Interoperability of Software Documents

Ethan V. Munson

December 15, 1993

Abstract

The software development process produces a diverse collection of documents ranging from requirements specifications to architecture diagrams to program source code to bug reports. Some are written in formal languages, but others, while highly structured, are written in natural language. The content of these documents is interconnected in complex ways. For instance, a *change report* might describe how *source code* was modified to conform to a *design specification*, the problem having been identified because *test output* was incorrect. The current state of the art in software development environments uses many different, and often incompatible, tools to manage these different documents. Many significant advances in interaction for the software development process will not be possible until all the documents it produces interoperate.

1 Software documents

Large software projects produce an impressive array of documents. In a sense, program source code is the most important type of document produced, since it is the most direct expression of the nature of the program itself. But it is often the case that the majority of software documents are not program source code. Table 1 shows a number of examples, categorized by the phases of the "waterfall" model of the software development process [2]. Some of these documents are descriptions of the program (e.g. user documentation and architectural design). Others describe the process

To appear in the Proceedings of the Workshop on Software Engineering and Human-Computer Interaction, Sorrento, Italy, May 1994.

Phase	Document Types
Requirements	system definition project plan software requirements specification
Design	architectural design specification detailed design specification preliminary user's manual software verification plan
Implementation	program source code walkthrough and inspection reports
Testing	test plans test data testing scripts bug reports
Maintenance	bug reports change reports program source code errata for user documentation

Table 1: Software document types categorized by the phases of the waterfall model of the software process. (Some of the document types listed here were taken from [2].)

of producing the program (e.g. project plan and walkthrough reports). Certain documents describe how other documents change over time (e.g. change reports and errata).

2 Interconnectedness of Software Documents

What stands out about these documents, when taken as a whole, is their interconnectedness. Admittedly, a few documents from the requirements phase make sense in isolation, but

- Design documents are motivated by material in requirements documents;

- Implementation documents are a more concrete expression of the ideas in design documents;
- Test phase documents are intended to make sure that certain implementation documents (particularly source code) meet the requirements and design specifications;
- Maintenance documents describe the process of maintaining the documents from the implementation phase.

These are the obvious, more-or-less linear relationships between the documents and phases. In practice, most software is developed in a series of design-build-evaluate cycles. The evaluation stage of one cycle motivates changes to the design and implementation of the next cycle. Thus, software documents have other interconnections that reflect the relationship between the cycles.

Ideally, each member of the community working on a software project should have fast, interactive access to on-line versions of every document produced by the project. As an example, a programmer working on a module, without leaving the workstation, should be able to find

- the design documents for the module,
- the higher-level design documents describing how the module will be used by other parts of the program,
- and the requirements documents that motivated the design.

Navigation among these documents could be performed by following links in the manner of hypertext. In fact, where it was appropriate, documents could include fragments of other documents. For example, a program source file could include fragments of the relevant design documents as part of its internal documentation. Furthermore, these included fragments could be *active*, so that when the design changed, they would either be updated automatically or their appearance could change in a manner signaling a possible conflict between the new design and the current source code.

Unfortunately, exploiting even simple interconnections between software documents is rarely possible. Typically, each division of the development team uses a document tool well-suited to the division's task, but incompatible with the tools used by other divisions of the team. Some tools that might be used are: program editors (for programs), easy-to-use word-processing programs (for internal natural language documents), and high-quality document composition systems (for external documents). While there exist some formats (e.g. ASCII text and PostScript) that can be used for interchange of data between these systems, a great deal of information and functionality is sacrificed in the interchange process.

The inability to exploit the connections among the various software documents is a significant hindrance to the development of better human-computer interfaces for software. In building a software system, one of the key difficulties is managing the large amount of information that is available, often in the form of documents. If the natural connections between the documents can be used as a navigational framework, it should be much easier to construct advanced interfaces.

3 Interoperability

The solution is to make software documents interoperate. If all software documents share a common framework that possesses sufficient functionality, it should be much easier to construct and experiment with advanced interfaces. Of course, if interoperability were easy to achieve, we would already have it. Instead, we must settle for a clear description of the kind of interoperability we desire and an understanding of the technical challenges that lie along the path to achieving it.

These are my desiderata for interoperability of software documents:

Support for structure: All documents have structure. In some documents, the structure is so simple as to be degenerate, but most software documents have structure of at least moderate complexity. Some software documents have the standard technical document structure of sections, paragraphs, and sentences. Others, such as bug reports, have a specialized structure all their own. Program source code (along with certain other formalized language documents) has the special quality that its structure can be determined by analysis of its content.

Whatever a document's structure, it can be exploited to create better interfaces. Structure can be used to support navigation both by direct traversal of the structure and by queries that exploit structural information (e.g. "find all bug reports issued by John Doe" or "find the declaration of the global variable x"). Structure can also be used as the basis for presentation. For example, a pretty-printing service uses the structure of a program to determine how the program is formatted.

Support for exploiting connections: It must be possible to take advantage of the connections between software documents. One way is through the use of hyperlinks (from hypertext or hypermedia), which allow navigation between arbitrary points in the same document or in separate documents. Another way is through active inclusions. If every software document can include arbitrary portions of other documents, then it should be easier to maintain correspondences between the documents. If the included portions are active, they can be automatically updated, if that is appropriate.

Tolerance for multimedia: Currently, many software documents use multiple static media (text, graphics, tables, raster images). The key exception to this rule has been program source code, which has been limited to a very restricted model of text. Research on literate programming [6, 11] has made a case for the benefits of applying high-quality document formatting techniques to program text. Further benefits will be derived if programs and all other software documents can *include* a wide variety of static and dynamic media. This does not require a complete reworking of our model of programs. Rather, our systems for handling programs (and other specialized document types) should be able to tolerate the presence of multimedia data. If our support of interoperability is general enough, this requirement may be easy to meet, because documents drawn from all media will interoperate.

Support for large systems: Any mechanisms for document interoperability must scale up to meet the needs of very large systems. It is the very largest systems whose complexity most strongly demands better interfaces. Thus, solutions that do not scale up sufficiently will probably never be used.

There are many technical obstacles to achieving the goal of software document interoperability. The following are some of the issues that must be addressed:

System Architecture: There are three basic approaches to building a system or collection of systems which provide interoperability.

Integrated: One approach is to construct a single integrated system which handles all document types. This approach is likely to provide uniform mechanisms for handling all document types. It is also much easier to share services and information between modules in an integrated system. A disadvantage to this approach is that it forces all users to work with the same large system. It is difficult to construct a single system that do a good job of meeting the specialized needs of every particular group of users.

Encoding Standards: Another approach is to store all documents using an encoding standard of sufficient generality to handle all types of software documents. The encoding standard becomes the common denominator for the various software document tools. Examples of such standards are SGML [3], ODA [5] and HyTime [8] (an extension of SGML suitable for hypermedia documents). The problem with this approach is that it may impose a complex and verbose storage representation on tools whose task is fundamentally quite simple.

Tool-based: A final approach involves the construction of small, specialized tools that cooperate. These tools might be similar in function to the incompatible programs currently used, but they would use a common communication protocol to cooperate. The tools can be carefully designed to meet the needs of their users and their relatively small size should reduce computing infrastructure requirements. Furthermore, unlike the encoding standards approach, the communication between pairs of tools can be tuned to improve efficiency. Two different ways to build tool-based systems have been explored in Tcl/Tk [9, 10] and OLE [7].

The problem with the tool-based approach is that it is hard to guarantee that the tools really share common models, particularly if they evolve independently. Also, fine-grained sharing is typically more difficult than in an integrated system.

These three approaches are not mutually exclusive. For instance, an encoding standard can be used as the lowest common denominator for document interoperability among a set of cooperating tools. But when circumstances require it, pairs of tools can use more efficient coding schemes for communication or sharing. Furthermore, the tools can share code for common operations in the form of various libraries. If the libraries become powerful enough, it may be hard to tell the difference between the tool-based and integrated approaches.

Formalized Language Support: Perhaps the hardest technical problem is providing advanced support for formalized languages (programs and specifications). It is very difficult to provide incremental program analysis services with interactive performance. Such services must share large amounts of fine-grained data, which is typically easiest to do in an integrated system. It is even more difficult to build a system that gracefully handles error states, a common situation if text editing is not restricted by the program [1].

Structured Documents: One model which has the potential to provide software document interoperability is the structured document model. Structured documents have hierarchical structure which is specified separately from the document instances (using grammar-like specifications). The model applies equally well to both formalized language and natural language documents and forms the basis of the encoding standards mentioned earlier (SGML, ODA, and HyTime).

One of the key benefits of the structured document model is that it separates document presentation from document content. Thus, it is possible to view the same document different ways, depending on current needs. One of the focuses of my own work on the Ensemble environment has been to provide support for multiple presentations of the same document [4].

While the structured document model can be applied to both programs and natural language documents, there are some problems. First, efficient incremental program analysis requires a much more complex tree model than is necessary for natural language documents. While such a tree model is both space and time efficient, its construction is a demanding engineering task. Second, the best way to make programs able to include non-program documents is not yet clear. One method would be to modify the program's grammar to explicitly allow inclusion of non-program documents at certain points (e.g. between statements but not in the middle of expressions). Another approach would be to create a special type of document which can intermix the program's text stream and arbitrary fragments of multimedia documentation.

4 Conclusions

Ultimately, every document produced in the software development process discusses the same topic: the program being developed. Because of this shared topic, the content of the various documents is highly interconnected. Yet these connections can rarely be exploited because the software currently used to produce the different documents is often incompatible. This obstacle to the creation of advanced interfaces can be removed by making software documents interoperable.

I have argued that any approach to interoperability must take advantage of the natural structure of software documents, exploit the connections among the various documents, tolerate multimedia, and scale up to meet the needs of large systems.

We, as computer scientists and software engineers, have a compelling interest in improving the software development process, which makes it a topic of great interest to us. Sometimess our work has little application outside our own domain. However, improvements in document interoperability should also benefit domains outside computer science. Other engineering domains have the same general problems with managing complex sets of documents covering the design, construction, and evaluation of their particular engineering artifacts. Furthermore, certain collections of business documents also share the quality of interconnectedness. Thus, document interoperability could benefit computer users outside the engineering fields.

References

[1] Robert A Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):96–127, January 1992.

[2] Richard Fairley. *Software Engineering Concepts*. Series in Software Engineering and Technology. McGraw-Hill, New York, 1985.

[3] Charles F. Goldfarb, editor. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879.

[4] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT*

Fifth Symposium on Software Development Environments, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[5] International Standards Organization. *Office Document Architecture*, 1986. Draft International Standard 8813.

[6] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[7] Microsoft Corporation, Redmond, WA. *Object Linking & Embedding, Programmer's Reference, Version 2.0 (Pre-Release)*, 1993.

[8] Stephen R. Newcomb, Neill A. Kipp, and Victoria T. Newcomb. The HyTime hypermedia/time-based document structuring language. *Communications of the ACM*, 34(11):67–83, November 1991.

[9] John Ousterhout. Tcl: An embeddable command language. In *1990 Winter USENIX Conference Proceedings*, 1990.

[10] John Ousterhout. An X11 toolkit based on the Tcl language. In *1991 Winter USENIX Conference Proceedings*, pages 105–115, 1991.

[11] Norman Ramsey and Carla Marceau. Literate programming on a team project. *Software — Practice and Experience*, 21(7):677–683, July 1991.