How Changes are Reflected in a Software Architecture

by Dave Busse

How Changes Are Reflected in a Software Architecture

CS8503 Special Project

Dave Busse

August, 1995

1. Introduction

This paper briefly explores what happens to a Software Architecture during maintenance. Or more specifically, when changes are made to an existing system that have a system wide impact, are these changes always reflected in the Software Architecture? An existing student system is presented with its software architecture. Changes to this system are described along with their effects to the architecture. It is shown that some changes that should be reflected in the architectural model are not present. An alternate model for the software architecture that seems to address this problem, is discussed.

2. Software Architecture

Software Architecture refers to a high level of software design. Software architectures can be specified and represented in different ways. This gives rise to the idea of different families of software architecture models. The most common are the structural models[1]. These models are often expressed as graph containing components and connectors. In such a software architecture diagram the components represent the computation (processes or data, depending on shape). The connectors describe the interaction between the components and are usually drawn as lines. In addition, various architectural styles are possible as described by Garland and Shaw[2]. Initially, the architecture for the example software system is a structural type shared data architecture as shown in Figure 1 on page 8.

3. Citation

The example software system used here is called Citation. The Citation system translates bibliographies from one type to another. Types currently supported as of this writing are: refer, bibtex, an Oracle database and the GTEC and COMP formats from GTEL (Georgia Tech Electronic Library). Work is currently going on to add an e-mail format. Citation was originally written as a project by the students of CS8113J: Software Generation, Testing and Maintenance in the summer quarter of 1993. It was enhanced by the students of that course in 1994. Its software architecture was documented as part of another course: Principles and Application of Software Design in the Spring quarter of 1995. This summer, the students of Software Generation, Testing and Maintenance are again, enhancing Citation. This brief history serves to show that Citation, like most “real” software systems, undergoes change.

Citation’s software architecture is shown in Figure 1 on page 8. There are three kinds of components shown. The single walled boxes are processes, the thick walled boxes represent data (bibliography formats) and the double walled box is a repository that holds an intermediate representation of a bibliography. The connectors in this architecture are the arrows that represent control and lines that represent data flow.

This design breaks CITATION down into modules using data hiding as one criteria. This is implemented in the following way. The parts of the software that access the repository are implemented in part with the Interface Description Language (IDL)[3] . IDL is a specification language used for describing data structures that are passed from one cooperating process to another. The language represents data in classes and nodes (structures) as well as descriptions of the processes that use the data. Tools exist to translate the specifications into various standard programming languages such as C. The resulting translated code not only contains structures to hold the data but also routines and macros to do I/O and to perform various manipulations on the data. Thus a certain degree of data-hiding is provided. The design has been modularized using access to the internal representation as a criteria, similar to the criteria suggested by Parnas[4] .

4. Changes

In this section several proposed changes to CITATION are described. The changes can be categorized by how they would effect the detailed design of the software. We can then imagine what the results would be to the architecture by abstracting out details from the detailed design level until we are at the software architecture level. Some changes would clearly be pervasive in implementation across the whole system. Such changes should be reflected in the software architecture.

Several changes to CITATION have been suggested. They are:

1. IDL Restructuring. This is a change to the format of the internal representation. The IDL language supports the idea of classes. Currently, the class hierarchy in the IR is used to define the type of reference. The proposed change is to build a class hierarchy with an abstract superclass that contains all common bibliographic items and a subclass for each supported bibliographic type. The kind of reference is encoded as an attribute of the classes that use it (e.g. bibtex).
2. The Batching Problem. When converting from a bibliography to the internal representation, the whole translation occurs in memory. Should an input reference be found in error, the program quits. This yields an all or nothing situation. The proposed change is to consider a bibliography a batch of references. Each reference in the batch is translated and written to the output. If an error is found in a reference, the

error is reported, the faulty reference is skipped and the program reads the next reference.

3. Robust Error Handling. This is a more general case of the Batching problem. What is proposed is an error handling module that reports problems of different severity levels and allows the program to continue if appropriate.
4. Fan-In/Fan-Out Substitution Problem. Certain bibliographic items do not correspond directly with items in other bibliographic types. An example is a publisher in one bibliographic type may not have a corresponding publisher item in the target bibliographic type. However, the target may have an item with a different name that may be appropriate, such as organization. This change proposes to address this problem by defining some sort of protocol. The protocol would allow a translator to convert an item found in the input IR to a reference item "close" to the same meaning in the target reference.
5. Change to add a new bibliographic type. This is the type of change expected if the software is successful; addition of a new feature.

IDL Restructuring affects every component and connector involved with the IR Repository, an IR bibliography or an IR reference. This constitutes at least half of the components and connectors. In spite of this system wide change, there is no way to express it in the component and connector type of architecture. What is needed is a way to express a change in the protocols between components.

The solution to the Batching problem involves changes the way control is handled. If error handling is to be part of the function of the "get a bibref" or "get a IR Ref" component we would need to add an error handling component that would handle error messages. Control would be passed to it and then back to the original component. This sort of change does show up in the Citation diagram.

The general error handling also requires a system wide change, adding of an error handling component usable by all other components. This could be expressed using the diagram notation But we need a further addition to represent how control is passed. Control may or may not be returned to the original component depending on the type of error. This is a kind of connector protocol, and the diagramming notation has no way to express it.

The Fan-In/Fan-out Substitution change is hidden in the transcribe to bib reference component. It can be solved without a system wide change and is an example of a change that would not show up in an architectural diagram. We will consider it no further here.

Adding a new bibliographic type to be translated does not show up as a change to the architecture. This change is an addition of a supported bibliographic type. Supported bibliographic type is one of the details that was abstracted out when this architecture was drawn. The architecture does show what the new feature will have to do. It does not convey exactly how to do it or what existing components or connectors might be reused.

Each of the changes that have a system wide effect fails to show up fully in the architecture of the system. This is because the component and connector notation have no way to express the protocols present in the connectors.

5. A Different Architecture

As mentioned above, a number of architectural models have been identified[1]. These six types of software architectural models are:

1. Structural Models. Architecture is viewed as being composed of components and connectors and “other stuff”. The “other stuff” is used to document important semantics about the architecture, but it is currently not well defined.
2. Framework Models. These are similar to structured models but put more emphasis on targeting a specific problem domain. A framework is an object oriented model. It consists of a set of classes (usually abstract) that forms an abstract design for a set of related problems. At this time there is not one standard notation to describe all aspects of a framework. However, all framework notation has class diagramming in common.
3. Dynamic Models. As the name suggests, dynamic models describe the behavior of a system at a high level. The changes described may have to do the computation or changes in the system configuration or other changes that characterize the software.
4. Process Models. These models describe the construction steps or processes that go into constructing the software system. The architecture is end product of following the process.
5. Functional Models. These models view a software architecture as a hierarchy of functional layers that supply services upward through that hierarchy.
6. Holistic Models. This is a view of software architecture that combines it with other design artifacts. It is probably an attempt to capture all the relevant semantics about the software but suffers from the effect of blurring the distinction between software architecture and other design artifacts.

Frameworks are used for the new Citation architecture. The development of frameworks has not yet reached the point where there is an agreed upon notation for completely describing them. One way of documenting frameworks is by use of patterns[5]. Patterns document a framework in a cookbook type manner. Patterns are arranged in a tree or graph, with the root giving the most general description of the problem domain the framework addresses. The edges of the graph are pointers to other patterns. The further along the graph one progresses the more detailed the documentation. The format of each pattern consists of three sections: a description of the purpose of the pattern, a description of how to use the pattern (with examples) and a description of the design. Patterns are considered part of the design of a system rather than part of architecture. Instead, informal descriptions of frameworks are often used as documentation in architecture[6]. Next we document the Citation framework in an informal way. Even though this sort of documentation is weak, it still serves the purpose well enough to demonstrate some advantage over the structured model. The possibility of a different approach is mentioned in section 7.

The architecture of Citation is shown as the bibXlate framework in Figure 2 on page 9. This is a class diagram using OMT[7] showing the differing classes and their interfaces that make up the Citation architecture. In addition some concrete classes are shown that illustrate the use of the framework, but are not, strictly speaking, part of the framework. These classes are shown with dashed line boxes. In the bibXlate framework the Translator abstract object specifies a protocol (the start message) that must be implemented by all concrete subclasses that might be created. Examples are refer2ir, ir2refer, bibtex2ir, ir2bibtex, gtec2ir, ora2ir and ir2ora. In the diagram the refer2ir and ir2refer concrete classes are shown as examples. A start method must create source and target bibliographic objects. Then one of translate_to_ir or translate_from_ir methods in the "user" bibliography must be called depending on the direction of the translation. The "translate" methods are declared in the abstract class user_bibliography. They must be implemented in any concrete subclass, for example bibtex_bibliography, refer_bibliography, gtel_bibliography or ora_bibliography. The translate_to_ir method, for example, makes use of the get_ref method from the bibliography class, the self2ir method of the user_reference class and the append_ref of the bibliographic class. In addition the report message from the error_reporting class may also be used.

Together, the framework object diagram and the above discussion form an architectural description that will more closely reflect the types of system-wide changes we used in our examples.

The IDL Restructuring change is addressed in this framework. The IR restructuring is plainly shown in the object diagram. It is explicit in the class hierarchy showing the "IR reference" class. The IR Reference holds attributes that are common to all reference types that are converted to the intermediate representation. There is a subclass for the unique attributes of each bibliographic type. The example shown is the refer_ir class that holds attributes unique to the refer references. The object oriented design of Citation for this change would show subclasses for the ir_reference class for all supported bibliographic types: bibtex_ir, refer_ir, ora_ir and gtel_ir.

As before, the solution to the Batching problem involves change to the way control is handled. Control is made explicit in the start method of the Translator abstract class. Each subclass of the Translator class must supply a start method that follows a general outline that was described in the documentation for the framework. This algorithm specifies that references be translated and written out one at a time. That errors be handled by sending a message to an error reporting object, skipping the faulty input reference and continuing to translate references until the bibliography is translated.

General error handling is made explicit by the Error_Reporting abstract class. Differing levels of errors can be created by subclassing Error_Reporting. The protocols of the error reporting classes make changes of this kind clear in the architecture.

Although patterns are not considered part of architecture, there inclusion as part of the design documentation would the problem of adding a new bibliographic type to be translated much simpler. This is because this is just the sort of problem that would be discussed in the patterns that describe the framework. Not only would changes of this type be evident, the patterns actually give instructions on how it should be accomplished. To reflect these changes in the architecture, we must either make it clear in the informal documentation and/or extend the object diagram with example concrete classes. Care must be taken to distinguish between architectural elements and concrete classes that are actually part of a specific implementation rather than part of the architecture. In the Citation framework object diagram examples are shown as boxes drawn with dashed lines. This indicates implementation concrete classes. Objects that are part of the framework architecture are drawn as boxes with solid lines.

6. Conclusions

It seems clear that it is easy to invent changes to existing systems that have system-wide effects but do not show up in the most commonly used architectural model. The framework architectural model seems to better reflect our example changes, but it is not without some difficulties of its own. Both types of architecture suffer from not having a well-defined formal description notation. However, the framework approach still seems more robust in reflecting changes made to a system and that alone should encourage its use.

7. Future work

There seem to be more questions than answers in this document. The following are a list of issues that came up during work on this question. There was not enough time to pursue them, but they seem important enough to note here.

Building a good framework is a difficult process and is most successfully done bottom up and in an interative fashion[8] . The bibXlate framework probably suffers because it has not had the benefit of such development. It is the first framework the author has ever attempted and probably needs enhancement. For example, it is difficult to decide, where architecture leaves off and object design begins. The signatures of the methods in the framework object model are not detailed here. Only the names of the methods are given not their types or parameters. This is intentional. Is this good enough?

The major difficulty remains how best to document an architecture to show system wide change. Framework models are more expressive than structured architecture models for showing the protocols involved with connectors. However, frameworks need a well-defined description. Path expressions have been suggested as a way to document the protocols. Investigating this possibility would be a logical next step.

Two software architecture types, structural and framework have been compared here. There are other software architecture types as listed in section 5. An interesting project might be to develop matrix showing how well each does in the area of reflecting system wide changes.

Another possible project would be to address the problem of scale in the example. The student system suffers from the perceived fault of being too small or in some other way not a “real life” problem. A “real life” system could be found and the comparison of software architecture could be done using this larger example.

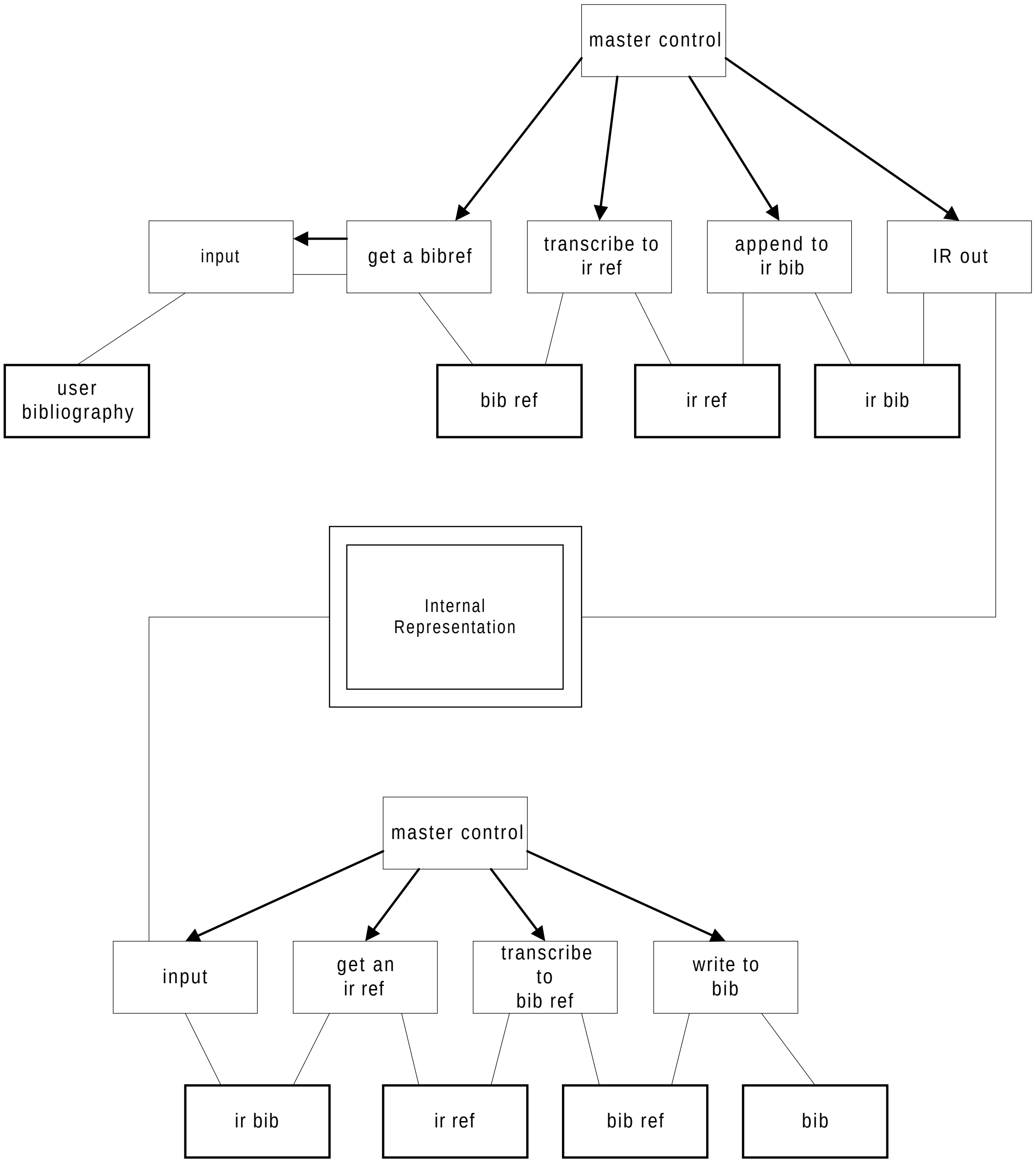

Figure 1: Citation Architecture

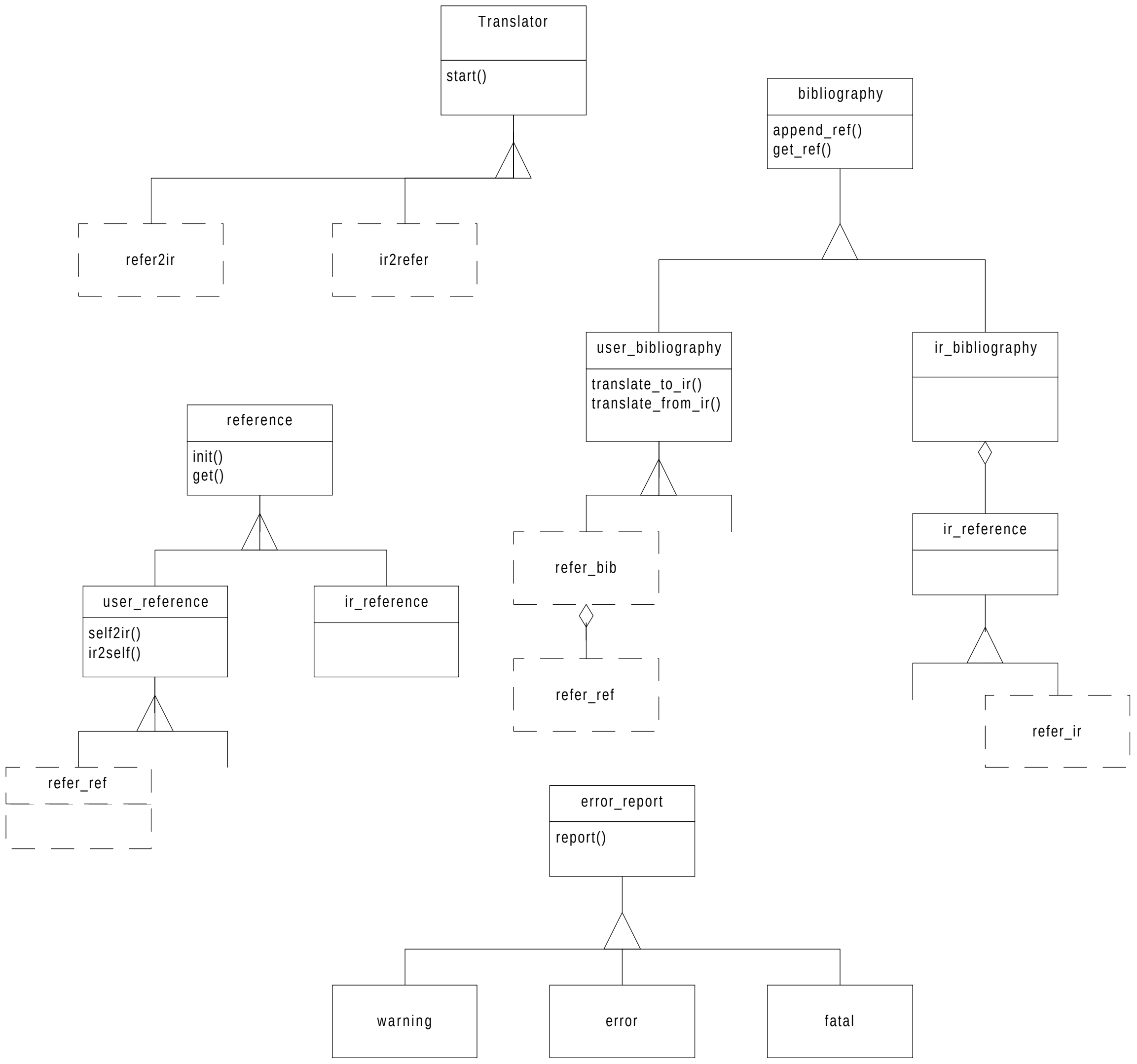

Figure 2: Framework for Citation

8. References

[1] David Garlan, editor, First International Workshop on Architectures for Software Systems Workshop Summary, *Software Engineering Notes*, Vol. 20, No 3, Pages 84-89, July 1995

[2] David Garlan, Mary Shaw, An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing Company, 1993

[3] Richard Snodgrass, *The Interface Description Laguage: Definition and Use*, Computer Science Press,1989

[4] L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol 15, No 12, Pages 1053-1058, December 1972

[5] Ralph E. Johnson, "Documenting Frameworks using Patterns," *Proceedings OOPSLA '92*, ACM SIGPLAN Notices, vol. 27, no. 10, Oct. 1992, pp. 63-76

[6] Ralph E. Johnson and Vincent F. Ruisso, "Reusing Object-Oriented Designs," UIUC DCS 91-1696, May 1991

[7] J. Rumbaugh, M. Blaha, W. Permerlani, F. Eddy and W. Lorenson. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991

[8] Ralph E. Johnson and Brian Foote, "Desgining Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22-35