Software Architecture

When to use Text and Graphical Formats for Communicating Ideas from Stakeholders to Software Developers

Johnny King and John L Williams Jr*
UCCS
Colorado Springs, CO, USA
jwilli11@uccs.edu
jking4@uccs.edu

ABSTRACT

Documenting software in a way that avoids verbosity, ambiguity, and confusion is a goal that all software architects should aspire to. The authors will identify and compare several formats that may be used to document software. The authors will list some of the pros and cons of each format, helping the readers to make an informed choice of the format to use for their own software projects.

Keywords

Software, Architecture, Documentation

1. INTRODUCTION

Software architects are responsible for communicating ideas from stakeholders to software coders with the goal that the software coders accurately and completely implement those ideas into a working software product.

With an estimated software project failure rate of over 40%, we need to investigate possible reasons for such a high percentage of failures. One such reason may be a failure to communicate effectively. Communication is required between stakeholders and requirements engineers, between requirements engineers and software architects, and finally between software architects and software coders and testers [6]. The authors will focus their investigation on communication failures between software architects and software coders and testers. This approach will assume that documents created prior to this communication phase are both complete and correct.

Software architects have several formats to chose from when it comes to creating documentation for software. We will assume that most software projects are too complex to support an all oral format using natural language to communicate the ideas, so we will assume that at least a written format using natural language is required. Other formats that are more formal and structured include UML diagrams and mathematical models.

*King and Williams are graduate students at UCCS

2. RELATED WORK

Miles and Hamilton describe why and how to use UML (Universal Modeling Language) for documenting software. They state that because informal languages do not have exact rules they will always suffer from the problem of verbosity, confusion, ambiguity and unnecessary details, which is an extremely dangerous way to model a system. Their solution is to use a formal modeling language, such as UML [3].

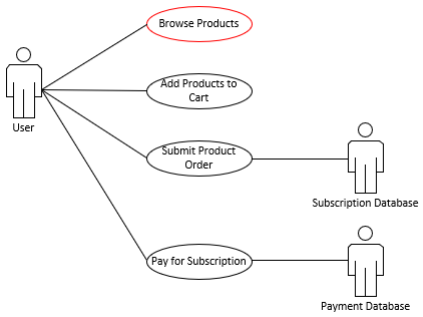
They list the following UML diagrams[3].

1. Requirements using **Use Case Diagrams**

   Use cases are the system's functional requirements and should be the first output from your modeling. "How can you begin to design a system if you don't know what it will be required to do?" [3] Use cases specify

what the system will deliver to users.

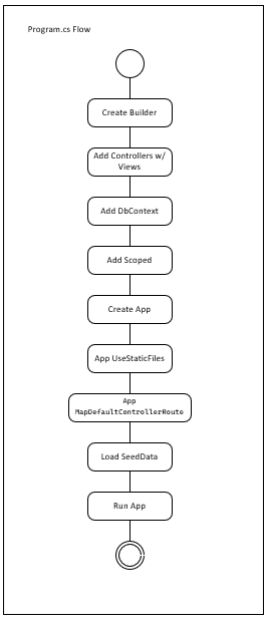
The figure below shows a use case diagram from our class project.



2. System Workflows using **Activity Diagrams**

   Activity diagrams show how the system will accomplish the requirements set out in the user case diagrams. High-level actions are linked together to represent a process that needs to occur in the system [3].

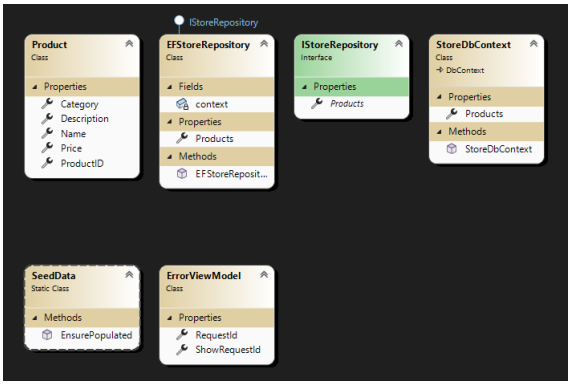
   The figure below shows an activity diagram from our class project.



3. A Systems Logical Structure using **Class Diagrams**

   Class diagrams show classes and their relationships [3].

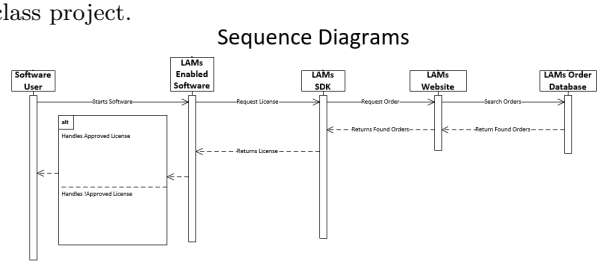
   The figure below shows a class diagram from our class project.



4. Ordered Interactions using **Sequence Diagrams**

   Sequence diagrams are an important member of a group known as interaction diagrams. They help accurately model how the parts that make up the system interact. They capture the order of interactions between parts of the system, describe which interactions will be triggered and in what order those interactions will occur [3].

   The figure below shows a sequence diagram from our class project.



5. Interaction Links using **Communication Diagrams**

   Communication Diagrams are also a member of the group known as interaction diagrams. They focus on the links between participants and are good at showing which links are needed to pass messages. They also show the links between participants that are required for events that make up the interaction [3].

6. Interaction Timing using **Timing Diagrams**

   Timing Diagrams are all about timing to model detailed timing information. These kinds of diagrams are mostly associated with real time and embedded systems, but may also be applied to other types of systems. Each event has the following information associated with it: when the event was invoked, how long it takes for another participant to receive the event,

and how long the receiver will be in that particular state [3].

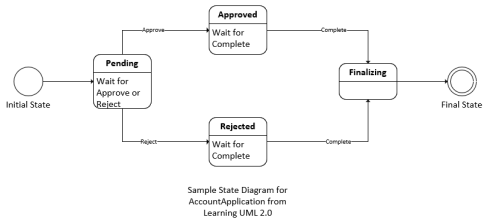
7. Interaction Picture using **Interaction Overview Diagrams**
   Interaction Overview Diagrams combine sequence diagrams, communication diagrams, and timing diagrams to show the higher-level picture. They result in having a better feel for what you are doing and the context in which you are doing it. Each part of an interaction overview is a complete interaction in itself. This diagram glues together the components in a way that makes the most sense to show how things work together [3].

8. Internal Class Structure using **Composite Structures**
   Composite Structures are part of the Logical View of your system. They come in handy when class or sequence diagrams don't quite work. These diagrams are relatively advanced in nature are are suited for situations such as internal structures which show the parts contained by a class, ports, and collaborations which show design patterns and how objects cooperate to meet a requirement [3].

9. System Parts using **Component Diagrams**
   Component Diagrams are part of the Development View which show how your system is organized into modules and components. This helps to manage layers within your systems architecture. Component Diagrams also help you manage complexities and dependencies between the parts. A component is an encapsulated, reusable, and replaceable part of your system which acts like a building block which can be combined into even larger component building blocks [3].

10. Organize Your Model using **Packages**
    Packages are part of the Development View and are used to group like UML elements together. In the case of grouping classes, namespaces are used. Namespaces are typically folders that contain like objects. For example, a folder named Models may be created to contain all classes that are model object classes. Packages help keep things organized. Also, dependencies between packages can be shown, which is an important fact to know [3].

11. Object State using **State Machine Diagrams**
    State Machine Diagrams are part of the Logical View and are used to show the state of objects and the events that trigger the objects to change state. A state diagram consists of states, transitions, and events. A transition represents a change of state or how to get from one state to another. The event is was causes the transition. A state becomes active when it is entered and inactive when it is exited. State diagrams are used frequently in real-time, mission critical applications, dedicated devices, and in games [3].
    The figure below shows a state diagram from [3].



Sample State Diagram for AccountApplication from Learning UML 2.0

12. The Deployed System using **Deployment Diagrams**
    Deployment Diagrams are part of the Physical View and show both the hardware and software components of the system and how they are deployed. This view is especially helpful to stakeholders to show them how the final delivered system will look. In these diagrams, hardware is referred to as a node and software is referred to as an artifact. These diagrams evolve over the lifetime of the project and contain enough information to support the systems needs [3].

A picture is worth 1024 words! The title of chapter 12 in Software Requirements. Wiegers and Beatty discuss the importance of using both textual and visual representations at different levels of abstraction to capture the full intentions of the intended system [8]. "Comparing the different representations of requirements reveals inconsistencies, ambiguities, assumptions, and omissions that would otherwise be difficult to spot" [8]. The authors stress that visual representations should augment and not replace natural language specifications [8].

Their book describes the following visual requirements models [8].

1. Data flow diagrams (DFDs)
2. Process flow diagrams
3. State-transition diagrams and state tables
4. Dialog maps
5. Decision tables and decision trees
6. Event-response tables
7. Feature trees (Ch 5)
8. Use case diagrams (Ch 8)
9. Activity diagrams (Ch 8)
10. Entity-relationship diagrams (Ch 13)

Wiegers and Beatty also briefly discuss the use of visual models in agile driven projects [8].

Weilkiens introduces a toolbox that can be used for modeling complex and distributed systems called SYSMOD. This toolbox helps to alleviate some of the problems due to rising complexity of systems. There is a great need to express component systems in a way that can be easily shared between team members. A shared language is important for being able to complete the design task [7].

The American Institute of Architects discuss the need for standard content and graphics to be used by Construction Architects to document their projects (Appendix E). [5]. Although software architecture and construction architecture are technically different fields, they share many common goals of turning stakeholder ideas into a real products.
The US CAD Standard (NCS) is a compilation of related documents published by several organizations for the purposes of creating a national standard for construction related CAD documents. [5].
A national CAD standard provides the following advantages [5]

1. Allows information to be transfered throughout the project cycle from one professional to another.
2. Results in better coordination between architects and engineers
3. Saves production time
4. Improves the overall design of the project

The Uniform Drawing System consist of 8 modules that help standardize the documents needed to convey information from the architect to the builder [5].

1. Module 1 - Drawing Set Organization
2. Module 2 - Sheet Organization
3. Module 3 - Schedules
4. Module 4 - Drafting Conventions
5. Module 5 - Terms and Abbreviations
6. Module 6 - Symbols
7. Module 7 - Notations
8. Module 8 - Code Conventions

Of particular interest is the recommendation of using of a drawing set hierarchy in the NCS.

Ingeno discusses methods for documenting and reviewing software architectures [2] In particular, he discusses

1. Uses of software architecture documentation
2. Creating architecture descriptions (ADs), including architecture views
3. Using UML to document software architecture
4. Reviewing software architecture documents

Other work that is somewhat related to Software Architecture documentation can be found by searching for articles related to Construction Architecture documentation. By replacing the verb construction with the verb software development, replacing the noun building with the noun application and replacing the noun material(s) with class or instance in these articles, one will see the striking resemblence between Software Architecture and Construction Architecture. One such article I found interesting can be found here

The Importance of Detailed Architectural Drawings in Modern Construction, Pinnacle Infotech, 2024

Bobek and Tversky presented a paper on how learning can be improved by using visuals [1]. They state that "creating visual explanations is superior to creating verbal ones. There are several notable differences between visual and verbal explanations; visual explanations map thought more directly than words and provide checks for completeness and coherence as well as a platform for inference, notably from structure to process." [1]

3. PROPOSED APPROACH

A table will be generated showing which format is recommended by each professional or professional association.

4. EXPERIMENTAL RESULTS

Table 1 summarizes the opinions of various experts in the field of technical documentation.

Table 1: Expert Recommendations for Document Formats

Expert	Text	Graphics	Both
Miles and Hamilton [3]			Yes
Wiegers and Beatty [8]			Yes
Weilkiens [7]			Yes
Ingeno [2]			Yes
Lamsweerde [6]			Yes
AIA [5]			Yes
NCS [4]			Yes

5. DISCUSSION

Table 1 shows that the majority of leading experts in software documentation agree that graphics, in conjunction with text, enhance the accuracy, completeness, and comprehensibility of software documents. It is encouraging to see current curriculum is CS studies including training on the use of UML diagrams in software documentation. The software industry would benefit from pushing this concept further by defining detailed drawing standards for using UML diagrams and other modeling techniques as is found in the NCS (National CAD Standard) papers and the AIA Standards for Graphics. For example, the creation of a National CAD Standard for Software Architecture (NCSSA) that mirrors the thoroughness of the NCS would be a good start. We must be aware that the creation of such a standard would take many years and require the input and buyin of many industry leaders to be successful.

6. THREATS TO VALIDITY

When we evaluate our findings while writing this paper, which explores the effectiveness or using both textual and graphical formats to improve communication between software architects and developers, we must acknowledge the limitations.

One challenge that this paper faces is the ambiguity of the definitions of "textual" and "graphical" documentation formats. Our study assumes that textual involves written natural language while graphical formats include diagrams such as UML that have been shown in this paper. However, Weigers and Beatty say that the integration of text and visuals is subject to varying interpretations, which can lead to inconsistent applications in real-world scenarios. [8]

The recommendations provided are based on existing literature which can include bias when selecting the sources. If all relevant works were not included in the studies, then it could limit the breadth of insight. The evaluation of diagrams in graphical format is subjective which means that different researchers may assess them based on their own individual experiences and project contexts.

The generalization of the findings could be limited due to individual basis in general software practices. The conclusions may not apply equally to all types of software projects, more specifically those specified in domains such as embedded systems, real-time systems, or exploratory research projects. Furthermore, the research does not account for the rapid evolution of documentation tools and their influence to the effectiveness of either text-based or graphical formats.

Some limitations of our study is that some of the conclusions come from existing literature and expert insight. This approach provides a foundation for understanding the topic but conducting new studies with direct observations or experiments would strengthen these findings.

7. CONCLUSIONS AND FUTURE WORK

Because natural language specifications can result in incomplete and/or inaccurate application builds, natural language only specifications should be limited to very few circumstances. Due to the fact that most applications need to both evolve over time and will eventually require maintenance, our conclusion is that the majority of software specifications need to include both natural language and graphical representations to fully document their functionality. This will lead to more complete and accurate implementation of specifications by software developers.

There are several opportunities for future work regarding software documentation. Here is a list of a few that we thought of.

1. Perform controlled studies were one group of software developers is given documents with text only and another group is given the same documents but are augmented with diagrams and graphics. Compare the results to see if there is a significant difference.

2. Examine CAD standards and determine how they may be applicable to software development documentation.

3. Examine software project failures and determine which percentage of those projects used documentation that included detailed diagrams.

8. ACKNOWLEDGMENTS

We would like to thank Dr. Kristen Walcott, Dr. Armin Moin and the Computer Science and Software Engineering Department at UCCS for helping us gain a deeper understanding of the challenges and opportunities in software architecture.

9. REFERENCES

[1] Bobek and Tversky. Creating visual explanations improves learning. Cogn Res Princ Implic. *National Library of Medicine*, 2016.

[2] J. Ingeno. *Software Architect's Handbook*. Packt Publishing, 2018.

[3] R. Miles and K. Hamilton. *Learning UML 2.0*. O'Reilly, 2006.

[4] NCS. National cad standard.

[5] A. I. of Architects. *Architectural Graphic Standards, 11th Edition*. John Wiley and Sons, Inc., 2007.

[6] A. van Lamsweerde. *Requirements Engineering*. Wiley, 2010.

[7] T. Weilkiens. *Systems Engineering with SysML and UML*. Morgan Kaufmann OMG Press, 2006.

[8] K. Wiegers and J. Beatty. *Software Requirements*. Microsoft Press, 2013.