JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 7, September-October 2005

Context

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

"Context" is the setting within which any statement is interpreted or any claim is verified. Reusable software is never actually reusable in every setting. There are constraints on the environment in which that software will be usable but often they are not stated explicitly. Even more often, the extent of the context is not known. In this issue of Strategic Software Engineering I will examine the strategic importance of context and some ways to manage its effects.

1 INTRODUCTION

In a recent paper, we illustrated that often the answer to a question to an engineering consultant from a client is, "It depends." [Boeckle 04] That is, the correct decision depends on the exact situation in which the problem occurs, which is never described completely in a question or even a set of requirements. The client almost always assumes we know what they know about their situation and assumes that they have given us all the information that impacts the decision.

When I begin to write a research paper I consider the audience for which it is intended. What do they know? What do they expect? What types of arguments do they believe? I am trying to understand the "context" in which my words will be placed by the readers. I need to make explicit as much of the context as I believe is critical for them to correctly interpret my words.

Research reports, articles, and books all contain context information. Did you ever read a murder mystery and realize a third of the way through the book that no one has died yet? This is a sure sign of a book that is giving you a lot of context - clues - to help figure out whodunnit, when it happens. Research reports include a background section that makes explicit the critical information that the reader should know to interpret the work they are about to read and a related work section that puts the new work being reported into the context of existing work in the same area.

Context is the setting in which any statement will, or perhaps should, be interpreted. We can think of context as a set of constraints, or limitations, that set boundaries and a set of assertions that establish certain facts. The boundaries may be beliefs, or logical propositions, or any definition upon which our work depends. The "best" design for a

Cite this column as follows: John McGregor: "Context", in *Journal of Object Technology*, vol. 4, no. 7, September - October 2005, pp. 35-44 http://www.jot.fm/issues/issue_2005_09/column4

given solution is context dependent. Change the context and the evaluation of the design may change. Designs that were correct in a batch mode data processing system often are incorrect in a real-time, embedded system.

Context plays a role in every thing we do. Sometimes it is explicit, but most often it is implicit. The secret to success is recognizing the context and the constraints it places on what you do and then controlling and modifying the context to your advantage.

An explicit form of context is found in the function call mechanism in many languages. A variable name is known within the function's calling context. A calling context is contained within the calling context from which it was called. The same variable name can be used in both the calling context and the called context to refer to memory locations as long as either only the variable in the most recent context is used or there is a means of explicitly referencing the other variable.

We choose a programming language, in part, for the variety of mechanisms that are provided for specifying the access context. We control the context for a statement by using these mechanisms. This explicit context information is used by the compiler to disambiguate memory references.

In an object-oriented programming language, an object forms an explicit context. Elements of an object - methods and attributes - are addressed by prefixing the element's name with the name of the object containing the element. Multiple instances of the same class can co-exist because the objects give their members a reference context.

Context is also illustrated in a software product line organization. One of the actions taken by a product line organization is to develop an explicit statement of the scope of the product line. This statement defines which products are part of the product line and which are not. The scope provides the high-level context in which reusability, among other 'ilities,' can be evaluated. A software component does not have to be reusable across the entire known software universe, just across the set of products within the scope of the product line. The scope is manipulated during product line planning to achieve an optimal return on investment by either expanding the scope to amortize costs over more products or narrowing it to have more commonality among the products.

This discussion is only of value if we have control over context, and I think we do. In the following sections I will explore some facets of context, some ways we represent context, and how we use context.

2 OUT OF CONTEXT

We have all heard someone defend themselves by saying, "..but my words are being taken out of context.." In other words a mistaken interpretation is being made because the context is not known or is being ignored. Software, taken out of its context, must be analyzed in the new context before use.

On June 4th, 1996, the Ariane-5 launcher veered off course and exploded within seconds after its flight was initiated. A detailed report of an independent board of inquiry stated that the cause of this disastrous failure was due to a software design error resulting

from software reuse [Lions, 96]. The error occurred because a software component from the Ariane-4 was reused in the Ariane-5 but the new context varied from the old sufficiently that an error occurred during execution and caused a failure. The critcal change in context being the difference in the size of the floating point number representation. The code was not tested extensively because it had been tested for the Ariane-4 context and had performed as expected.

Many efforts at reusing software result in a similar situation although the effects usually aren't so obvious. A developer needs a certain function and remembers a piece of code that will do the trick. Best of all, it is already tested and trusted code – less effort. Little does he realize that the code only worked for a much smaller range of values than the data types used to declare variables and parameters. Tests are often conducted over the range of expected inputs not the range of possible inputs, even when the possibility is identified by data type declarations. Used by itself in the new context even a smaller digression from this range may produce failures.

Lesson #1: The explicit context often is different from the implicit context.

3 FACETS OF CONTEXT

"The context" is composed of pieces of context from many sources. If we consider reusable software, the context comes from the operating system, libraries that support windowing, the database, and other sources. Each of these contributes constraints, dependencies, and specifications to the context a designer must consider.

In fact, a context is composed of interacting, and sometimes cross-cutting, concerns. Further, the context that affects the GUI builder is different from the context that affects the builder of the computational engine which is different from the context faced by the manager of the project. Each role in a project works within its own context.

One of the problems with being multi-faceted is that to manage context, we have to analyze the context systematically and recognize its component parts. We have to separate those concerns into pieces that can be managed and manipulated. This allows us to reason about the effects that a change will have on the context and hence the operation of a program. Consider performance design decisions for a system where the current hardware may be replaced by a machine with a faster processor, faster disk drive, but slower bus. We have to understand the effects of each change before we can reason about the effcets of the total change.

The design patterns community calls the facets of a problem's context *forces*. In some pattern templates there is a section on context separate from the forces but I will treat the two as synonymous.

Context can be decomposed into a set of forces to support reasoning about specific decisions. A force will have dependencies on other forces. A solution that resolves one force may cause another force to exceed acceptable values. By associating a facet with a

force, when a decision causes the violation of a constraint, it is relatively easy to evaluate the implications of violating that constraint. We may accept the violation or even be able to make changes that remove the constraint altogether.

In a recent engagement the attempt to make a specific system design decision was confounded by a complex context in which the reliability requirements on the hardware, the funds available, and the need for an innovative architecture made for a complex reasoning process. We decomposed these facets of the context, prioritized the forces, and in the process came up with a design proposal that was not envisioned until the forces were clearly, separately delineated. Rather than violate a constraint, we did some research and identified a different approach that changed some basic assumptions to remove the constraint.

Decomposing the facets of the context using the origin of the facet as one means of decomposition, such as whether the facet is the result of the operating system or a library being used in the project or some other source, can result in a useful organization. If a constraint, that prohibits a particular choice, is imposed by the operating system, we may have to accept the limitation since the options are few but if it comes from one of the libraries selected for the project we may be able to find an alternative library that does not impose such a restriction. Another possibility is to use the architecture for decomposing the context. That is, a portion of the context may be the result of the graphical user interface while another portion is a result of choices made for the database interface. The point is to identify the source of the constraints as a means of knowing more about how to handle that portion of context.

Lesson #2: The source of the context will determine how to handle it.

4 SPECIFICATIONS AND CONTEXT

Specifications describe what something is to do. They should include the context considered when the specification is written. For example,

float squareRoot(int x);

post: return*return = x

This specification describes what the squareRoot function does, but it omits the fact that, in most computing environments, it will not compute correct answers for every integer. Adding

pre: x >=0

makes explicit that the specification for squareRoot exists in the context of the real number system as opposed to the complex number system.

The Unified Modeling Language's (UML) Object Constraint Language (OCL) recognizes the importance of context information and the need for it to be explicitly stated. The full syntax for this specification would be:

Context: RealMathFunctions::squareRoot(x:int)

pre: x>=0

post: return*return = x

In this expression OCL views RealMathFunctions as the *contextual type*. The pre and post-conditions for squareRoot are evaluated in the context of a specific instance of the contextual type, referred to as the *contextual instance*. This is an important distinction because one of the most frequent specification mistakes is to fail to understand the difference between a class definition and the objects generated from that class.

Lesson #3: The target of the context needs to be clearly delineated.

5 MANAGING CONTEXT

A colleague of mine said recently he believes context just happens, you can't create it. I disagree but I will agree that we often do not have control over every facet of the context. We usually have control over the development method we choose, but we seldom have much control over the client. I will start by talking about how to manage existing context and in the next section I will discuss how to establish context.

Recognize context

The first step in solving your problem is to admit you have one. I see many organizations that go along bumping into context without recognizing that these constraints can be identified and managed. This has been evident for some time in organizations that use large scale reuse schemes such as frameworks and commercial off-the-shelf (COTS) software as the basis for products. A framework imposes context on the projects in which it is used. It implies certain assumptions and imposes constraints on design decisions particularly flow of control issues. Resources are often consumed unnecessarily where designers attempt to defeat these constraints when they are actually inherent in the framework and defeating the context is really redesigning the framework.

A software architecture defines a context. The context is in the form of "architecture qualities." The Attribute Driven Design technique creates an architecture by identifying and prioritizing the qualities that the final product should have. Decisions are made using reasoning frameworks [Bass 05]. Each framework provides a means of thinking about a specific quality and the level of that quality that will be present given a specific structure. A change in the priorities among these qualities changes the context and changes the decisions that are made.

The standards for documentation for architectures specifically address ways to make the context explicit [Clements 02]. Designers using the architecture determine whether certain design decisions are valid by examining the documentation.

Consider the MVC architecture. The decomposition into model, view, and controller components defines a context in which data and the functions that manipulate that data only fit the context provided by one of the three major divisions. This context is imposed to ensure the extensibility of the View portion of the architecture. Any design decision that ties the Model and View more tightly violates the context of the architecture and degrades the extensibility quality.

Manage context

Managing a context requires decomposing the context into manageable facets, communicating it to others and making decisions based on those facets. I have already discussed decomposing the context so I will focus on communicating and decision making.

As the context is being decomposed, the individual facets can be described and the dependencies among the facets made explicit. A Unified Modeling Language (UML) use case diagram can be used to represent this. The facets of the context are similar in nature to the aspects in an aspect-oriented model. The facets are not orthogonal nor are they limited to one point in the system. They "weave" among other items.

Using the extension mechanism in the use case diagram provides a means of modeling facets and their decomposition. Each oval in the use case diagram is either a base level context or a context fragment. The extends relationship between the ovals indicates that the pointed-to oval extends the pointing oval without modifying it. Notice that there are multiple base contexts. The context for an architect is different from the context for a developer. The architect considers architectural qualities but not language capabilities while the opposite is true for the developer. This diagram is a useful mechanism for communication.

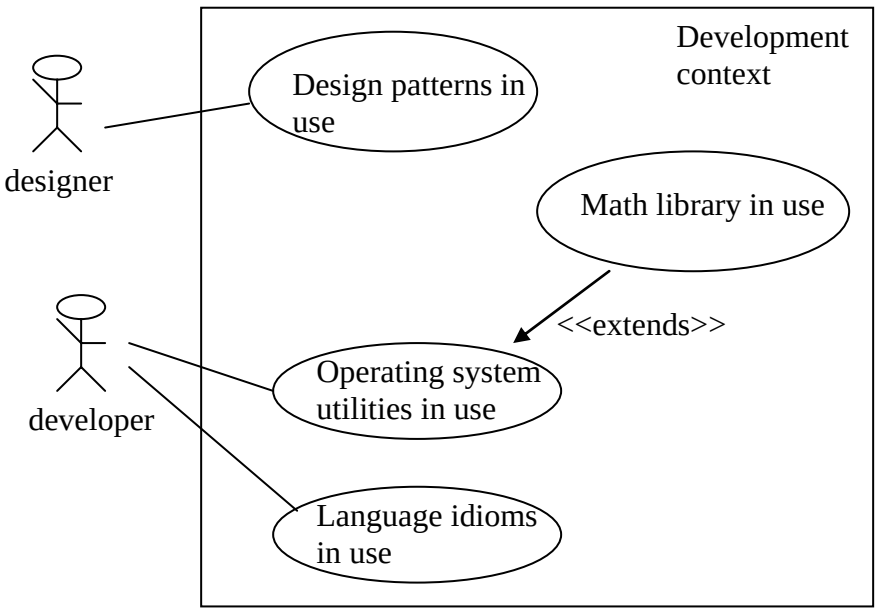


Figure 1 - Facets of a Context

This model is an important reference for decision making. It supports reasoning about trade-offs for decisions. The decomposed facets allow us to identify those fragments that impact a specific decision and to predict their affect on the various design options. I have also found that explicitly examining the context often reveals assumptions that simply are not true. We often are not as constrained in a real way as much as we imagine we are. A detailed analysis reveals these false assumptions.

Lesson #4: The real context is often different from our supposed context.

6 ESTABLISHING CONTEXT

Establishing context requires two types of techniques:

technique for determining what the context should be
technique for enforcing the context.

Determine the appropriate context

Remember that the context is the setting in which statements will be interpreted. So, we can start by considering what assumptions are necessary for those statements to be true. That is, we can set limits on our work. Incremental software development often is accompanied by an expanding context.

As I mentioned in the Introduction, determining the scope of a product line is an effort to determine the context of product development. The scope is expanded or contracted by adding or removing products. The context may be changed as the scope is changed if constraints are added or removed by the change in scope. For example, my manufacturing context may only need to encompass static binding until a product is added that will provide a web page interface that requires dynamic binding. Previous constraints become invalid and are changed.

Enforce the context

By enforcing the context I mean actively ensuring that actions don't violate the context. This is not a strong point in our industry. We try to be all things to all clients. The software product line approach is to clearly differentiate between what is possible within the product line and what is not. Products that violate the context are not attempted by the organization.

Tools can be used to enforce the constraints and assumptions of a design. The obvious concrete example is a compiler that checks whether values being assigned to a variable are of the appropriate type given the assumptions about that variable. Another example is the architecture description that can be type checked. AADL uses a formal grammar for describing the architecture to provide a means of enforcing the constraints of the architecture model on the designer creating an instance of the architecture. Most

UML tools – as opposed to paint programs – enforce the syntax to some degree by either refusing to allow, or at least flagging, incorrectly formed statements.

Lesson #5: Explicitly control the context to your benefit.

7 PUTTING IT ALL TOGETHER

I have discussed several techniques related to context. Now I want to put "context" into context by considering where context enters the development process.

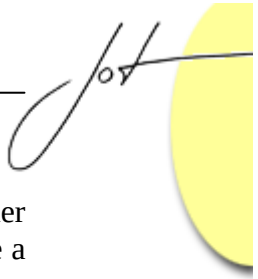
Requirements for a product heavily rely on context. Even government procurement assumes that the readers and writers of the requirements share a common understanding of the domain. This means that not everything needed to understand the problem is written. This usually means there is a vendor selection criteria about vendor experience. Including domain modeling and domain analysis results in the development process reduces the reliance on implicit context.

In the architecture design and detailed design activities the context usually includes actual and ad hoc standards. When the specific architecture design method is added in, additional context is added. For example, the Attribute Driven Design (ADD) method assumes designers have common definitions of specific qualities. ADD also assumes that design decisions are made with quantitative criteria and a clear set of priorities among the criteria – the qualities. The design context is made explicit by using a glossary of terms that includes the quantitative definitions of the qualities and by using design patterns to make rationales apparent.

Finally, during implementation the context becomes even more important. The implementation of a product uses specific language idioms depending upon the qualities that the system must exhibit. For example, the use of iterations, selections and other control statements have performance implications. A developer with extensive experience in the language will select the optimal technique. Coding standards within an organization can make this context explicit. Be careful though, getting a set of standards off the net may impose someone else's context rather than your own.

8 SUMMARY

Managers and engineers make decisions all the time. They use their experience to guide those decisions. That experience is valid only within certain limits, the context of their experience. The challenge is to realize when decisions are being influenced by an inappropriate or incorrect context. The simplest way to address this is to make the context explicit. Viewing the context as an aggregate of information from several sources allows decision makers to consider the impact of the anticipated evolution of the source of a piece of context.



Context, because it is implicit, creeps up on us. We often don't explicitly consider the implications of context on our decisions. What I have attempted to do is to illustrate a few ways in which context can be made more explicit and can be controlled to our benefit.

Context is strategically important. Many new directions in software relate to context-sensitive or context-aware systems that adapt to their environment. For example autonomous agents must be context-aware. But it is not enough to be context aware, you must control and manage the many contexts that affect the achievement of your strategic goals. By explicitly considering context, essentially we are taking control of yet another force and making it a competitive advantage.

ACKNOWLEDGEMENTS

Special thanks to John Hunt for many conversations and ideas about context.

REFERENCES

[Bass 05] Len Bass, James Ivers, Mark Klein, Paulo Merson. Reasoning Frameworks, CMU/SEI-2005-TR-007

[Boeckle 04] Guenter Boeckle, Paul Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. "Computing Return on Investment for Software Product Lines" IEEE Software, v 21, n 3, May/June 2004.

[Clements 02] Paul Clements http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Paul%20Clements/002-7991720-3272019

Felix Bachmann http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Felix%20Bachmann/002-7991720-3272019

Len Bass http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Len%20Bass/002-7991720-3272019

David Garlan http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=David%20Garlan/002-7991720-3272019

James Ivers http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=James%20Ivers/002-7991720-3272019

Reed Little http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Reed%20Little/002-7991720-3272019

Robert Nord http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Robert%20Nord/002-7991720-3272019

Judith Stafford http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author-exact=Judith%20Stafford/002-7991720-3272019: Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.

[Jacobson 03] Ivar Jacobson: "Use Cases and Aspects - Working Seamlessly Together", in Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 7-28, http://www.jot.fm/issues/issue_2003_07/column1

[Lions 96] R.L. Lions, Ariane 5, Flight 501 Failure, Report by the Inquiry Board, http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Ariane/Esa//ariane5/COPY/ariane5rep.html, July, 1996.

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.