documentation include:

- Requirements – Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what will be or has been implemented.
- Architecture/Design – Overview of software. Includes relations to an environment and construction principles to be used in design of software components.
- Technical – Documentation of code, algorithms, interfaces, and APIs.
- End user – Manuals for the end-user, system administrators and support staff.

- Marketing – How to market the product and analysis of the market demand.

Requirements documentation

Requirements documentation is the description of what a particular software does or shall do. It is used throughout development to communicate how the software functions or how it is intended to operate. It is also used as an agreement or as the foundation for agreement on what the software will do. Requirements are produced and consumed by everyone involved in the production of software, including: end users, customers, project managers, sales, marketing, software architects, usability engineers, interaction designers, developers, and testers.

Requirements comes in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started by right-clicking a configuration file and select the 'build' function*), and anything in between. They can be specified as statements in natural language, as drawn figures, as detailed mathematical formulas, and as a combination of them all.

The variation and complexity of requirements documentation makes it a proven challenge. Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people who shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult — and therefore more error prone (decreased software quality) and time-consuming (expensive).

The need for requirements documentation is typically related to the complexity of the product, the impact of the product, and the life expectancy of the software. If the software is very complex or developed by many people (e.g., mobile phone software), requirements can help to better communicate what to achieve. If the software is safety-critical and can have negative impact on human life (e.g., nuclear power systems, medical equipment, mechanical equipment), more formal requirements documentation is often required. If the software is expected to live for only a month or two (e.g., very small mobile phone applications developed specifically for a certain campaign) very little requirements documentation may be needed. If the software is a first release that is later built upon, requirements documentation is very helpful when managing the change of the software and verifying that nothing has been broken in the software when it is modified.

Traditionally, requirements are specified in requirements documents (e.g. using word processing applications and spreadsheet applications). To manage the increased complexity and changing nature of requirements documentation (and software documentation in general), database-centric systems and special-purpose requirements management tools are advocated.

Architecture design documentation

Architecture documentation (also known as software architecture description) is a special type of design document. In a way, architecture documents are third derivative from the code (design document being second derivative, and code documents being first). Very little in the architecture documents is specific to the code itself. These documents do not describe how to program a particular routine, or even why that particular routine exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation. It may suggest approaches for lower level design, but leave the actual exploration trade studies to other documents.

Another type of design document is the comparison document, or trade study.

This would often take the form of a *whitepaper*. It focuses on one specific aspect of the system and suggests alternate approaches. It could be at the user interface, code, design, or even architectural level. It will outline what the situation is, describe one or more alternatives, and enumerate the pros and cons of each. A good trade study document is heavy on research, expresses its idea clearly (without relying heavily on obtuse jargon to dazzle the reader), and most importantly is impartial. It should honestly and clearly explain the costs of whatever solution it offers as best. The objective of a trade study is to devise the best solution, rather than to push a particular point of view. It is perfectly acceptable to state no conclusion, or to conclude that none of the alternatives are sufficiently better than the baseline to warrant a change. It should be approached as a scientific endeavor, not as a marketing technique.

A very important part of the design document in enterprise software development is the Database Design Document (DDD). It contains Conceptual, Logical, and Physical Design Elements. The DDD includes the formal information that the people who interact with the database need. The purpose of preparing it is to create a common source to be used by all players within the scene. The potential users are:

- Database designer
- Database developer
- Database administrator
- Application designer
- Application developer

When talking about Relational Database Systems, the document should include following parts:

- Entity - Relationship Schema (enhanced or not), including following information and their clear definitions:
    - Entity Sets and their attributes

    - Relationships and their attributes
    - Candidate keys for each entity set
    - Attribute and Tuple based constraints
- Relational Schema, including following information:
    - Tables, Attributes, and their properties
    - Views
    - Constraints such as primary keys, foreign keys,
    - Cardinality of referential constraints
    - Cascading Policy for referential constraints

    - Primary keys

It is very important to include all information that is to be used by all actors in the scene. It is also very important to update the documents as any change occurs in the database as well.

Technical documentation

It is important for the code documents associated with the source code (which may include README files and API documentation) to be thorough, but not so verbose that it becomes overly time-consuming or difficult to maintain them. Various how-to and overview documentation guides are commonly found specific to the software application or software product being documented by API writers. This documentation may be used by developers, testers, and also end-users. Today, a lot of high-end applications are seen in the fields of power, energy, transportation, networks, aerospace, safety, security, industry automation, and a variety of other domains. Technical documentation has become important within such organizations as the basic and advanced level of information may change over a period of time with architecture changes.

Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class.

Technical documentation embedded in source code

Often, tools such as Doxygen, NDoc, Visual Expert, Javadoc, JSDoc, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, or Universal Report can be used to auto-generate the code documents—that is, they extract the comments and software contracts, where available, from the source code and create reference manuals in such forms as text or HTML files.

The idea of auto-generating documentation is attractive to programmers for various reasons. For example, because it is extracted from the source code itself (for example, through comments), the programmer can write it while referring to the code, and use the same tools used to create the source code to make the documentation. This makes it much easier to keep the documentation up-to-date.

Of course, a downside is that only programmers can edit this kind of documentation, and it depends on them to refresh the output (for example, by running a cron job to update the documents nightly). Some would characterize this as a pro rather than a con.

Literate programming

Respected computer scientist Donald Knuth has noted that documentation can be a very difficult afterthought process and has advocated literate programming, written at the same time and location as the source code and extracted by automatic means. The programming languages Haskell and CoffeeScript have built-in support for a simple form of literate programming, but this support is not widely used.

Elucidative programming

Elucidative Programming is the result of practical applications of Literate Programming in real programming contexts. The Elucidative paradigm proposes that source code and documentation be stored separately.

Often, software developers need to be able to create and access information that is not going to be part of the source file itself. Such annotations are usually part of several software development activities, such as code walks and porting, where third party source code is analysed in a functional way. Annotations can therefore help the developer during any stage of software development where a formal documentation system would hinder progress.

User documentation

Unlike code documents, user documents simply describe how a program is used.

In the case of a software library, the code documents and user documents could in some cases be effectively equivalent and worth conjoining, but for a general application this is not often true.

Typically, the user documentation describes each feature of the program, and assists the user in realizing these features. It is very important for user documents to not be confusing, and for them to be up to date. User documents don't need to be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very valuable. User documentation is considered to constitute a contract specifying what the software will do. API Writers are very well accomplished towards writing good user documents as they would be well aware of the software architecture and programming techniques used. See also technical writing.

User documentation can be produced in a variety of online and print formats.[1] However, there are three broad ways in which user documentation can be organized.

1. **Tutorial:** A tutorial approach is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks.[2]
2. **Thematic:** A thematic approach, where chapters or sections concentrate on one particular area

of interest, is of more general use to an intermediate user. Some authors prefer to convey their ideas through a knowledge based article to facilitate the user needs. This approach is usually practiced by a dynamic industry, such as Information technology.[3]

3. **List or Reference:** The final type of organizing principle is one in which commands or tasks are simply listed alphabetically or logically grouped, often via cross-referenced indexes. This latter approach is of greater use to advanced users who know exactly what sort of information they are looking for.

A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two. It is common to limit provided software documentation for personal computers to online help that give only reference information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a program is left to private publishers, who are often given significant assistance by the software developer.

Composing user documentation

...

Like other forms of technical documentation, good user documentation benefits from an organized process of development. In the case of user documentation, the process as it commonly occurs in industry consists of five steps:[4]

1. User analysis, the basic research phase of the process.[5]
2. Planning, or the actual documentation phase.[6]
3. Draft review, a self-explanatory phase where feedback is sought on the draft composed in the previous step.[7]

4. Usability testing, whereby the usability of the document is tested empirically.[8]
5. Editing, the final step in which the information collected in steps three and four is used to produce the final draft.

Documentation and agile development controversy

"The resistance to documentation among developers is well known and needs no emphasis."[9] This situation is particularly prevalent in agile software development because these methodologies try to avoid any unnecessary activities that do not directly bring value. Specifically, the

Agile Manifesto advocates valuing "working software over comprehensive documentation", which could be interpreted cynically as "We want to spend all our time coding. Remember, real programmers don't write documentation."[10]

A survey among software engineering experts revealed, however, that documentation is by no means considered unnecessary in agile development. Yet it is acknowledged that there are motivational problems in development, and that documentation methods tailored to agile development (e.g. through Reputation systems and Gamification) may be needed.[11][12]

Marketing documentation

For many applications it is necessary to have some promotional materials to encourage casual observers to spend more time learning about the product. This form of documentation has three purposes:

1. To excite the potential user about the product and instill in them a desire for becoming more involved with it.
2. To inform them about what exactly the product does, so that their

expectations are in line with what they will be receiving.

3. To explain the position of this product with respect to other alternatives.

See also

- API Writer
- Comparison of documentation generators
- Design by contract
- Design document
- Docstring
- Documentation
- Literate programming
- README files

- User Assistance
- Unified Modeling Language UML

Notes

1. *RH Earle, MA Rosso, KE Alexander (2015) User preferences of software documentation genres. Proceedings of the 33rd Annual International Conference on the Design of Communication (ACM SIGDOC).*
2. *Woelz, Carlos. "The KDE Documentation Primer" . Retrieved 15 June 2009.*
3. *Microsoft. "Knowledge Base Articles for Driver Development" . Retrieved 15 June 2009.*

4. *Thomas T. Barker, Writing Software Documentation , Preface, xxiv. Part of the Allyn & Bacon Series in Technical Communication, 2nd ed. Upper Saddle River: Pearson Education, 2003. ISBN 0321103289 Archived May 13, 2013, at the Wayback Machine*
5. *Barker, pg. 118.*
6. *Barker, pg. 173.*
7. *Barker, pg. 217.*
8. *Barker, pg. 240.*
9. *Herbsleb, James D. and Moitra, Dependra. In: IEEE Software, vol. 18, no. 2, pp. 16-20, Mar/Apr 2001*
10. *Rakitin, Steven. , "Manifesto elicits cynicism." IEEE Computer, vol. 34, no. 12, p. 4, 2001*

11. *Prause, Christian R., and Zoya Durdik. "Architectural design and documentation: Waste in agile development?" In: International Conference on Software and System Process (ICSSP), IEEE, 2012.*
12. *Selic, Bran. "Agile documentation, anyone?" In: IEEE Software, vol. 26, no. 6, pp. 11-12, 2009*