# Do We Think In Terms Of Objects and What Are The Consequences For Software Reuse, Architecture and Patterns?

Jason Baragry
Norwegian Computing Center.
Postboks 114 Blindern, N-0314 OSLO
Visiting address: Gaustadalléen 23, Oslo. Norway.
Jason.Baragry@nr.no

## Abstract.

We present an overview of some of the research being performed at the Norwegian Computing Center. We believe many of the problems related to software engineering in general and object-oriented development in particular result from too little attention being paid to the role of modelling, both in terms of how we understanding problems and how we attempt to solve them using software systems. We present a summary of our research direction and how it is being applied to areas such as software reuse, patterns, and architecture.

## 1. Introduction.

One of the claimed advantages of object-oriented development is that developers can use objects in a uniform modelling approach throughout the development process. Popular object-oriented software engineering texts (e.g., [1-5]) all suggest similar approaches to the phases of the development process and the artefacts produced by them. During requirements elicitation, the customers, users, and developers use use-cases to develop and represent the system requirements in a manner that can be communally understood. During the analysis phase, the collection of use-cases are refined by identifying and reusing similar concepts, specifying the functionality with greater precision, and removing ambiguities. The result is the generation of an analysis model that identifies the significant concepts of the problem and the way they need to interact to provide a satisfactory solution. As use-cases are refined during the analysis phase, the system moves closer to design because the identified concepts and their interactions become more formal. Indeed, some object-oriented researchers argue that the product of the analysis phase, the analysis model, can be viewed as an initial version of the design model [1] (p. 178).

The boundary between analysis and design, however, is not clearly defined. The goal of analysis is to specify exactly what the system is to do without necessarily how it is supposed to do it. However, constructs used to represent concepts during the analysis phase often have direct analogues in the design phase. Therefore, it may not be clear where the analysis concept ends and the design concept begins. Despite this blurred boundary, the goal of system design is to transform the analysis model into a design model. The design model is comprised of constructs that are directly realisable in the chosen implementation medium, that is, the combination of hardware environment and software programming language(s).

Traditional OO methods suggest the transition from requirements to analysis to design is smooth and easy, however in practice it has been shown to be quite difficult [6]. Similarly, both researchers and developers are beginning to question the assumption that object-orientated development is advantageous because it allows developers to more easily implement their model of reality (see for example [7]). While improvements in analysis techniques are certainly helping (e.g., [8]) we believe the practical problems associated with moving from analysis to design are far greater and more deeply rooted than currently thought and that to solve them will require a closer examination of how developers understand their problems and how those ‘understandings’ can be represented in software.

This paper presents an overview of the relevant issues and the research direction at the Norwegian Computing Center that is aimed at investigating them. It begins with the concept of modelling. Most researchers and developers agree that software development revolves around models, however far too little work has been done on the exact role of modelling. The next section provides an overview of the problems with modelling in software development and then summarises relevant theories about modelling from the disciplines of philosophy and psychology. Section 3, then applies these theories to our understanding of software development and highlights different ways of thinking about software reuse, architecture, and patterns.

## 2. What are conceptual models how do we build them?

The artefacts produced during the transition from requirements elicitation to the design representation and implementation are often talked about as models. Books and papers describing software development methodologies constantly refer to terms such as conceptual models, use-case models, analysis models, design models, process models, architectures, architecture styles, design patterns, programming idioms, design paradigms, implementation mediums, and programming constructs. However, the use of those terms is certainly not consistent throughout the discipline.

Though people have been aware of the issue for some time, it is becoming increasingly apparent that the ‘conceptual construct’ is the essential part of achieving a good software design. This issue was discussed at the first software engineering conference in 1968 [9]. Brooks, in his famous *No Silver Bullet* paper, noted,

> “The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed. *I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor or representing it and testing the fidelity of the representation.* We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.” [10] [Brooks’ italics].

Indeed, his earlier book, *The Mythical Man-Month* [11], dedicated a chapter to the importance of the conceptual construct and the importance of conceptual integrity in system design.

Despite the fact that we have known about its importance for decades, the issues surrounding the creation and utilisation of the conceptual construct still cause problems in our attempts to engineer software systems. Consider the issue of software reuse. Mili et al note, “software development cannot possibly become an engineering discipline so long as it has not perfected a technology for developing products from reusable assets in a routine manner, on an industrial scale” [12]. The belief that conceptual structures can be designed and implemented using an ‘engineering’ approach that incorporates significant amounts of reuse requires one significant assumption on the part of the software development community. That is, the identification of items that can be reused from previous applications, from the requirements analysis stage to the implementation stage, assumes that different clients and developers experience the same reality and can model it using similar collections of distinct concepts and concept relationships. Moreover, those concepts and relationships can be specifically defined in terms of essential features and represented the same way in two different applications using the implementation medium of software development – hardware and software constructs.

We have argued that this assumption is at the root of many problems in software engineering and stems from the belief that software systems can be understood as being analogous to traditionally engineered systems, that is, software engineers have an artefact engineering view of software development (see [13] for more information). This assumption also underlies one of the major myths of object-orientation – the belief that objects allow us to directly implement our view of reality, and that these objects can be easily reused by other developers. This assumption is simply not supported in practice. For example, an earlier study of automotive cruise control systems has examined how software developers and traditional engineers approach the same problem (see [13, 14]). Seven of the software designs were object-oriented. Despite the fact that the requirements for those seven systems were almost identical, each of the object-oriented designs identified a different collection of objects to represent the same problem to be solved (see table 1). Even such a small and well-defined problem as automotive cruise control resulted in seven different models of the problem.

| Design Example | Objects Identified |
|---|---|
| Booch | Driver, Brake, Engine, Clock, Wheel, Current speed, Desired speed, Throttle, Accelerator. (9) |
| Yin & Tanik | Driver, Brake, Engine, Clock, Wheel, Cruise control system, Throttle, Accelerator. (8) |
| Birchenough | Driver, Wheels, Accelerator. (3) |
| Gomaa (JSD) | Cruise control, Calibration, Drive shaft. (3) |
| Wasserman | Cruise controller, Engine monitor, Cruise monitor, Brake pedal monitor, Engine events, Cruise events, Brake events, Speed, Throttle actuator, Drive shaft sensor. (10) |
| Appelbe & Abowd | Driver, Brake, Engine, Clock, Wheel, Cruise controller, Throttle. (7) |
| Gomaa (Booch OO) | Brake, Engine, Cruise control input, Cruise control, Desired speed, Throttle, Current speed, Distance, Calibration input, Calibration constant, Shaft, Shaft Count. (12) |

**Table 1: Cruise Control ‘Objects’.**

If we are to improve the way we build software, what is required is a closer examination of the underlying principles of software systems. Those underlying principles are related, somehow, to concepts, models, abstractions, theories, and how they are used by the human mind to understand reality and solve problems. Those issues have been explored by other disciplines for many years and their theories serve as a starting point for uncovering the foundations of software engineering. Philosophy, especially in the fields of epistemology and metaphysics, has a long history of identifying the concepts that constitute reality and how they are represented in knowledge. Additionally, psychologists, especially in the fields of conceptual development and cognition, have devised experiments and theories to explain how concepts are used to capture reality, how they are devised, and how they evolve. Finally, theories in the history and philosophy of science explain how models and theories are used to explain the world, how those theories can be verified, and how they evolve over time.

Unfortunately, those disciplines do not offer ready-made explanations of the underlying principles of software engineering. Nevertheless, different theories from those fields have been cited in software engineering research as justification for proposed ideas. To ensure our research does not simply adopt one of the many different philosophical and psychological positions to support a presupposed understanding of software engineering, it examines, in detail, the major theories from those disciplines that are related to the underlying principles of software systems. Moreover, it has also involved researchers form those disciplines. An attempt here to compress two thousand years of thought into a handful of notes is, perhaps, over ambitious. However, without this material, the basis for our direction is unlikely to be clear.

Our analysis identifies two phases of thinking about the underlying issues. The classical way of understanding concepts and theories dates back to the philosophies of Plato and Aristotle and begins with the assumption that people can be considered as separate from their environment and that all things can be classified in terms of essential attributes. Those assumptions result in a belief that all people observe the same objective reality, and that concepts are derived by inferring abstractions from that reality. As people operate in the world, they associate objects with known concepts by identifying the essential attributes. Furthermore, because reality is objective, theories used to explain phenomena capture the natural order of the world.

However, as progress occurred in both philosophy and psychology, a different way of thinking about the issues emerged. Many philosophical arguments and psychological experiments highlighted anomalies in the classical way of understanding. Subsequent research showed that people's conception of reality cannot be considered as separate from an objective reality. As people interact with the world, they automatically and subconsciously apply their accrued concepts and theories to the observed phenomena in order to understand it. Consequently, people's explanatory theories do not capture the natural order of the world. Rather, they are subjective to the person using them and different theories cannot be evaluated as being better or worse depictions of reality. Each can only be evaluated in terms of the usefulness it provides the person using that theory towards understanding and operating in the world. An additional contradiction to the classical way of understanding concerns the definition of concepts. Experiments and dialectic debate have shown that people do not classify phenomena into different classes of concepts based on the existence of essential attributes. Instead, concepts are defined in terms of the roles the play within people's explanatory theories of the world.

Its important to remember that although these conclusions contradict the classical way of understanding, researchers observe that the classical way of understanding still pervades the philosophical assumptions of people who have not studied the relevant philosophical and psychological research. That is evident in the justification of many views of software development.

These philosophical foundations provide a different way of understanding the software development process. For example, the popular Unified Process book [1] provides a table on page 219 comparing the analysis and design models produced as part of the process. While there are specific differences between the two it is assumed that the design model is based, in part, on a refinement of what exists in the conceptual analysis model. However, the issues of philosophy and psychology show there are more significant differences than those presented in conventional object-oriented design literature.

| Analysis Model | Design Model |
|---|---|
| Concepts and relationships cannot be precisely defined by essential attributes. | Concepts and relationships must be defined by essential features and specific functionality |
| The precise meaning of concepts and relationships is dependent on the context of the theory in which they are contained | The precise meanings of concepts and relationships, their definitions, are independent of the system in which they are implemented. |
| Concepts and relationships are constrained only by the previous experience and imaginative ability of the stakeholders in the development process | Concepts and relationships are constrained by the constructs provided by the implementation medium and the execution model of the virtual machine that executes it. |

**Table 2: Comparison of the Analysis Model and Design Model based on the Philosophical and Psychological Foundations.**

While it does not provide a comprehensive explanation for all our problems, we believe these foundations can be used to improve software engineering research by providing a basis with which to evaluate and justify both existing and future research ideas.

## 3. The Consequences for Software Engineering.

Our investigations have concluded that the discipline-wide understanding in software engineering research has been dominated by analogies with traditional engineering disciplines [13]. Those guiding assumptions are not always explicitly stated and practitioners are not always aware of them. However, those guiding assumptions set research agendas, direct investigations, bias observations, and justify conclusions. Moreover, those sets of guiding assumptions change as a discipline evolves and researches based on different sets of guiding assumptions are not always commensurable with each other. However, an alternative approach, based on philosophical foundations, offers the potential for an improved way of thinking about software systems and how they are developed. The remainder of the presentation explores that potential concentrating on the areas of software reuse, architectures, and design patterns.

### *3.1. Software Reuse.*

The first concerns software reuse. Substantial gains have been made as a result of our efforts to reap the benefits of widespread software reuse. However, we have yet to achieve the same scales of reuse that has been achieved by traditional engineering disciplines. The philosophical foundations of the model building view may provide some insights to explain this. The first insight concerns the difference between requirements/analysis concepts and design/implementation concepts. Concepts are identified during the requirements/analysis stage of the development process and have to be precisely defined and implemented as software constructs during the design/implementation stage. However, the concepts we entertain in our explanations of the world do not identify objective real-world parts and they cannot be universally defined by essential attributes. They are theory dependent and are subjective to the person using that theory to understand the phenomena under investigation. This results in two different types of concepts. The first (referred to here as $\text{concepts}_1$) are the fuzzy, theory-dependent concepts applied to sensory experience to assist human understanding. The second (referred to here as $\text{concepts}_2$) are the independent, rigorously defined structures of software design and implementation. The identification of a $\text{concepts}_1$ concept can result in an infinite variety of $\text{concepts}_2$ definitions. If a concept is identified during the development process of a system, then its definition, the resulting software construct, is only a realisation of that concept within the theory used to understand the problem at hand. For example, if the object-oriented analysis of a problem identifies a class, ‘Customer’ ($\text{concepts}_1$), then its definition ($\text{concepts}_2$) only provides the required features of a ‘Customer’ within the confines of the problem that the system solves. The philosophical foundations of software engineering suggest that if the analysis of a different problem also identifies a ‘Customer’ ($\text{concepts}_1$) during its analysis stage, then the original ‘Customer’ definition ($\text{concepts}_2$) may not be applicable in the new context. It may be possible to reuse the ‘Customer’ definition in the new situation, but it equally well may not be. This contradicts the idea of software reuse based on the classical theory of understanding and the artefact engineering view of software development.

Nevertheless, some successful reuse efforts have been achieved and they are explained with the foundations provided by a model building view. The first concerns the observation that reuse is more successful when the designer browses an asset library before beginning design rather than searching for and retrieving assets to match the concepts of a proposed design [12]. The human mind applies known concepts and theories to a situation in order to explain it. That is, humans understand a situation in terms of how they understand previously encountered situations. Having knowledge of what is already in a reuse repository before design commences exploits that innate conceptual ability by allowing the mind to devise a solution to a problem in terms of that knowledge. As the designer interacts with the problem, knowledge of those artefacts will be automatically and subconsciously applied to the situation to determine if they provide a useful explanation. Therefore, the human conceptual apparatus makes it a lot easier to design a system to reuse known artefacts than it is to find artefacts to meet a designed system.

Analyses of software reuse also notes that software product lines provide the most dominant form of systematic software reuse today [12]. Additionally, user interface components are often used as examples in explanations of successful reuse theories. These facts are also explained as a consequence of the model building view though they are not detailed here.

### *3.2. Software Patterns.*

These foundations can also be applied to develop an understanding of software design patterns. Software patterns have become extremely useful in systems development. Their historical link with the patterns of Christopher Alexander are well documented, however the model building view may provide a different and more useful explanation as to why they are so constructive and provide insights into how they can be better utilised. .

Despite the successful application of design pattern theories to software development, research in the area fails to resolve anomalies that exist between software systems and traditionally built artefacts. Alexander himself questions the validity of the analogy between software patterns and his building patterns [15]. Additionally, consider his comment, “the ultimate object of design is form” (Chapter 2: *Goodness of Fit* in [16]). Software systems do not have a notion of form that is analogous to that found in traditionally built artefacts. Hence, it is not clear what Alexander's term, "the coherence of the created whole" (Chapter 2: *Goodness of Fit* in [16]), means in the context of software systems when using the artefact engineering view of software development.

However, if software development is understood as model building rather than artefact engineering, some explanations of how patterns are utilised in the model building process become evident. People automatically and subconsciously apply their accumulated concepts and theories to the world in order to understand their experience. In software development, the subconsciously applied concepts and theories are made explicit and captured during the requirements/analysis stage of the process. They are then converted into a collection of constructs and connections that can be precisely specified and implemented during the design/implementation stage. However, the process of creating a useful analysis model and the transformation of that analysis model into a design model is quite complex. The solution embodied in the analysis model is the developer's theory for explaining the problem and that theory is not completely specified until it is implemented in code. However, it may take a long period of interacting with the problem before a satisfactory explanatory theory can be generated that cannot be falsified. Indeed, it may not be until the design is in the implementation stage that anomalies between the requirements and the explanatory theory become apparent. However, when successfully utilised collections of concepts and theories have been used to capture the understanding of a problem, and those concepts and theories are known to be implementable in the constructs of software and hardware, they can be made explicit for use by other developers. Moreover, those concepts and relationships can be represented at a higher-level of abstraction to make them applicable to analogous problem situations. Software patterns provide a format for capturing those higher-level concepts and relationships. They do not capture naturally occurring aspects of an objective reality. They capture successfully used ways of understanding a subjective reality that are known to be implementable in software and hardware constructs. To reiterate, people naturally explain the situations they encounter in terms of concepts and theories they have used before and modify those concepts and theories according to the new context. Software patterns make explicit and capture aspects of the natural thought processes of human understanding.

## 3.3. Software Architecture.

The last issue to be examined concerns software architecture, specifically, software architecture views. Many anomalies exist between the practice of software architecture and the theories provided by software architecture research. Those anomalies exist because of the prevailing influence of the artefact engineering view of software development. However, they can be explained by changing to a model building view. During the software development process, the designer must create an initial conceptual model that makes explicit the concepts and relationships, the explanatory theory, which the designer believes explains the problem. That conceptual model can consist of many different types of concepts and relationships, at many different levels of generality, and are limited only by the designer's experience and imagination. However, to implement that conceptual model, the collection of concepts and relationships must be transformed into a collection of constructs and connections provided by the implementation medium. Those constructs and connections may have the same labels as the concepts and relationships in the conceptual model, however the philosophical foundations of the model building view show that one is not simply a refinement of the other. The types of concepts and relationships are fundamentally different. As stated previously, one set of concepts is the fuzzy, theory-dependent concepts used in human understanding ($\text{concepts}_1$). The other set of concepts is the formally specified, context-independent concepts of software implementation (concepts$_2$). Finally, to realise the required system, a computer must execute the implemented constructs. That implementation can exist across many different machines, many different processes, and may include many different instantiations of the one software system.

The model building view of software development suggests three different types of high-level system representation are required during the development process. Those different types of representation are not different abstractions, or subsets, of the complex implementation detail. They are fundamentally different and are required because of the unique nature of software systems. First, representations of the conceptual model are required. These are produced as the initial step in the design process and represent the model that is to be implemented as a solution to the problem. They consist of the concepts and relationships that constitute the designer's explanatory theory for the problem. Second, representations of the static implementation are required. These depict the implementation of the system in terms of software and hardware constructs and their dependencies. They represent the structural form of the implemented system but do not contain enough explicit information to depict the control flow through the executing system. While the constructs in the static implementation may appear similar to the concepts in the conceptual model representations, they are fundamentally different and one is not merely a refinement of the other. Third, representations are required to represent the dynamic operation of the system. These depict the behaviour of the system and may consist of concepts from the conceptual model, concepts from the static implementation model, concepts used by the computer in the execution of the system, and concepts depicted to the user such as user interface constructs. Each of these types of high-level system representation are fundamentally different and those differences can be only be satisfactorily explained by rejecting the prevailing artefact engineering view of software development and accepting a model building view. See [17] for more detail.

## 4. Conclusion

This paper has provided an overview of the some of the software engineering research being performed at the Norwegian Computing Center. That research is focussing on the role of modelling in software development and how it can be improved using relevant research from fields such as philosophy and psychology. As an overview, this paper has touched on many areas while providing little detail and has also omitted some areas of interest. Moreover, because it describes a research direction rather than a set of research results there still remain many unresolved issues and points of contention. Nevertheless, we believe the direction is worth pursuing and can result significant improvements in the way people understand and subsequently build software systems. Finally, we would like to point out that while this direction may appear to contradict some existing software engineering theories, we believe our approach can be complementary and welcome the opportunity to working with others in the field.

## 5. References.

1. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Developent Process*. 1998: Addison Wesley Longman.
2. Bruegge, B. and A.H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. 1999: Prentice Hall.

3. Larman, C., *Applying UML and Patterns: an introduction to object-oriented analysis and design*. 1997.
4. Oestereich, B., *Developing Software with UML: Object-oriented analysis and design in practice*. Object Technology Series. 1999: Addison Wesley Longman.
5. Pooley, R. and P. Stevens, *Using UML: software engineering with objects and components*. Object Technology Series. 1999: Addison Wesley Longman.
6. Kaindl, H., *Difficulties in the Transition from OO Analysis to Design.* IEEE Software, 1999(September/October).
7. Hatton, L., *Does OO Sync with How We Think?* IEEE Software, 1998(May/June): p. 46-54.
8. Fowler, M., *Analysis Patterns: reusable object models*. Object Technology Series. 1997: Addison-Wesley. 355.
9. NATO, *Report on a Conference Sponsored by the NATO Science Committee. Garmisch Germany, Oct 7-11 1968.*, in *Software Engineering Concepts and Techniques: proceedings of the NATO conferences*, P. Naur and B. Randell, Editors. 1976, Mason/Charter.
10. Brooks, F.P., *No Silver Bullet.* IEEE Computer, 1987. **20**(4): p. 10-19.
11. Brooks, F.P., *The Mythical Man-Month: Essays in Software Engineering*. 1975: Addison-Wesley Publishing.
12. Mili, A., et al., *Toward an Engineering Discipline of Software Reuse.* IEEE Software, 1999(September/October).
13. Baragry, J., *Understanding Software Engineering: from analogies with other disciplines to a philosophical foundation.* PhD thesis *Dept of Computer Science and Computer Engineering*, La Trobe University. Australia. p. 350. Available from the author.
14. Baragry, J., *Is Software Development Analogous to Traditional Engineering? A Comparison Of Designs for Automotive Cruise Control.* Submitted to IEEE Transactions on Software Engineering., 2001.
15. Alexander, C., *The Origins of Pattern Theory: the future of the theory and the generation of a living world.* IEEE Software, 1999(September/October): p. 71-82.
16. Alexander, C., *Notes on the Synthesis of Form*. 1964: Harvard University Press.
17. Baragry, J. and K. Reed. *Why We Need a Different View of Software Architecture*. in *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2001. Amsterdam, The Netherlands.