Analysis and visualisation of defects in software projects

Student: Stefan Wink
Supervisor: Michel Chaudron
Special thanks to: Ariadi Nugroho

October 2008
LIACS, Leiden University

Table of Contents

Abstract:
This report discusses the results of my parser. The main aim of the research was to visualise defects in software projects, which have been encoded into .xmi files, and to analyse the generated images for patterns/notable things.

Introduction

The idea for this project began when I had an oral exam with my Software Engineering teacher, Michel Chaudron. We had a discussion on how UML (Unified Modelling Language) models [1] are actually related to Java [2] code. I then started a search on the internet and was surprised how little I found.
Later, when the bachelorprojects [3] started, I went to Michel to ask if he had anything on this subject. He did have something interesting, which would involve programming, UML models, software defects and coupling, combined leading to error visualisation. This report is about my findings.

Subject background

UML is a standardised visual specification language for object modelling. It is used to create an abstract model of an object-oriented system. It contains a variety of diagram types, including class diagrams, which we'll use later on. A class diagram consists of classes (drawn as a rectangle with the class name, attributes and operations) and their relationships (drawn as a line) to one another. A relationship can be for instance an association or generalisation.

An association defines a relationship between two classes of objects which allows one object instance to cause another to perform an action on its behalf.

Unless specified otherwise, the relation is bidirectional. In case there is a specification, it is displayed in the form of an arrow pointing from one class to another.
There is also multiplicity. This indicates how many instances of the class at this particular end of the association can be linked to an instance of the class at the other end of the association. E.g. *1* (one and one only), *0..1* (zero or one) or *1..** (at least one).

Associations are usually used in the context of "sending a message" or "invoking a method" from one class to another.

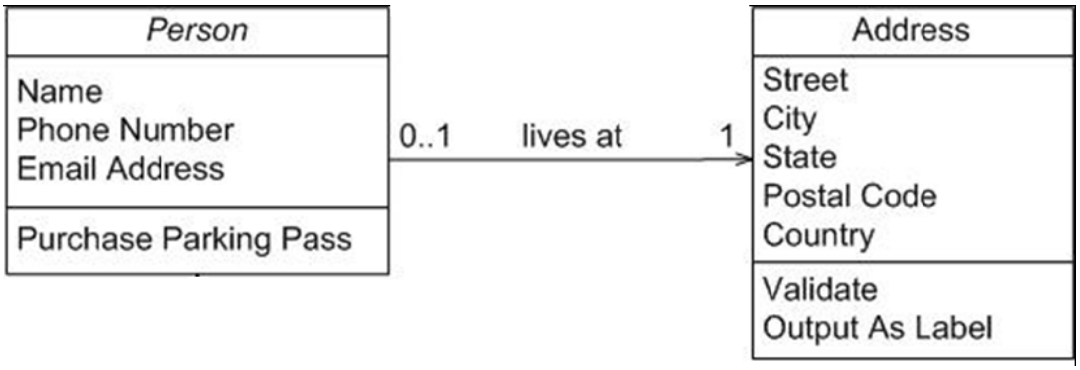


Figure 1: UML Class Diagram with an association

The corresponding code for *Figure 1* would be:

```
public class Person {
  String Name;
  String Phone_Number;
  String Email_Address;
  void Purchase_Parking_Pass();
  private Address personAddress[1];
}

public class Address {
  String Street;
  ...
  void Validate();
  ...
}
```

There is an association from `class Person` to `class Address`, which is one-way only. Content (let's take `String Street` as an example) from the `class Address` can be retrieved through calling `personAddress.Street`.

Coupling is a phenomenon which occurs when there are interdependencies between one module and another. It can be used as an indication for software quality. In general, the more tightly coupled (high degree of interconnections) a system is, the harder it is to understand and modify the system, because of the interdependencies. A change in one module will probably lead to changes in the other module(s) (because they often share variables or functions). A network of interdependencies makes it hard to see at a glance how a certain component works. If you want to use a specific component that is coupled to other components, you will have to import all of the components in order for them to work.

A loosely coupled system is preferable, since when changing a system you'd have to make sure that the change doesn't affect other classes in a way you do not want them to. Therefore a loosely coupled system will generally contain less error-prone classes, leading to a better system in terms of maintaining and changing.
To reduce coupling, you have to reduce the number of connections between modules and the strength of the connections. Only use coupling when it makes the modelling easier, which will lead to a program that is easier to maintain.
Coupling is not necessarily a bad thing, it just has to be used in a controlled manner.

Lethbridge & Laganiere [4] state several types of coupling:
1) Content coupling: where one component repeatedly changes internal data from another component.
2) Common coupling: the use of global variables in a program
3) Control coupling: function's parameter determines what the function does. This is also referred to as polymorphism.

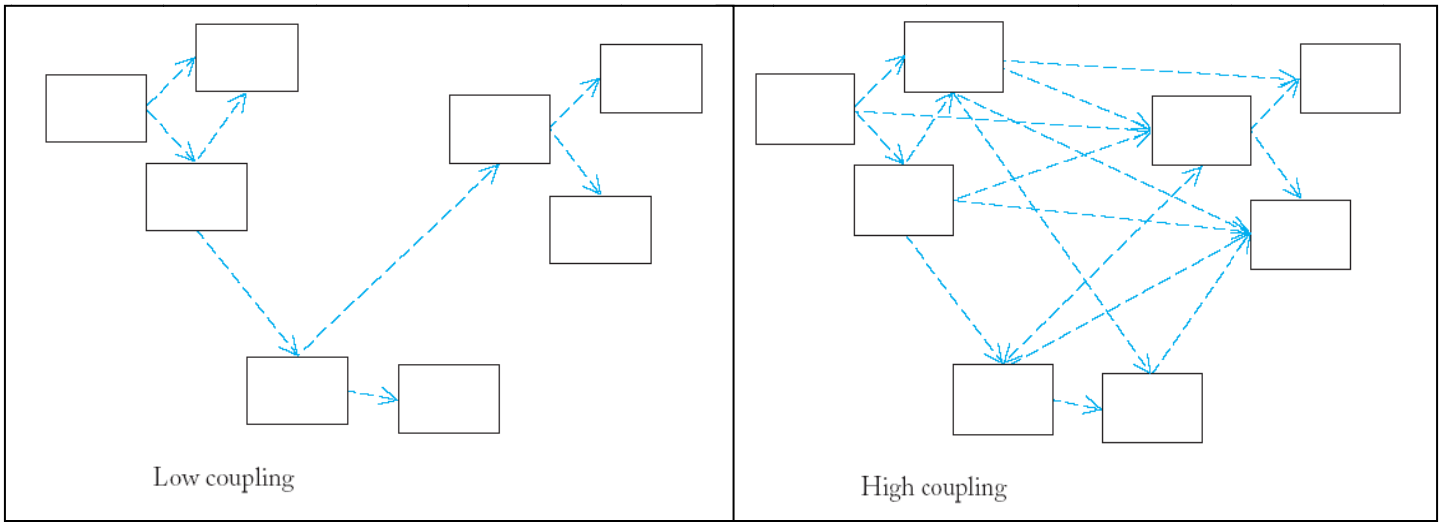


Figure 2: Two identical class diagrams with a different degree of coupling

Coupling is essentially a form of an association.

Let's take a look at some definitions on defects used in Lethbridge & Laganiere's book which was mentioned on the previous page:
A failure is unacceptable behaviour exhibited by a system.
A defect is a flaw in any aspect of the system including the requirements, the design and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures. A defect is also known as a fault.
An error is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system. It can be made at any stage of the software development process, from requirements to implementation and maintenance.
Improved education and disciplined approaches to software engineering should lead to fewer errors, and hence fewer defects and fewer failures.

Solution approach
UML diagrams can be (and often are) stored in a file type called XMI (which stands for *XML Metadata Interchange*). It's an integration of four industry standards [5], including UML and XML, and has an XML-like layout but it has a few constraints on file index, headers and tags. A very small simplified portion of the project file we've been provided with is this:

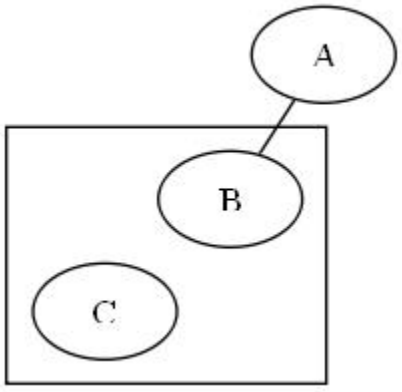
```
<UML:Package name = "medical" xmi.id = "{F12312}">
      <UML:Class name = "AppointmentForm" xmi.id = "{AB41G2}">
      </UML:Class>
      <UML:Interface name = "DiaryBusinessInterface">
      </UML:Interface>
      <UML:Class name = "DiaryForm" xmi.id = "{B12B51A}">
      </UML:Class>
</UML:Package>
```

The structure of the file is pretty clear. XML is a member of the mark-up language family, which uses tags (both opening and closing) extensively. In XML, every opening tag must have its own closing tag. The number of opening and closing tags in the entire file is therefore equal. By keeping track of how many opening and closing tags there have been while processing a file you can calculate the "depth" of the current tag (by subtracting the amount of closing tags from the amount of opening tags).
Tabs are used so that the structure of the file becomes more comprehensible.
In this example there are declarations of one package, two classes and one interface. The classes and interface are declared within the package, which means that in the UML model they are encapsulated in the package *medical*.

To visualise these diagrams we use GraphViz [6]. This is a program which allows for fast graph (matrix) drawing, using its own syntax. The syntax is quite simple:

```
graph G {
      node A;
      subgraph clusterBC {
            node B;
            node C;
      }
      A -- B;
}
```

This example would draw three nodes, A, B and C, of which B and C are in the same cluster and A and B are connected to each other through a line:



This is really all I need to visualise the UML diagrams, since I'm focussing on packages, classes, interfaces and associations. Lines correspond to associations, nodes correspond to classes or interfaces and clusters correspond to packages. As a bonus, some tweaking with node shapes, colours and line thickness/styles can be applied.

GraphViz incorporates different rendering algorithms, of which the most notable are *dot* and *neato*. *Dot* rendering is used for directed graphs, *neato* is used for undirected graphs. Since I'm not interested in the direction of the associations, I'll go with *neato*. More precisely: I will use *FDP*, which is an extension on *neato*, supporting clusters (to visualise packages).

The XMI file has to be converted somehow to an input format with which *FDP* can render a nice image. Therefore, a parser had to be written. Java seemed like a good option, because for Java there are readily available libraries which support XML parsing, for instance DOM [7] or SAX [8]. It was just a matter of importing the right libraries and make use of the parsing functions. I started experimenting with the SAX libraries and they worked right away, so I continued with SAX to write the parser.

SAX uses a couple of functions which are called when the parser recognises a certain part of the XML file (e.g. a tag, information between a tag, a closing tag).
Such SAX functions are *startDocument*, *endDocument*, *startElement(String name, Attributes atts)*, *endElement(String name)*, *characters(char[] chars, int start, int length)*.
Say we have the following example:

```
<UML:Class name = "MedicalDataLoader" xmi.id = "{A221-123-321}">
```

In this case, the *startElement* function is called and the String name will be filled with "Class", and attributes is a dataset which is filled with the attributes. They can be extracted by using *atts.getLocalName(i)* and *atts.getValue(i)*. In this case, *atts.getLocalName(1)* would return "xmi.id" and *atts.getValue(1)* would return "{A221-123-321}".

By storing the data of classes and interfaces (these will become the nodes) and combining these with the associations (these will become the interconnects of the nodes) we can compose a nice image through GraphViz visualisation.

GraphViz's syntax also allows for different shapes of nodes, thickness of edges and nodes and the use of colours. We'll use all of these options to try to create a clear image, where differences can be spotted easily.

GraphViz's output (the image) can be in a number of formats, such as *gif*, *jpg*, *png*, *svg*, and plenty more. While jpg-like formats work fine for small inputs (say <100 nodes), they don't work for larger inputs. The program will keep fully utilising CPU and memory for a couple of minutes and then aborts. This is because GraphViz tries to render the image within the memory and processes it through a certain algorithm. It tries to minimise the number of overlaps, but at a certain point you'll receive an error message which tells you that the *iterate number* has exceeded the value of *INT_MAX*.
As said, it does work for smaller inputs, but they would still return large images. A 50 node input with a couple of associations gave a 32 megapixel (resolution of 8000x4000 pixels) image. Since the project has 1000+ nodes, this clearly wasn't the way to go. Luckily, GraphViz has support for *svg* (Scalabe Vector Graphics). With vectors, you don't have the problem that an image is built up pixel by pixel. Rendering an *svg* is much faster than jpg (seconds as opposed to minutes), and the *svg* remains a very small file

(<1MB). The other great thing about *svg* is that you can zoom at any level without loss of quality.

To view these *svg* files, you could use a regular browser with a plugin for *svg*, but this doesn't work smoothly. Therefore I recommend ZGRViewer [9], a standalone Java application specialised in viewing files generated by GraphViz. The great thing about it is that it keeps track of the names and ID's, which can be viewed by doing a mouse-over on them. This isn't possible with browser plug-ins.

ZGRViewer needs to be linked to the GraphViz render algorithm and this has to be set manually. How to do this is explained in the Appendix.

I've been provided with a defect database which corresponds to the original UML model. Essential data from the database was extracted to an XML document, of which a portion is shown below.

```
 <row>
  <field name="defectsid">PARTS00000305</field>
  <field name="component">diary</field>
  <field name="classname">DiaryBusinessBean.java</field>
  <field name="classtype">business</field>
  <field name="headline">Business rule. Appointment that end before it begins is possible.
UC 015</field>
  <field name="description">Changing an appointment thus, that it ends before it has
begun is possible. Seems rather ridiculous to me... This applies also to Use cases: 006,
007 and 016. MC, 2004-07-30: See notes.</field>
  <field name="finding_type">Test</field>
 </row>
```

The information of each defect is put within the `<row>` tags. A defect has several attributes. There is a *defectsid*, which is used for identification of the defect. Typically, a defect has one to five associations (lines) to affected nodes. In the case of four affected nodes by a single defect, there will be four `<row>` declarations of a defect with the same *defectsid* while each will have a different *classname*.

A defect has a *defectsid* (which is not necessarily unique because one defect can affect several classes), the affected *component* (cluster/group/package), *classname*, *classtype* (layer type), a *headline* which briefly explains the defect, a larger description of the problem and a so called *finding type*, where defects are classified. There are four kinds of *finding types* and each has its own level of severity of the defect. By scrolling through the defects file and comparing *finding types* with their *descriptions* you get a general idea of what's what:

Finding Type	***Defect example***
Acceptance	Small GUI corrections / bugs
Integration	Small GUI mistakes (e.g. printing doesn't print a date)
Review	Errors in similarity of original UML model and final program
Test	Serious error in the final program (functionality)

Acceptance and *integration* usually refer to small inconsistencies in the GUI, such as typos, text alignment or in some occasions small bugs. These are not really the kind of defects we're after here. *Review* defects are inconsistencies between the original UML models and the implementation in the final program. This will not necessarily lead to faults in the program, but can do so. *Test* defects are serious errors in the program.

The main focus is to visualise defects (of the severe type), so *test* defects will be taken into account.
It might also be interesting to look at *review* defects and compare these results to *test* defects. If there is a relation between these two types, it might show that deviating from the original UML model/specification will lead to defects in the final program.

Results
I've been provided with a software engineering project which we're going to analyse in this section. The program appears to serve some medical application, given the name of several classes (e.g. *MedicalDossierForm*, *MedicalDataLoader*, or *Patient*). There are two sets of data, a UML model of the program itself and a corresponding defect database, both in XML format. I've written a parser which uses Java SAX libraries that will take these two files as its input. The parser will create a graph that will be passed to ZGRViewer, which will render a neat image of the project, with certain enhancements applied. On demand, a *csv* file (Comma Seperated Values) will be written which contains all classes with information such as defect amount, parent package and base package. This file can be viewed in Microsoft Office Excel as a spreadsheet, with the ability of sorting data or creating graphs.

The parser has a GUI in which a lot of options can be selected. A full documentation can be found in the appendix.

In the rest of this section all settings will be left on their default settings, except for the defect settings. Note that selecting different finding types can result in very diverse images.

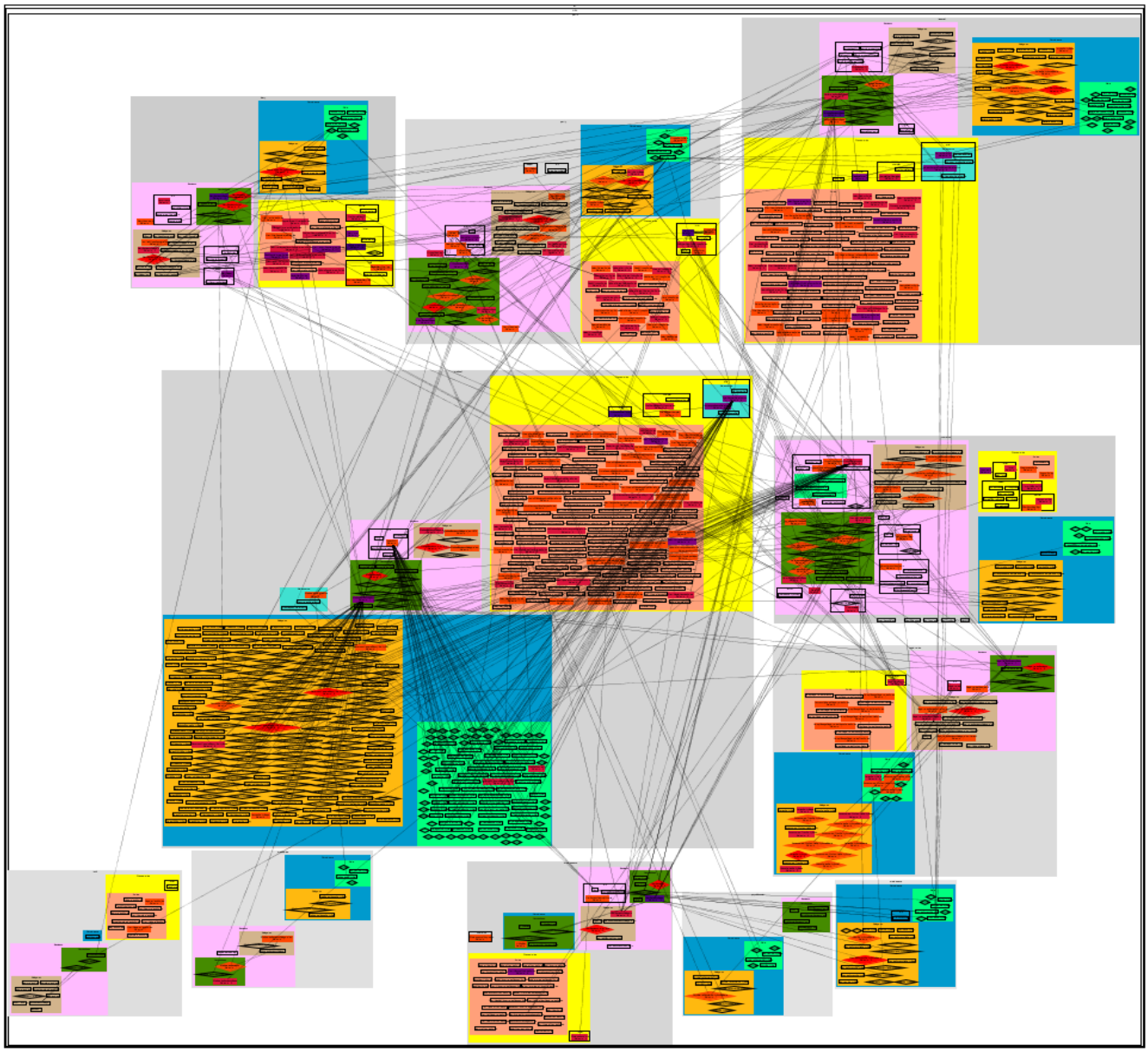


Figure 3: Overview

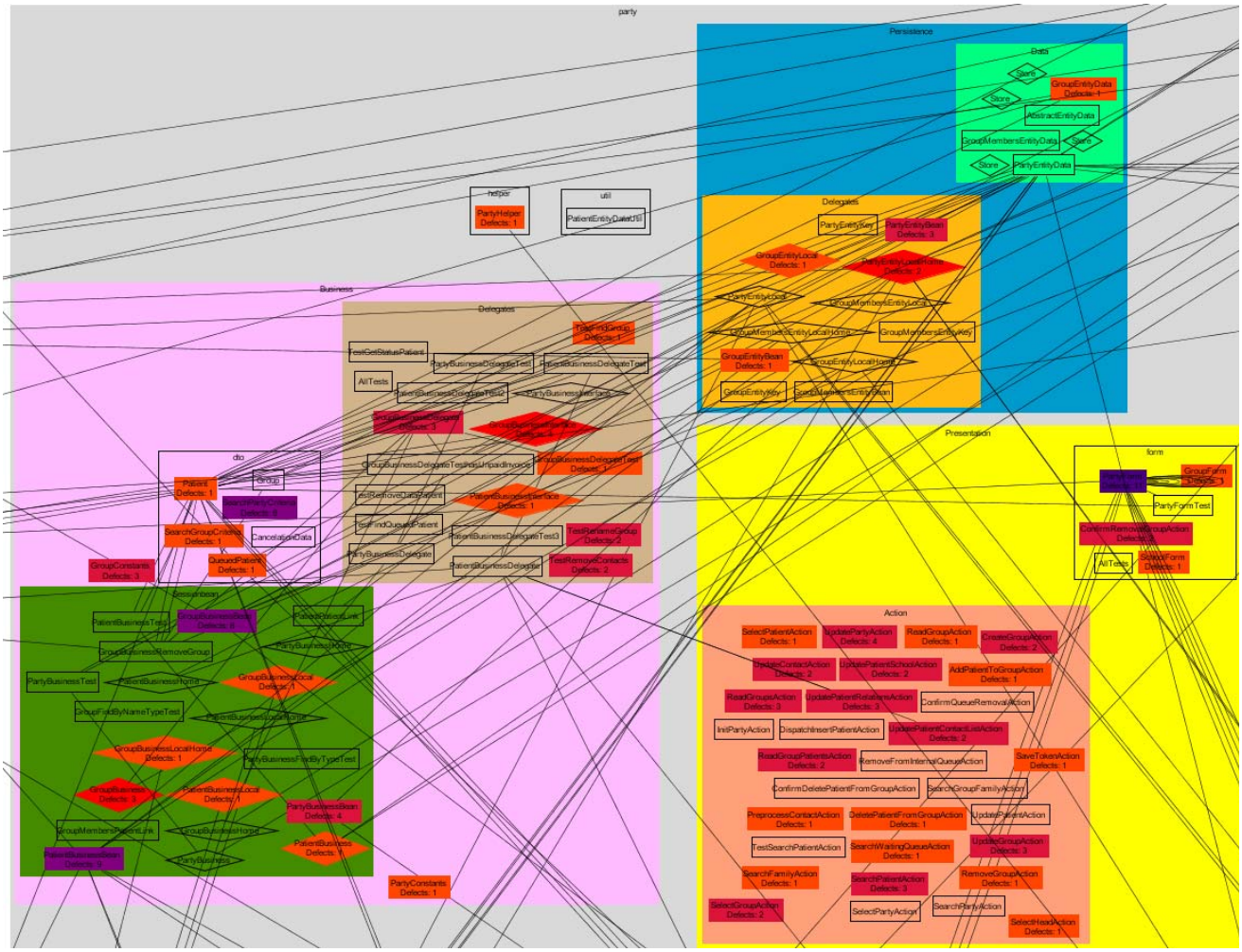
Figure 3 was obtained by having all parse settings at their default settings, except for the defects. In this case, defects of finding type *test* were selected. The image contains a lot of information. Let's first cover what's to be seen, and then we'll start looking into a deeper level of analysis.

Throughout the image grey blocks are visible. These correspond to (UML) packages at the base level. They will be referred to as base packages from now on.

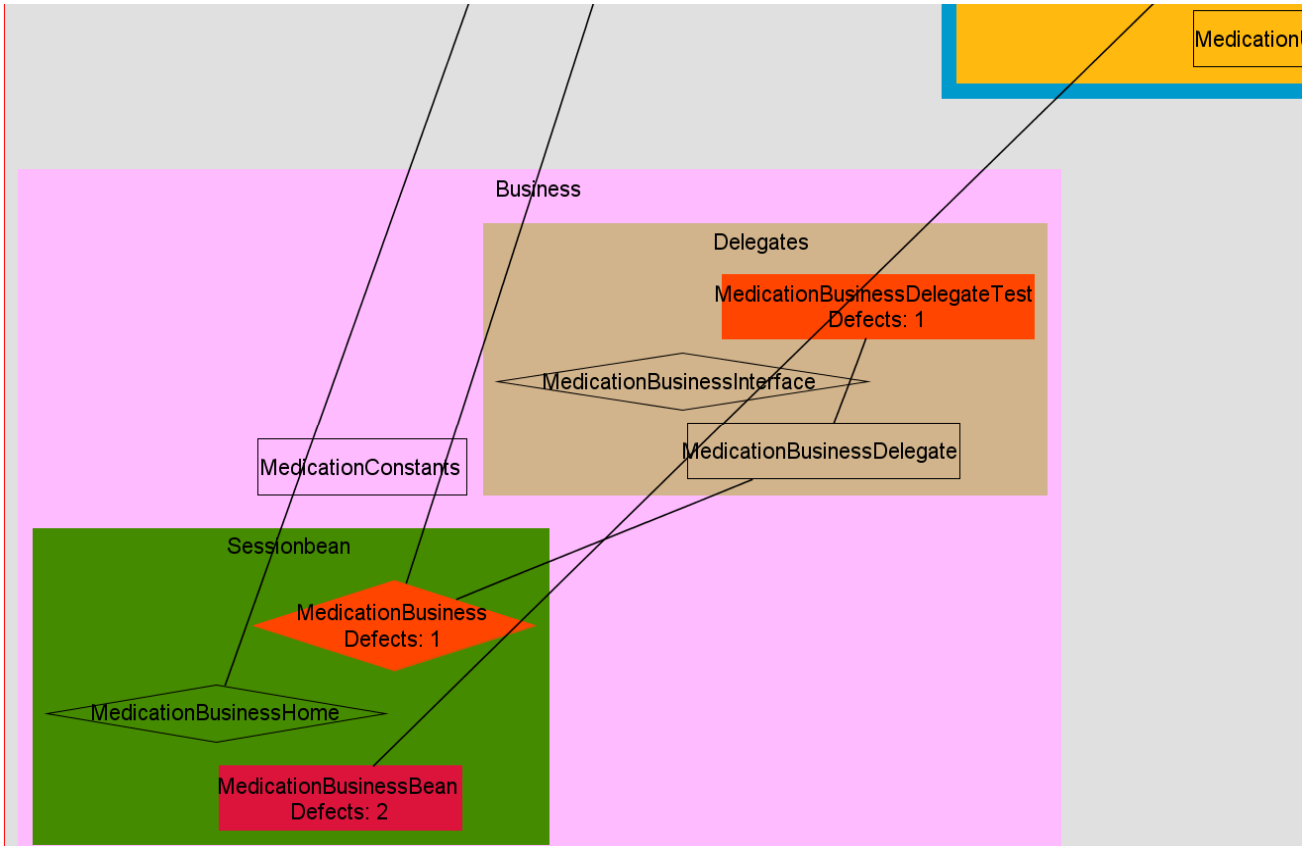
Within the base packages, more packages are visible (block shaped, most of them are coloured). Most of these packages have distinct colours, so that they can easily be distinguished. Some colours occur frequently, like yellow, blue and purple. We'll get to that later.

The small rectangular and diamond shapes with text inside packages correspond to respectively classes and interfaces, also referred to as nodes. *Figures 4* and *5* show a close up of a portion of *Figure 3*. By comparing *Figure 5* and *Figure 3* it becomes clear that this is a large project: there are over 1000 nodes in *Figure 3*!

The lines are associations, which for simplicity are all treated identical (no distinction between directions or multiplicity).



Top: Figure 4, a close-up of *Figure 3*



Bottom: Figure 5, an even closer look at *Figure 3*

The images show resemblance with UML class diagrams. Packages, nodes, interfaces and associations are quite similar to those as described in the UML standards.
The program was designed with layering in mind. Here, a three layer approach was used, from top to bottom: *presentation*, *business*, *persistence* (data), as can be seen in *Figure 6*:

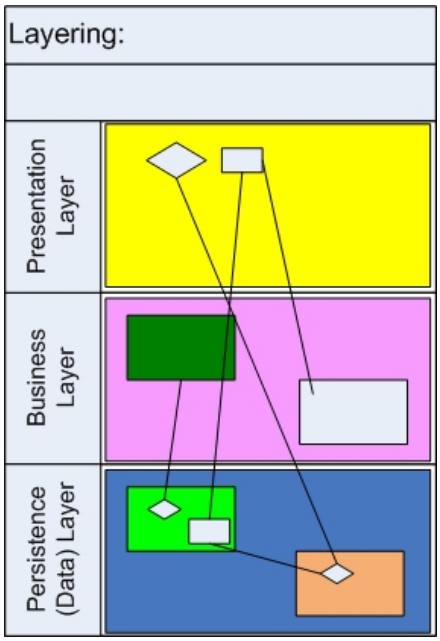


Figure 6: Layering within the program

Components (packages, classes, interfaces) can be placed within these "layer-packages" (coloured yellow, pink and blue) to improve maintainability. For instance, a patient database should be placed in the bottom layer, where the data resides, while the GUI should be placed in the top layer, which holds components with relation to presentation.
Similar packages (in terms of name/functionality) that occur frequently are given identical colours, so that the structure is more clear at a glance than without this property. Here's a small list of packages that have been given colours:
Every base package has a distinct grey colour (on a 256 colour grayscale each differ 5 colours).
The *presentation* package (yellow) and within this package there can exist package *action* (orange-red) and package *datasource* (light blue).
The *business* package (pink) contains package *sessionbean* (dark green) and package *delegates* (brown).
The package *persistence* (blue) holds packages *delegates* (orange) and *data* (light green).

In *Figure 5* the classes and interfaces are visible. Classes are rectangular shaped while interfaces are diamond shaped. By default, only the name and the amount of defects is shown, but more information can be viewed via a mouse-over or by enabling the option in the parser GUI. Classes and interfaces always reside inside packages. In this case, the class *MedicalBusinessBean* is inside the package *sessionbean* (coloured darkgreen), which is inside the package *business* (pink), which is inside the base package *medication* (note that the title of this package cannot be seen in *Figure 5*).

Classes and interfaces can contain defects. This is visualised by giving these nodes a colour, depending on the amount of defects. If there are no defect at all, the node is transparent (it will take the colour from its parent package). If there are defects, the node will be coloured somewhere between orange-red and deep purple according to this distribution:

Defect amount	Colour
1	orangered
2-4	crimson
5-9	purple
≥10	indigo

The amount of defects is also mentioned in the node (textual), but with a colour coding scheme like this it's easier to tell the severity of different defect nodes apart.

The results I will present on the next pages have been obtained by enabling defects of type "Test" only. It provides an overview of how many defects are contained in the grey "base" packages. The values from the tables originate from the *defects.csv* file, generated by the parser by keeping track of the amount of defects during the parsing process.

Let's find out if a class with a large amount of defects is likely to have a high number of associations:

Classname	Defect amount	Association amount
AppointmentForm	14	1
UserBusinessBean	14	6
MedicalDossierForm	11	11
PartyForm	11	17
DiaryBusinessBean	10	10
InvoicingBusinessBean	10	19

Lots of associations per node could cause defects, but the class *AppointmentForm* shows that this is not necessarily so. Let's see if there are nodes where it is the other way around: no associations, yet still defects. Or let's see if there are nodes without any defects and a lot of associations.

There are four classes with 20+ associations (*MedicalDataLoader*, *TopicsDTO*, *MedicalBusinessBean* and *PrintDossierDataSource*, they form a diamond in the center of Figure 3 and are easily spotted because of the high density of associations). They don't exhibit huge numbers of defects (ranging from 1 to 7). This is probably because all these classes load data from (many) other classes. They don't modify the data, so it isn't very likely that coupling errors occur here.

The node with the most associations yet keeping a clean sheet is *TestRemoveAttendencies* with 8 associations and 0 defects.

It turns out you can't draw conclusions based on only the numbers of associations and defects. You need to look at the nature of the classes in order to find out what function they serve. A class which modifies data of its associated classes is likely to have a higher defect amount than a class which only reads a parameter from another class.

It can be interesting to look at how the defects are distributed among their *finding types*. Here these results were obtained by selecting defects of finding type *Test* only.

Base Package	**Amount of defects**	**Amount of nodes that contain defects**	**Amount of total nodes within package**
Common	45	27	113
Diary	97	27	76
Financial	105	41	191
Maintenance	37	11	80
Medical	144	71	496
Medication	4	3	15
Party	117	52	106
Psychdossier	1	1	18
Registration	54	31	64
Task	2	2	25
Total	606	266	1184

It’s interesting to look at relative (normalised) data in this case. So here goes:

Base Package	**Percentage of affected nodes within package**	**Average amount of defects over the affected nodes**	**Average amount of defects over all nodes within package**
Common	24%	1,67	0,40
Diary	36%	3,59	1,28
Financial	22%	2,56	0,55
Maintenance	14%	3,36	0,46
Medical	14%	2,03	0,29
Medication	20%	1,33	0,27
Party	49%	2,25	1,10
Psychdossier	6%	1,00	0,06
Registration	48%	1,74	0,84
Task	8%	1,00	0,08
Total average	24%	2,05	0,53

Let's take a look at the defect statistics from a layer point of view:

Layer	**Defect amount**	**Defect nodes**	**Total nodes**	**Percentage of defect nodes on total**	**Average amount of defects per affected node**	**Average amount of defects per node over all nodes**
Presentation	342	148	409	43%	2,31	0,83
Business	197	78	255	31%	2,53	0,77
Persistence	64	39	514	8%	1,64	0,12

Up next we take a look at defects from a different finding type, *Review*.
The following tables correspond with the defects of type *Review* only:

Base Package	**Amount of defects**	**Amount of nodes that contain defects**	**Amount of total nodes within package**
Common	81	40	113
Diary	42	20	76
Financial	41	25	191
Maintenance	71	22	80
Medical	202	114	496
Medication	0	0	15
Party	88	39	106
Psychdossier	5	4	18
Registration	34	20	64
Task	2	2	25
Total	566	286	1184

Normalised data of the above graph:

Base Package	Percentage of affected nodes within package	Average amount of defects over the affected nodes	Average amount of defects over all nodes within package
Common	35%	2,03	0,72
Diary	26%	2,10	0,55
Financial	13%	1,64	0,21
Maintenance	28%	3,23	0,89
Medical	23%	1,77	0,41
Medication	0%	0,00	0,00
Party	37%	2,26	0,83
Psychdossier	22%	1,25	0,28
Registration	31%	1,70	0,53
Task	8%	1,00	0,08
Total average	22%	1,70	0,45

If the gathered data is put into a graph the similarities start to show:

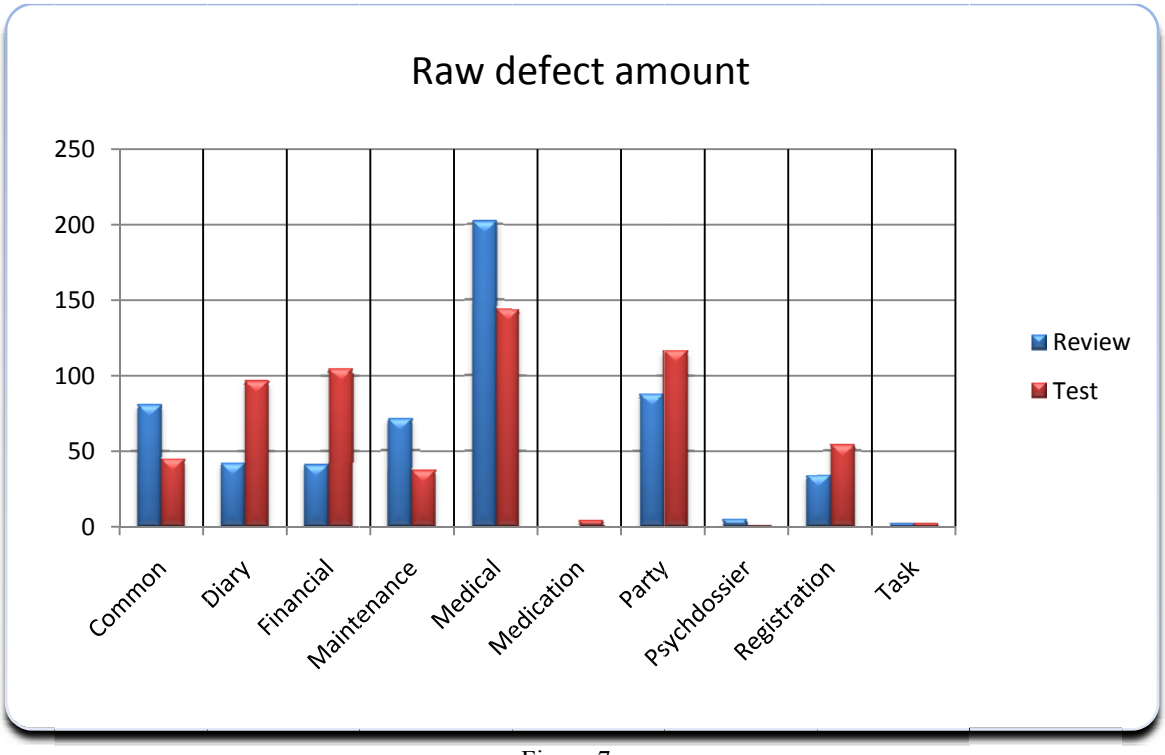


Figure 7

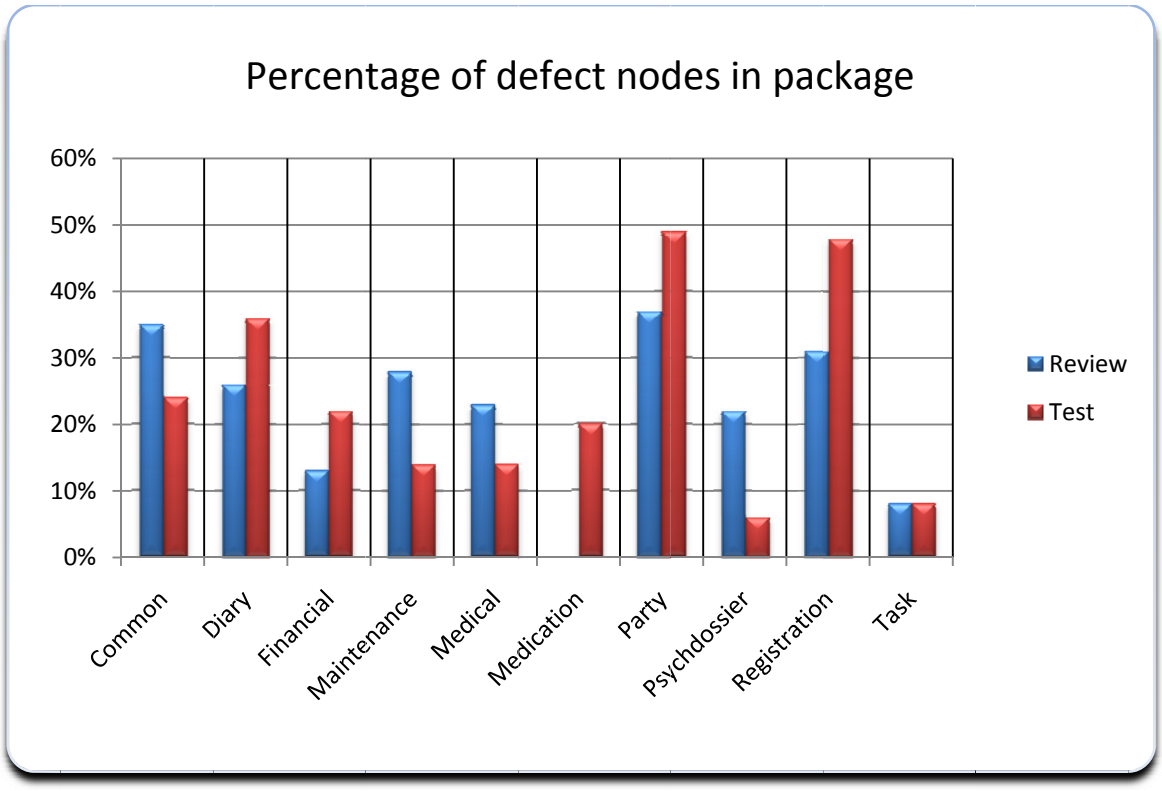


Figure 8

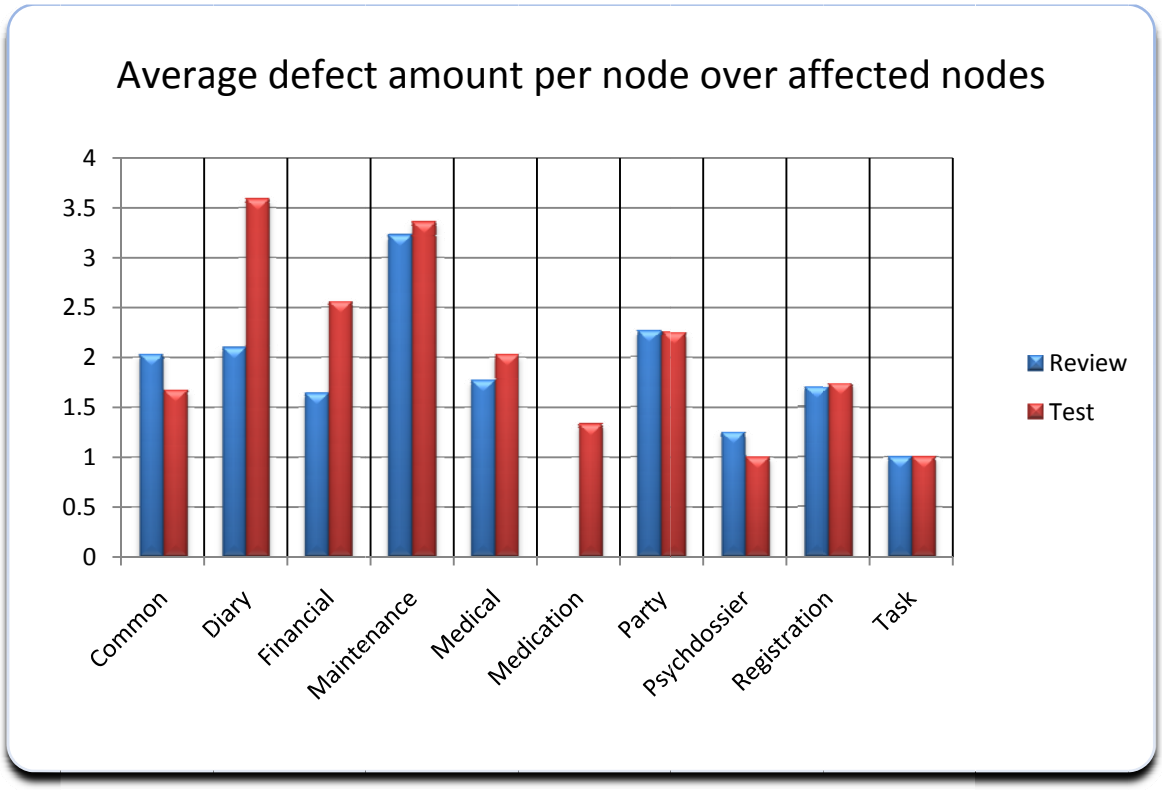


Figure 9

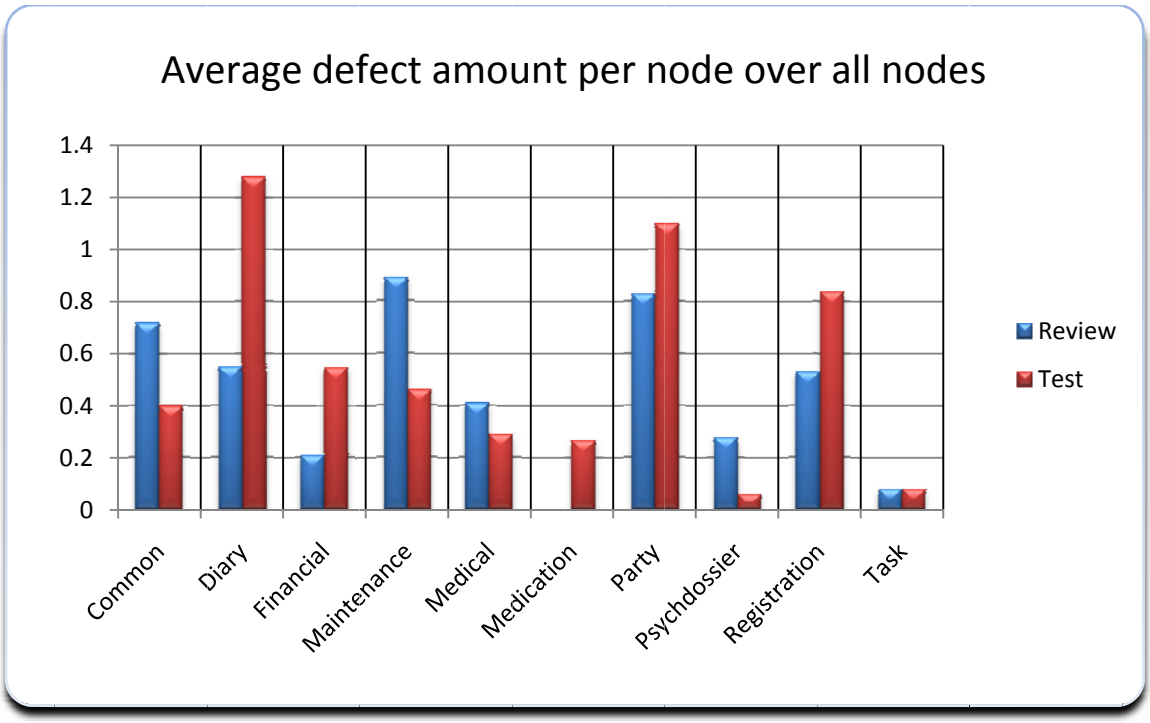


Figure 10

Figure 9 in particular shows striking similarities.
It looks like there is a relation between the two finding types *review* and *test*. This means that there is a relation between deviating from the original UML model while coding and the amount of defects in the final program.

The base package *Medical* holds the most defects of the project. The class *MedicalBusinessBean* has 7 defects of type *Test* and 12 defects of type *Review*. Let's take a look at the comments from the defect database on this particular class, to find out what's wrong with it.

PARTS00003683	Test	Not implemented according to design
PARTS00004353	Test	Error is not thrown (try & catch)
PARTS00004355	Test	Function parameters are literals instead of constants
PARTS00004402	Review	Over 650 violations by calling function from other class
PARTS00004656	Review	No optimistic locking
PARTS00004837	Review	Code lookup happens multiple times, should be only once
PARTS00005172	Test	Link not working
PARTS00005305	Review	Some values should be combined
PARTS00005306	Review	Function loads incorrect values

The first comment states that the coders deviated from the original model. If this is done with care it wouldn't necessarily lead to other defects, but unfortunately that's not the case here, judging by some of the comments.

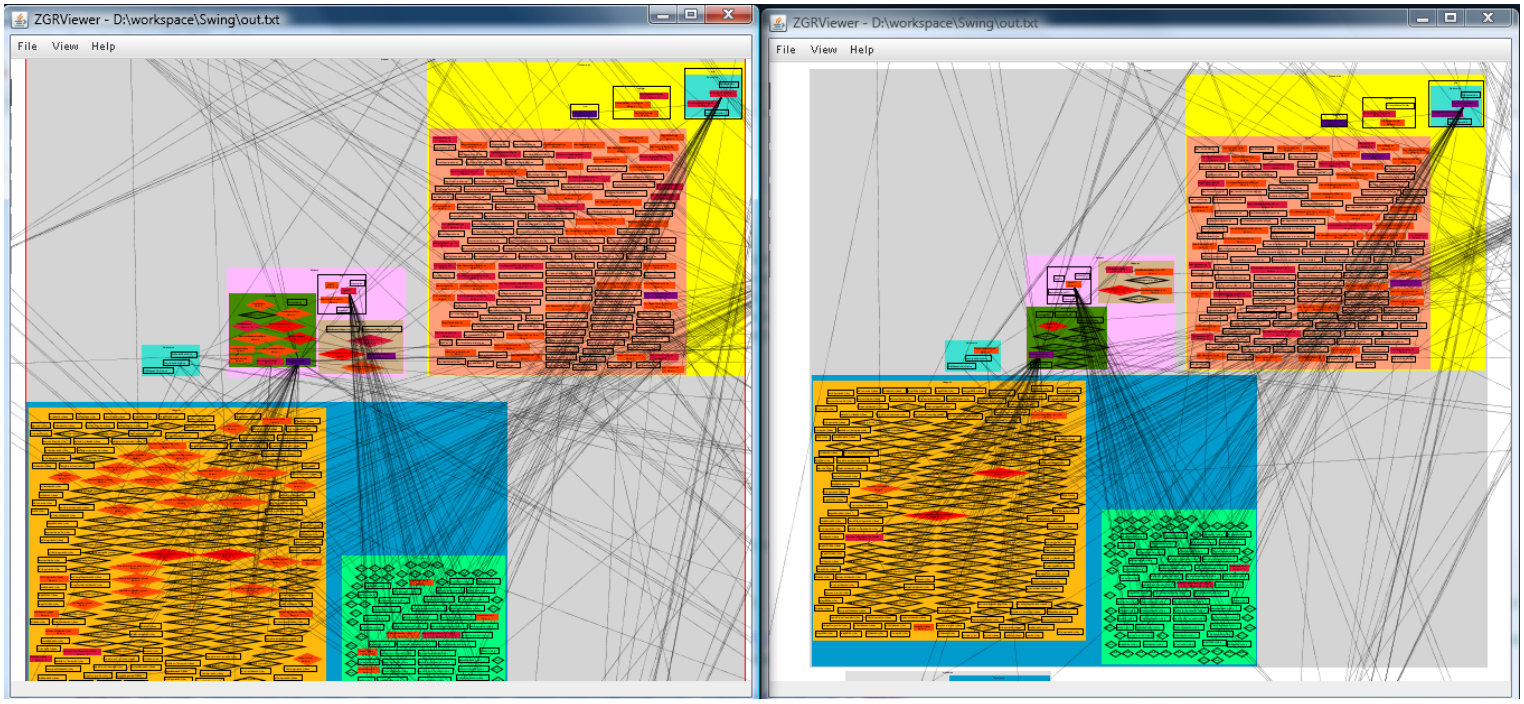


Figure 11: Base package *medical* with *Review* defects (left) and *Test* defects (right)

Figure 11 shows a side by side view of the generated images using different finding types, in an attempt to show similarity in the affected nodes. It turns out there is some similarity, but it isn't as similar as I was hoping to see, judging from the graphs in the previous pages. Compared to the yellow and blue packages, the pink packages show the least similar behaviour.

Comparing images is tough because GraphViz renders each input differently. The locations of packages and nodes are roughly the same, but not entirely. This is odd since the only thing that's changing is the colour of the nodes, not their location. GraphViz has more funny phenomena, which I'll discuss in the next section.

Issues

GraphViz has some issues/bugs. Positioning doesn't work under FDP rendering. It only appears to work when associations are drawn between these nodes. Also, the syntax isn't completely clear. I've emailed Emden Gassner [10], who is in charge of graph layout issues, about this on Friday June 11th, 2008. I'm awaiting response at the moment. If there appears to be a fix for this I still might implement this into the parser. It will be used to position certain sub-graphs (clusters) within other sub-graphs. E.g. inside the large greyed financial cluster, the action (yellow), business (purple) and persistence (greenish) should be aligned to show hierarchy. This should help understanding the structure of the program more easily. However, this is not possible at the moment. A (rather dirty) solution would be to connect every node with the other, but then the lines would have to be invisible. This would solve the positioning problem, but incorporates another: the "overlap = false" statement would become unusable due to the extreme number of interconnections in the graph.

Update: got a response from Emden (see Appendix). He says it's possible to 'steer' the nodes into the right direction with small graphs, but when dealing with larger graphs the algorithm simply ignores the positioning parameters. It is possible however to manually override this, but then you have to do that for every single node in the graph, which isn't feasible (we're talking 1000+ nodes in the complete model).

It would be nice if we could get a visual representation of the layering used throughout the program. Nearly all base-packages (coloured grey) contain the layers *persistence*, *business*, and *presentation*. These layers correspond to how the program is designed in an abstract way (data layer at the bottom, presentation layer at the top, business layer in between). This is visualised in *Figure 6*.
This is only possible if we don't group the packages into clusters, because GraphViz doesn't support overlapping clusters (one node cannot be in more than one package). We can give up on package-clustering in favour of layer-clustering, but then the resulting image would be three layers high, but immensely wide. This will not improve the comprehensibility of the image.

Future work
In theory, the parser works for any XMI model, given that it shares the same structure of this project. This means that the information must be *inside* of the tags, rather than *in between* them. For example the parser will correctly parse `<xmi.id="1"></xmi.id>` but fails at `<xmi.id>1</xmi.id>`. The reason for this is that in the first case you'll need to extract the information as an attribute from the function *startElement* while in the latter case the information has to be extracted through the function *characters*.
This problem could be solved by checking first with what kind of file we're dealing with, thus determining which functions have to be used to extract which information.

The colours have been added manually (manually triggered that is), so when a different project with different names is parsed, all packages will be transparent. This problem can be overcome by first scanning the file for names of packages that occur frequently and then assign colours to those names. Then parse the project using this information. That still doesn't solve everything, because when there are many colours it could very well happen that packages coloured pink, light pink, light purple and dark pink are placed next to each other (decreasing distinguishability). A solution to this might be an implementation of a heuristic for the NP-complete graph-colouring problem [11].

Conclusion

The main aim of the project was the visualisation, which worked out pretty well. The images show the architecture of the project with use of colours and shapes. It would be nice to enforce a layout in which the layering is taken into account, but with GraphViz this remains impossible. ZGRViewer is a nice tool with which the *svg* images can be viewed quickly. The great thing is that you can zoom all the way in to class level without the loss of image quality, since the image is made out of vectors.

Besides visualisation I wanted to do some analysis, mainly on defects and coupling, as these two subjects appealed to me. Analysis proved to be difficult since there are a lot of factors you must take into account. The amount of defects itself doesn't say much, you really have to look in the defect database to get an insight of what's going on. This database is quite large, so I was only able to look at a couple of specific classes.

Along this way I started to make assumptions, of which some turned out to be true and some turned out to be false. Either way it was educational to take a look at a large software project, which, as it turns out, doesn't come error-free.

References

[1] UML: http://www.uml.org/
[2] Java: http://www.sun.com/java/
[3] Bachelorproject at LIACS: http://www.liacs.nl/edu/bachelor.html
[4] Lethbridge/Laganiere: Object-Oriented Software Engineering
[5] XMI: http://en.wikipedia.org/wiki/XML_Metadata_Interchange
[6] GraphViz: http://www.graphviz.org/
[7] Java DOM: http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html
[8] Java SAX: http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/package-summary.html
[9] ZGRViewer: http://zvtm.sourceforge.net/zgrviewer.html
[10] Emden Gassner: http://www.research.att.com/viewPage.cfm?PageID=431
[11] Graph colouring problem: http://en.wikipedia.org/wiki/Graph_coloring

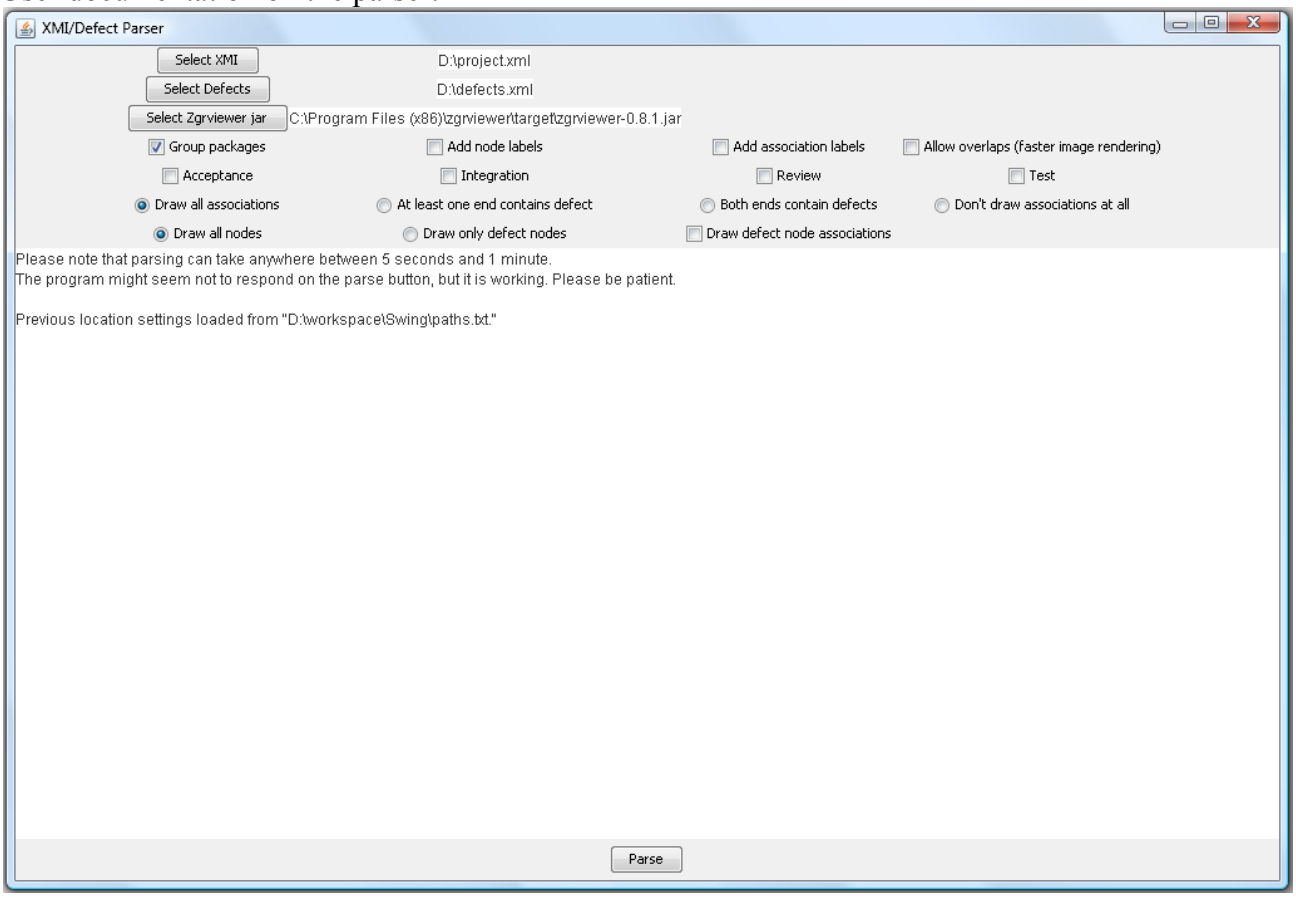
Appendix

Subject: Re: GraphViz FDP pos rendering
From: "Emden R. Gansner" erg@research.att.com
Date: Fri, July 11, 2008 6:10 pm
To: "Stefan Wink" <swink@liacs.nl>

```
Stefan Wink wrote:
> Hello Emden,
>
> I'm having some trouble with node positioning using FDP/Neato.
>
> When I try:
> graph G {
>         a [pos="0,0"];
>         b [pos="0,5"];
>         c [pos="0,10"];
> }
>
> the nodes a, b and c will appear on the screen, but they're completely
> ignoring the pos attributes.
>
> When I try:
> graph G {
>         a [pos="0,0"];
>         b [pos="0,5"];
>         c [pos="0,10"];
>         a -- b -- c;
> }
>
> the nodes a, b and c do are positioned in the way I want them to.
>
> When I try:
> graph G {
>         a [pos="0,0"];
>         b [pos="0,5"];
>         c [pos="0,10"];
>         a -- b;
> }
>
> the nodes a and b are positioned according to the pos attribute, but node
> c is then placed somewhere else.
>
>
>
> Why does there have to be a link between nodes in order for them to be
> positioned accordingly? Am I missing something here?
>
>
Are you using neato for graph layout or rendering? If the former, the
algorithm will start with the positions
you give it, but it is extremely unlikely that these positions will be
used in the final layout unless you pin
them down. In the cases above where you say they are positioned as you
want them, they are no longer
where you put them originally, although they lie on a vertical line. It
just happens that, starting with your
initial positions, the algorithm doesn't have to change the x
coordinate, but just needs to adjust the y
coordinate. If you had started instead with

graph G {
    a [pos="0,0"];
    b [pos="0,5"];
    c [pos="0,10"];
    d [pos="10,10"];
    a -- b -- c -- d
}

you'd find the nodes curving up and to the right from a to d.

The effect you noticed, where it appears that nodes need to be connected
```

```
to get the layout you want,
happens because neato doesn't handle unconnected graphs. It lays out
each component separately, and
then combines them into a single layout by packing. See

   http://www.graphviz.org/doc/info/attrs.html#d:pack
   http://www.graphviz.org/doc/info/attrs.html#d:packmode

Again, it is largely just an accident that the connected component
appears to look the way you expect it to.

If you really just want to render  your graph, using exactly the
positions you supply, use

    neato -n

This tells neato not to do any layout, but just use the graph positions
as given in points.

     -- Emden
```

User documentation on the parser:



We'll run through the program top-down, as that the way the fields should be filled in. On top there are three buttons, "*Select XMI*", "*Select Defects*" and "*Select ZGRViewer jar"*. Click on these to select the corresponding files. *Select XMI* should point to the source XMI file where the program is modelled (in packages, classes, interfaces and associations). *Select Defects* should point to a (if available, else this will be ignored) defects file which contains information on which classes/interfaces contain defects. *Select ZGRViewer jar* should point to the location of the ZGRViewer jar executable, which by default resides in the C:\Program Files\ directory (on Windows machines, that is). This information only has to be entered once, as it will remain stored by the parser.

Explanation on the check boxes (classes and interfaces together will be called nodes from now on):
Group Packages: check this box if you want to view the package structure of the model. Nodes will be placed in their parent package. If unchecked, all nodes can appear randomly on the screen.

Add node labels: By default only the node name is displayed. By checking this more info on the node is shown. This leads to a more crowded image.

Add association labels: Adds labels to the associations (drawn as lines).

Allow overlaps: The parser tries to obtain an image without overlapping lines/nodes. However, in the case of very large graphs the calculation might take very long. It is then wiser to check this box and bypass this neatness in exchange for faster calculation.

Acceptance/Integration/Review/Test: When checked, defects of these finding types will be shown as coloured nodes.

Draw all associations/At least one end contains defect/Both ends contain defects/Don't draw associations at all: Here you can decide which associations (lines between nodes) should be drawn. If you're only interested in how nodes are grouped for example, your best bet is to draw no associations at all, since this will lead to a more neat image.

Draw all nodes/Draw only defect nodes: Selects which nodes to be drawn.

Draw defect node associations: Within the defects file is stated which nodes are affected by which defect. This option links those defects to a new node, which represents the defect.

Hit the *Parse* button to show the image.

Anomaly: If you select *Draw only defect nodes* and *Draw all associations* together, you'll get a bunch of seemingly random nodes bounding the graph. This is because the program wants to draw associations between non-defect nodes (which will not be drawn).

ZGRViewer has to be set up first in order to work correctly. It uses references to executable files from GraphViz, but these have to be set manually.
Start ZGRViewer by running (%zgrviewer dir%)\target\zgrviewer-0.8.1.jar.
In the *View->Preferences->Directories* window, set the *GraphViz/neato executable* to the location of *FDP.exe* in your GraphViz installation directory, for instance C:\Program Files\GraphViz\bin\fdp.exe. Also set a *tmp* directory, ZGRViewer will not render without one. Save these settings.

So, in summary, you need:
GraphViz installed.
ZGRViewer installed with the reference to FDP.exe from the GraphViz folder.
Set the locations of

1) The xmi file
2) The defects file
3) ZGRViewer's jar executable

in the parser.

Once this is done, you can fire up the parser anytime without ever having to do all this again.