

LK Rekryterings System

Pre-analysis Report

2008-01-31

Rüdiger Lincke

Overview

We analyzed a .NET-based application called *LK Rekryterings System*. It consists out of compilation units written in C# and ASP being distributed in 7 packages.

Raw data extracted

The following provides an overview about the data extracted from the source code. It lists the single entities and relations according to their type. The later analysis will be based on these.

Nodes 1230, Edges 3526

Node data

Condition 365
EnumerationValue 5
Field 137
Method 324
Event 3
Property 165
Enumeration 2
Interface 1
Class 68
Package 7
Constructor 72
Loop 80
Solution 1

Edge data

Invokes 742
AccessProperty 763
Accessess 632
AccessEnumeration 17
Extends 27
contains 1229
InvokesConstructor 115

Global Statistics

The global statistics provide an overview about the entities being of highest interest. We can see that we have in average approximately 10 classes per package, 4.7 methods per class, 4.4 fields and properties per class, which are quite rather low values, but point at a good modularity.

| | |
|------------|-----|
| Packages | 7 |
| Classes | 68 |
| Interfaces | 1 |
| Fields | 137 |
| Properties | 165 |
| Methods | 324 |

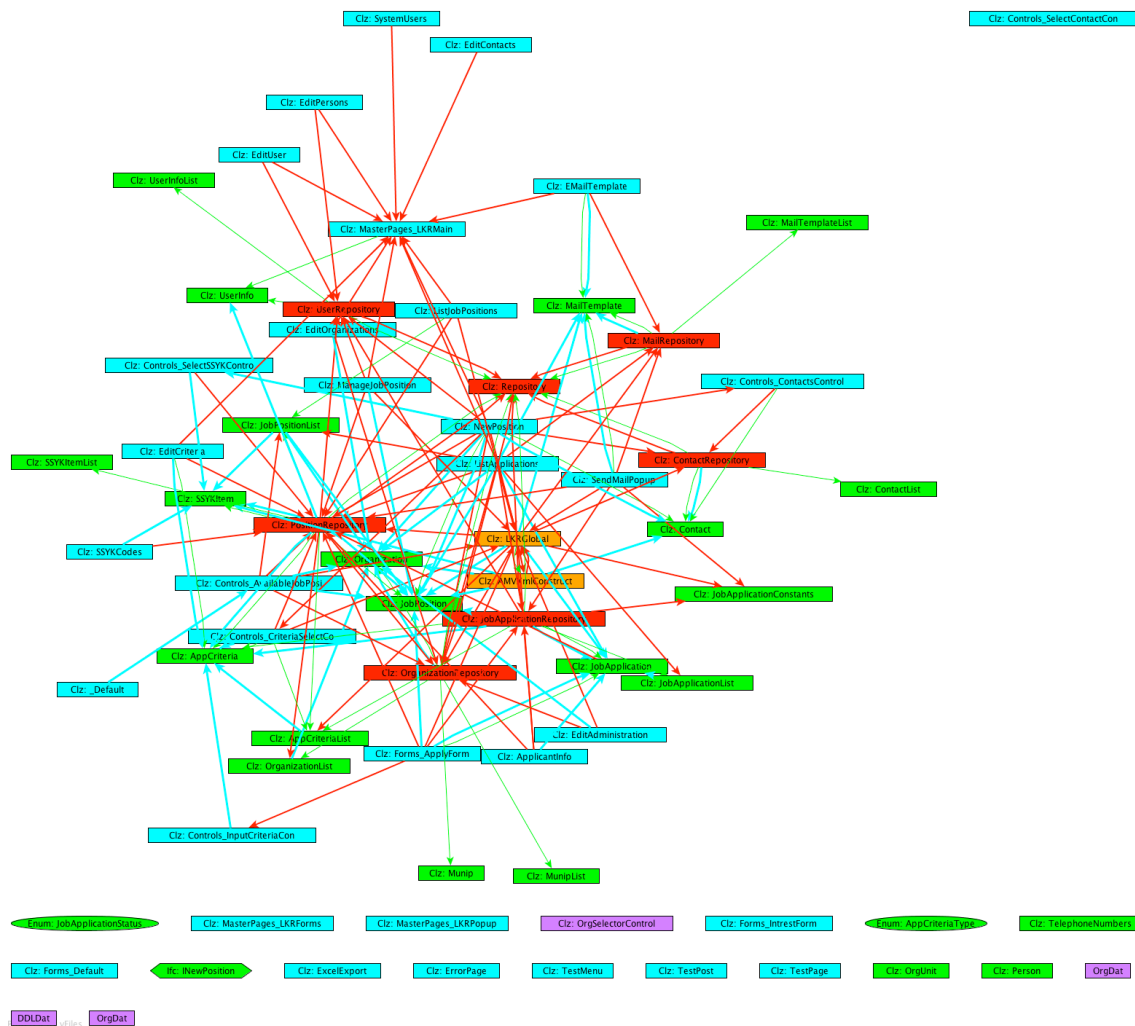
Remarks: After discussion it turned out that classes, fields and methods are of rather low importance. The main focus is on the compilation units, that is .aspx, .ascx, .webcomp, and .cs.

Architecture Analysis

We performed an analysis of the system architecture from three different views. First we tried a clustering to identify components having a high internal cohesion and a rather low external coupling. Secondly, we tried to identify layers, and last, we looked for a hierarchy in the call structure.

Class Call Graph – Components

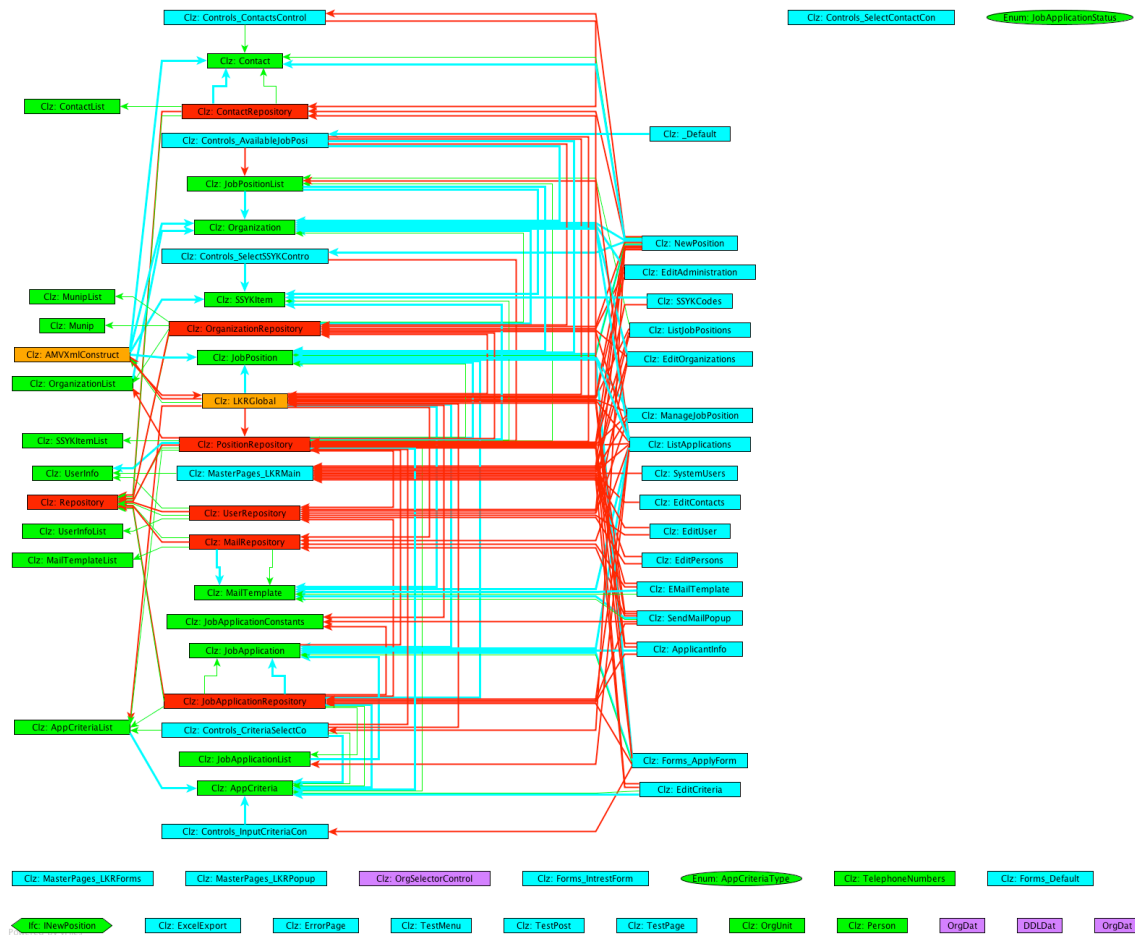
The created class-call-graph shows no clear component structure, when looking for natural components.



Remarks: After discussion, we agreed that there are no particular components foreseen in the current architecture.

Class Call Graph – Layers

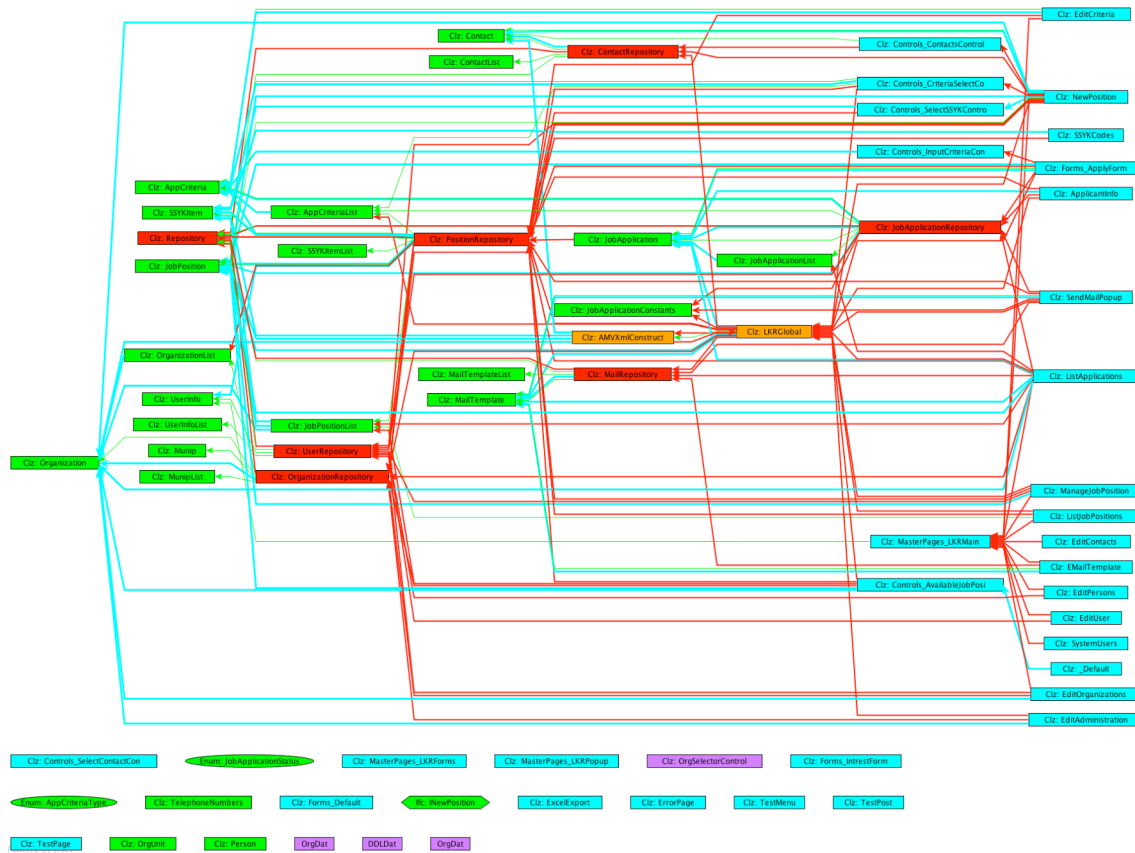
The created class-call-graph shows a layered structure. When looking for three-tier architecture layers we could identify a server layer (left) and a client layer (right), both are communicating with a middle layer. The classes of the layers are from different packages.



Remark: After discussion we found that there is no strictly layered architecture. The blue classes represent the view. The red classes represent the data repository. The green classes represent data types. The orange nodes represent classes of the global logic. The blue, orange and red classes use the green classes to exchange information.

Class Call Graph – Call hierarchy

Looking at the call hierarchy there are no large cycles.



Remarks: There are no particular remarks. The blue nodes belonging to the view control call methods of the other nodes, which are either data structures (green), global logic (orange) or data storage/repository (red).

Metrics Analysis

Status: in progress

Interesting classes (high coupling):

- NewPosition
- ListApplications
- PositionRepository
- LKRGlobal

Remarks: The identified classes were well known to the developers. Further analysis is necessary to identify the attributes of interest, and to create a suitable software quality model for them.

Problems

Partial Class Declarations, we are not sure how to treat them. Currently we aggregate them.

Remarks: After discussion, we found out, that we should not aggregate the partial classes to represent whole classes. It is more important to see the single parts and their relations.

Questions

Architecture:

- What is your picture of the architecture?
- We did not find clearly separated components, but layers.

Other questions:

- Your impression of quality

Answers

The intended architecture of the LK Rekryteringssystem was described as following:

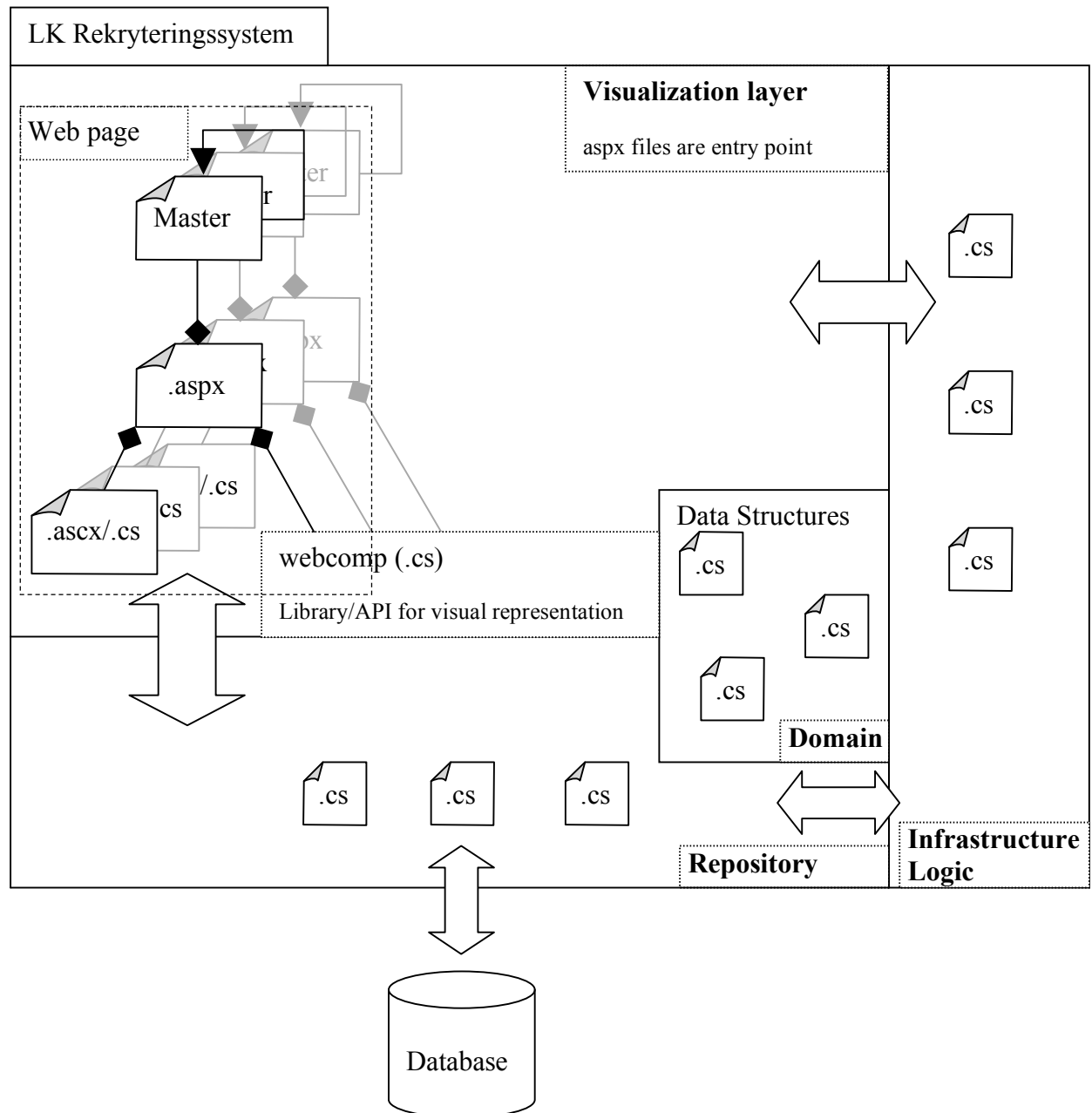
There are three layers, a visualization layer and a repository layer being horizontal to each other, and an infrastructure layer being vertical to the other two layers. The visualization layer is the user interface, controlling the various actions. It sends and retrieves data via the repository layer from the database. None primitive data types are transmitted using .cs files from the Domain. The visualization and repository layer use the Infrastructure/Logic layer for utility functions.

The Infrastructure/Logic layer contains only .cs files defining control classes.

The Repository layer contains .cs files defining control classes for storing and retrieving data from the database.

The Domain contains .cs files defining data classes containing non-trivial data types.

The Visualization layer contains .aspx pages which are the entry points to the web application. They are independent of each other representing the user interface. The .ascx and .cs files implement functionality used by the .aspx page. They can be shared between different .aspx pages. .aspx pages never reference each other. .aspx pages include one or more Master pages describing visual appearance being common to the different .aspx pages.



Possible rules/patterns to check:

- Classes defined in Domain are simply abstract data types. They should not contain complex methods and program logic. Their data should be encapsulated and only accessible through getter/setter methods. Classes of all layers can use them.
- Classes defined in Infrastructure/logic are control classes. They are expected to contain complex method and program logic, but they should not contain data. They might use classes of Domain for transmitting data. Classes of the visualization and repository layer communicate with them.
- Classes defined in the repository layer are control classes. They are expected to contain complex method and program logic, including SQL statements. They should not contain data. They might use classes of Domain for sending/retrieving data. Classes of the visualization and infrastructure layer communicate with them.
- Classes defined in the Visualization layer are:
 - o *Webcomp*, which are library classes providing reusable methods. They can be used by .aspx pages.
 - o *.ascx/.cs*, are classes implementing visual elements of .aspx pages. They are can be used by the owning .aspx page.
 - o *.aspx* pages, inherit visual appearance and behavior from one or more Master pages. The **must not** reference other .aspx pages.
 - o *Master pages*, are reusable .aspx pages possibly inheriting form other Master pages. They are reused by .aspx pages.

Main interest:

- no connections between .aspx pages are allowed.
- Dependency structure between .aspx pages, master pages, and .ascx/.cs files.

How to continue from here

We discussed two directions for continuing:

1. Architecture representation (Rüdiger is looking into this)
 - a. We define a better visualization of the architectural concepts focusing on the dependencies between the different view parts (visualization layer)
 - b. We define an architecture checker, warning if .aspx pages reference each other.
2. Metrics evaluation
 - a. We adapt our automated metrics calculation to meet the specifics of the current project/architecture. (Rüdiger is looking into this)
 - b. We collect validation metrics about a selection of classes from the Rekryteringssystem for validation of the automated metrics. (Artisan is looking into this)

Next meeting

The next meeting is to be determined. It will possibly be in about 2 weeks to discuss the results of the metrics collection.

Proposal for collecting validation metrics

We propose two ways of collecting validation metrics. One implies that an expert assesses the quality of selected classes. The other implies that we correlate quantitative data, like bug reports, to parts of the system.

Expert opinion about quality of classes

We would like to get the expert opinion of Artisan (Jimmy) about the quality of a selection of classes regarding maintainability, and its four sub-characteristics, which are:

- *Analyzability*, allows drawing conclusions about how well software or parts of it can be analyzed. That is the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
- *Changeability*, allows drawing conclusions about how well software can be changed. That is the effort needed for modification, fault removal or for environmental change.
- *Stability*, allows drawing conclusions about how stable software is. That is the risk of unexpected effects as result of modifications.
- *Testability*, allows to draw conclusions about how well software can be tested and is tested. That is the effort needed for validating the software and for assessing the test coverage.

For each selected class (10-15 of all) the above four characteristics should be assessed on a scale between 1 and 5. (1 = low, 3 = normal, 5 = high).

Jimmy is looking into this.

Quantitative data

We did not discuss this in the meeting, but we would like to know what project related information is available which could relate to quality. E.g.:

- Do you have change requests?
- Do you have bug reports?
- Do you have data about duration, cost, classes involved for implementing the bug/change requests?

Background is, that we try to find, e.g., classes which contained many bugs, or took a long time to change, because they were “bad” quality, or classes being “good” quality, because they were easy to change, had few bugs associated.

Maybe Susanne could look into this.