A Survey about the Intent to Use Visual Defect Annotations for Software Models

Jörg Rech[1] and Axel Spriestersbach[2]

[1] Fraunhofer IESE, Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
+49 (0) 631 6800 2210, Joerg.Rech@iese.fraunhofer.de
[2] SAP Research, Vincenz-Prießnitz-Str. 1, 76131 Karlsruhe, Germany
+49 (0) 6227-752525, Axel.Spriestersbach@sap.com

Abstract. Today, many practitioners have consolidated their experience with software models in collections of design flaws, smells, antipatterns, or guidelines that have a negative impact on quality aspects (such as maintainability). Besides these quality defects, many compilability errors or conformance warnings might occur in a software design. Programming IDEs typically present problems regarding compilability in or near the code (e.g., icons at the line or underlining in the code). Modeling IDEs in MDSD follow a visual paradigm and need a similar mechanism for presenting problems in a clear, consistent, and familiar way. In this paper, we present different visualization concepts for visualizing quality defects and other problems in software models. These concepts use different dimensions such as color, size, or icons to present this information to the user. We used a survey to explore the opinions held by practitioners showing that 89.9% want to be informed about potential defects and prefer icon-, view- and underscore-based concepts to other types of concepts.

Keywords: Visual Annotations, Software Models, Intelligent Assistance, Quality Defects, Software Diagnostics, MDSD

1 Introduction

Model-driven software development (MDSD) drastically alters the software development process, which is characterized by a high degree of innovation and productivity. MDSD focuses on the idea of constructing software systems by designing visual models that are translated into executable software systems by generators. These characteristics enable designers to deliver product releases within much shorter periods of time compared to the traditional development methods. In theory, this process makes it unnecessary to worry about an executable system's quality, as it is "optimized" by the generators.

However, just as in current software programming, people make mistakes that result in defects of the software model. These defects might prevent the compilation/transformation of the model, deteriorate its quality aspects such as its maintainability, or violate conformance to a modeling standard. In programming errors and warnings are usually presented to the programmer in the textual editor by means of

problematic parts being underlined or marked with little icons. In MDSD, problems are typically not annotated directly in the visual models.

In this paper, we present several concepts for annotating software models in order to provide architects, designers, and modelers with additional information about problems regarding the compilability, quality, or conformance of a software model. Our primary research goal was to identify the acceptance of our concepts by practitioners and explore their intent to use these visualization concepts.

After presenting related work in section 2, we describe details of the annotation concepts in section 3. The instrument for the evaluation of visual mockups, used to evaluate the acceptance of our annotation concepts, as well as the findings of our survey is described in section 4. Finally, we conclude our contribution in section 5

2 Related Work

In order to inform the modeler about the quality defects in his software model (e.g., the PIM), we need to annotate the visual diagrams presented to him. However, annotating UML with information about quality defects is not a straightforward task. UML is intended to describe the structural (and, with Action Semantics, the behavioral) elements of a software system. Nevertheless, in UML (2.1) [13] several mechanisms exist to store additional (non-standard) information in the software model. However the additional information are either not shown in the diagrams (e.g., UML annotations) or would flood the diagrams (e.g., UML comments).

2.1 Defects in Programming IDEs

In programming IDEs, icons are typically used to pinpoint problems such as compiler warnings or errors. For example, the Eclipse IDE (V3.3) [5] uses markers to annotate problems in the source code and icons are anchored directly beside the respective line, while underlines are used to pinpoint the exact position. A problem view is used to list all problems in all projects and decorators are used to annotate problems in a class or other files shown in the project file tree. Netbeans (V5.5) [7] also presents defects as icons in the source code directly beside the respective line and underlines the exact position in the text. In Visual Studio .Net 2003 (V7.1) [15], defects are listed in the Task List after the build process is started.

Another source for error or defect visualization are tools used for bug tracing and error detection. A couple of such tools exist for the Java language. Most of them are extensions to the Eclipse environment. During our survey, we looked at the most commonly used tools: Findbugs [6], PMD [9] and CheckStyle [2]. All tools are based on Eclipse and use the features Eclipse provides, such as the "problem view", to report problems to the user. While these annotations are very useful in textual IDEs, they are not sufficient for visual IDEs in MDSD. Here we need to annotate elements in 2D graphs with multiple defects of various severity, priority, and effects.

2.2 Defects in Modeling IDEs

In software modeling, the visual annotation of defects in software models is scarcely explored. Current modeling tools can be classified as either being based on the Eclipse framework or being standalone tool. Those tools integrated into Eclipse usually use the known Eclipse features to visualize errors:

- The *problem view* lists errors and warnings from multiple sources and aggregates them into one list. The list usually supports icons to indicate the level of seriousness, such as error, warning, or info.
- The *outline view* is a representation of the model tree and usually contains information on errors, which are indicated by mini-icons (i.e., so-called decorators).
- Errors in the *textual editor* area are usually indicated by an icon directly beside the line of code and the text block containing the error is highlighted by underscores. To facilitate navigation, errors are marked right besides the scrollbar.

Typical examples of model editors built on top of the Eclipse functionalities are Topcased and OmondoUML. Topcased [12] displays errors in the UML models as tasks in the Eclipse Problem List and also little icons directly in the diagram as well as the tree view of the UML model. The icons are rather small. Only one error type is used. Errors in hidden parts of the tree view are not shown in collapsed nodes. One example that combines all methods is OmondoUML [8] that is also based on Eclipse and used within Java projects. It attaches Java coding directly to the model. Errors in the Java coding are shown in multiple locations such as the UML diagram, the code, the Eclipse Problem list, and the project (resp. package) tree.

Examples of tools that are not based on Eclipse are Poseidon [10] and Enterprise Architect [4]. Poseidon [10] is the commercial variant of ArgoUML [1], which presents some errors in the UML diagram using icons and most others in a list. Enterprise Architect [4] supports model validation and detects errors and warnings found in the model. They are visualized as lists similar to the Eclipse Problem list. However, these tools do not use differentiating icons in the error list or in the diagram.

2.3 Safety Signs and Defect Pictograms

Beyond our own domain people in the field of technical documentation are using icons, signs, and pictograms to warn about dangerous substances or situations. ISO 3864-2 2002 and ANSI Z535.2-2002 are standards for environmental & facility safety signs that includes plain warning symbols, detailed message panels, and short header sections (see Figure 1a). It is intended to establish "the safety identification colors and design principles for safety signs to be used in workplaces and in public areas for the purpose of accident prevention, fire protection, health hazard information and emergency evacuation". Other standards include ISO 7010 (see Figure 1c), IEC 61310, ISO 16069, IEC/TR 60878, (see Figure 1b), ISO 7000, or IEC 60417. Similar to the passenger/pedestrian symbols developed in the 1970s by the AIGA and the U.S. Department of Transportation (DOT), they represent uniform and consistent visual languages.

However, while these pictograms can be used in the context of MDSD to annotate defects in software models, they are not tailored to represent different types, severities, priorities, or effects of these defects.

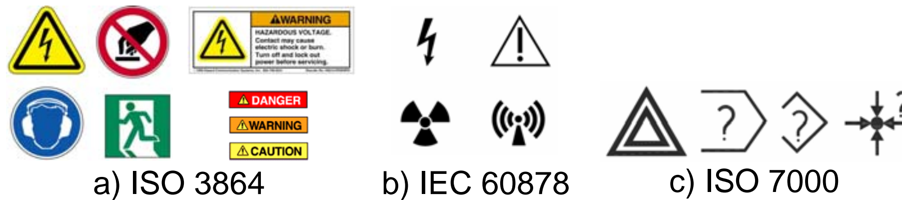

Figure 1 Warning Symbols and Pictograms

3 Visualization Concepts

For the annotation concepts, we used an exemplary UML diagram contaminated with multiple defects. This example was also used in the survey as the guiding scenario. As presented in Figure 2, the UML class diagram contains a package (Opportunity) and two class declarations (Opportunity and SalesForecast). The class Opportunity is meant to represent an opportunity for a sale (e.g., during project acquisition) and SalesForecast the prognosis / forecast about the chances and benefits of this opportunity.

Since the focus of this survey was on the defect annotation of the software models, we introduced five defects on different levels that are marked with numbers in the following diagram:

1. **Relation defect**: Defect in the relations between the two classes. In this case, a circular inheritance was introduced.
2. **Attribute defect**: Defect in the attributes of the class SalesForecast. In this case, one identifier is specified twice.
3. **Method defect**: A defect in the method declaration of the class SalesForecast. In this case, too many parameters (i.e., the code smell "Long parameter list").
4. **Class defect**: Defect in the Opportunity class itself: The class has too few methods and attributes (i.e., the code smell "Lazy Class").
5. **Diagram defect**: Defect in the diagram: The diagram has too few elements and might be superfluous.

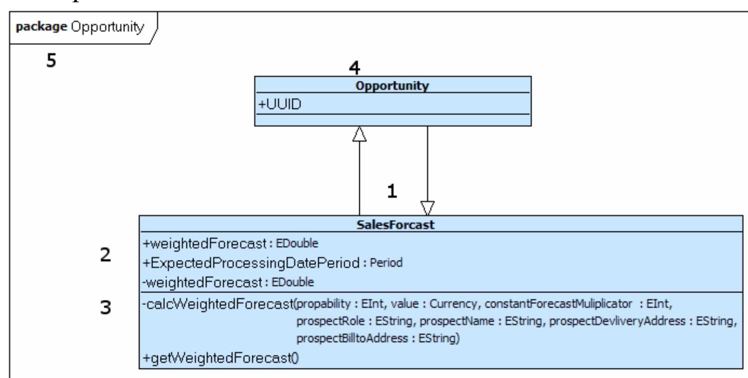

Figure 2 Scenario Mock-up

We collected several kinds of annotations during a brainstorming session. We identified 11 techniques applicable to two-dimensional software models listed in Table 1. The first five (Icons to Aura) including the last one (Views) were selected for evaluation as they appeared to be the most promising. The concepts are characterized, as to whether they can be applied in entities (e.g., boxes for classes or packages), relations between entities, the connectors of relations (e.g., the diamond shaped ends of an aggregation), or additional notes or comments. Furthermore, the positioning of the annotation is differentiated in the body (e.g., within a box or connector), the line, the frame (e.g., borderline of a box or connector), and the aura (e.g., directly outside the box or connector).

	Entities			Relations		Connectors			Notes		
	Body	Frame	Aura	Line	Aura	Body	Frame	Aura	Entity	Relation	Connect.
Icons	●	-	●	-	●	o	-	●	●	●	●
Color	●	●	●	●	●	●	●	●	●	●	●
Bold	o	●	-	●	-	-	●	-	●	●	o
Dashed	-	●	-	●	-	-	●	-	o	●	●
Aura	●	●	-	●	-	●	●	-	●	●	●
Form	○	●	●	●	●	○	●	●	o	●	●
Size	○	●	●	●	●	○	●	●	o	●	●
Pattern	●	○	●	○	●	●	○	●	●	o	●
Opaque	o	●	o	●	-	o	●	-	●	●	●
Tilting	○	●	o	-	o	-	●	o	o	-	○
Views	-	-	-	-	-	-	-	-	-	-	-

Table 1. VISUALIZATION CONCEPTS AND AFFECTED ELEMENTS

In order to annotate software models in MDSD with information on defects, we developed the previously described concepts that should visually present defects. As we developed these concepts, we identified the following constraints:

- They should be *integrated into the UML* (Version 2.1) and not change the meaning and standard representation of the language's elements (e.g., by changing the form of class-boxes).
- They should be easily *integratable into today's UML modeling environments* (i.e., this excludes animated or 3D visualizations).
- They should *pinpoint the location* of the defect as exactly as possible.
- They should enable *differentiating between different defects* (e.g., based on type).
- They should enable the annotation of one modeling element with *multiple defects*.
- They should *not distract modeler* from his work (i.e., excluded large annotations).

Finally, the annotations should be presented to today's software developers and modelers in a familiar way.

Icon-based Concept

The icon-based concept is centered around the idea of representing every defect by a distinct icon positioned at a package, class, method, attribute, or relation in a UML model. As depicted in Figure 3, the five previously mentioned defects are positioned

very close to the defective element. However, only one icon per element can be attached (except for packages/diagrams, classes, and possibly (long) relations) and, if more defects were diagnosed, they have to be hidden (i.e., stacked) below the first one. The amount of icons positioned at packages/diagrams, classes, and relations is basically constrained by their individual size.

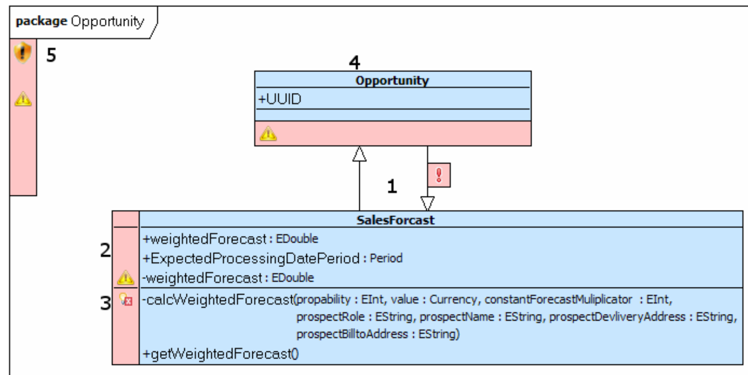

Figure 3 The Icon-based Concept

We varied the icons in order to represent different defects, since in reality, several different kinds of defects (e.g., compiler errors, maintainability defects, security flaws, etc.) would be shown. However, as several hundreds of these defects are known [11] it is probably not possible to represent every individual defect by one distinct icon (esp. considering the size limitations of 16 x 16 pixels). Nevertheless, groups of defects regarding one specific quality aspect (cf. ISO 9126) might be represented by one icon.

Color-based Concept

The color-based concept is centers around the idea of representing every defect with a distinct color for a package, class, method, attribute, or relation in a UML model. Here the five previously mentioned defects are indicated by different colors.

As with icons, the concept allows only one color to be assigned for each individual element and therefore only one defect/error type. Multiple colors can represent several different kinds of defects (e.g., compiler errors, maintainability defects, security flaws, etc.), but if multiple defects need to be assigned to one model element, only the color of the defect with the highest priority can be presented.

However, as color can and is used in UML tools to distinguish entities such as classes, this concept has to be handled with care. Furthermore, if color is used it should affect the readability of the diagram (e.g., a red text on a red background would be hard to read).

Boldness-based Concept

The boldness-based concept is very similar to the color-based concept but represents defects in boldface text for package, class, method, attribute names and with thicker/bold lines for relations and boxes in the UML model. The concept is expected to be less intrusive or distractive, however, the drawback is that the visualization cannot distinguish different types of defects.

Underscore-based Concept

The underscore-based concept adapts a concept for error visualization from code editors as well as from spellcheckers in word processors. Since VIDE visual syntax goes beyond textual, this concept has been extended to diagrams where needed. While package, class, method, and attribute names are underlined, relations are *overlaid* with a dashed line. The concept may be extended to other diagrams and connectors as needed. To distinguish different defect types, different colors can be used similar to the color-based concept and as depicted in Figure 4.

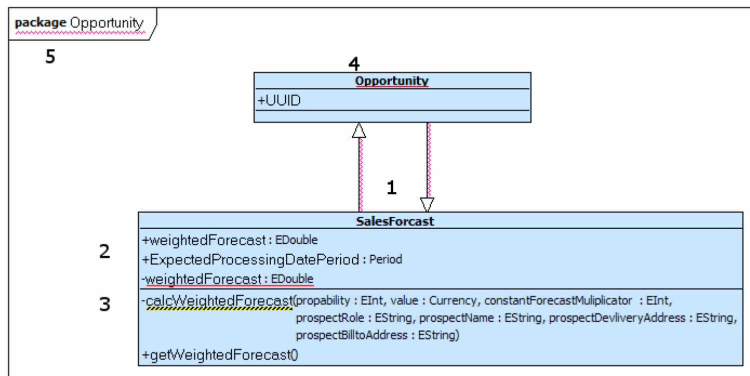

Figure 4 The Underscore-based Concept

Aura-based Concept

The aura-based concept represents a fusion of the underline-based concept with the color-based concept in order to maintain consistency of the annotated diagram. In this concept, diagrams, entities, relations, and text are enriched with a colored aura or halo. The aura-based concept surrounds all elements (package, class, method, attribute, and relation) with defects/errors in a consistent way.

The Views Concept

A separate view that lists all errors, defects, warnings etc. is the most common and accepted concept. Today, the majority of development, bug tracing, and diagram modeling tools feature a separate view that shows identified defects. The views are organized as simple lists that often support (hierarchical) categories and sorting.

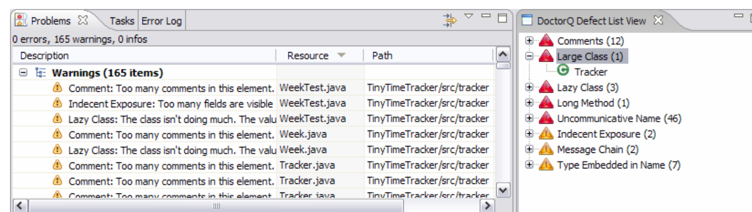

Figure 5 Concept Using Views

Icons are often used to facilitate understanding. These icons should be the same (also semantically) as those used in other views (i.e., a diagram or the coding/textual view.).

4 Evaluation

In order to evaluate these annotation concepts for architectural models, we conducted a survey using an electronic questionnaire. The survey was aimed at eliciting which annotation concept practitioners prefer and when, what kind, why, and where they prefer the annotations.

We conducted the survey between 24 September and 8 October 2007. The main target groups were architects, designers, as well as programmers and testers in software organizations who are involved in daily software development activities. Our respondents consisted of a total of 292 individuals, of whom 78 completely finalized the questionnaire – 48.3% from SMEs and 51.7% from large enterprises.

To develop the survey pages and make them available on the Internet, a commercial tool called OPST from the company Globalpark (http://www.globalpark.de/) was used. The questionnaire was designed using multiple choice questions (mostly based on a 7-point semantic differential or five-point Likert scale) wherever possible, as these are more likely to be answered, and it is easy to statistically analyze the answers. To allow unexpected answers, most concepts had an open question with some extra space for comments.

Figure 6 summarizes the respondent profile of our survey. The respondents had 6-10 years of experience on average and consisted of 54.2% architects, 18.6% developers, and only 27.1% other (of which 1.5% stated to be project managers).

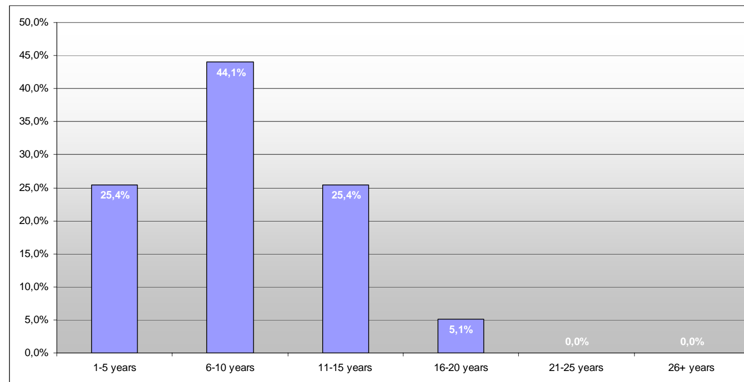

Figure 6 Experience with Software Modeling

Furthermore, 67.9% of their employers had been using modeling techniques for over 2 years for their commercial software, 66.1% for their internal software, and 50.9% were applying the model-driven approach (i.e., code generation).

The respondent profile obtained met our prior expectations, considering the basic user group of assistance in software engineering tools. Non-management employees and project managers are the group that is supposed to have the most contact with tools in this domain.

4.1 Findings

The following results are extracted from the answers to the survey and are provided in graphical format for reasons of brevity. The main feedback on the evaluated visualiza-

tion concepts is depicted in Figure 7. To compare the concepts we defined eight factors that are targeted to explore the intent to use a visualization concept. We asked the following questions using a semantic differential between 7 and 1, with 4 representing the neutral center:

- *Useful vs. useless*: To identify if the participants perceive the visual concept as useful or not.
- *Unobtrusive vs. distracting*: To identify if the participants perceive the visual concept as distracting or not.
- *Appealing vs. repelling*: To identify if the participants perceive the visual concept as appealing or not.
- *Understandable vs. incomprehensible*: To identify if the participants perceive the visual concept as understandable or not.
- *Consistent vs. inconsistent*: To identify if the participants perceive the visual concept as consistent or not.
- *Precise vs. imprecise*: To identify if the participants perceive the visual concept as precise or not.
- *Easy to use vs. hard to use*: To identify if the participants perceive the visual concept as easy to use or not.
- *Familiar vs. strange*: To identify if the participants perceive the visual concept as familiar or not.

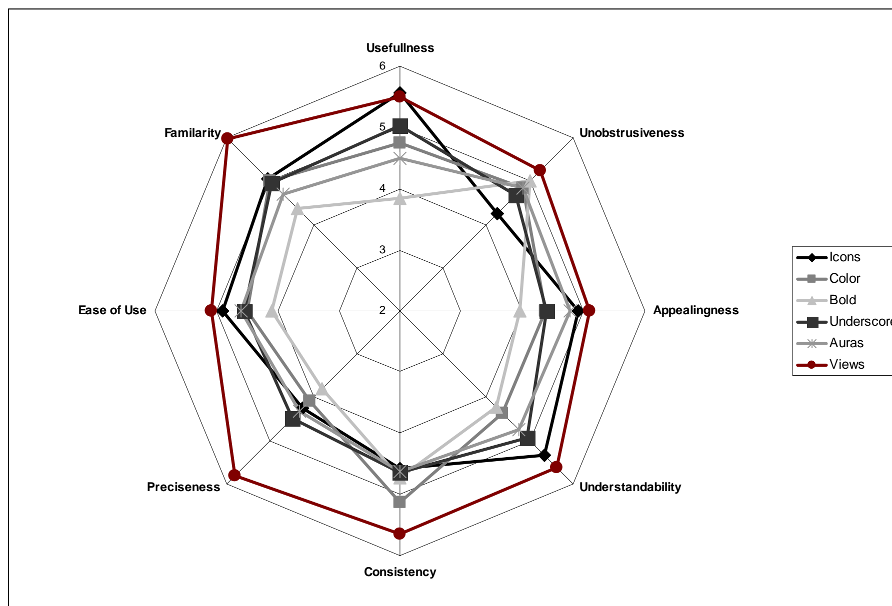

Figure 7 Kiviat Graph Overview

In order to compare all annotation concepts and rank them, we mapped the scores from all questions onto one value. As shown in Figure 8 the concepts with the highest overall scores are the views, icons, and underscore concepts. While no concept was found to be inadequate, the participants were almost undecided regarding the bold-

based concept. The color- and aura-based concepts were slightly accepted and are only marginally behind the underscore-based concept.

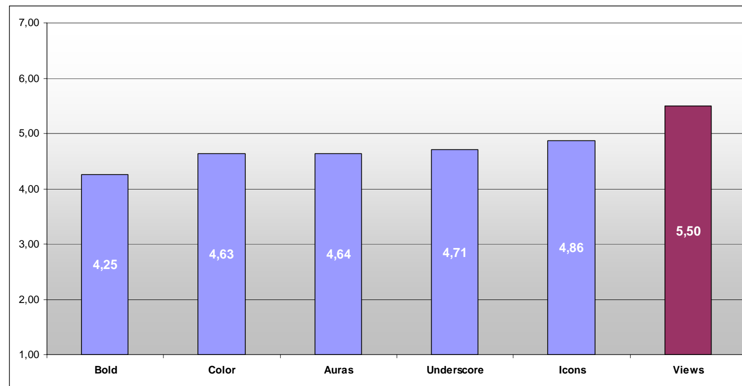

Figure 8 Average Score for Concepts

The general preference of visual in-model presentation concepts (plus additional ones such as form or opaqueness) is shown in Figure 9 question was: “Assuming I had access to a modeling system (e.g., an UML tool) with a defect annotation extension, I would use it if it presented defects with the following types of annotations”. While the participants had no picture that described the concept the results seem to indicate that they are interested in the “Pattern (e.g., the background of a box)” concept. Concepts based on size (e.g., larger boxes and font sizes) or forms (e.g., deformed boxes) seem to be very undesirable.

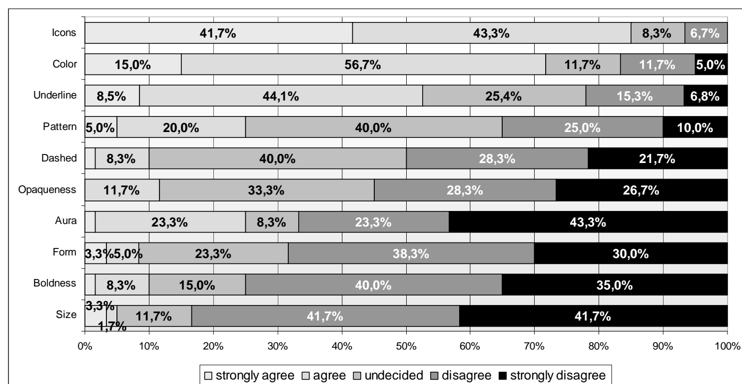

Figure 9 General Preference of Concepts

4.2 Discussion

In this paper, we present different visualization concepts and used a survey to explore the practitioners’ opinions. In this section, we discuss the strengths and weaknesses of our work. Beside the data presented in section 4, we have observed that the participants were very interested in the matter and gave long comments (3-4 sentences) to each concept.

Strengths
The survey described above has enabled us to undertake a transparent collection of opinions regarding the perceived usefulness and acceptance based on the technology acceptance model (TAM). While the original TAM (as well as UTAUT [14]) questions are not aimed at evaluating mockups, Davis and Venkatesh analyzed TAM for the user acceptance testing of pre-prototypes/mockups [3].

This survey was constructed to record information on visualization concepts for quality defects in software models. In retrospective, this approach and our implementation had the following strong points:

- The survey is *replicable* due to the approach described, the search terms used, and the selection process.
- It is *transferable* to other, new visualization concepts and can be used to compare them to the concepts we selected for the review.

Whether the results really show the intended use, i.e., whether the results of Davis and Venkatesh will hold, has to be investigated in the future.

Weaknesses & Threats to Validity
However, besides these strong points, we are aware that there may be weak points to our survey. From our point of view, it had the following weak points:

- The participants only evaluated a static representation of the visualization concept and could not "work" with it. However, we follow the research done by Davis and Venkatesh [3] that the evaluation of mockups can be used to judge the real usage behavior.
- We should have used more "obviously" bad concepts in order to find more explicit differences between the concepts. However, due to the constraints regarding the time participants are willing to invest, we assume that only few (i.e., 2-4 more for a total of 8-10) concepts could be added, as one concept requires approx. 3-5 minutes to evaluate.

Finally, it is unclear if the results of the survey are truly representative, as no information on the basic population in the field of MDSD is documented. However, as we have elicited answers from people in organizations of various sizes and in different domains, as well as from different experience levels we assume that there was no large systematic error.

5 Conclusion

The findings of this survey provide a general characterization of the preferences of practitioners regarding the annotation of software models in MDSD. The findings helped us to identify the most promising candidate for quality defect annotation and might be used as a starting point for people interested in the development of intelligent assistance systems and annotation languages.

The survey results provided the following observations about visual annotations of software models as perceived by the participants:

- Almost all participants (89.8%) want to be informed about defects in their software models.

- Of the visual concepts presented, the icons-based concept was preferred above all others.
- Views that present a list of defects are clearly preferred by many people and should not be replaced by purely visual concepts.
- The most useful, appealing, understandable, easy to use, and familiar concept seems to be icons.
- The most unobtrusive concept seems to be bold.
- The most consistent concept seems to be color.
- The most precise concept seems to be underscore.

As MDSD is becoming more and more getting productive and enables software engineers to visually develop software systems, we expect to see our or similar concepts integrated into visual IDEs. Therefore, we are currently working on the implementation of defect annotations into an open source IDE for MDSD and are constructing a visual language for the differentiation of individual defects.

References

1. ArgoUML, "ArgoUML IDE," http://argouml.tigris.org/, last accessed on 27 November 2007.
2. Checkstyle, "Checkstyle Defect Detector," http://checkstyle.sourceforge.net/, last accessed on 27 November 2007.
3. F. D. Davis and V. Venkatesh, Toward preprototype user acceptance testing of new information systems: implications for software project management T2 - Engineering Management, IEEE Transactions on, Engineering Management, IEEE Transactions on, vol. 51, no. 1, pp. 31-46 (2004)
4. EA, "Sparx Enterprise Architect IDE," http://www.sparxsystems.eu/default.asp?nav=3x6&lid=32, last accessed on 27 November 2007.
5. Eclipse, "Eclipse IDE," http://www.eclipse.org/, last accessed on 27 November 2007.
6. Findbugs, "Findbugs Defect Detector," http://findbugs.sourceforge.net/, last accessed on 27 November 2007.
7. Netbeans, "Sun Netbeans IDE," http://www.netbeans.org/, last accessed on 27 November 2007.
8. Omondo, "Omondo UML IDE," http://www.omondo.com/, last accessed on 27 November 2007.
9. PMD, "PMD Defect Detector," http://pmd.sourceforge.net/, last accessed on 27 November 2007.
10. Poseidon, "Gentleware Poseidon IDE," http://www.gentleware.com/products.html, last accessed on 27 November 2007.
11. J. Rech and A. Spriestersbach, Quality Defects in Model-driven Software Development, Deliverable, Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, D4.1 (2007).
12. TopCased, "Topcased IDE," http://www.topcased.org/, last accessed on 27 November 2007.
13. UML-2.1.1, Unified Modeling Language (UML), version 2.1.1, Object Management Group, Inc. (OMG), Needham, MA, USA2007).
14. V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, User acceptance of information technology: Toward a unified view, MIS Quarterly, vol. 27, no. 3, pp. 425-478 (2003)
15. Visual-Studio, "Microsoft Visual Studio IDE," http://msdn2.microsoft.com/en-us/vstudio/default.aspx, last accessed on 27 November 2007.