Depiction of Additional Node-related Elements in Graph-based Software Visualisations

Yijun Yu
Michel Wermelinger
Haruhiko Kaiya
Prof Bashar Nuseibeh

19th December 2008

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk
Depiction of Additional Node-related Elements in Graph-based Software Visualisations

Yijun Yu\textsuperscript{1} Michel Wermelinger\textsuperscript{1} Haruhiko Kaiya\textsuperscript{2} Bashar Nuseibeh\textsuperscript{1}

\textsuperscript{1}Department of Computing, The Open University, UK
\textsuperscript{2}Department of Information Engineering, Shinshu University, Japan

Abstract

Many of the ways to depict software are based on graphs, although what nodes and edges represent differ from visualisation to visualisation. In this paper we present a light-weight approach to enrich graph-based visualisations so that nodes can represent more information. The idea is to show in each node a rectangle of pixels, each representing a certain element associated to the node, and the colour of each pixel representing up to three attributes of that element. The order of the pixels is user defined and may convey additional information. The approach is generic and allows data obtained through completely different reverse engineering processes to be shown together in a compact way that preserves the meaning of the graph layout. We illustrate our approach by showing how software architecture and defects can be related: a graph depicting the high-level components and their dependencies is enriched with information about the bugs reported for each component.

1. Introduction

Software systems are complex entities comprising several artefacts (requirements, source code, test cases, etc.), each one typically consisting of many elements (like variables, functions, and modules in the case of source code). Such a heterogeneous and large collection of elements entails a rich collection of relationships between them, both within the same artefact (e.g. call relations between functions) and across artefacts (e.g. traceability relations between requirements and code).

Brooks has argued that, due to its nature, “software has no reality in space” and hence “it is invisible and unvisualizable” \cite{6}. In particular, he argues that the complexity of software structures leads to superimposed graphs that are neither planar nor hierarchical. This makes it very difficult to capture the essential difficulties of software engineering in a visual way that could help comprehension and communication.

In spite of this ‘warning’ more than 20 years ago, graph-based representations have been very popular in the literature and in tools. They usually just concentrate on particular kinds of elements, relationships and stakeholders (typically the developers). Due to their generic nature, graphs are very flexible: nodes and arcs have been used to represent a vast array of elements and relations, respectively. A typical example is control-flow graphs, where nodes represent some executable entity and arcs represent the order of execution.

Graph-based representations of software structures can quickly become complex and large and the graph’s layout can make or break its understandability. In fact, the layout may have a semantic meaning, which disallows arbitrary changes to the layout. For example, CCVisu is a tool for force-directed layout of graphs, in which various energy models can be used to determine the distance between nodes, thereby showing clusters of elements that are strongly related \cite{2}. For example, if nodes represent source code files, distance may be used to represent co-change: the closer together the nodes are, the more often the corresponding files were changed together. When such visualisations are used, it is advisable to keep the exact graph layout if one wants to view the evolution history through animations \cite{3}. Graphs representing software architecture are another example where the layout may convey a particular meaning, e.g. it may show certain architectural patterns like layers.

The different kinds of information that can be represented in a single graph are often relatively constrained, in order to keep the graph fit for purpose. Using various types of nodes and edges (e.g. through different shapes and icons), or heavy labelling of nodes and edges, in order to represent different kinds of
software elements and relations, can quickly render a graph difficult to understand.

One common approach to fit more information into a graph without resorting to different node shapes and additional labels, and without changing the layout, is to colour the nodes in order to represent an additional attribute of the corresponding elements. Each different colour (or different shade of the same colour) represents a different value of the attribute. In the CCVisu example cited earlier, nodes represent files and colours are used to show to which subsystems the files belong. If nodes closely clustered together have different colours, it means that co-changes span across subsystems, which can point to a poor modularisation of the system.

In this paper we present a generalisation of using colour to represent an additional attribute of the elements in a graph. First, we allow representing up to 3 attributes with a single colour, by mapping each attribute to one of the 3 dimensions of the hue-saturation-brightness (HSB) colour model. Second, instead of using colour just for the node’s background, we insert a coloured bitmap into each node, each pixel of the bitmap representing one element (potentially of another artefact) associated to the element of the graph.

We illustrate our approach with a graph representing the architecture of the Eclipse SDK, where each node represents a component. Each node includes a bitmap with one pixel for each bug associated to that component in Eclipse’s Bugzilla database. The colour of each pixel represents three attributes of that bug: its severity, priority and status. The various values for those three attributes are mapped to various values for hue, saturation and brightness.

In the next section we present just the necessary details about Eclipse, Bugzilla and the HSB model, in order to make the paper self-contained. In section 3 we explain how we obtained the visualisations presented and discussed in section 4. Those visualisations are intended to show the flexibility of the approach. It is not this paper’s purpose to prescribe any particular way of relating bugs to software architecture. Section 5 relates this work to existing approaches, and the last section presents concluding remarks.

2. The case study and the colour model

2.1 Eclipse

The development of Eclipse is a vast undertaking, and it has been divided into different projects, like the Java Development Toolkit (JDT), the Plugin Development Environment (PDE) or the core Platform. Each project is composed of some coarse-level components like the user interface. Each component is made of sub-components called plugins.

Each plugin provides zero or more extension points. These can be used at run-time by other plugins in order to extend the functionality of Eclipse. Typical examples are the extension points provided by the ui plugin: they allow other plugins to add at runtime new GUI elements (menu bars, buttons, etc.). It is also possible for a plugin to use the extension points provided by itself. Again, the ui plugin is an example thereof: it uses its own extension points to add the default menus and buttons to Eclipse’s GUI.

In the remaining of the paper, we say that plugin X statically depends on plugin Y if the compilation of X requires Java classes that belong to Y, and we say that X dynamically depends on Y if X uses at run-time an extension point that Y provides. Note that the dynamic dependencies are at the architectural level; they do not capture run-time calls between objects.

2.2 Bugzilla

Bugzilla is a widely used open-source system for tracking bugs during software development and maintenance. It provides a web interface for users and developers to report bugs.

Each bug has a unique identification number, assigned consecutively. In order words, the order of the id numbers indicates the chronological order of bug reporting. We will use this fact later on.

Each bug is also given a priority, which indicates how quickly the bug should be fixed, and a severity, which indicates how serious the bug is. The priority ranges from 1 (highest) to 5, while severity ranges over the following seven values, from high to low severity: Blocker blocks development and/or testing work, Critical crashes, loss of data, severe memory leak, Major major loss of function, Normal normal, Minor minor loss of function, Trivial cosmetic problem like misspelled words.
Enhancement request for enhancement

Each bug report also has a status field, which indicates the current state of the bug. A bug’s lifecycle goes through the following seven states:

- **Unconfirmed**: The bug has been added to the database but it is not yet known whether it really is a bug.
- **New**: The bug has been confirmed, and added to the assignee’s list of bugs and must be processed.
- **Assigned**: This bug is not yet resolved, but is assigned to the proper person.
- **Resolved**: A resolution has been taken, and it is awaiting verification by QA.
- **Verified**: QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken.
- **Closed**: The bug is considered dead, the resolution is correct.
- **Re-opened**: This bug was once resolved, but the resolution was deemed incorrect.

Each bug report also includes a description of the bug and other information, but in this paper we only use the severity, priority and status, which are explained in more detail at [https://bugs.eclipse.org/bugs/page.cgi?id=fields.html](https://bugs.eclipse.org/bugs/page.cgi?id=fields.html).

In the case of Eclipse, each bug report also indicates in which component the bug is believed to be and we use this information to assign bugs to components in the architectural graph (see Section 3).

### 2.3 The HSB colour model

The HSB model represents a range of colours using three dimensions.

- **Hue**: represents what common language designates as red, blue, etc. The set of hues is represented by a colour wheel, and therefore values range from 0 to 360 degrees.

- **Saturation**: indicates how strong a colour is. Values range from 0 to 1, with zero indicating absence of colour. In that case the result will be a shade of gray, independently of the hue value.

- **Brightness**: values range from 0 to 1, with zero representing the absence of light. In that case the result is always black, independently of the values of the other two dimensions.

The HSB model can be represented as a cone, as seen in Figure 1. Saturation increases from the centre to the border, and brightness from the bottom to the top. If we were to vertically halve the cone, we would see the various shades of grey in the centre (saturation = 0), from black in the bottom (brightness = 0) to white in the top (brightness = 1).

![Figure 1. The HSB colour model](https://example.com/hsb-figure.png)

We will use the fact that the three dimensions are not independent.

### 3. Data collection and processing

Eclipse keeps information about its architectural elements and relations in XML and text metadata files, saving us from having to delve into source code. We wrote bash, AWK and XSLT scripts that read those metadata files, extract the relevant information, and produce text files which encode the relations, such as **required** build dependencies and **provided/used** extension points (Section 2.1), in generalised Rigi Standard Format (RSF) [14]. We use the relational calculator Crocopat [4] to compute derived relations. For example, from the **used** and **provided** relations between plugins and extension points, a Crocopat script computes the dynamic dependency relation among plugins. More details about this process and a discussion of its advantages can be found in our previous work about the evolution of Eclipse’s architecture [20, 21].

The bugs reported for Eclipse are not classified by plugin names but by the name of an Eclipse project, each project corresponding to a set of plugins as documented in the project’s *developer resource* page. Using those documents, we manually added a binary relation **contain(component, plugin)**, describing which plugins belonged to which Eclipse component (project) and then wrote a Crocopat script that lifted the static and dynamic dependency relations from plugins to components, i.e. a component X is said to statically (resp. dynamically) depend on component Y if at least one of X’s plugins statically (resp. dynamically)
depends on one of Y’s plugins. The resulting component dependency relations are translated by a further script into a text file describing a graph to be laid out by the Graphviz graph drawing tool [7]. The result is shown in Section 4; the figures of Section 4.1 use the dynamic dependency graph, while those in Section 4.2 use the static dependency graph. We agree that the layout can be improved, but that is besides the point of this work, which aims to add more information to given graphs, no matter how they are laid out.

We also wrote scripts to extract from Eclipse’s Bugzilla database1 the id number, severity, priority, status and the component associated to each bug. Note that we extracted this info from a single Bugzilla database and therefore do not have the history of each bug, i.e. we only have the current values for the status and other attributes. We do not know when and how they were changed in the past, but that does not matter for our purposes.

Finally, we changed the scripts that generate the Graphviz file from the RSF file with the component dependency relations, in order to also show information about the bugs associated to each component as customised shapes.

The shapes are generated by a Java program we wrote. In this paper, shapes are squares in which each pixel represents a bug. The square is filled with these pixels from left to right, and from top to bottom. The square is padded at the end with white pixels, if necessary.

Our scripts are quite flexible and take as input a text file in which the user can state the mapping between attributes and colour dimensions. For attributes with discrete values, as is the case for severity, priority and status, the user can also assign a dimension value to each attribute value. It is possible to just visualise a subset of the attributes: the unused colour dimensions will be set at their maximal value. Last but not least, the user can state which attribute should be used to order all elements associated to a node. The sort attribute does not have to be one of the visualised attributes. For our example, we have only one additional attribute besides the three that can be visualised: the bug’s id number, which can be used to sort bugs by chronological order.

We will see in the next section how all this flexibility can be put to use.

4. Results and Analysis

Due to space constraints, we can only present some of the graphs we are able to generate, and we had to keep the size of the figures quite small. However, graphs like these are not supposed to be seen on paper, printed in black and white, but on large displays. All figures presented in this section can be downloaded from http://mcs.open.ac.uk/yy66/wcre08.

We present the graphs in increasing order of complexity, first the graphs that visualise only a single bug attribute for a subset of bugs, then graphs that show all 3 attributes for all the 207743 correctly formatted bug reports in our data set. Our scripts could not automatically process the 8.75% of badly formatted reports in the database.

4.1 Single attribute, last release

We first tried out our approach using only a subset of the data, namely the 543 bug reports for release 3.3.1, the last Eclipse release included in the database. We decided to visualise two attributes of the bugs, their status and their severity, and we ordered the bugs both by the attribute in question and by date. These four combinations were chosen in order to address the following research questions:

• What is the status of bugs: are most bugs about this release still unconfirmed, have some already been closed?
• What is the severity of bugs for a mature release? Are there still many critical bugs or are most of them enhancement requests?
• Are bugs handled mostly in the order they are reported, i.e. are earlier bugs in a more advanced status?
• Are more severe bugs usually reported first?

Given that the bug report data is superimposed on the architectural graph, we further wished to see if:

• the number and severity of bugs relates to the type of component;
• the number and status of bugs relates to the layer the component belongs to.

An example of a possible relation between layer and status is that so called responsible components [13], i.e. those that are used by many other

---

1 The database we used is available as a single file (http://pag.csail.mit.edu/msr_challenge2008/eclipse-bugs-000001-2T3000.zip), part of the benchmark data for the Mining Software Repository 2008 Challenge.
components, have substantially fewer bugs because most of them were found before the release shipped.

We can now proceed to the visualisations. The small number of data points would lead to small bitmaps. To make them larger we have used 4 by 4 pixels for each bug. Figures 2 and 3 use hue to show the status of the bugs reported for release 3.3.1, ordered by status and by date, respectively.

![Figure 2. Status of bugs in last release, ordered by status, shown in hues](image)

Ordering by status means that all bugs with the same status form consecutive lines of pixels of the same colour, making it easier to see the relative number of bugs for each status. In Figure 2, one can see large portions of NEW (orange) and RESOLVED (light blue) bugs and fewer bugs that are ASSIGNED (green), VERIFIED (dark blue), and REOPENED (dark pink). There are no UNCONFIRMED or CLOSED bugs. This reflects the fact that the bugs of the last release are relatively new in the software process.

![Figure 3. Status of bugs in last release, ordered by date (same hues as in Figure 2)](image)

The size of the bitmaps shows the number of bugs related to the components. As the components are of non-uniform size (ranging from 1 to 20 plugins), the ones with more bugs are not necessarily more problematic. From the graphs, we can hardly see any relation between the number of bugs and the layer, or the responsibility, of a component, although some low-level components do have fewer bugs than the high-level ones. On the other hand, the distributions of status within individual components do reveal that almost all VERIFIED bugs are in the UI and Core components, probably because they are more important for user-satisfaction than other components. However, the percentage of unresolved bugs (i.e. NEW or ASSIGNED) varies greatly among components (even with the three UI components) and does not seem to be related to layers or responsibility.

Ordering by date means it is possible to see how early bugs are different from those reported later. Figure 3 shows that the early bugs reported for release 3.3.1 of one of the UI components are actually reopened bugs from previous releases. One can also see that in the Update component a systematic process has been followed, with early bugs already resolved and later bugs still in the new state, whereas in other components there is no relation between the bug’s current state and the time it was reported.

In both figures we assigned status to hue, thereby having saturation and brightness at their maximal levels. The legend in Figure 2 shows how status values were mapped to colours. The rationale was to use ‘hot’ reddish hues for those reports that haven’t been assigned to anyone yet, green for those being handled, and different ‘cool’ hues of blue for those that are in the closing down phase. Furthermore, we made an analogy between the hue circle and the ‘cycle of life’ of
a bug: unconfirmed and reopened states are mapped to
nearby hues in the colour circle.

Figures 4 and 5 show the severity of bugs for the
same release, again ordered by value and by date,
respectively. This time we map the attribute to
brightness instead of hue: people with colour
disabilities might find brightness easier to distinguish
than hue. The maximal hue value (360°) corresponds to
red. Hence, the highest severity bugs are depicted in
bright saturated red, while the lowest severity bugs are
depicted in black (zero brightness).

It is apparent that, in percentual terms, there are few
blocking bugs and enhancement requests in the last
release, which may be due in part to Eclipse’s maturity.
Figure 5 shows that at the beginning almost no trivial
bugs or enhancements are reported; they appear at later
stages.

4.2 All attributes, all bugs

When mapping 3 different attribute values to a
single colour, the assignment of attributes to colour
dimensions plays an important role. In Figures 6 and 7
we contrast two different assignments, but in both cases
the order of pixels is by colour. Since the colour
ordering first uses hue, then saturation and finally
brightness, all of them in inverse order from high to
low, a sequence of greyish pixels (potentially from
white to black, depending on the actual underlying
values in the data set) marks the end of a initially
saturated hue.

In Figure 6, we mapped status to hue, priority to
saturation and severity to brightness. In this way, we
are basically stating that we do not care about
enhancements: they all will be represented as black, no
matter their priority or current status. With this
assignment and the colour ordering, within each status
the bugs will progress from high priority (i.e. the hue
becomes less saturated), and within each priority the
bugs are ordered from critical to enhancements (i.e. the
pixels become darker).

However, it is immediately apparent from the figure
that no strong colours are visible. In other words, the
saturation in Figure 6 is relatively uniform: most bug
reports have a medium priority. This probably points to
a difficulty in bug submitters to have a precise idea
about which priority a bug should be given. The
priority attribute is probably not being used in its most
effective way by Bugzilla users and developers to
discriminate which bugs should be handled first.

The black stripes clearly show the status of the
enhancements: in most components, most
enhancements have already been verified because the stripe comes before the purple hue, while in the SWT component, the enhancements are more evenly distributed between assigned and resolved bugs.

Finally, since the major ordering is by hue, it is easy to see the percentage of bugs in each state. For example, in the Core, Debug and Ant components, about half of the bugs are already closed, a much larger percentage than in the other components, which could be a sign that those components are rather important and require bugs to be solved quickly. We can also see that, over all releases of Eclipse, the percentage of unconfirmed or new bugs is negligible, but there are sufficient reopened bugs to be visible in the nodes.

In Figure 7 we assign status to brightness and severity to hue, keeping priority assigned to saturation. Note that the flexibility of assigning a colour dimension value to each attribute value made it possible to change the order of values: reopened and unconfirmed bugs were assigned to the top of the brightness scale and closed bugs to the bottom. In this way, we are stating that we don’t care about already closed bugs, no matter what their severity or priority was: they are all depicted as black. As for hue to represent severity, we mapped low severity bugs to bluish hues, and high severity ones to reddish hues.
Figure 7 does not exhibit any thick black stripes, which indicates that the closed bugs are spread throughout the various severity levels. From the distribution of the hues within components, one can see that most bugs have normal or lower severity. Enhancements (in the bottom of each component node) also make for a non-negligible percentage of bug reports, and their dark colour indicates that most enhancements have already been resolved.

To sum up, while Figure 6 groups bugs by status, Figure 7 groups them by severity. The former makes it easier to see what kinds of bugs are being processed in each state (e.g., what are the currently assigned bugs?), while the latter makes it easier to see what is happening to critical bugs.

5. Related Work

Graph visualisations were found to be useful in understanding software systems, for various relations in requirements [8, 19], design [11] and implementation [9]. Hence, general or special purpose visualisation systems such as [1, 2, 3, 7, 14, 17, 18, 22-24] have been used in program understanding, software architecture, software evolution, clustering, and so on. Despite the need to further specialise these visualisation systems, there is also a need to combine the views in a non-intrusive way such that one can view related metrics from other sources without losing focus on the primary visualisation subject. Such a hybrid view might help understanding traceability among different software viewpoints [10] and different development phases [15].

Most visualisation systems have mechanisms to integrate different structures through standardised graph exchange formats [12], but they mostly split the window into separate views for different structures. To show traceability links between different views explicitly, some systems require one to explicitly document them as arrows, but they further clutter the desktop space [5, 16]. In this paper, we present an alternative to allow one to focus on the primary subject while reflecting the related information as decorating colours.

Ordering pixels by date to visualize time-series data was initially used to visualise the cache behaviour in program executions [23], in which we assigned colours to different types of cache hits/misses. The pixels align vertically when a recurring period of cache behaviour divides the width of the bitmaps, showing regular patterns in loop executions. However, we do not see such apparent vertical patterns in the bug traces. The change of density of colours over time, on the other hand, does show some commonality in the time series bug data among related components. When consistently applied using a circle of hues, we found it natural to reflect that fact of recurring lifecycle of bugs.

We previously used the HSB colour to decorate requirement goal graphs with multiple dimensions of colours [8]. Although the visualisation helps in understanding the static view of the goal model, it fell short in depicting the evolution of the status of goals. Although in principle it is possible to visualise such evolution using storyboard-like animations [3], it is found harder to memorise the animated states of components along hundreds and thousands of transitions. On the other hand, animation helps to view the snapshot of the graphs in synchronized way. To do so in the proposed visualisation, bug states of components can in sync using a global clock, i.e., by filling white pixels for bugs not belonging to the associated component. Similarly, a spectrum graph visualisation system developed by [22] can present evolutionary data statically. It uses only saturation to show punctuated evolution patterns. Our proposed visualisation can be regarded as a generalisation of the spectrum graphs, where hue and brightness can also play a role in presenting and relating multiple dimensions of metrics.

6. Concluding remarks

We have presented an approach to enrich existing graph-based software visualisations with more information, in a lightweight and compact way that does not obfuscate the existing graph (e.g. by changing its layout or creating more labels).

The approach is based on inserting coloured bitmaps into each node, each pixel in the bitmap representing one element associated to the node. The colour of each pixel is determined by assigning one to three element attributes to dimensions of the HSB colour model. Although we could have adopted the RGB colour model, we believe the orthogonal nature of the HSB dimensions makes it much easier for users to interpret the meaning of the colours they see. For example, while olive green can be easily interpreted via the HSB model as corresponding to a green hue with medium saturation and brightness, for the RGB model the user would need to understand the contribution of the red and blue components to make a pure green change to a dark greyish one.

The approach is completely flexible: the user can define which attribute is assigned to which colour dimension, how attribute values are mapped to dimension values, and in which order the pixels should
be drawn (which may be based on a different attribute than those used for colouring). All this combinatorial flexibility generates a wide variety of possible visualisations, from which each user can pick those more suitable not only for the application and research question at hand, but also for his or her personal likings.

Moreover, it means that this approach generalises existing visualisations in which a node has a single colour, representing a single attribute, and where different attribute values map to different hues or different saturations.

To show the usefulness of the approach, we have taken the architecture and the bugs of Eclipse as case study. Each node contains one pixel per bug associated to that component and the colours represent the bug’s attributes: severity, priority and status. We have shown visualisations that concentrate on a single bug attribute and others that mix all attributes. We have ordered pixels by the bug’s date and by colour (the bug’s attribute values). Ordering by date may e.g. help project managers to see if a significant number of early bugs remain unresolved.

The default of leaving unassigned dimensions at their maximal values and of attribute values being proportional to dimension values, means that high attribute values are mapped to bright saturated red, making it easier to spot the problematic cases (e.g. unconfirmed bugs with high priority and severity).

A further advantage of our approach comes from the HSB dimensions not being completely independent: zero brightness results in black independently of hue and saturation, and no saturation makes the colour grey, independently of hue. Hence, through the assignment of attributes to dimensions the user can focus on some attributes while neglecting others: e.g., assigning severity to brightness leads all enhancements to be mapped to black, independently of their priority or status.

Moreover, the approach of superimposing bitmaps on the architectural graph allows one to quickly see which components have more bugs and what the situation for each component is (e.g. are most of its bugs still unresolved? Does it have a higher percentage of critical bugs than other components?).

In our case study we couldn’t find any relationships between the component dependencies and bugs, but we believe this is due to the very high level of abstraction and coarse granularity of components, not to any intrinsic limitation of the general visualisation approach.

References


