring more specificity and provide the list of program parts which are the most critical from the perspective of the tasks we have declared in introduction:

– General application ecosystem – part that organizes communication between all layers of application, defines project structure, provides with a context at all points of program life cycle.
– User interface components and interacting with user layer – part that is responsible for displaying the information to user, reaction on his actions, invocation of appropriate logic.
– Component "Poll" – part that can display set of questions of different types to user.
– Voting access control - part that provides an ability to specify which groups of people can access a particular voting by different conditions.

It was already decided to use Java language and Java Development Kit (JDK) platform in development, so it came possible for us to extend technological stack with ready solutions which implement necessary part of architecture.

Below we take a closer look at each of parts introduced above.

General application ecosystem

Summarizing requirements we defined before, and finding known techniques for building flexible independent components we selected Spring framework as the one which implements a lot of architecture design parts we need: IoC container, annotation driven configuration, managing objects life cycles. As result we have very flexible context with easy configuration.

To split huge application on the big reusable parts, responsible for a certain type of task we divided it on projects, which will be compiled on the separate binary modules:

– *common* – library which contains common utility methods, object definitions which will be widely used by almost all other parts
– *domain* – models in terms of MVC methodology annotated with persistence annotations. This is also a library which doesn't contain any business logic.
– *api* – library contains interfaces and public facades for application interfaces. This will allow as in future easily prepare SDK for 3-rd party developers or change business logic covered by default implementations of these interfaces.
– *service* – this library contains actual implementation of the business logic. Most of the classes implements interfaces defined in api project.
– *web-common* – library which contains common reusable UI components.
– *web-corporate* – actual web application for corporate edition.
– *web-personal* – web application for Personal Feedback (simplified and personal-oriented service).
– *worker* – daemon in linux terms, which performs scheduled background tasks like bulk e-mailing, cleaning db, etc.

Figure 6 illustrates all these projects in UML

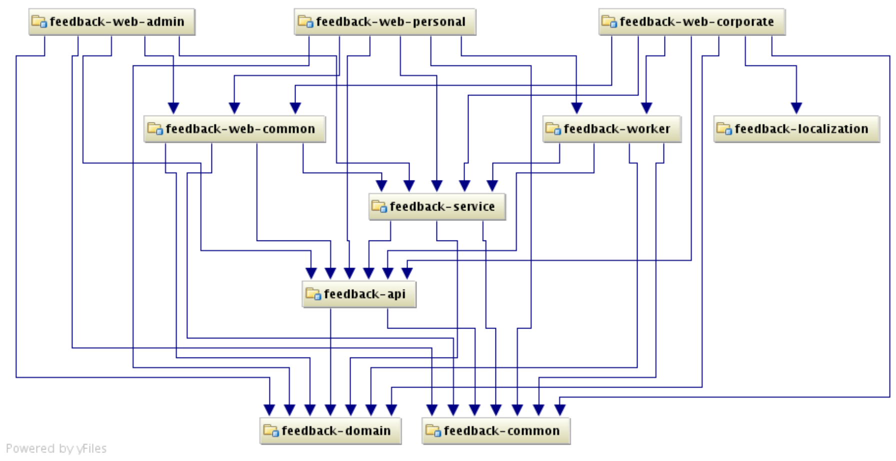

Fig. 6. *Project structure*

Such approach along with idem Spring allows us to add or even replace implementations of the defined interfaces without recompiling previously created modules. The following example from our project illustrates using of dependency injection for building flexible application.

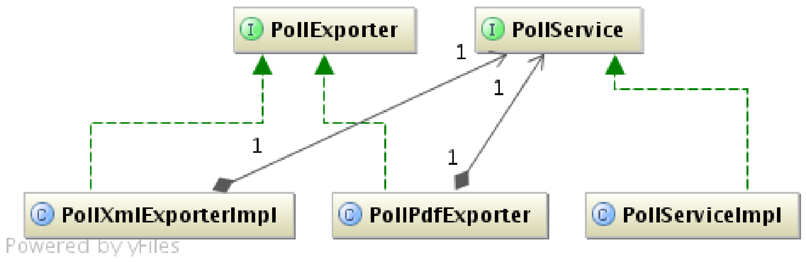

Fig. 7. *Typical dependencies between classes from the Service project in UML*

On the Figure 7 we have interface PollService which represents a collection of business logic methods related to polls, its implementation, called PollServiceImpl, interface PollExporter which represents the object responsible for exporting given poll in different formats and finally concrete realizations of the PollExporter for exporting in XML and PDF.As you see from UML abowe both of exporter implementations use PollService interface below the hood, but doesn't know anything about implementation of the PollService. This allows us to replace implementation each of these classes from the one hand and add another implementation from the other.

The most interesting part there is creating and linking concrete classes. IoC container is responsive for doing this job automatically before application start.

Let's look at the code snippets below.

```
PollServiceImpl class
@Component
public class PollServiceImpl extends EntityServiceImpl<Poll> implements PollService {
..........................
}

PollXmlExporter class
@Component
public class PollXmlExporterImpl extends AbstractXmlExporter implements  PollExporter {
    ...................
    @Autowired
    private PollService pollService;
    ...................
}

PollPdfExporter class
@Component
public class PollPdfExporterImpl extends AbstractXmlExporter implements  PollExporter {
    ...................
    @Autowired
    private PollService pollService;
    ...................
}
```

We use annotation driven configuration which is more flexible and allows avoiding XML declarations for each managed bean. Each class which is annotated with *@Component* annotation will be picked up by IoC container. Such objects in terms of Spring are called *managed objects or managed beans* [7]. During context initialization, typically immediately after application start Spring will instantiate each managed object and perform dependency injection basing on existing meta information, defined in XML file or using annotations.

To configure dependency injection we use *@Authowired* annotation which indicates that class field should be initialized with object instance from context. Since annotation doesn't provide any information about required object injection will be done by type. This means that after initialization of the PollXmlExporterImpl for each field annotated with *@Autowired* IoC will try to find suitable object instance in context. In this example it will try to find a class which implements *PollService* interface since annotated field is of type *PollService*.

The other good example is also widely used in project. It allows creating repositories of the objects basing on the classes available in classpath. Let's look at the other UML.

In this case we have another service which is for exporting votes to various formats. Again, each exporter implements *VoteExporter* interface and is annotated with *@Component* annotation. The main difference is in the *VoteServiceImpl*:

```
VoteServiceImpl class
@Component
public class VoteServiceImpl extends EntityServiceImpl<Vote> implements VoteService {
   .....................................
    @Autowired
    private List<VoteExporter> voteExporters;
   .....................................
}
```

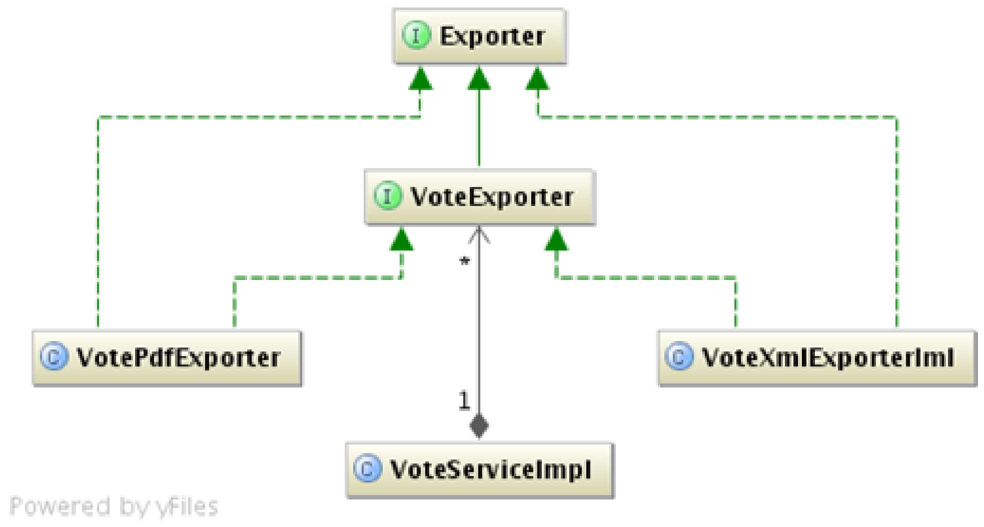

Fig. 8. *Repository of the Vote Exporters in UML*

Instead of auto wiring single object we annotated field which stores collection of *VoteExplorers*. Spring will detect this and put all available objects from the container which implements this interface. This feature allows us to extend functionality without modifying existing components. Thus it is easy enough to create a separate library, compiled independently from the main application which extend it with for instance new type of report or question fragment. Let's assume that we would like to add exporter which allows exporting files in RTF format, but our application is in production already and we don't want to rebuild it. Thus, here are the steps we need to go through:

1. Create new project which depends on *Api*, *Domain* and *Common* projects. All these projects contain only interfaces and model definitions without business logic.
2. Implement appropriate exporter interface and annotate it with *@Component* annotation. It's important to say that we are able to use any of the services here by simple adding private field of the appropriate type and annotating it with *@Autowire* annotation. Even though we haven't any of the implementations in our project they will be available in application context.
3. Compile project in the .jar file and put it under the main application context path.

After that our new exporter will be handled by IoC container and injected in *VotingService* along with other ones.

User interface components and interacting with user layer

Defined scope of functionality requires a lot of user interface views and forms. The problem is that components are not trivial and need to be reused in different places. Along with that we have very complex logic that depends on the data that user has entered. Obviously it would be nice to have the business and user interface logic separated. Thus we decided to try to use the concept of MVP described before. But unfortunately Java does not provide ready MVP solution "out-of-box". However there are a lot of free frameworks which do it. We stopped our choice on Apache Wicket framework, which in addition brings such benefits as completely pure HTML templates at View level, reusable components of Presentation layer and well abstracted at Model level.

Besides, inside of the components we can use helper objects from the IoC container, which allows us to move all complex logic to special handler. Moreover we involved Callback pattern that gives to class' users ability to inject custom handling logic for particular events (e.g. form was submitted, dialog was closed, etc.) Thus particular component contains generic logic of user interface objects creation, validation and internal interaction. Fig. 9 shows simplified class diagram for poll editor:

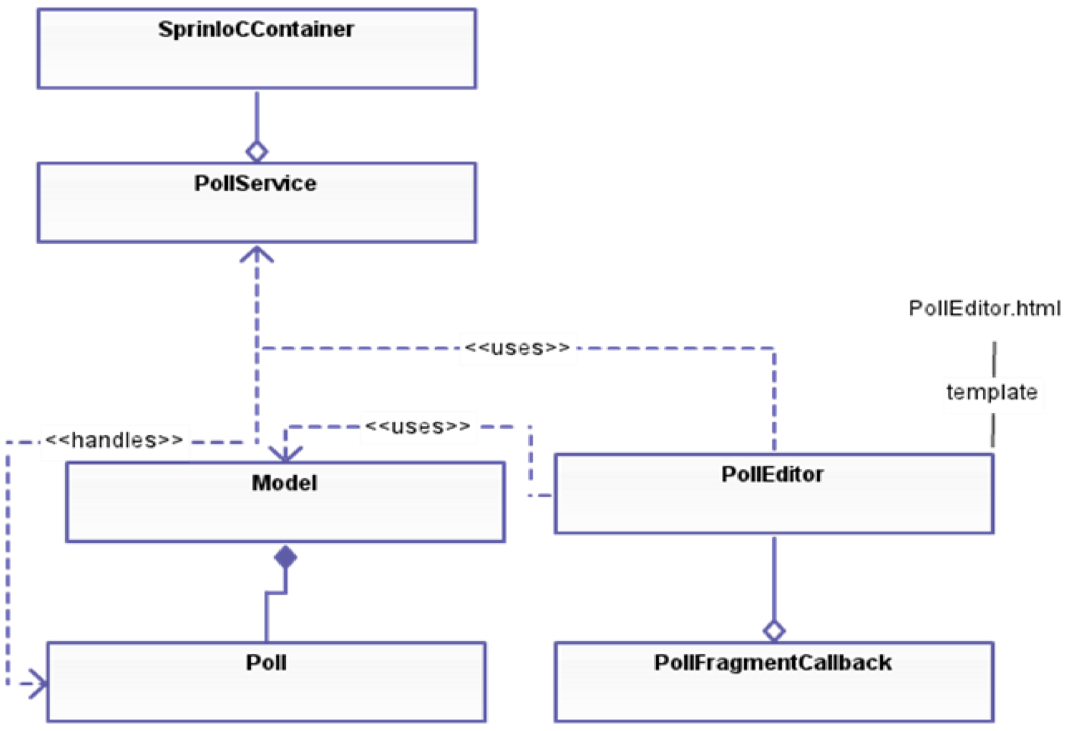

Fig. 9. *Simplified UML class diagram shows relations between PollEditor components*

As you can see PollEditor object works with Model (Model layer) that encapsulates Poll domain object. All handling logic of it is done through *PollService* obtained from SpringIoCContainer. Clients of *PollEditor* (Presenter layer) can add their additional handling by setting *PollFragmentCallback* facade which methods will be called on particular events of *PollEditor* life cycle. View level is represented by pure HTML markup file that is linked to *PollEditor*.

The great benefit here is that we can reuse each written component and have business logic separately. All communications between service logic and Presentation layer are done via Model abstraction that encapsulates business domain object. Thus components can be reused everywhere in context that operates with the same business objects.

Voting access control

Because of "KSU Feedback" specificity, it is very important to control the access to the voting. Moreover, it is one of the main project features which give user several types of access limitation for voting process. In internal project terms the group of people which can reach the voting process through defined by user access criterias is target audience.

Each target audience provides its own specific way of authorization. User can specify the details in editor component and indicate what audiences he wants to include for particular survey. Then, it must be able to authorize the respondent by data he has provided. So each target audience has to deal with a lot of responsibilities which we splitted into several classes. Fig. 10 shows class set that needs to be created per one concrete target audience (*SinglePasswordAudience*).

The problem is that we do not have full list of target audience and most likely this part will be extended with new types of access in a future. Along with this we need to write a core component that will work with target audience in generic way without relying on any specific type. This task looks quite hard as handling process is type sensitive. That means, for example, that only one type of editor can edit one specific type of target audience; the same situation is with the rest of the components.

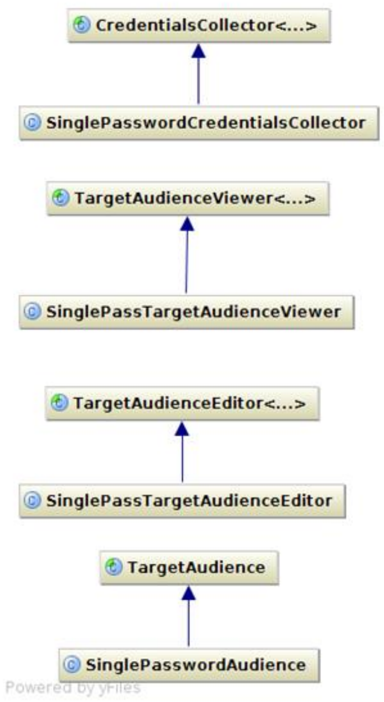
CredentialsCollector<...>
SinglePasswordCredentialsCollector
TargetAudienceViewer<...>
SinglePassTargetAudienceViewer
TargetAudienceEditor<...>
SinglePassTargetAudienceEditor
TargetAudience
SinglePasswordAudience
Powered by yFiles

Fig. 10. *Set of classes for concrete target audience*

Obviously, it would be much easier if we would have some object that contains all necessary information about relationship between target audience and its helper components. For this reason we created custom annotation *@TargetAudienceMeta* for encapsulation of meta information about each component that handles particular target audience. It must present above declaration of each class that participates in working cycle of target audience. For example:

```
@TargetAudienceMeta(type = SinglePasswordAudience.class)
public class SinglePasswordCredentialsCollector{...}
```

As you can see TargetAudienceMeta annotation indicates that *SinglePasswordCredentialsCollector* collects credentials information for target audience of type *SinglePasswordAudience*. The similar thing is done with editor and viewers components. The final step was to write special class scanner that goes through specified package and looks for annotated classes. As the scanner class implements event model and thus each time it founds appropriately annotated class it calls register method of *TargetAudienceResolver* class which implements the Repository design pattern. By the end of initialization we have fully instantiated repository that provides us with useful methods for resolving the types of helper objects for target audience (see Fig. 11).

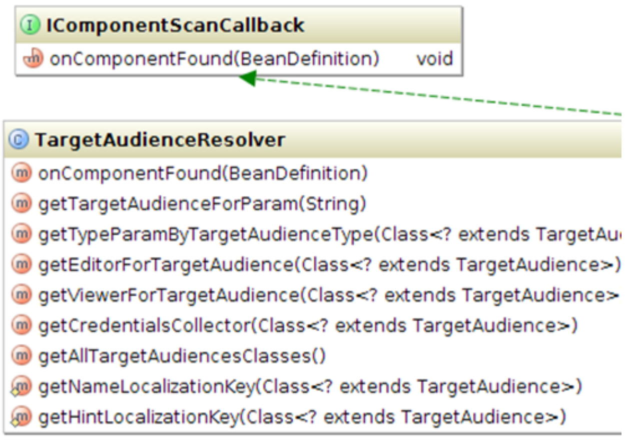
IComponentScanCallback
onComponentFound(BeanDefinition) void
TargetAudienceResolver
onComponentFound(BeanDefinition)
getTargetAudienceForParam(String)
getTypeParamByTargetAudienceType(Class<? extends TargetAu
getEditorForTargetAudience(Class<? extends TargetAudience>)
getViewerForTargetAudience(Class<? extends TargetAudience>
getCredentialsCollector(Class<? extends TargetAudience>)
getAllTargetAudiencesClasses()
getNameLocalizationKey(Class<? extends TargetAudience>)
getHintLocalizationKey(Class<? extends TargetAudience>)

Fig. 11. *Target audience repository in UML*

Such approach gives us an ability to create a common logic that works with target audiences despite their type. It will get required information from *TargetAudienceResolver* instance. Moreover it helps split the work into independent units which are connected with each other only at meta-information level.

Conclusion

The problem of incompletely defined requirements for the critical parts of application demands big flexibility from architecture in order to be able to write the components in a future without any code refactoring. We made investigation of best practices which can be applied to help achieve it. Based on the obtained results we tried to use them in architecture of most important parts of "KSU Feedback".

The result we got fully corresponds to SOLID principles we have used for estimation of quality of a solution, and that is necessary, however not sufficient condition that we designed an optimal architecture.

But we already had a chance to approbate it as we have come through several iterations of "KSU Feedback" using this architecture. As a result it helped us to achieve the points listed below:

4. It is easy to extend with new functionality (we were able to create formal instruction of how to extend particular parts).
5. We write only project-specific code for personal and corporate versions of feedback.
6. We always have runnable version of application.

Of course we are sure we did not find the ideal way that can be applied for each product in all situations but at least it looks like it covers the goals we had at the beginning of our research. And even for our case it also how its downside. The problem is that architecture is very complex and requires big experience in programming and deep knowledge of several Java frameworks from developers. This was omitted during architecture design phase but can become a serious problem during the implementation, since each new team member should already have good experience in Java enterprise development. However this problem obviously is out of scope of this article and needs to be considered separately.

REFERENCES

1. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices (2002)
2. Martin, R. C.: "The Open-Closed Principle", C++ Report (1996)
3. Freeman, E; Freeman, El.; Sierra, K., Bates, B.: Head First Design Patterns (2004)
4. Fowler M.: Inversion of Control Containers and the Dependency Injection pattern (2004), http://martinfowler.com/articles/injection.html
5. MSDN Blogs, Using the Model-View-Presenter (MVP) Design Pattern to enable Presentational Interoperability and Increased Testability, http://blogs.msdn.com/b/jowardel/archive/2008/09/09/using-the-model-view-presenter-mvp-design-pattern-to-enable-presentational-interoperability-and-increased-testability.aspx
6. Johnson, R., Vlissides, J., Helm, R., Gamma E.: Design Patterns: Elements of Reusable Object-Oriented Software (1994)
7. Walls C.: Spring in Action (2011)

Стаття надійшла до редакції 17.03.2013.

Співаковський О.В., Тітенок С.О., Березовський Д.О., Сторожук Я.І., Литвиненко О.А., Клименко Н.О.

Херсонський державний університет

ПРОБЛЕМА РОЗРОБКИ АРХІТЕКТУРИ СКЛАДНОГО ВЕБ-ДОДАТКУ "KSU FEEDBACK " В КОНТЕКСТІ ЧАСТКОВО ВІДОМИХ ВИМОГ

Проблема розробки гнучкої архітектури для критичних частин додатку "KSU Feedback", вимоги та об'єм робіт для яких визначені не вповній мірі. Досліджені рекомендовані практики для рішення такого типу задач, і показано як вони застосовані в архітектурі “KSU Feedback”.

Ключові слова: feedback 360, розробка, цільова група, розробка програмного забезпечення, архітектура, анкета

Спиваковский А.В., Титенок С.А., Березовский Д.А., Сторожук Я.И., Литвиненко А.А., Клименко Н.О.

ПРОБЛЕМА РАЗРАБОТКИ АРХИТЕКТУРЫ СЛОЖНОГО ВЕБ-ПРИЛОЖЕНИЯ “KSU FEEDBCK ” В КОНТЕКСТЕ ЧАСТИЧНО ИЗВЕСТНЫХ ТРЕБОВАНИЙ

Проблема разработки гибкой архитектуры для кртических частей приложения “KSU Feedback”, требования и объем работы для которых не определены в полной мере. Исследованы рекомендуемые практики решения такого типа задач, и показано как они применены в архитектуре “KSU Feedback”.

Ключевые слова: feedback 360, разработка, целевая группа, разработка программного обеспечения, архитектура, анкета