Architectural Documentation

The Purpose of Architectural Documentation

- The documentation shall give the reader good understanding of the application's architecture and design.
- The reader shall be able to modify and extend the application without violating architecture or design.
- The documentation shall explain architectural decisions and show what has been considered. The reader shall *not* be led to reiterate discussions or considerations.

The Prototype is Also Part of the Documentation

- The documentation consists of two parts:
    1. A prototype consisting of a first version of the application under development. It shall contain enough code to explain the architecture and design. Note that this shall be production-quality code which has the standard (tested, commented, structured, etc.) we want the final application to have.
    2. A written documentation (often called architecture document) explaining the architecture and design.

The Documentation Must Be Clear and Concrete

- *What matters is that the architecture is explained as clear and precise as possible*.
- There are many templates for architecture documents, remember that they must be adapted! They are a great source for ideas, but do often contain unnecessary sections that must be removed. They might also lack sections that are actually needed.

How To Explain?

- A good approach is to start by writing body text, and then clarify the text with code snippets, UML diagrams, user interface screenshots and other images.
- Text and images complement each other, both are needed. Images shall be numbered, have captions and be explained in the body text.
- Assume that the reader has the same knowledge as those who wrote the prototype, except that she/he does not know anything about this particular application.
- Ask yourself what you would like to know if you were to take over maintenance of this application.
- Pay extra attention to thoroughly explain things you found hard to solve.

What to Write About?

- The documentation is divided into sections (*views*) explaining different aspects of the application. The following list is a suggestion, it is often necessary to add or remove views.
  - *Functionality*, what the user can do with the application.
  - *Design*, how the application is structured in layers, which patterns are used, important flows and components.
  - *Security*, for example authentication, authorization, encryption and logging.
  - *Data*, both OR-mapping and database design.
  - *Non-functional requirements*, describe all that are considered, e.g., transactions, usability, response time, availability.
  - *Deployment*, explains processes and hardware.
  - *Implementation*, describes the delivered files.
  - *Problems*, are there unsolved issues? Here you can also cover discarded solutions.

Functionality View

- The *functionality* section shall explain why the application exists.
- Explain what the user can do with the application.
- Cover only the most important features.
- Illustrate with user interface screenshots.

Design View

- The *design* section shall explain architecture and design.
- Explain all layers.
- Illustrate with UML diagrams that show major components and how they communicate with each other. Do not forget sequence (or communication) diagrams.
- If any part of the design is especially tricky it can be explained with text and diagrams in a separate subsection.

Design View (cont'd)

- It is often good to include code snippets, but choose carefully which parts of the code to show. You may also want to remove some lines, e.g., exception handling. If so, mention that the code is not complete.
  - Code should be treated like images, i.e., placed in a figure with number and caption, and explained in body text.
- Explain important architectural patterns and design patterns.
- Explain how to add new functionality. The architecture should not be destroyed by new developers taking over. Maybe give a concrete example of adding new functionality.

Security View

- The *security* section shall explain all security solutions.
- Indicate if a problem is known but not solved.
- Also mention which security issues have not been considered at all.

Data View

- The *data* section shall explain how data is stored and how the application accesses the data, i.e., integration layer and OR-mapping.
- Illustrate with interaction diagrams, class diagrams and/or code snippets.

Deployment View

- The *deployment* section shall explain how the program is divided in different processes running on different hardware, and how these components communicate.
- Describe the physical nodes (hardware) and the components running on these, e.g., web server, database server. Also describe how they communicate.
- Illustrate with deployment diagram.

Implementation View

- The *implementation* section shall explain which files the application consists of and what to do with these files, for example to install and start the application.
- Explain exactly how the program is compiled, installed, deployed and started. Mention if the executable files are compiled from the delivered source code.
  - Remember to test run your instructions.

Implementation View (cont'd)

- Explain what each unzipped directory contain.
- Specify whether something else, e.g., Java, must be installed. If so, specify which version is required.
- Indicate hardware and software requirements, such as memory, CPU and supported operating systems.

Problems View

- The *problems* section shall cover remaining unsolved issues. Also include descriptions of discarded solutions that do not fit anywhere else in the document.
- For each problem, state problem, solution and rejected solutions. If rejected solutions are not properly described there is a risk they will be reconsidered again in the future.
- Illustrate with code snippets, UML diagrams or user interface screenshots.
- Do not invent problems, but do not hesitate to mention those that actually appeared.

A Sample Architecture Document

1. Title Page with author, application name, organization name.
2. Revision History
3. Table of Content
4. Introduction, briefly explains the application and why and where it is developed.
5. Functionality View
6. Design View
7. Security View
8. Data View
9. Non-Functional View
10. Deployment View
11. Implementation View
12. Problems
13. References

Common Mistakes

- No or irrelevant images.
  - Images make it a lot easier to understand the body text, but they must be relevant and referenced in the body text.
- Very little text.
  - Even though images enhance understanding they can never substitute body text.
- Too big UML diagrams.
  - UML diagrams that are generated from the code by a tool will include *all* the application's details. Such diagrams obscure the important parts unless they are edited.

Common Mistakes (cont'd)

- No or to few user interface screenshots
  - Such images should be included everywhere there is a reference to the user interface. They make it a lot easier to understand.
- No code snippets
  - Sometimes the text is better illustrated with a code snippet than with a UML diagram. This is true especially when the text describes a programming problem instead of design. Choose carefully which parts of the code to include. You may also want to remove some lines, e.g., exception handling. If so, mention that the code is not complete.
  - Code should be treated like images, i.e., placed in a figure with number and caption, and explained in body text.

Common Mistakes (cont'd)

- No interaction diagrams
  - Most people find it harder to draw interaction diagrams than class diagrams. However, interaction diagrams are important and facilitate understanding a lot.
- Irrelevant information
  - Make sure not to add information just to make the document look fancier or to follow a particular template. It is very important to be precise and concrete throughout the document.