The Complexity Map

Mark W. Hissink Muller
DNV-CIBIT (Det Norske Veritas)
Bilthoven, The Netherlands
mark.hissink.muller@dnv.com

Abstract

The Complexity Map describes a way of organizing and looking at software architectures, focusing on the integration of functional 'areas of responsibility'. When an application's source code is structured according to the Complexity Map layout rules, a visualization which is deemed understandable for non-technical users, can be automatically generated from the source code by the Complexity Map application. Using this visualization, quality and project management related attributes can be visualized in the different sections of the application, providing easy overview into the status of these attributes for end users, developers and management. The Complexity Map application is available under open source license from http://complexitymap.com.

1. Introduction

In our globalizing and increasingly networked economy, organizations and their business processes are becoming more and more intertwined. Many of the applications that support these business processes have interfaces with other systems, both within the same organization as beyond. As the number of applications and processes that are integrated increases, the importance of addressing the integration of applications in a way which is suitable for later changes increases likewise.

As is argued in [1] and [2], organizations are learning that a visual overview of their organization application landscape helps business managers and architects to govern (in particular manage the integration aspects of) the application portfolio. Although for a whole portfolio of applications, this insight is becoming more common, *within* an application and its software architecture, a visual approach of laying out the application is not always seen. Software architectures which are not laid out according to the most efficient *functional* approach lead to systems that are more difficult and thus more costly to maintain and change.

This paper presents an approach which we claim helps effective design of software architectures for applications where *integration* is a predominant aspect. Based on this *design approach*, we present an application that visualizes architectures which follow the Complexity Map layout guidelines. The application can be used to monitor the amount of *technical debt* [3] in an application in a way which should be easy understandable by non-technical managers and which will help them to use this information in the decision making process.

This paper is structured as follows. Section 2 describes the symptoms which led to the development of the Complexity Map [4]. Section 3 introduces the Complexity Map diagramming style, which is used in section 4 to show quality attributes can be added to the visual representation of the source code to manage *technical debt*. Section 5 describes the current functionality of the Complexity Map application and future directions for development. Section 6 concludes this article.

2. Symptoms from the industry

The basis for the work presented in this paper is found in the following problematic symptoms, which have been encountered in certain applications in the industry during DNV-CIBIT's audit and consulting engagements. For reasons of confidentiality, no reference can be made to the actual applications or organizations where these symptoms were found.

- Architectural diagrams are often too informal in nature (regarding the structure that is described) and lack the *use of direction* as a means to add information about architecture that is described.

- The gap between how a business user views the application structure and the actual situation is not always minimal, as it should. Business people should, when explained, be able to recognize various parts of an application which support a business process, down to the lowest level.

- Naming of *areas of responsibility* is underappreciated in the design and construction of (in particular) administrative systems. *Semantics*, or giving things the proper name, is a notoriously hard topic and project managers allow very little time to adapt and refactor an application when later insights occur.

- Functional decomposition is not always leading over technical decomposition, as it should. Developers tend to naturally use technical concepts to apply ordering at the lower levels of the structure of an application. A striking example often found is the segmentation according to the *model*, *view* and *controller* paradigm [5], which results in application packages with equal names; not a good practice.

- A conceptual model of the application does not always span the entire end-to-end system, as it should, but often focuses on a part in isolation, e.g. the application's domain model.

3. Complexity Map diagramming style

The Complexity Map diagramming style, as presented in this section, addresses the symptoms and concerns discussed in the previous section.

3.1. Layout principles and rules

Applications which are laid out according to the Complexity Map diagramming style adhere to the following rules.

- **Overall context is relevant.** A *namespace hierarchy* is used to bind all *areas of responsibility* to a single conceptual tree. In a two-dimensional layout, this concept results in boxes within boxes.

- **Horizontal direction of the visualization is relevant.** The map is read from left to right. Users and systems that *use* the particular system are positioned on the left hand side of the architecture, systems that are *used by* the system are positioned on the right hand side. In this way, the internals of the application's software architecture mimic the systems which surround the application.

- **Vertical direction of the functional areas is relevant.** The map is read from top to bottom. Functionality that is either more important or used earlier in time is positioned higher, functionality that is used by other packages is positioned on the lower side.

- **The first ordering is technical, the further layering functional.** The technical layers follow a pre-defined structure.

Using these layout rules, figure 1 shows an example of the Complexity Map diagramming style for an imaginary software architecture of the Amazon's online bookstore [6]. On the left hand side of the figure, from top to bottom, areas for the `web`-user, `message` and `batch` jobs which *use* the system are indicated. On the right hand side of the figure, areas where integration with the external and internal systems and the database is handled are displayed. Within the outer structure in the application (`com.amazon.store`) the technical layers for `service`(s), `domain` objects, `integration` and `persistence` layers are also explicitly positioned. The boundaries of the various *areas of responsibility* are indicated by the thick frames.

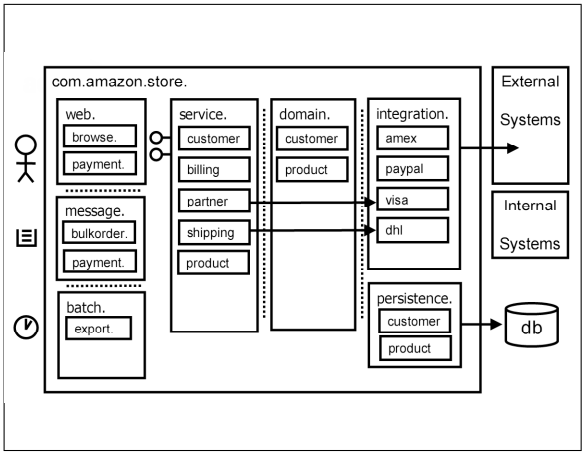

Figure 1. Amazon architecture

Reading the diagram from the outside inwards, the abbreviated *position* of a `ProductService` implementation in the *namespace hierarchy* would be: `...store.service.product.ProductService`,

The diagram intends to serve as a source code package design, guiding the development team during application construction.

3.2. Generating a visualization from code

If an application is laid out according to the Complexity Map layout rules, a directed visualization can be generated automatically from the application's source code. Figure 2 shows the Amazon architecture in a diagram generated from source code by the Complexity Map application using its `MatrixPaneLayout` TreeMap-positioning algorithm.

It is always possible to visualize source code using the Squarified Treemap [7] algorithm (of which no example is displayed in this paper), but when an application is laid out according to the Complexity Map layout rules, it is possible to create a directed map (as displayed figure 2) which - as we argue - has added value.

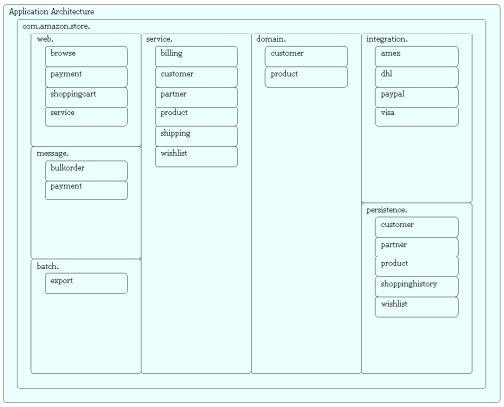

Figure 2. Generated wireframe visualization

4. Managing technical debt

When aggregated quality attributes are added to the Complexity Map wireframe visualization, the result can be used as a *management dashboard* of *technical debt*, useful for guiding application development (see 4.2).

4.1. Visually indicating quality attributes

When both business and technical users use the same visual model to think about the application and its integration challenges, this visual representation can be used to indicate quality attributes, as is shown in figure 3.

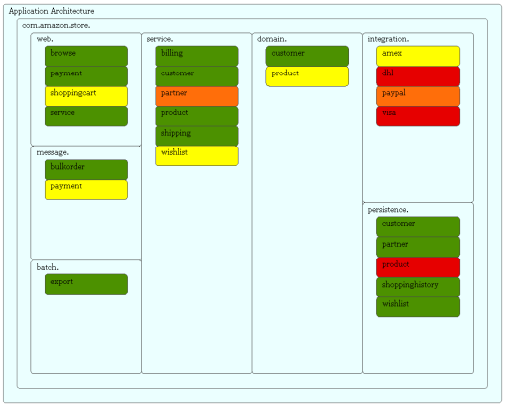

Figure 3. Quality indicated

From this Amazon-example, it shows that the quality for the application is sufficient (*green* or *yellow*) in many areas, but that there are some challenges (*orange* or *red*), mainly in the areas of integration with external systems and where the integration with the database is solved.

Although these diagrams were created using a mockup and no actual Amazon source code was used in the process, the benefits are clear. Attributes in color added to the diagram, provide an overview at a glance, showing non-technical stakeholders about (quality) problem spots. More problematic findings per area will turn an *area of responsibility* from *green*, via *yellow* and *orange* to eventually *red*, where thresholds can be set by the user.

The (quality) indications will allow business owners or management, who understand the wireframe representation of the source code, to ask questions like: *"Why was this area colored yellow?"*, *"What extraordinary problems in the source code justify the red color for this area?"* or *"What do we need to do to make all boxes yellow or green?"*.

As a basis for the color indication, currently *software metrics* and other measured findings which are collected using open source tools, are used. Metrics can be presented in various layers, either individually or in aggregated form. Since some metrics are, within a certain context, more important than others, metrics can be weighted before aggregation. The summarized *complexity torque* (or the amount of 'pressure' due to complexity) for a certain *area of responsibility* is used to color the area. The selected *sensitivity scale* of the Complexity Map selects the intervals which are used for the various colors. This functionality can be used by the end user to perform *sensitivity analyses*.

4.2. Guiding application development

As managers are using the Complexity Map to guide their teams and developers little by little remove an application's *technical debt*, the relative weight of the negative findings which underlie a certain coloring can be modified, thus increasing the team's ambition level. The development team's output bar is raised, which will lead to applications which are better maintainable and eventually, better skilled developers.

When a business manager gets a little bit more involved (in a white box way) with how an application he/she's responsible for is structured, it will ease - we argue - the overall life cycle management of the application.

5. Complexity Map application

The concepts presented the previous sections have been implemented in the Complexity Map application, for which a working proof of concept is available in the Java-language [8]. Although it relies on several third party components, notable is Prefuse [9], a toolkit which greatly eases the creation of impressive visualizations. Current features of the Complexity Map application include:

- support for *static* or *dynamic* source code measurement tools via adapters

- displays source code violations in individual or aggregated and weighted layers
- contains a zoomable interface to allow quick analysis of large scale enterprise codebases
- drill down to the individual finding-level (and sideways from this level) for detailed analysis
- save layers of findings for historical trend analysis

The current version of the Complexity Map application uses Checkstyle [10] and PMD [11] to analyze source code of Java applications.

For applications which need to be visualized using the Squarified Treemap layout (since their architecture was not created with the Complexity Map layout rules in mind) it is still possible to visualize attributes of the source code in multiple layers. Building on the work presented in [12], the Complexity Map application allows various layers to be aggregated to one weighted, summarized layer, which will ease and allow for quick management interpretation.

The Complexity Map application proves to be valuable when used in industrial context, in particular for initial analysis of large codebases. The seamless zoom functionality allows large codebases to be analyzed quickly, whereas the drill down functionality allows findings to be considered per *area of responsibility* or for the *codebase as a whole* and to be sorted by e.g. severity of the violation or by category.

Possible future extensions or additions to the Complexity Map application could entail; support for more programming languages, support for used-added findings or complexity *trend analyses* based on Complexity Maps taken at various moments in time.

The Complexity Map application is available under GPLv3 license [13] from [4].

6. Concluding remarks

For applications which need to integrate with more than just a simple database, a conceptual map of the application will help facilitate the discussion about the correct *position* of functionality and provide a common understanding of the system for both end users as the development team.

The Complexity Map describes a way of organizing the application's software integration architecture, which can be visualized using the Complexity Map application. The colored quality indications allow *technical debt* to be identified by management and kept at low levels which will keep applications fit for change.

The Complexity Map application is available under GPLv3 license and developers are invited to contribute.

References

[1] Matthijs Maat, Jochem Schulenklopper, and Gert Florijn. IT Landscape Photography vital for decision making. *DNV-CIBIT website (`http://www.cibit.com`)*, 2007.

[2] Jürgen Laartz, Ernst Sonderegger, and Johan Vinckier. The Paris guide to IT architecture. *The McKinsey Quarterly*, August 2000.

[3] Ward Cunningham (quoted by Martin Fowler). Tecnical Debt. `http://www.martinfowler.com/bliki/TechnicalDebt.html`.

[4] Mark W. Hissink Muller. The Complexity Map application. `http://complexitymap.com`, 2007-2008.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns (Hardcover)*. Wiley, 1996.

[6] Amazon online bookstore. `http://www.amazon.com`.

[7] D.M. Bruls, C. Huizing, and J.J. van Wijk. Squarified treemaps. *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, January 2000.

[8] James Gosling, ea. The Java Programming Language. `http://java.sun.com`, 1996-2008.

[9] Jeffrey Michael Heer. The Prefuse Visualization Toolkit. `http://vis.berkeley.edu/papers/prefuse/`, 2005.

[10] Oliver Burn, ea. Checkstyle. `http://checkstyle.sourceforge.net/`, 2001-2007.

[11] InfoEther. PMD - don't shoot the messenger. `http://pmd.sourceforge.net/`, 2002-2008.

[12] D. Holten, R. Vliegen, and J.J. van Wijk. Visual Realism for the Visualization of Software Metrics. *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (Proceedings of VISSOFT 2005)*, pages 27–32, January 2005.

[13] GPLv3 License. `http://www.gnu.org/licenses/gpl-3.0.html`.