Object-Oriented Programming: Bringing Perspective to the Claims and Counter-Claims

A dissertation presented to the

Department of Information Systems

University of Cape Town

By

Len Naidoo

15th February, 2009

In partial fulfillment of the requirements for the Coursework and Dissertation Masters

(INF5005W) Course 2008

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.

2. I have used the APA convention for citation and referencing. Each contribution to and quotation in this dissertation, *Object-Oriented Programming: Bringing Perspective to the Claims and Counter-Claims*, from the work(s) of other people has been attributed, and has been cited and referenced.

3. This dissertation, *Object-Oriented Programming: Bringing Perspective to the Claims and Counter-Claims,* is my own work.

4. I have not allowed and will not allow anyone to copy my work with the intention of passing it off as his or her own work.

5. I acknowledge that copying someone else's assignment or essay, or part of it, is wrong, and declare that this is my own work.

Signature: Signed by candidate .Date 26/5/2009

Full Name of Student: Len Naidoo

Abstract

This study attempts to bring insight into the claims and counter-claims made about Object-Oriented Programming (OOP). A rich understanding of OOP enables us to maximize OOP's potential and uncover its limitations in practice. OOP rose from relative obscurity in the 1960s to the mainstream of software development. While adoption of OOP continues to grow steadily fervent criticism mounts against OOP from certain quarters. Detractors believe that OOP's early promise has not been fulfilled and resentment deepens against its all-encompassing embrace. Negative, self-serving rants needs to be separated from genuine causes for concern about software development. Equally, the software development community must guard against the blind adoption of technology in an industry driven by excessive market hype and vested interests. Given the decidedly human nature of the task of programming and the varying interpretations and perceptions that exist about programming styles, it is unlikely that scientific methods can conclusively prove the truth of arguments made for or against OOP. Therefore this interpretive study uses hermeneutics to understand and explain the claims and counter-claims made in the OOP discourse in order to forge a deeper understanding of OOP. The 'meanings' of OOP in the online discussions taking place in three developer communities encounters the literature on OOP through hermeneutic principles of understanding. Three different interpretations of OOP emerge from this encounter. Against these newly appropriated insights of the OOP phenomenon the prevailing OOP discourse is critically assessed through the hermeneutic lens. This study concludes that the current quest for atemporal categories and casual explanations of OOP is futile; we can only deeply understand OOP if we are willing to accommodate multiple interpretations of OOP in a hermeneutic understanding of the phenomenon of OOP.

1 Introduction

The approach followed when developing software systems is vital not only to the software development community but to an entire modern society that has become increasingly reliant on information technology. The choices made when developing software impacts on the quality of software systems reverberating on the productivity of organizations and the lives of individuals. Software development is plagued by ongoing issues of poor delivery and poor quality (Brooks, 1987). The continuous effort to improve the quality of software and the software development process has lead to new approaches and ideas (Raccoon, 1997). Object-Orientation (OO) is one such approach. The ideas in Object-Oriented Programming (OOP) have been around since the late 1960s however it only reached popularity and mainstream development since the 1990s (Capretz, 2003). While this study acknowledges the broad impact of OO on the entire software development process the focus of this study is OO programming.

Object-Orientation is popularly conceived of as a revolutionary software programming paradigm radically departing from more 'traditional' methods (Johnson, Hardgrave & Doke, 1999). A central concept to our generally accepted notion of OOP is the concept of an 'object'. Rumbaugh, Blaha, Premerlani, Eddy and Lorensen (1991) defined the term 'object-oriented' as the organization of software as a collection of discrete objects that incorporate both data and behaviour. Booch (1994) informally defined an 'object' as a tangible entity that exhibits some well-defined behaviour. This is said to be in contrast to traditional programming in which data and behaviour are only loosely connected. Taylor (1998) states that the most fundamental idea behind object technology is that real-world objects or objects of the problem domain are represented in software systems through logical units called objects.

OOP has been hailed by its supporters as a revolution in software development yet others claim that it has not fulfilled its early promise in practice; some doubt that it ever will. Against this backdrop various claims and counter-claims have been made; at one extreme

OOP is sold as the solution to the software crisis as presented by Brooks (1987), while at the other end of the argument OOP is blamed for impeding the growth and further advancement of software development.

OOP was initially received with high expectation. Meyer (1988) expressed these expectations as follows:

> I do not think object-oriented design is a mere fad; I think it is not trivial (although I shall strive to make it as limpid as I can); I know it works; and I believe it is not only different from but even, to a certain extent, incompatible with the software design methods that most people use today – including some of the principles taught in most programming textbooks. I further believe that object-oriented design has the potential for significantly improving the quality of software, and that it is here to stay. (p. xiii).

As OOP started to become more widely accepted the virtues of OOP where expounded in contrast to traditional software tools and techniques. According to Jorgensen et al. (2002), the modest promises of OO soon gave way to bolder and grander promises. These promises that were intended to further separate OOP languages from the more traditional languages included:

- Naturalness - OOP models the real world better because everything in the world is an object.
- Reuse - OOP makes programming faster and easier because of the reuse of existing, previously tested classes from a library. Reuse can also be accomplished by inheritance.
- Development Life Cycle - OOP makes faster development of programs because rapid prototyping of models is achieved due to the reuse of existing models of a corporation's processes.
- Maintenance - OOP makes maintenance easier because code only needs to be changed in one place. This is made possible by encapsulation.

- Quality - OOP makes testing easier and more reliable because components already exist and have been previously tested.

Jorgensen et al. (2002) concludes that OOP has only partially kept its promise.

Deepening resentment and growing criticism of OOP has accompanied the rise in the popularity of OO. Gabriel (2002) expressed the sentiments of detractors in the opening remarks of a debate on the question 'Objects Have Failed' at OOPSLA 2002 in Seattle, Washington:

> What can it mean for a programming paradigm to fail? A paradigm fails when the narrative it embodies fails to speak the truth or when its proponents embrace it beyond reason. The failure to speak truth centers around the changing needs of software in the 21st century and around the so-called improvements on OOP that have obliterated its original benefits. Obsessive embrace has spawned a search for purity that has become an ideological weapon, promoting an incremental advance as the ultimate solution to our software problems. The effect has been to brainwash people on the street. The statement everything is an object says that OOP is universal, and the statement objects model the real world says that OOP has a privileged position. These are very seductive invitations to a totalizing viewpoint. The result is to starve research and development on alternative paradigms.

This study is motivated by a need to understand and explain the arguments in the OOP discourse. The approaches we follow in developing software needs to be properly evaluated in the context of the contrasting positions of market hype and blind adoption, and the sheer reluctance to change from old ideas. Deeper insight can be brought to the meaning of OO not only by making sense of both the exalted virtues of OOP and the scathing negative criticism leveled against OOP but also by examining the more subtle tensions in the OOP discourse. In doing so it is hoped that OOP can be made easier to understand and more effectively leveraged to create better software. The primary objective of this research is:

To make sense of the claims and counter-claims made in the OOP discourse in order to develop a deeper understanding of OOP so that we are able to maximise OOP's potential and also uncover its limitation in practice.

Hermeneutics serves as the ontological, epistemological and methodological basis for this study. While many of the questions asked about programs, programming and software development benefit from scientific research methods this study considers the task of programming to be a decidedly human activity. It takes the viewpoint that much of the fundamental beliefs that underlie approaches to programming are creations of the human mind derived from and contributing to 'shared understandings', therefore it seems unlikely that many of the arguments about software development will be conclusively resolved by 'scientific' proof. This study analyzes the claims and counter-claims made in the online discussions of three developer communities through hermeneutic principles of understanding. The rich meanings derived from the online discussion will encounter the preunderstandings of OOP derived from the OOP literature allowing the hidden and confused interpretations of OOP to emerge into a coherent whole. Hermeneutics will also serve as the basis for critique and further explanation of the prevailing OOP discourse.

This dissertation is organised as follows. The research approach section discusses the philosophical underpinnings and principles of understanding employed in this study. The literature section reveals this studies preunderstanding of the subject matter of OOP. The text section discusses the nature of the text used in the analysis. The analysis section analyzes the text for meanings 'in' the text. The discussion section seeks a deeper understanding 'beneath' and 'beyond' the text. The conclusion section summarizes the key conclusions reached by this study.

2 Research Approach

Research can be undertaken with many different assumptions about reality, knowledge, and method therefore it is particularly important for the researcher to make clear the epistemological, ontological and methodological underpinnings of any study so that it can be properly conducted and evaluated (Myers, 1997). The ontological and epistemological basis for this study is hermeneutics. Therefore the philosophical underpinnings of this study derive from hermeneutics. The text of OOP discussions taking place in three online developer communities is hermeneutically understood by applying principles of hermeneutic interpretation therefore it is also the method of this study. Hermeneutics also forms the theoretical basis for the critical stance taken and further explanation of the prevailing OOP discourse.

The rest of this section both justifies and explains this study's particular application of hermeneutics.

2.1 Hermeneutics

According to Ricoeur (1981), "Hermeneutics is the theory of the operations of understanding in their relation to the interpretation of texts". Hermeneutics is primarily concerned with the meaning of a text (text in the broadest sense also includes a situation that a researcher comes to understand though oral or written artifacts). Hermeneutics presupposes that texts and text-analogues that are distanced by time and culture, or that are veiled in ideology and false consciousness, would necessarily appear chaotic, incomplete, contradictory and distorted, and that they need to be systematically interpreted to unveil their underlying coherence or sense (Demeterio, 2001). The choice of hermeneutic compliments the objective of this study given the nature of the prevailing OOP discourse.

In order to truly understand the complex subject of hermeneutics the inter-twining of its multiple layers of meanings and concerns need to be untangled. Hermeneutics has a long tradition from ancient times to the present day. The meaning of hermeneutics and our knowledge of the nature of understanding has evolved over time. There are various separate and sometimes contrasting hermeneutic theories (Arnold & Fischer, 1994). Later

views on hermeneutics may contradict or incorporate elements from earlier views. A general theory of interpretation does not exist therefore it is important for any hermeneutical study to clarify its understanding or interpretation of hermeneutics.

Hermeneutics refers to the praxis, methodology and philosophy of interpretation (Demeterio, 2001). Hermeneutics started out as praxis of interpretation from the ancient times. The term 'hermeneutics' came into use from the seventeenth century however the operations of textual exegesis and theories of interpretation of religious, literary and legal texts date back to antiquity (Palmer, 1969). The Greeks, Jews and Christians read and interpreted their precious texts like the Homeric epics, the Torah and the Holy Bible. Gradually they developed sets of rules for doing interpretation therefore hermeneutics as a methodology of interpretation started to evolve from the praxis of interpretation. Hermeneutics was fully developed as a methodology of interpretation during the Renaissance period. Hermeneutics during this period proliferated into a collection of contradicting, incoherent and confusing methodological systems. Hermeneutics developed from its initial roots in the interpretation of religious and humanist text to include the social sciences and other fields of study. The meaning of text grew to include anything that has something to do with man and society. All these developments lead towards a generalized hermeneutics with a philosophical foundation for interpretation (Demeterio, 2001).

According to Palmer (1969) the hermeneutical problem as a whole is too important and too complex to become the property of a single school of thought. Care must be taken not to take certain narrow perspectives of hermeneutics as absolute. Hermeneutics must remain a field of study open to contributions from many different and sometimes conflicting traditions. The schools of hermeneutic thought represented by Ast, Schleiermacher, Dilthey, Heidegger, Gadamer and Ricoeur represent important milestones in the evolution of hermeneutics for this study. This study does not apply a particular hermeneutic; instead it integrates ideas and concepts from these various schools of thought into a coherent and meaningful whole that make sense for this study.

2.1.1 Ast

Initially the task of textual interpretation followed different methods determined by the type of text; legal text was interpreted using of juridical hermeneutics, sacred scripture required biblical hermeneutics and philological hermeneutics was applied in the interpretation of literary texts. These various methodologies are now referred to as regional hermeneutics in contrast to a 'general hermeneutic' that encompassed interpretation of all types of text, which developed later. Ast's contributions in the field of philology exemplified the sort of regional hermeneutics that provided the foundation for a 'general hermeneutics' (Ormiston & Schrift, 1990).

Ast distinguished between three corresponding forms of explication:

- The hermeneutics of the letter
- The hermeneutics of meaning
- The hermeneutics of the spirit

The hermeneutics of the letter involved an explanation of the language and style, and the subject matter in an historical context. The hermeneutics of meaning attempts to recapture the particular meaning intended by the author when the works first appeared. The hermeneutics of the spirit explains the text in terms of the one unifying idea that guides the text as a whole. The hermeneutics of the spirit lead Ast to the conceptualization of the circular structure of understanding that was later to become known as the hermeneutic circle. Understanding must be forged in the context of a dialectical relation between part and whole. The underlying principle of all understanding is to find the spirit of the whole in the particular, and to comprehend the particular through the whole. Ast's work provided the background for Schleiermacher's pursuit of a 'general' hermeneutics (Ormiston & Schrift, 1990).

2.1.2 Schleiermacher

Schleiermacher directly addressed the phenomenon of understanding, for the first time. He sought to uncover the interpretive technique, which operate universally within

understanding for all kinds of text interpretation, whether spoken or written. He distinguished between grammatical interpretation and a technical interpretation. Grammatical interpretation is the objective rules for understanding a language in terms of its original audience, while a technical understanding attempts to uncover the author's original intended meaning (Ormiston & Schrift, 1990).

The art of 'explanation', which had previously constituted a large part of hermeneutics, was held by Schleiermacher to fall outside of hermeneutics. According to Schleiermacher, hermeneutics does not concern itself with the operation of bringing something to speech; it concerns itself with the understanding of what is spoken. Understanding takes place in a dialogical relationship where the speaker constructs a sentence and the hearer interprets its meaning through the hermeneutic process. Hermeneutics is the art of hearing (Ormiston & Schrift, 1990).

Understanding is a referential operation because we understand something by comparing it to something we already know. We understand the meaning of individual words in the context of the emerging whole while the individual words form the whole meaning. This dialectical interaction between the whole and the parts each giving each other meaning is referred to as the hermeneutic circle. Likewise a dialectical interaction between the grammatical interpretation and the technical interpretation illustrates another dimension to the circular nature of understanding (Ormiston & Schrift, 1990).

2.1.3 Dilthey

Dilthey expanded the concept of text to include all disciplines focused on understanding man's art, action and writings. He saw hermeneutics as the foundation for all the humanities and social sciences, all those disciplines which interpret expressions of man's 'inner life', since the norms and ways of thinking of the natural sciences did not apply to the study of man. The purpose of science is to explain nature while the human studies understand expressions of life. Understanding grasps the particular for its own sake while science sees the individual as a means to arrive at the general. A methodology of understanding must transcend the reductionism of the sciences and return to the fullness of human experience (Palmer, 1969).

Dilthey's hermeneutic formula is a systematic relationship between 'lived experience', 'objectified expressions' of those 'lived experiences' and 'understanding'. A 'lived experience' is the unity of meaning that arises from several encounters with a phenomenon. Lived experience objectifies itself in social and human phenomena. Art, poetry, language, laws and ideas are all objectified expressions of lived experience. Objectification of lived experiences enables understanding to be focused on a fixed, 'objective expression' of lived experience instead of introspection. Introspection is an unreliable way to understand. Understanding is the mental process by which we comprehend living human experience. Understanding is not a mere act of thought but a re-experiencing of another's lived experience (Palmer, 1969).

Dilthey asserted that man is an historical being. Understanding is intrinsically temporal and meaning is contextually dependent on the past and the future. An interpretive approach cannot ignore the historicality of lived experience by applying atemporal categories to historical objects for this will distort our understanding of the phenomenon (Palmer, 1969).

2.1.4 Heidegger

Heidegger redirected hermeneutics towards an ontological quest for the processes of understanding and interpretation through which things appear. Hermeneutics is the primary act of interpretation which first brings a thing from concealment. Understanding is a mode of 'being-in-the-world' and fundamental to existing in the world; it is not something metaphysical, beyond man's existence. The hermeneutics of Heidegger is often referred to as phenomenological hermeneutics (Palmer, 1969).

According to Heidegger's school of thought, interpretation without supposition is impossible. The clear separation between subject and object is not possible in the world of phenomenological hermeneutics. Reality does not exist independently of our understanding of it. We create our reality in the act of interpreting and understanding through our presuppositions. Hermeneutics prior to Heidegger assumed that reality exists independently of our interpretation or understanding or it (Palmer, 1969).

Heidegger moved beyond Dilthey's conception of hermeneutics as the methodological foundation of all humane disciplines. Hermeneutics became the ontological basis for all understanding. The historical-scientific dichotomy introduced by Dilthey is overturned in the assertion that all understanding is rooted in the historical character of existential understanding. The ground is cleared for Gadamer's philosophical hermeneutics (Palmer, 1969).

2.1.5 Gadamer

Like Heigegger, Gadamer is a critic of the separateness of subject and object in understanding and gaining knowledge. Truth is not reached methodically but dialectically. In method the subject leads, controls and manipulates; in dialectic the object can also question and lead the subject. Gadamer sought to answer the question of how understanding was possible in the whole of man's experience of the world rather than to seek a methodology or finding the right principles of interpretation (Linge, 1976).

Gadamer also emphasized the historicality of understanding. We come to any understanding with prejudgements. We neither can nor should we temporarily suspend our prejudgements, instead we should be willing to risk our preunderstandings in the process of understanding. Since there can be no presuppositionless interpretation the notion of one correct interpretation is an impossibility. This means that there may be multiple interpretations that are all valid because the presuppositions that underlie each interpretation are different. The task of hermeneutics is primarily to understand the subject matter and not the intentions of the author. Both the interpreter and the author participate in an interpretation of the subject matter, a sort of dialogue. Therefore the task of understandings is not one of abandoning one's own prejudices (good or bad) in a quest to uncover the authors point of view but rather to allow a dialectic interplay between one's own horizon of understanding and the horizon of meaning emanating from the text resulting in a 'fusion of horizons'. A 'fusion of horizons' is possible because one does not seek to become master of what is in the text but rather to become servant of the text. One follows, participates in and here's what the text is saying however this is not a passive openness but a dialectical interaction with one's own preunderstandings. It is the willingness to subject one's horizon of preunderstanding to modification by encountering

the meaning of the text. Thus Gadamer brought the subject preunderstanding into the hermeneutic circle (Linge, 1976).

2.1.6 Ricoeur

Ricoeur sought to find a place for critique within hermeneutics. According to Ricoeur, from Heidegger onwards the recognition of a critical instance is a vague desire constantly reiterated, but constantly aborted, within hermeneutics. The hermeneutical experience itself as described by Heidegger and Gadamer discourages the recognition of any critical instance since it involves the overcoming of distanciation between the 'horizon' of the subject and the 'horizon' of the object. However, Ricoeur views distanciation as a positive element which makes interpretation possible, it does not contradict understanding (Ricoeur, 1981).

Ricoeur views the text as much more than a particular case of intersubjective communication, it is inherently distanciated. The relation between writing and reading is not a particular case of the relation between speaking and hearing. Writing renders the text autonomous with respect to the intention of the author's psychological and sociological conditions. Distanciation is not the product of methodology and hence something negative but constitutive of the phenomenon of the text as writing. It is the very condition of interpretation and not what understanding must overcome (Ricoeur, 1981).

According to Ricoeur, appropriation is dialectically linked to the distanciated characteristic of writing. Distanciation is not abolished by appropriation, but a counterpart of it. The concept of distanciation does not only apply to the inherent nature of text but also to the interpreter who understands through moments of distanciation and appropriation. The hermeneutics of Gadamer closes this possibility by premature introjection. The concept of appropriation, to the extent that is directed against distanciation, demands an internal critique. The understanding of a text arises in the dialectic of distanciation and appropriation (Ricoeur, 1981).

Ricoeur asserts that a further condition for hermeneutics to account for a critical instance is to reconcile the dichotomy between explanation and understanding introduced by

Dilthey. This dichotomy arises from the conviction that any explanatory attitude is borrowed from the methodology of the natural sciences and this has no place in the human sciences. However, not all explanation is naturalistic or causal. The text calls not only for a description but also an 'explanation' that mediates understanding. Explanation must help move hermeneutics from a naive interpretation to a critical interpretation. Hermeneutics is not just 'understanding' but the dialectic of explanation and understanding that moves us towards a deeper understanding of the text. The hermeneutic circle is broadened to include the dialectic of distanciation and appropriation, and explanation and understanding (Ricoeur, 1981).

2.2 Research Ontology & Epistemology

The ontological and epistemological basis of this study is drawn from hermeneutics therefore this is an interpretive study. The philosophical base of interpretive research is the phenomenological hermeneutics of Heidegger and the philosophical hermeneutics of Gadamer. The broad methodological choices facing a IS researcher are those embodied in positivism, interpretivism and critical research. According to Myers (1997), IS research is positivist if there is evidence of formal propositions, quantifiable measures of variables, hypothesis testing, and the drawing of inferences about a phenomenon from a representative sample to a stated population. Critical research offers social critique in the hope of helping to realize human potential; it seeks to emancipate society from unwarranted alienation and domination (Myers, 1997). IS research is interpretive if it attempts to understand phenomena through the meanings that people assign to them by focusing on the complexity of human sense making as the situation emerges (Myers, 1997).

Interpretive researchers start out with the assumption that we can only know reality through social constructions such as language, consciousness and shared meanings (Myers, 1997). A phenomenon is understood through the meanings that people assign to it. Walsham (2006) argues that that our knowledge of reality is a social construction and shared meanings are a form of 'intersubjectivity' rather than objectivity. Even though interpretivists may accept the idea that some sort of reality may exist out there we cannot

know this reality directly. We each have our subjective interpretations of this reality but we also have a shared understanding of this reality, that is, the meanings we constantly negotiate with others with whom we interact about this reality.

Interpretivism, and its underpinning philosophy derived from hermeneutics, is not the mainstream approach to the study and practice of software engineering. However, its value is beginning to be recognized. According to West (1997), when applied to systems and computer science, the minority hermeneutic paradigm centers on concepts of autonomy, multiple perspective, negotiated and ephemeral meaning, interpretation, emergence, self-organization, change, and evolution. West (1997) contends that we can only build computer-based systems that create global networks, whose complexity rivals that of the natural world, which respect and complement the dignity of human beings, and which can adapt to support a world in constant flux if we have a computer science that can recognize why objects are different, why hermeneutics are important, and why formalism is limited in its application. Berntsen, Sampson and Osterlie (2004) support the claim that interpretive research has the potential to produce deep insights into computer science, information systems and software engineering.

While the subject matter of OOP and software programming in general may appear very absolute and scientific (in the natural science sense) it is the viewpoint of this research that software concepts are largely creations of the human mind derived from the human experience and shared understandings rather than irrefutable, fundamental laws of nature. Software concepts are continuously redefined, appropriated or misappropriated for uses removed from their original conception, and continuously evolved to take on new meanings in an ever changing socio-technical environment. Many aspects of the complex subject of OOP, discussed in the OOP discourse, would be difficult to quantify and measure against this backdrop therefore this study aims to understand and explain the OOP discourse qualitatively and interpretively rather than scientifically falsify OOP claims (or counter-claims). Additionally a qualitative, interpretive approach offers a different perspective on what is largely considered to be a technical matter only explainable through 'hard science'. A hermeneutic approach will offer new and enlightening insights into the subject matter of OOP and software programming concepts.

2.3 Methodology

Further to hermeneutics forming the ontological and epistemological basis for this study the following principles of understanding derived from the various schools of hermeneutic thought guide the interpretation of the text in this study. These principles describe both the nature of the understanding this study sets out to achieve and also how it intends to achieve its objectives. The principles below were derived directly from the hermeneutic schools of thought discussed previously. The hermeneutic principles for this study are congruent with the principles for conducting and evaluating interpretive IS studies that have been identified by Klein and Myers (1999).

2.3.1 The Principle of Circular Understanding

A common theme across the various schools of hermeneutic thought from the time of regional hermeneutics of Ast to the critical hermeneutics of Ricoeur is that understanding is inherently circular. Understanding is forged in the context of a dialectical relation between part and whole. We find the spirit of the whole in the particular and we comprehend the particular through the whole. The concept is pervasive in deriving meaning from the parts of the text in the context of a broader emerging understanding, and understanding the broader whole from the parts. As also note by Klein and Myers (1999), the principle of circular understanding is the over-arching principle within which the other principles of understanding operate.

In the case of this particular study, arguments made for or against OOP cannot be dealt with individually and in isolation. They need to be considered in the light of the broader understanding of the OOP discourse as a whole that emerge through the application of the hermeneutic circle and the other hermeneutic principles of understanding that follow. Therefore a hermeneutic understanding of the OOP discourse requires that an understanding of the individual claims in the online discussion also help forge an understanding of the OOP discourse as a whole through which we are able to achieve a deeper understanding of those very same claims. This study makes apparent the dialectic, circular nature of hermeneutic understanding. It explicitly shows how our understanding of the OOP discourse as a whole emerges from the individual claims and counter-claims

and how the individual claims and counter-claims can be understood in the light of the broader empathetic understanding of the OOP discourse as a whole. The concepts of parts and whole are relative. The broad empathetic understandings of the online OOP discussion that emerge during the analysis phase become parts in the discussion phase where an ever expanding understanding of the entire OOP discourse continues to emerge. The principle of circular understanding permeates the entire understanding process dialectically and iteratively.

2.3.2 The Principle of Prejudgement

The concept of prejudgement is largely borrowed from Gadamer but modified by Ricoeur's argument for a critical instance in hermeneutics. We come to any understanding with prejudgements. We neither can nor should we temporarily suspend our prejudgements, instead we should be willing to risk our preunderstandings in the process of understanding. Therefore the task of understandings is not one of merely abandoning one's own prejudices (good or bad) in a quest to uncover the author's point of view but rather to allow the dialectic interplay between one's own horizon of understanding and the horizon of meaning emanating from the text. Neither the author's intended meaning, nor the interpreter's prejudgements, hold privileged positions. One does not seek to reign over the text and neither is one enslaved by the text. One follows, participates in and here's what the text is saying however this is not a passive openness but a dialectical interaction with one's own preunderstandings. It is the willingness to subject one's horizon of preunderstanding to modification by encountering the meaning of the text and yet be willing to reflect critically upon ones newly derived insights. The critical instance and the explanatory stance will be covered further in the principle of appropriation and distanciation.

This study makes explicit prior understandin

gs of OOP and the OOP discourse, and risks those preunderstandings in this study. The preunderstandings of the subject matter of OOP appropriated from the OOP literature are revealed. The meanings of OOP in the literature are not privileged or untouched by this study. This study cannot approach the text with fixed understandings or "atemporal"

categories of OOP. The meanings in the text and the meanings in the literature encounter one another in this study opening up both meanings to the possibility of change.

2.3.3 The Principle of Appropriation and Distanciation

Understanding develops through the dialectic of appropriation and distanciation. Ast, Schleiermacher and Dilthey emphasize the importance of understanding the language and intentions of the author therefore the meanings 'in' the text are appropriated by being led by the text and listening with openness to what the text is saying. Heidegger and Gadamer emphasize the importance of allowing the meanings 'in' the text and ones preunderstandings to encounter one another in order to develop a deeper understanding of the subject matter therefore appropriating the meanings that lie 'beneath' the text.

Ricoeur emphasizes that understanding cannot be just a naïve reading of the text. Distanciation plays an important part in uncovering a deeper understanding of the text. Distanciation is not an alien imposition on the text since the text is by nature inherently distanciated from the intentions of the author, psychological and sociological conditions. Additionally, the processes of distanciation and appropriation are also at work within the interpreter as he comes to understand. The interpreter must distance himself from his newly appropriated understandings in order to be critical and to take an explanatory stance (bearing in mind that all explanation need not take the casual form found in scientific studies). Distanciation opens the door for critique and explanation but only as the means of appropriating a deeper meaning that lies 'beyond' the text.

This study appropriates the meanings 'in' the online discussion by listening with empathetic openness to the arguments made. The meanings 'in' the online discussion encounter the preunderstandings derived from the OOP literature revealing the broader understanding of the OOP discourse and the meanings 'beneath' the OOP arguments. Distanciation from our newly appropriated understandings of the OOP discourse allow for critique and explanation that help clarify the OOP discourse even further revealing the meanings 'beyond' the text. However, any critique and explanation of the OOP discourse must not contradict the philosophical underpinnings of this study. The nature of this study will only allow for critique and explanations of a non-causal, temporal nature.

Hermeneutics as a theory of understanding forms the perspective from which critique and explanation is offered.

2.3.4 The Principle of Multiple interpretations

This study adopts the ontological basis of hermeneutics proposed by Heidegger and Gadamer. We cannot know the true nature of reality for understanding is not 'subject' making sense of an atemporal objective reality. Reality is constituted in the interaction of the subject with the object. Therefore interpretation or 'understanding' is a reality creating process. There is no objective viewpoint from which to know reality directly therefore the very nature of understanding and very nature of reality allows for the possibility of multiple interpretations or understandings of a phenomenon.

This study is open to multiple interpretations OOP. The OOP arguments and their respective counter-arguments may originate from different perspectives of OOP. The hermeneutic philosophy and principles adopted in this study precludes any kind of 'scientific proof' that will explain why one interpretation of OOP is more scientifically valid than another. However, as discussed earlier hermeneutics does not close the door on critique and explanation.

2.4 Process

The prevailing OOP discourse is chaotic, incomplete, contradictory and distorted. This study applies the principles of hermeneutic understanding in order to systematically interpret the OOP discourse thereby unveiling the underlying coherence of the OOP phenomenon. The 4 principles of hermeneutics will be used to explain the claims and counter-claims made in the OOP discourse in order to forge a deeper understanding of OOP. Neither the literature nor the 'text' are privileged positions in this study rather this study will allow the meanings of OOP in the online discussions taking place in developer communities to encounter the literature on OOP through hermeneutic principles of understanding.

It is the premise of this study that OOP and the subject matter of software development in general are temporal, shared understandings that emerge from the software development

community as a whole rather than only the definitions and theories postulated by OOP experts within the community. We can only appropriate a deeper understanding of OOP if we are willing to consider the points of view of both the well informed and those that may be perceived to be ill informed. Therefore this study cannot depart from an unshakeable preconceived understanding of OOP gleaned from the 'expert' opinions revealed in the literature. Rather the preunderstandings of OOP revealed in the literature encounters the meanings appropriated in the online OOP discussions where both the meanings in the literature and the meanings in the online discussions are subject to change and modification. A fusion of understandings will take place allowing a deeper meaning of the subject matter of OOP to be appropriated.

The 'text' is the OOP discussion taking place in the online software developer communities. This study does not regard the online discussions as a source of expert opinion on OOP either. While this study cannot claim that the online discussion is completely representative of the entire OOP discussion taking place among members of the software development community it is indicative of the broader communities' understanding of OOP. The inclusiveness and pervasiveness of the Internet broadens the discourse on OOP in a manner that would not be possible otherwise.

The process of hermeneutic understanding will unfold as follows:

- This study reveals its prejudgements of the OOP phenomenon. A background study of the literature shapes this studies preunderstanding of OOP. These preunderstandings of the subject matter of OOP appropriated from the literature are revealed and acknowledged.
- The analysis of the online discussions will appropriate the meanings 'in' the text. The principle of circular understanding is an over-arching principle for the other hermeneutic principles, however its explicit application in the analysis will allow broad themes of discussion and salient points to emerge from the individual claims.
- The discussion will reveal the meanings 'beneath' the arguments made in the online discussion as the fusion of meanings in the text with the meanings in the

literature will result in a deeper understanding of the broader OOP discourse. Hermeneutics opens up the possibility for multiple underlying interpretations of OOP. The various claims and counter-claims made in the online discussion and the OOP discourse in general will be reconsidered in the light of newly appropriated interpretations of OOP.

- Through the principle of distanciation a critical and explanatory stance will seek to uncover the meanings 'beyond' the text and the prevailing OOP discourse. Distanciation from newly appropriated understandings of OOP opens the door to critique and a further explanation of the OOP discourse in general. Hermeneutics as a theory of understanding is also the perspective from which critique and explanation can be offered.

3 The Literature – preunderstandings of OOP

According to the hermeneutic principle of prejudgement we come to any understanding with preunderstandings. We neither can nor should we temporarily suspend our preunderstandings, instead we should be willing to risk our preunderstandings in the process of understanding. Therefore a hermeneutic study must first reveal and acknowledge its preunderstandings or prejudgements. This study commenced with a background study of the existing literature on OOP. The preunderstandings of the subject matter of OOP appropriated in the background study are revealed here.

The preunderstandings from the literature do not put a stake in the ground as to the meaning of OOP. It is not a privileged position from which the meanings 'in' the text can be explained or validated. The meanings appropriated from the literature will encounter the meanings appropriated 'in' the text. The meanings 'in' the text are also not privileged. It's in the fusion of the two horizons of understanding where each is subject to modification in the knowledge of the other that a deeper meaning of the subject matter of OOP is appropriated. This 'fusion of horizons' unfolds later in the discussion section.

3.1 Paradigms of Programming

According to Normark (2007), a programming paradigm serves as a school of thought for programming of computers. It is a philosophical and theoretical framework within which computation and programming concepts are formulated. Therefore it is essentially a high level model of what computation is about. Normark (2007) identifies four main programming paradigms:

- The Imperative paradigm
- The functional paradigm
- The Logical Paradigm
- The Object-Oriented Paradigm

Functional programming originates from a purely mathematical discipline, the theory of functions. All computations are done by applying or calling functions. Functional programs do not require variables or any other abstractions of program state. The logic paradigm works well in problem domains that deal with extracting knowledge from basic facts and relations. It is based on axioms, inference rules and queries. Program execution is a systematic search in a set of facts, making use of a set of inference rules (Normark, 2007).

The main focus of this study is the object-oriented paradigm, which also necessitates a closer examination of traditional methods of programming. Therefore the object-oriented paradigm and the imperative paradigm will be dealt with in greater depth in the next sections.

3.2 Object-Oriented Programming

3.2.1 OO Programming

Booch (1994) defined OO programming as a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. He stated that there are three important parts to this definition:

- OO programming uses objects instead of algorithms as the fundamental logical building blocks.
- Each object is an instance of some class.
- Classes are related to one another via inheritance.

A program that omits any of these elements is not an object-oriented program. Programming without inheritance for example is not OOP; it is programming with abstract data types (Booch, 1994).

3.2.2 OO Analysis & Design

While the focus of this study is OO programming the influence of analysis & design on OOP cannot be ignored. OO is not just a style of programming but also an approach to systems analysis and design. Object-Oriented analysis is a method that examines the requirements from the perspective of the classes and objects found in the problem domain. It emphasizes the building of real-world models, using an object-oriented view of the world. Traditional structured analysis is said to focus upon the flow of data within a system. OO design is a method of design that uses object-oriented decomposition utilizing a notation depicting both, logical and physical as well as static and dynamic models of the system under design. OO uses class and object abstractions to logically structure design while structured design uses algorithmic abstractions (Booch, 1994; Post & Kagan, 2000; Quatrani, 1998).

While the early 1990's saw a profusion of OO analysis & design methodologies a few clearly prominent methods appeared by 1994 (Grossman, Aronson & McCarthy, 2004):

- Rumbaugh's Object Modeling Technique
- Jacobson's Object-Oriented Software Engineering
- Booch's Object-Oriented Analysis & Design

These methods were independently evolving towards each other. It made sense to continue the evolution together rather than apart from each other (Booch, Rumbaugh & Jacobson, 1999). This culminated in the creation of UML that attempted to fuse, or unify, the best practices from these various approaches. UML has achieved popularity and is rapidly becoming the standard for OO systems development (Grossman et al., 2004).

In a web-based survey of UML users worldwide Grossman et al. (2004) conclude that the UML phenomenon presents an enigma. It is being increasingly adopted throughout the world yet there is no consensus on how it should be used or whether it is really beneficial. The literature points to a technology that is complex, poorly understood, and used

inconsistently. While users seem eager to adopt UML they are unable to determine if it makes any real difference to them (Grossman et al., 2004).

3.2.3 OO Databases

OO informs every aspect of software development including data storage and database technology. According to Bloor (2004), early OO languages gave no thought to data storage therefore OO database technology only developed later in response to the problems encountered when OO programs had to work with traditional relational databases. Relational databases store only data while OO databases store the entire object, both its properties and its methods. The organization of the objects in storage maps more directly to the objects in the program design and the conceptual model (Bloor, 2004; Deng & Fuhr, 1995). Bloor (2004) states that a relational database is not optimal for expressing all the relationships that exist between objects and that between 35 percent and 40 percent of effort is wasted on disassembling and mapping objects to relational databases. Despite the advantages of OO databases for OOP relational databases continue to dominate and the uptake of OO databases is sluggish. Bloor (2004) attributes this situation to the momentum and market penetration of relational databases.

3.2.4 The Origins of OOP

According to Pawson (2004) OOP was invented in Simula in the 1960s and refined in Smalltalk in the 1970s. Simula was created to simulated real-world phenomena such as industrial processes, engineering problems and disease epidemics by building systems out of objects. Software objects model the properties and behaviours of the real-world entity. Simula broke away from tradition by challenging the practice of explicitly separating software into procedure and data (Pawson, 2004). Pawson's (2004) perspective of the origins of OOP is emphasized in the literature creating the dominant perception of OOP.

Capretz (2003) contradicts the invention theory proposing that OO evolved by improving upon already existing practices. The term "object" is said to have emerged almost independently in various branches of computer science. All of these were attempts to cope with the complexity of software:

- The Simula programming language introduced the concept, 'classes of objects', used to simulate real-world applications. The execution of a computer program was organized as a combined execution of a collection of objects. Objects sharing common behaviour are said to constitute a class.

- Monitors introduced the idea of an enclosed area as a software unit to deal with the issues of process synchronization and contention for resources among processes.

- Abstract Data Types are data types that can only be manipulated by operations that are exclusively encapsulated within a protected region.

- Frames used for knowledge representation captured the idea that behaviour goes with the entity whose behaviour is being described.

While the viewpoint that OOP was invented is the projected perception of OOP that exists 'out there', there is support in the literature for the idea that OOP concepts like encapsulation, inheritance and polymorphism build on basic programming principles. These principles are said to incrementally advance traditional programming. According to Sircar, Nerur and Mahapatra (2001) fundamental programming logic (decision and loop statements) is an essential aspect of both traditional and OO programming. Lewis (2000) states that mutual exclusivity of OOP and procedural programming is a myth. OOP does not abandon the concepts that we admire in a procedural approach; it augments and strengthens them. Solving a problem by dissection into multiple and manageable pieces, modularity and encapsulation have been known for years. The best procedural programs, according to Lewis (2000), are the ones that are the most object-oriented. According to Fraser et al. (2005) it is not correct to say that algorithmic abstractions have been superseded by object-oriented ones. The beginnings of object abstractions are in structured methods while object-oriented methods echo algorithmic abstractions. Both structured and object-oriented techniques play a role in the construction of contemporary systems (Fraser et. al, 2005).

3.2.5 Key OOP Characteristics

According to Armstrong (2006) any study of OO must be underpinned by a thorough understanding of the fundamental concepts that characterize OO. A clear understanding of what concepts characterize OO is of paramount importance to both practitioners in the midst of transitioning to the OO approach and researchers studying the transition to OO development. She believes that we cannot hope to achieve the productivity gains promised by the OO development approach, effectively transition software developers, or conduct meaningful research toward these goals, when we have yet to identify and understand the basic phenomena.

There has been no consensus around the fundamental concepts that define the OOP. Armstrong (2006) believes that there are various reasons for this: differences in perspectives (conceptual versus implementation), emphasis in life cycle (analysis, design, implementation), and orientation (computer science versus information systems) may provide insurmountable obstacles to agreement around what fundamentally defines OOP. This is especially confusing for developers learning the OO approach.

In a survey of journals, trade magazines, books, and conference proceedings Armstrong (2006) identified 8 concepts that were identified as characteristic of OO by the majority of sources. These were inheritance, object, class, encapsulation, method, message passing, polymorphism, and abstraction.

Based on her survey, Armstrong refined these concepts as follows:

- Inheritance is a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.
- An 'object' is an individual, identifiable item, either real or abstract, which contains data about itself and descriptions of its manipulations of the data.
- A 'class' is a description of the organization and actions shared by one or more similar objects.

- Encapsulation is a technique for designing classes and objects that restricts access to the data and behavior by defining a limited set of messages that an object of that class can receive.

- A 'method' is a way to access, set or manipulate an object's information.

- Message passing is the process by which an object sends data to another object or asks the other object to invoke a method.

- Polymorphism is the ability of different classes to respond to the same message and each class implement the method appropriately.

- Abstraction is the act of creating classes to simplify aspects of reality using distinctions inherent to the problem.

The class programming construct was first introduced in the SIMULA 67 language released in 1967 (Sklenar, 1997). The class encapsulates data and process into a single abstraction. The traditionally separate branches of data abstraction and process abstraction merge in the class abstraction. Classes are neither exclusively abstractions of data nor process, they are anything that can be 'named' and treated as a conceptual whole. Classes are used to represent abstractions of real objects. The class is used as a template to create one or more runtime instantiations called objects. The objects are said to belong to that class. In hybrid OOP languages classes are treated as part of the type system of the language and are handled in a similar way to built-in, primitive data types like integers and characters, and other user defined data types.

Inheritance allows a predefined class to share is properties and behaviours with another class obviating the need to redefine the details again in the other class. A derived class inherits all the properties and behaviours of the base class however the derived class can modify the class from which it inherits by adding, removing or changing one of more of the methods of the base class that it inherits (Armstrong, 2006). The methods are the encapsulated behaviours of the object.

Methods are usually implemented as traditional functions in hybrid OOP languages using the function call to invoke methods however the pure OOP Smalltalk invokes methods using the richer message passing method. Unlike the function call an object is always capable of responding to a message since a default response is available if the message is not directly supported. Message can also be forwarded by the original receiver to other objects to handle (Smalltalk dot org).

Inheritance combined with the dynamic binding of classes implement polymorphism. Polymorphism allows different behaviours to be referred to by the same name. If a base class can be 'abstract' describing only the interface or outward representation of the abstraction. Derived classes can implement the interface differently. Client code written 'to' the interface is polymorphic because it will behave differently depending on which derived class (implementation) we choose to bind to the abstract base class at runtime. Polymorphism can create more reusable and simpler code in certain circumstances (Armstrong, 2006).

3.2.6 Prototype-based OOP

The class serves as a template for the creation of objects while inheritance shares behaviour in mainstream OOP. Prototype-based OOP is an alternative approach to class-based OOP since it does not require either the class or inheritance. Prototype-based OOP is based on a theory that humans identify new phenomena by comparing their properties to phenomena they already know. A new phenomenon is then perceived as 'like a well known phenomenon' but with some properties modified and new properties added. In contrast class-based OOP classify phenomena according to common properties. Prototype-based OOP instantiate objects from another object called the prototype object that represents default behaviour. Behaviour is shared via delegation, which is essentially the forwarding of messages from one object to another (Lieberman, 1986).

According to Lieberman (1986) prototype-based OOP has been neglected despite its advantages because the origins of OOP in Simula and Smalltalk have firmly entrenched class-based OOP. Classes-based OOP fixes the communication pattern between objects at compile time while delegation allows communication patterns to be dynamically changed

at runtime. Madsen (1996) feels that prototype-based OOP will be useful in the initial exploratory phases of analysis, design and implementation because one has little understanding of the domain and it is useful to describe new phenomena in terms of well known phenomena. However, once a better understanding of the phenomena has been obtained it may be more desirable to classify them in terms of common properties.

3.2.7 OOP Languages

OO programming languages can be considered to fall into two main categories, pure OO languages and hybrid languages that support OOP. Pure OO languages have only objects and only class methods. Process cannot exist outside the class and data types don't exist. Everything is an object, even traditionally primitive data types like integers. By this definition Smalltalk and Eiffel are pure OO languages while C++ is a hybrid language. Popular OOP languages like Java and C-Sharp are also not pure by this definition (Gosling, Joy, Steele & Bracha, 1996; Stroustrup, 2007; Smalltalk dot org; Microsoft Developer Network Library).

Cox (1987) stated that OO programming can be added to nearly any conventional programming language by adding a small number of the new syntactic features alongside the existing capabilities of the language. This is reflected in practice of implementing OO concepts in many languages that have been traditionally regarded as imperative programming languages. The best illustration of this trend is the Basic programming language, which now supports OOP in its latest incarnation Visual Basic Dot Net (Gosling, Joy, Steele & Bracha, 1996; Stroustrup, 2007; Smalltalk dot org; Microsoft Developer Network Library).

3.2.8 OOP Promises

Jorgensen et al. (2002) explains that as OO started to become more widely accepted, the virtues of OO were expounded in contract to traditional software tools and techniques to further separate OO from the more traditional approaches. The promises of OO include:

- Naturalness - OO supposedly models the real world better because everything in the world is an object. Meyer (1988) stated that OO designers do not need to

spend their time in academic discussions of methods to find objects. Since OO designs are an operational model of some aspect of the world, the objects are there for the picking. The software objects will simply reflect these external objects.

- Reuse - OO is supposed to make programming faster and easier because of the reuse of existing, previously tested classes from a library. Reuse can also be accomplished by inheritance. Jacobson, Christerson, Jonsson and Overgaard (1992) concluded that prior problems with reuse included the finding, understanding and appropriateness of the things to be reused. OO gives a technique that never existed before that strongly supports these issues.

- Maintenance - OO promises to make maintenance easier. According to Meyer (1988), OO has a decisive edge over functional decomposition because functions are not the most stable part of a system. Functional decomposition leads to a brittle architecture. Traceability is said to improve in OO because an object identified during analysis can be found again in the code (Jacobson et al., 1992).

- Development Risk – OO is supposed to reduce development risk by supporting iterative and incremental development better than traditional methods (Booch, 1994). In the iterative and incremental development software is deliberately built to satisfy fewer requirements initially, but is constructed in such a way that it is easy to incorporate new requirements as they evolve.

3.2.9 OOP Criticisms

A deepening resentment and growing criticism of OO has accompanied the rise in the popularity of OO. Jacobs (2006) claims that the benefits of OO cannot be extrapolated to everyone. Just because OOP 'prophets' find OO better for themselves or their teams does not mean that this can be extrapolated to everyone. Software developers think very differently from each other. The human mind and business cultures come in many flavours therefore our programming paradigms should also do the same. Counter-claims against OOP include:

- Naturalness - Neubauer and Strong (2002) concluded that the procedural paradigm is more "natural" than OO for various reasons. Students learn mathematics as procedural processes applied to data from the beginning of their schooling. Encapsulating methods into objects implies that those objects are animated in some form yet most objects we encounter in real life are not animated. A recipe is an algorithm that is applied to a set of ingredients. We view our world as containing many inanimate objects that we control and manipulate. In sports like baseball, the ball does not pitch itself neither does the bat swing itself.

- Reuse - Early expert opinion may have created the false impression that reuse would follow almost automatically because abstraction, encapsulation and inheritance were inherent features of the OO paradigm (Ambler, 2001). The main protest of OO detractors is that reusability is not a particular and unique advantage of OO (Bezroukov, 2007; Graham, 2003). While abstraction and encapsulation are essential characteristics of modularity, which assists reuse, these concepts are not unique to the OO paradigm. Inheritance on the other hand may actually discourage reuse because it can create strong module coupling that adversely affects maintenance and testing (Bieman & Karunanithi, 1995).

- Maintenance – According to Hatton (1998), the complexity of an OO application is much greater than a procedural application in practice. This makes error identification much more difficult. In OO systems the failure is a long way away from the fault that caused it. The inability to narrow down the location of the fault brings all the code into scope for maintenance. Additionally, the average OO application has significantly more lines of code than its procedural counterpart. Therefore the maintenance programmer must sift through more code to find the fault which translates into increased effort, resources and cost.

- Development Risk - While the OO software community may have been some of the prominent advocates and early adopters of iterative and incremental style development, iterative and incremental development cannot be uniquely identified

with OO. Agile methodologies represents the current thinking within the software development community on iterative and incremental development yet the Agile community do not explicitly link their approach to only the OO paradigm (Fowler, 2005).

3.3 Traditional Programming

The traditional approach to programming is mainly characterized by imperative programming while 'Structured Systems Analysis & Design' and relational databases have also shaped the prevailing perception of traditional programming. These older yet firmly entrenched approaches to software development continue to maintain their hold over software development in the face of the growing popularity of OOP. In contrast to the prevailing notion of OO traditional software development consistently separates data and process at the analysis & design, programming and storage stages. Process is seen as acting on data, transforming it as it flows through the software.

3.3.1 Imperative Programming

An imperative program is essentially an ordered set of instructions that modifies the state of a program incrementally over time. It defines a sequence of commands for the computer to perform. Imperative programs prescribe 'how to do it' in terms of giving a step-by-step set of instructions while declarative programming styles like functional and logic programming express 'what needs to be done'. The imperative program derives its fundamental nature from the basic Von Neumann architecture of the computer hardware that is inherently designed to execute code imperatively (Normark, 2007).

3.3.2 Structured Systems Analysis and Design

Traditional programming is also influenced by analysis and design techniques, in particular the Structured Systems Analysis & Design (SASD) approach. SASD evolved from earlier approaches that concentrated on either process or data. SASD models both process and data, but separately. Data flow diagrams, entity-relationship diagrams and a data dictionary are used as conceptual models during the requirements and analysis phases. The data flow diagrams are converted into one or more high-level functions that describe the software systems overall processing during the design phase. The high-level

functions are functionally decomposed and implemented. Functional decomposition repeatedly breaks down the high-level functions into smaller less complex functions until the resulting functions are simple to understand. The entity-relationship diagrams and data dictionary produce the data organization for the database (Power, Cheney & Crow, 1990; Schach, 1996).

3.3.3 Relational Databases

The relational model of data first proposed by Codd in 1970 continues to dominate database technology today (Bloor, 2004). Relational models structure the data into a collection of related tables. Data redundancy is reduced in the database through the process of normalization. Relational databases use a standardized data access language, SQL. Relational database technology supports separateness of data, from process, found in traditional programming because it allows data to modeled and stored independently of the processes that act on the data. Relational databases became the default method of data management displacing other data storage paradigms over time because of its simplicity, uniformity and formal theoretical basis (Elmasari & Navathe, 1989).

3.3.4 Abstraction in Traditional Programming

Booch (1994) concludes that the entire history of software engineering is characterized by rising levels of abstraction. The primary way that humans deal with complexity is by abstraction; as software complexity has grown, so too has approaches to abstraction (Fraser et al., 2005). The central theme of the large number of research efforts to control software cost and quality focus on controlling software complexity through abstraction (Shaw et al., 1978).

According to Wulf (as cited in Booch, 1994), humans use the exceptionally powerful technique of abstraction to deal with complexity. Unable to master the entirety of a complex object we choose to ignore its non-essential details and deal instead with the generalized, idealized model of the object. Similarly Shaw (as cited in Booch, 1994) defines an abstraction as a simplified description of a system that emphasizes some of the details while suppressing others. Berzins, Gray and Naumann state that a concept

qualifies as an abstraction if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it (as cited in Booch, 1994).

The use of abstraction to control complexity has taken imperative programming through many paradigms of programming that still remain within the imperative approach:

- Assembly Language - Abstracting from binary coding to mnemonic coding.
- 1st to 3rd Generation Languages - Abstracting execution and state away from the computer hardware architecture.
- Structured Programming - Abstraction for well-formed control structures, repetition and decision control structures.
- Procedural Programming - Abstracting process to high-level functions.
- Modular Programming – Abstracting process to conceptually meaningful groupings of high-level functions.

3.3.5 Assembly Language

Up until the 1950s most programming was done with the native machine language of the underlying hardware (Mitchell, 2002). The enormous potential of the computer, increasing processing capacity and drop in hardware costs meant that more kinds of problems could be solved using the computer, especially business applications (Booch, 1994). The complexity of software increased due to the inherent complexity of problem domains and the increase in size of software systems (Booch, 1994). Programming in the computer hardware's intrinsic machine language proved difficult for humans trying to write complex software given the limitation of humans to work effectively with binary patterns (Booch, 1994).

Assembly language represents the first step in abstracting code away from the computer hardware to a form that makes it easier for humans to handle. The essence of the abstraction from machine language to assembler is that binary code of machine language is replaced by mnemonics. Humans generally find symbols made up of letters and

decimal numbers easier to recognize and understand than endless patterns of zeros and ones. The individual byte patterns representing the machine instructions and binary data is replaced by combinations of meaningful letters and numbers (Sebesta, 1996).

3.3.6 High-level Languages (1st to 3rd generation)

After machine language and assembly language, high-level languages evolved that essentially abstract away the computer's hardware architecture. Code is written for a virtual computer that can perform high-level instructions on decimal numbers and textual data. Finally this code is compiled into binary code. High-level programming languages allow the creation of computer programs that have instruction sets (execution) and data types (state) that are more aligned with human understanding (Sebesta, 1996).

While the mnemonic code of assembler helps to reduce complexity it still required the programmer to write code at a level of abstraction not too far removed from machine language instructions. Machine language instructions are very simple and basic. A simple program in either machine language or assembler requires many instructions and a very detailed knowledge of the computer hardware (Sebesta, 1996).

High-level programming languages maintain the 2 fundamental program elements of execution and state but at a higher level of abstraction than that found in machine language and assembler:

- Execution abstraction – the binary nature of machine and assembly language instructions are replaced by instructions at a higher level of abstraction that perform mathematical and string manipulation operations.

- State abstraction - Notions of the computer hardware architecture, internal registers and main memory, generally do not exist in a 1st to 3rd generation languages. Program state is abstracted from memory to variables. Data is stored in variables that can hold decimal numbers and text.

3.3.7 Structured Programming

Structured programming introduced control structure abstractions for repetition and decision flow control ('do loops' and 'if' statements) into 3rd generation languages. This encourages a more understandable or readable program design by creating properly nested and discernable decision and repetition code blocks that allows one to trace or follow the program execution easily.

Prior to repetition and decision statements the 'goto' instruction fulfilled the flow control function. While the 'goto' can be used to create discernable, easy to follow blocks of code it is left to the programmer to make correct use of the 'goto' statement. Injudicious use of the 'goto' can produce an unnecessarily complex code structure. Control structures are abstractions of well-formed patterns of 'goto' usage. Their introduction effectively ended the widespread use of the 'goto' (Sebesta, 1996).

3.3.8 Procedural Programming

Procedural programming emphasizes the use of the function as a code reuse mechanism. Functions abstract code into functionally meaningful units. Functions encourage reuse because they allow the same code to be used at more than one place in the software system. The function call transfers control to the block of code wherever the function call is made. The function call represents the entire operation, the details of which are defined in the actual function (Sebesta, 1996).

Programmers quickly realized that they were coding the 'same' functionality over and over again. This meant duplication of effort during development and also during the maintenance phase. The function can perform the same operation on different sets of data each time by allowing the calling code to pass it new values each time. Without the use of the function abstraction a program consists of a single monolithic main procedure. Common custom libraries and vendor supplied libraries of functions encouraged procedural programming (Sebesta, 1996).

3.3.9 Modular Programming

Modules are encapsulation mechanisms that group functions and data structures into a single physical unit. They can also control access to their 'internals' commonly known as information hiding where certain data structures and functions are only accessible within the module and are not visible outside the module (Sebesta, 1996).

According to Sebesta (1996) the module allows the software system to be physically broken up and composed of a collection of modules. The single monolithic program with all its data structures and functions lumped together is both difficult to comprehend and compile. Breaking up a software system into modules that can be developed, maintained and compiled separately significantly reduces the complexity of these tasks (Sebesta, 1996).

Early attempts at modularizing code were arbitrary causing maintenance issues because the software structure bore little resemblance to the structure of the problem. Functional decomposition starts at the overall function of the program breaking it down into a set of lower levels functions that accomplish the higher-level function. This process is repeated as lower level functions become the high-level function for the next cycle of decomposition, until the functions become small and simple. Functional decomposition produces a code structure that can be physically modularized into conceptually meaningful modules. It is an important design concept in structured systems analysis and design (Power, Cheney & Crow, 1990).

4 The Text – Online OOP discussions

The 'text' is the OOP discussion taking place in online software developer communities. While this study cannot claim that the online discussion is completely representative of the entire OOP discussion taking place among members of the software development community it is indicative of the broader communities understanding of OOP.

This research focuses on the study of online software developer discussions because it offers an easily accessible source of insight into the thoughts and feelings of the developer community. Given the skills and knowledge of the software development community one can reasonably assume that they have embraced the Internet in large numbers. The global reach of the Internet offers all members of the software development community an opportunity to participate in the OOP discourse. The Internet is more affordable than most other forms of media giving voice to members in poorer countries, members from different backgrounds and the novice. Participation in these online communities is not restricted. No undue censorship based on differences of opinion about the subject matter is applied. The inclusiveness and pervasiveness of the Internet broadens the discourse on OOP in a manner that would not be possible otherwise.

This study does not regard the online discussions as a source of expert opinion on OOP. The objective of this study is to make sense of the OOP discourse in the broad software development community irrespective of whether their opinions are informed or uninformed (contrary to expert opinion found in the literature). It is the premise of this study that OOP and the subject matter of software development in general are temporal, shared understandings that emerge from the software development community as a whole rather than the definitions and theories postulated by OOP experts within the community only. Since there are no explicit restrictions that bar entrance to these online communities anyone with an interest in the subject matter, whether novice or expert, can participate in the discussion. This serves the purpose of this study well since we can only appropriate a deeper understanding of OOP if we are willing to consider the points of view of both the 'informed' and the 'uninformed'.

The text used in this study is drawn from three online software developer communities:

- http://www.artima.com/forums/
- http://lambda-the-ultimate.org/forum
- http://c2.com/cgi/wiki?WelcomeVisitors

30 'Lambda the Ultimate' discussions, 28 'Artima' and 33 'Ward and Cunningham' discussions were found to be pertinent to the subject matter under scrutiny.

While the Internet hosts a large number of online software discussion the challenge was to find communities that are holding or have held intense discussions on OOP as a general programming style. It became apparent that the vast majority of online software communities are dedicated to education and skill building around particular vendor products. After an intense search of the internet three online communities providing the kind of 'text' necessary for this study emerged. (See the Appendix, for a list of the discussion threads selected for this study.)

5 Analysis

The analysis appropriates the meanings 'in' the online discussion. The objective of the analysis to capture the richness of the arguments that emanate from the text. At the same time the details must be assimilated into a cohesive whole. In this way the online OOP discussion can be understood in breadth and depth.

The analysis is lead by the text. The text is 'listened' to with a quiet mind, the 'speaker' is empathized with and there is openness to what the text is saying. It is not the intention of the analysis to deliberately impose an external understanding on the text but to allow the meanings of OOP to emerge from the text itself. The text is the sole source of the claims and counter-claims referred to in this analysis and the meanings of OOP noted in the analysis derive entirely from the text. The analysis attempts to capture much of the debate in the words of the participants themselves. Categories of claims were not conceived prior to the reading of the text. Broad conceptualizations of the arguments were allowed to develop and evolve during the actual reading and analysis of the text. These became the themes of discussion that emerged from the online discussion itself.

The understandings derived were continually revised through several passes through the text. After the initial selection of the threads of discussion for further analysis the individual discussion threads were analyzed. The meaning of the specific claims and counter-claims were appropriated as broad themes of discussion and the salient points made under each theme emerged. The detail and the whole were reconsidered over several more passes through the discussion threads until the final revision of the analysis presented here.

5.1 The Theme of 'Meaning'

Unsurprisingly, the very meaning of OOP is vigorously debated. A common complaint in the online discussion is the lack of clarity about '*what OOP is really all about*'. "*The OO waters are already so hopelessly muddied that no single definition will please even a significant fraction of 'OOP People'.*" (Appendix, LTU 8). It is argued that we cannot discuss the advantages and disadvantages of OOP without a precise meaning of the term OOP. There's also the belief that OOP is '*much too young*' for its theoretical framework

to be solid. OOP supporters feel that the lack of consensus on what OOP really means is used by detractors of OOP to try and discredit OOP.

According to certain proponents of OOP the core principle of the OOP paradigm is the simulation or modelling of reality: One simply models the phenomena: *"Originally, OO meant to simulate the world around us, by transferring the concepts of the real world to the computer."*(Appendix, ART 8). As one OOP supporter explains further: "*The goal of 99% of all system developments is to make the computers in a knowledge domain act and understand the domain like the humans do, in the same knowledge domain. This is, and has always been, the goal of OO development since it first started...*" (Appendix, ART 27). This particular meaning of OOP is closely related to the 'naturalness' argument.

The meaning of OOP is strongly linked to traditional programming by some participants. "*OOP is nothing more than procedural programming organized in a different way. OOP essentially binds code together with data. Procedural programming does not enforce organization of code.*" (Appendix, LTU 21). OOP is described as a means of associating the right combination of procedures with state in order to process a message therefore it is regarded as an important step in the evolution of programming paradigms in this school of thought. OOP is seen to provide a way of fully implementing the process of abstraction: "*What I am claiming is that non-OOP languages are lacking some of the fundamental requirements of abstraction (specifically association of state with things that perform actions), which thus means attempts at abstractions in those language will never be flexible (or even good).*" (Appendix, W&C 3). However, OOP should still be regarded as a '*paradigm shift*', according to one OOP proponent; even though OOP may have probably started as a collection of good practices that helped organize large procedural projects, once it was formalized it actually produced a new way of thinking about problems (Appendix, W&C 5).

OOP skeptics in the online discussion dismiss OOP as just a new name, re-invention or extension of old concepts. *"If slots also contain functions then it essentially ends up being OOP."* (Appendix, LTU 6). An object also can be regarded as just a namespace at the basic quantum level nothing more: *"One could say that in OOP, namespaces are*

first-class values." (Appendix, ART 17). Namespaces are said to provide many of the key OOP concepts; encapsulation is achieved because a namespace is a collection of variables and functions; inheritance statically extends the scope of the 'base namespace" to the 'derived namespace" while polymorphism dynamically scopes functions at runtime. Similar arguments are made with regard to the module, OOP can be regarded an extension of the module concept if one adds support for polymorphic module interfaces, or that abstract data types with inheritance (and polymorphism) is OOP.

An important thread of discussion within the broader 'meaning of OOP' dialogue is the contentious role that 'key' concepts like objects, classes, encapsulation, inheritance, polymorphism and message passing actually play in defining OOP. *"To some classes with inheritance is central to OO while others may consider programming with prototypical objects (without classes) as still being OOP."* (Appendix, LTU 1). The argument is made that OO concepts came ahead of any formal framework or method which describes OOP therefore the definition of OOP is '*straightforward*', simply using classes, objects, encapsulation, inheritance and polymorphism in ones software makes it OOP; everything else is irrelevant to the OOP definition argument. (A more specific analysis of the online discussion on each of the key OOP concepts follows in section 6.6, 'Key OOP concepts'.)

It is proposed that we take our definition of OOP from '*the founding fathers*' since they are the originators or OOP. Alan Kay, creator of the Smalltalk programming language, is said to have defined the ubiquity of objects, message passing, classes and inheritance as key attributes of OOP. However, it is disputed that these attributes were meant to define OOP: *"...it is made clear that the above are the design principles behind Smalltalk, but nowhere it is implied that they should also serve as definition of ObjectOriented."* (Appendix, W&C 2). The rebuttal to the 'founding father' definition is that the '*street*' that defines OOP; common usage should define OOP: "*The issue often revolves around whether common usage should define it, or the originators. If the originators define it, then do we go with Alan Kay or Kristen Nygaard? Are their definitions out of date? After all, over time the industry learns more."* (Appendix, W&C 15). Computer pioneer Kristen Nygaard, creator of the Simula programming language, is said to have defined

OOP as follows: *"A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world."* (Appendix, W&C 17). There are also contentions between Kay's and Nygaard's definitions of OOP. Certain OOP purists dismiss Nygaard's invention Simula, as *'nothing but ALGOL with classes'* compared to Smalltalk, which they regard as the *'holy grail'* of OO.

Salient points made under 'Meaning' theme:

- The meaning of OOP is so confused that any definition will please only a fraction of the software development community.
- The lack of clear definition of OOP is used by detractors to discredit OOP.
- We can only have a meaningful dialogue about the advantages and disadvantages of OOP if we have a precise definition of OOP.
- OOP is too young to have a sound theoretical framework.
- The simulation or modeling of reality is fundamental to OOP; this is the reason why OOP came be in the first place.
- OOP is a way of organizing a traditional program to a higher level of abstraction therefore it is an evolutionary step in traditional programming.
- OOP is nothing new it is just a re-invention or new name for old concepts like namespaces and modules.
- Simply using the concepts of classes, inheritance and polymorphism in program makes it OOP.
- Classes and inheritance don't necessarily define OOP since programming with prototypical objects that don't require either classes or inheritance can still be considered OOP.
- We must take our definitions of OOP from the founding fathers.

- The 'street' defines what OOP means.
- The programming language Smalltalk defines OOP.
- OOP is a procedural programming organized in a different way.
- OOP provides a way of fully implementing the process of abstraction in procedural programming.

5.2 The Theme of 'Hype'

OOP detractors claim that hype is the real driving force behind the rapid adoption of OOP by the developer community: *"OOP is for some reason a very productive 'fad factory'. It generates all kinds of buzzwords and ill-defined concepts that everybody is encouraged to use, but the why's, when's, and how's are not very well defined. This includes patterns, SOA, responsibility-driven design, MVC, etc."* (Appendix, W&C 3).

The hype is supposed to emanate from various sources:

- The unscrupulous marketing tactics of corporate software vendors
- The overinflated claims originating from OOP experts,
- Inconclusive academic studies.

It is argued that large corporations convince companies that their OOP tools are the '*golden hammer*'. OOP tools are sold and advertised for all kinds of purposes, far exceeding their real scope of application. Developers who are employed by these companies or supply software to them are then compelled to use these tools even if they believe them to be inappropriate. The programming languages Java and C-Sharp are touted as examples of this practice. Programming languages provide the means to manipulate programmers: "*Providing a language gives one a very powerful method to influence software construction, control programmer behaviour and help produce the kinds of software one is interested in*". (Appendix, LTU 10).

Overinflated claims from OOP experts are also blamed for influencing the software development community into accepting OOP. Early '*evangelists*' of OO tried to present OO as a revolutionary change: "*I feel much of the backlash is against the 'revolutionary' portrayal rather than a reasoned review of the advancements solidified in OO specific languages.*" (Appendix, W&C 5). It is claimed that '*Argument from authority*' is accepted by zealots as fully legitimate evidence yet the benefits of OOP claims are largely unproven. These OOP experts are seen to be promoting OOP for their personal gain.

Some critics ask for empirical evidence to back up OOP claims. They do not believe that there is much empirical evidence to support (or perhaps refute) OO claims: "*It's the job of OOP proponents to submit convincing evidence in support of their claims.*" (Appendix, LTU 1). Whatever evidence does exist is not regarded as conclusive. Critics argue that while some empirical studies show support for OOP claims other studies refute these same arguments. Yet other participants believe that the theoretical aspects of OOP can be discussed sensibly without empirical evidence. For advocates of OOP's naturalness the '*naturalness*' of OOP is obvious and does not require further objective validation. Additionally, there's the belief that new techniques, like OOP are judged by the marketplace and mindshare and not by scientific methods. A practitioner of OOP comments: "*OOP in practice is different from OOP in theory therefore practitioners of OOP know better than academic theorists?*" (Appendix, LTU 21); the counter-argument being that it is reasonable to expect professionals and academics to know the subject matter of OOP far better than amateur developers who appear to be the main anti-OOP campaigners.

The hype theory of OOP detractors is treated with skepticism not only by fervent OOP proponents but also by more neutral observers. The claim of hype is bandied about, yet equal argument is made that there is no evidence that the popularity of Java or C-Sharp, for example, is actually due to 'over-selling'. As one hype skeptic comments: "*Is it possible that the commercially successful languages like Visual Basic, Java and C-Sharp gained popularity because they provided programmers with what they wanted rather than what someone thinks they need?*" (Appendix, LTU 10).

Between the claims that OOP is mainly just hype and the counter-claim that much of the hype is justified is the more tempered argument that OOP has indeed been oversold but OOP cannot simply be dismissed as a mere fad with no substance. There are real benefits to OOP. It is not the fault of OOP itself that its benefits have been overstated.

Salient points made under the 'Hype' theme:

- OOP's popularity and acceptance is driven mainly by hype; claims are accepted without conclusive proof.
- There is no real empirical evidence to support OOP claims; empirical studies in support of OOP thus far are inconclusive because they contradict one another.
- OOP benefits like its 'naturalness' are obvious so they do not require proof.
- OOP is now popular among software developers because it fulfills genuine needs.
- OOP detractors are mainly uninformed amateurs therefore these detractors should accept the benefits touted by the OOP experts because the experts know better.
- OOP may be oversold to a certain extent but it cannot be dismissed as a mere fad because there are real benefits to OOP underneath all the hype.
- OOP is sold and advertised for all kinds of purposes, far exceeding its real scope of application.
- Much of the backlash is against the revolutionary portrayal of OOP.
- OOP in practice is different from OOP in theory.

5.3 The Theme of 'Purity'

Proponents of OOP purity in the online discussion regard a pure OO program as one where the only types are objects and all code is method code (i.e. functions cannot exist outside the class). The 'Purists' regard languages like Smalltalk and Eiffel that strictly

enforce these rules as being pure OOP languages in contrast to languages like C++ and Java that are regarded as hybrid languages because they do not enforce these constraints.

On the one extreme the argument is made that a language can only be considered OO if it is pure: *"To qualify as OO, a language must be pure. Smalltalk is pure OO; everything is an object. Hybrid languages like C++, with OO features and a bunch of other stuff to boot, don't qualify."* (Appendix, W&C 11). The claim is made that some vendors hack OO features into their language rather than design OO as a core part of the language. OOP purists are critical of C++: "*C++ is a real mess perhaps because a language like this is not really designed, it just happened.*" (Appendix, LTU 12). Purists consider OOP to be most useful in pure OOP languages.

It seems that not all practitioners of OOP require the programming language itself to be 'pure' in order for them to develop OO programs. They separate language support for OOP concepts such as encapsulation, inheritance and polymorphism from the OO purity of the language. "*Java isn't pure OO because some types aren't objects. C++ is even less pure, because some code isn't method code. But Java and C++ are still OO because they provide capabilities such as encapsulation, inheritance, polymorphism, abstract data types, classes ...*" (Appendix, W&C 12). A developer claims that there are reasons why in certain languages like Python one can't subclass every single object. According to him, there are good reasons why everything is not an object. While those reasons may not be acceptable it never compromised the design of his applications. He simply put some wrappers in place which made real objects out of primitives (Appendix, ART 11).

Detractors of purity consider the quest for OO purity to be fanatical rather than being based on sound practical considerations: *"When we consider everything an object, we miss good design choices."* (Appendix, LTU 1). They believe that OOP design does not effectively deal with a large category of problems. One should be able to '*mix-and-match*' paradigms to the development effort, using OOP concepts where useful and other paradigms if needed. It is purported that even the most ardent OOP defenders would resort to procedural ways of programming in practice when necessary. Anti-purists

believe that the state of programming is never going to improve if programmers are taught to approach problems in only one way.

The pursuit of OO purity is contrasted with the advantages of multi-paradigm programming. C++'s flexible nature is praised: *"One of the advantages of C++ is its multi-paradigm nature. It supports the OO paradigm, imperative programming, functional programming and also generic programming."* (Appendix, LTU 30). Proponents of the multi-paradigm approach want to be able to combine programming concepts dependent on the problem at hand rather than be straight-jacketed into a particular programming paradigm. They doubt whether the programming paradigms are really separate since no real differences seems to exist between the so-called paradigms. It is claimed that even some OOP experts have called that the so-called object paradigm a hodgepodge of the paradigms that preceded it.

Detractors of the multi-paradigm approach claim that the more concepts a programming language supports or implements the more difficult the language becomes to learn: *"Maybe in some circumstances multiple paradigms would reduce the code size or maintenance effort by a few percent, but is it worth mixing paradigms to get this few percent? It may take twice the training to get that few percent."* (Appendix, W&C 14). While supporters of the multi-paradigm approach concede that multi-paradigm programming languages are more difficult to learn initially they choose a steeper learning curve over restriction of alternatives in order to achieve the best solution.

A key contention within the multi-paradigm debate is the relationship between the OOP paradigm and the relational paradigm. Relational database technology is considered by its supporters to be a very mature and elegant technology for the modeling and storage of data. Proponents of relational database technology believe that OOP advocates unnecessarily reinventing the database, '*in too many ways*', in the quest for OO purity. The elegance of the relational paradigm is praised: *"The relational model is a lot more elegant and fundamental than some OO approaches. OO behaviours should be built on top of a relational model using composition and delegation."* (Appendix, LTU 13). Multi-paradigm advocates believe that OOP and the relational paradigm can get along if

OOP is restricted to a programming technique and doesn't try to model data; OOP and the relational paradigm need not be mutually exclusive or in opposition to each other.

Salient points made under the 'Purity' theme:

- OOP Purity is so vital to the OOP paradigm that there can be no talk of OOP without the concept of purity.
- OOP can only be effectively developed in a pure OO language since hybrid languages simply hack in OO features rather than design then into the core of the language.
- Hybrid languages are more complex requiring a steeper learning therefore the effort outweighs the benefits one gains from hybrids.
- The quest for the ubiquity of OO is fanatical because elegant non-OO technologies like the relational paradigm are re-invented in pure OO without sound reasons.
- A single programming paradigm that excludes other ideas is not good for the advancement of software development in general.
- The multi-paradigm approach is more flexible and adaptable to the problem at hand since it does not prescribe a uniform approach to all problems.
- Programming paradigms are not really inherently separate or mutually exclusive since they borrow and build on ideas from each other.
- A program does not need to be OO pure in order to be considered OO; it is the use of concepts such as concepts encapsulation, inheritance and polymorphism that make it OOP.
- OOP behaviours should be built on top of a relational model.
- OOP and the relational paradigm need not be mutually exclusive.

5.4 The Theme of 'Naturalness'

Proponents of OOP's naturalness argue that the core principle of the OOP paradigm is the simulation or modelling of reality: *"Originally, OO meant to simulate the world around us, by transferring the concepts of the real world to the computer."*(Appendix, ART 8). As one supporter of OOP's naturalness explains further: "*The goal of 99% of all system developments is to make the computers in a knowledge domain act and understand the domain like the humans do, in the same knowledge domain. This is, and has always been, the goal of OO development since it first started...*" (Appendix, ART 27). OOP's naturalness apparently closes the conceptual distance between the actual problem space and the computerized solution. This is supposed to make it psychologically simpler for humans to translate or map the conceptual design in the problem domain to a software solution. According to supporters of OOP's naturalness one simply models the phenomena; OO concepts like inheritance reflect the hierarchy of categories formed by the nouns in the spoken language.

Teaching and learning OOP is supposed to be made simpler by its naturalness: *"OO requires minimal training because the idea of objects is so close to real life conceptually."* (Appendix, LTU 21). Yet it is conceded that mindsets evolved for other programming paradigms are more difficult to change; even a good academic background does not make OOP easy in these cases: *"Indeed a good computer science background can make learning OOP more difficult because they fail to move from the imperative or functional styles to a truly OO style."* (Appendix, LTU 17).

Perhaps no other OOP claim is treated with the same degree of contempt as the claim of OOP's naturalness. *"The concept of modeling the real world has been so badly abused in the OO community, that I want to find the person who first coined the notion and flay him alive (not really) ."* (Appendix, ART 14). Strong argument against OOP's naturalness also comes from those who are not necessarily against OOP concepts in general. *"The concepts of OO are relatively simple (polymorphism, inheritance, wrapping data with its operators, etc.) It is just the application of these concepts to the real world that is the messy part."* (Appendix, W&C 18). The feeling is that even if we accept that the world

consists of objects this does not mean that it is natural to map the objects in reality to software objects and classification hierarchies. It is more correct to say, according to the 'anti-naturalists', that we are using vocabulary from the world; it is the vocabulary and relationships of the problem domain represented in our objects and methods.

Counter-claims against OOP's '*naturalness*' stem from OOP's supposed assumptions about the real world. These assumptions are regarded as false, artificial and restrictive. The most basic assumption made that everything in the real world is an object is treated with contempt by the 'anti-naturalists'. There are apparently a few common concepts that people universally use to understand and describe all systems that does not fit into the object mold. One such concept referred to in the debate is the sequence of the routine itself, what comes before under what conditions based on what causality, has no meaningful representation in OOP. Another is the apparently false OOP assumption that there is only one primary noun per action or operation. Critics push their point home by questioning the '*absurdity*' of the assumption that the world divides nicely into type taxonomies: *"The real world does not form and change in a hierarchical way in some domains."* (Appendix, W&C 3).

On a more philosophical level, the question posed is whether we can model a '*real world*' when we cannot agree on exactly what the '*real world*' is: *"What exactly is the 'real world'? There are possible answers as the number of human beings on the face of the earth."* (Appendix, ART 14). Critics find the real world modeling idea proposed by experts hard to believe because '*meta-physicists have been grappling with the true nature of reality for centuries*' without any success. In essence the universality of OO is questioned: *"OO may model some minds better, but certainly not all."* (Appendix, W&C 3).

Critics claim that the real value of OOP is good program organization. Most program classes have to do implementation: *"Developers mainly use class hierarchies for modularizing and parameterizing code rather than modeling the problem domain."* (Appendix, LTU 21). Code it seems gets split into classes in order to facilitate reuse, not because of the ontology of the problem being tackled. Therefore class instances don't

correspond so much to objects in the problem domain, as to components in the system that solves it. It follows from this point of view that many objects in the software are seen to have no counterpart in the real world. Collection classes, singleton classes and many of the classes discussed in OOP design patterns are presented as examples of classes that are purely software constructs. Criticism against 'naturalness' also comes from the area of distributed computing, objects in distributed computing '*get cloned across time and space*' yet this is not observed in reality by the critics.

Salient points made under the 'Naturalness' theme:

- The simulation or modeling of reality is fundamental to OOP; this is reason why OOP came be in the first place.
- Humans find it psychologically easier to translate the conceptual design to a software implementation when using OOP.
- Inheritance enables us to naturally model taxonomies found in the real world.
- OOP requires minimal training because of its naturalness however mindsets evolved for other programming paradigms confuses the issue since it can indeed make the transition to OOP more difficult.
- The naturalness of OOP is the most abused and misleading notion of OOP.
- The OOP concepts of classes and inheritance and OOP are not applied to model the real world rather these concepts are used to modularize and parameterize code in order to facilitate reuse in practice.
- While the vocabulary and the relationships of the problem domain are represented in the OOP software system we do not model reality in OOP.
- Philosophical assumptions of the real world made by naturalness supporters are false since not everything in the real world is an object.

- Collection classes, singleton classes and many of the classes discussed in OOP design patterns are presented as examples of classes that are purely software constructs.

5.5 The Theme of 'Software Quality'

OOP proponents claim that OOP improves the quality of software. OOP supposedly makes maintenance easier, improves the structure of code, improves correctness and increases reuse. *"It's often much easier to create correct code with OO off-the-bat, in my experience."* (Appendix, ART 8). However, these same supporters also caution about not realizing these virtues if developers do not design correctly or if OOP concepts are incorrectly applied: *"OOP does have value however it can make programming more difficult if it is applied incorrectly."* (Appendix, LTU 20).

It is contended that the best procedural programs are effectively OO in structure and could easily be translated to OO programs in an OOP language. The claim goes even further as OOP is seen as '*codifying*' some of the best practices in procedural programming. OOP is believed to give procedural code better structure: *"Average OO code tends to have better structure than average procedural code,... For the majority of developers, however, OO structure constrains them to better practices."* (Appendix, W&C 5)

Indeed the very quest to produce quality software, it seems, has led us towards OOP. As one contributor explains, the size of programming projects increased our need for more structure to help us manage the complexity of the project led us to OOP: *"OOP gave us well defined sets of procedures operating on well defined data structures to help us manage large numbers of procedures."* (Appendix, W&C 5). A developer explains how he discovered OOP '*on his own*' in the quest to keep his code organized. He discovered that the best way to keep large C programs manageable was to impose rules upon himself in order to keep the code organized. Structures where built to hold data and he restricted himself to only accessing those structures through a set of functions. The structure and its functions all went in a common header file by themselves, with a single .c file for the function implementations. It was only later that he learnt, to his surprise no doubt, that

this technique was an already existing technique called Object-Oriented Programming (Appendix, W&C 5).

While OOP may help software quality some participants feel that this is not guaranteed; the generally accepted key concepts of OOP (encapsulation, inheritance and polymorphism) do not by themselves make software '*good*'. OOP cannot save a project from bad practices for example. OOP proponents don't see this as a problem specific to OOP since no other programming paradigm would do any better. It is argued that OOP is often unfairly blamed: *"One thing I've noticed is that a lot of failures are blamed on OO or Java or C++ when, in fact, they are just due to plain old bad project management practices."* (Appendix, W&C 18). OOP practitioners warn that OOP concepts should be applied pragmatically, *"If it's not helping, don't use it."*. While it is conceded that OOP may provide the tools it is criticized for not doing a good job of telling us how to use them correctly: "*...what does make an OO app "good" has been difficult to describe. OO may describe the brush and the paint can, but it's not describing how to paint a room well yet.*" (Appendix, W&C 18).

The extreme OOP detractor position is that OOP does not contribute anything to the improvement of software quality; in fact it is seen to be doing the exact opposite. Inheritance is said to increase coupling and reuse is supposedly a farce. Correctness apparently takes too much effort: *"Having seen how much effort it takes to make an OO program "steady" (i.e. to approach, let's say, 90% correctness), I do not think OO has contributed anything to solving the issue of correctness, which is fundamentally a problem of mathematics."* (Appendix, ART 29).

OOP detractors claim that OOP makes code less agile, that is, less changeable and more difficult to maintain. Inheritance apparently increases coupling because derived classes can inherit code from a common base class therefore any change to the internals of the base class can impact on the correct functioning of its subclasses even if there no external behaviour or interface changes. OOP proponents retort that inheritance is not dangerous because it introduces judicious coupling, in a controlled manner, necessary to maintain code cohesion or the logical grouping of related segments of code: "*...often what*

happens is that a base class is created solely to represent common functionality, so coupling between this base class and inherited classes is to be expected." (Appendix, ART 4). From the OOP proponent's perspective it is impossible and somewhat meaningless to remove coupling altogether; the derived class should be regarded as an extension of the base class rather than coupled to their base classes: *"A derived class is logically equivalent to a single class that combines the functionality of the base and the derived class."* (Appendix, ART 4). It is conceded that multiple classes inheriting from the same base class inherit all of the same shortcomings of the model but this should not be taken to mean that they are coupled.

OOP detractors feel that OOP cannot lay special claim to enabling reuse: *"In order to write reusable code, you have to have a strategy for reuse. A lot of people will tell you OO is one such strategy and they are wrong."* (Appendix, ART 22). OO is said to be very useful in creating reusable components but it's not the only one and by itself doesn't solve anything: *"Just because you use objects in code doesn't make them reusable."* (Appendix, ART 22). Paradigms apparently cannot just produce reusable components by themselves. Another factor raised in the reuse debate is the freedom of choice OOP gives to developers: *"The only way to achieve guaranteed interoperable and truly reusable components is to straitjacket the programmer into a unified framework that solidly enforces the right way to do things while not restricting what things are possible."* (Appendix, LTU 13).

OOP critics also accuse OOP of allowing global variables at class scope creating the same kind of problems that global variables cause in procedural programs. The software quality issue linked with global variables is that global variables are not clearly associated with the code the uses them making maintenance difficult. OOP supporters refute this accusation. *"This isn't as big a problem as the global variables of yore yet OOP is like spaghetti code in its own way. The situations is much better now because instead of having one large platter of spaghetti you have several smaller plates of spaghetti."* (Appendix, LTU 24). They claim that the real issue here is the misuse of the class mechanism. *"If the use of instance variables becomes as confusing as system-wide global variables, then the class is much, much too big. It should be split into multiple,*

collaborating classes." (Appendix, LTU 24). It seems that just as a global variable is not a problem in a small program, so to an instance variable is not a problem in a small class.

Salient points made under the 'Software Quality' theme:

- OOP improves maintenance by solving problems like the global variable problem (if the class concept is applied correctly).
- OOP class and inheritance concepts help improve reuse.
- It is easier to create correct code in OOP from the very start (however OOP concepts must be applied correctly).
- OOP codifies the best practices in procedural programming; the best procedural programs are really OO programs.
- The very quest for quality in large software systems has led us to OOP.
- OOP is unfairly blamed for project failures and poor quality software when bad project management is really the real culprit.
- If OOP has not achieved reuse in practice then it's only because OOP tool implementations gives developers too much choice currently; they need to be straitjacketed into a particular OO framework that enforces the right way of doing things.
- Simply applying OOP concepts does not seem to guarantee software quality since OOP provides the tools but it does not tell us how to use them correctly.
- OOP degrades software quality because the concept of inheritance increases coupling and the reuse that is being claimed has not materialized in practice.
- If takes just as much effort to achieve a steady (correct) OOP system compared to other types of systems since the problem of correctness cannot be solved by OOP because it is fundamentally a mathematical issue.

- OOP cannot lay special claim to improving reuse since what is really required is a reuse strategy, which will also lead to improved reuse in other paradigms, not just OOP.

- OOP does not solve issues like the global variable problem because instance variables are still really global to the class.

5.6 The Theme of 'Key Concepts'

5.6.1 Encapsulation

Encapsulation is the fundamental core of OOP, according to some participants: *"Most of what we call OO is in fact just a set of techniques or programming language constructs that allow for the organization of code and data into separate components with localized scope."* (Appendix, LTU 21). OOP is said to rely heavily on encapsulation to create high-level objects. Encapsulation is apparently the '*major issue*' of OOP because it allows the management of the context and scope of related elements of data. It is claimed to be the one OOP principle that is common to all definitions of OOP therefore it is argued that any approaches and design that ignores encapsulation is not OOP.

Encapsulation is seen as the key distinction between OOP and procedural programming. In even broader terms, encapsulation is supposedly central to the evolution of programming languages: *"Agreed - encapsulation is the key. We can see that OO programming is simply the next generations of programming languages - so called 4GL languages are a side branch, not part of the main sequence. At each stage - assembler, macro assemblers, procedural, object oriented - stuff gets wrapped in containers. You can make a thing, test that it works, and then wrap it in a container and use it by name."* (Appendix, W&C 22).

However, the uniqueness of encapsulation to OOP is questioned since a procedure can be regarded as a type of encapsulation because it combines a series of computer instructions and hides the detailed instructions and local variables. Encapsulation is seen to be closely related to the idea of abstraction as a whole: *"Data abstraction and encapsulation are entirely independent of OOP; the class facility is just one way of implementing them"*.

(Appendix, LTU 21). According to one developer, encapsulation is the only good thing about OOP because it forces one to put things together, something that a disciplined programmer can easily achieve without explicit support from OOP.

Encapsulation seems to also implicitly refer to the concept of information hiding because participants make no clear distinction between the two concepts. *"The fundamental trade-off in encapsulation is that it hides implementation details and reduces code duplication."* (Appendix, ART 6). Too much information hiding is said to reduce flexibility: *"Classes can be too encapsulated as in some APIs that are over-engineered and less useful that they could be. Designers try to anticipate every need except the need to be flexible."* (Appendix, ART 6). Encapsulation (inclusive of information hiding) supposedly makes the API '*thicker*' and harder to understand.

Salient points made under the 'Key Concepts - Encapsulation' theme:

- Encapsulation is a key defining concept of OOP since the class is an organization of code and data into separate components with localized scope.
- Encapsulation is fundamental to programming language evolution as a whole defining each stage of the evolution of programming.
- Encapsulation is not a concept unique to OOP; the function for example, is also a form of encapsulation.
- Classes can be too encapsulated rendering then less useful because too much of the implementation is hidden and not directly accessible.
- OO is in fact just a set of techniques or programming language constructs that allow for the organization of process into separate components with localized scope.
- Encapsulation is the only good thing about OOP because it forces one to put things together

5.6.2 Classes

The status of the class as a defining concept of OOP is debated. It is argued that classes are not necessary for OOP: *"Special objects called prototypes can be used instead of classes"*. (Appendix, LTU 21). The prototype mechanism works by cloning from a prototype object instead of instantiating objects from a class. Supporters of the prototyping approach claim that the prototyping mechanism is a more natural way of thinking about objects compared to classes.

The class mechanism is also accused of trying to do too much: *"The class mechanism does too much: encapsulation, modularization and subtyping. There should be separate mechanisms for separate things. Sub-typing and implementation heritance should be separated."* (Appendix, LTU 21). This is believed to confuse the role that the class mechanism plays in OO programs.

Salient points made under the 'Key Concepts - Class' theme:

- The class is a key defining concept of OOP.
- The class does not necessarily define OOP since OOP using prototypes does not require OOP.
- Too many OOP semantics are associated with the class mechanism making it a confusing concept.

5.6.3 Inheritance

The first point of contention is whether inheritance is a core OOP. It is argued that OOP without inheritance is just traditional programming with abstract data types. Other '*key concepts*' like polymorphism are supposedly dependent on inheritance: *"...However when polymorphism is needed, we have to use inheritance. There is no other way*" (Appendix, ART 12). Equally, polymorphism is said to be able to '*exist*' fine without inheritance. The concept of polymorphism is explored further in the next section.

While the majority point of view is that inheritance is a key defining concept of OOP there exists the claim that delegation is the real intent behind inheritance: *"Inheritance*

also isn't required in a prototyped based object system. Inheritance is really a form of delegation, so are prototypes, so delegation may be more a base requirement than inheritance." (Appendix, W&C 15). The diminishing importance of inheritance in OOP is apparently an '*admission of guilt*' that inheritance was initially oversold. Supporters of inheritance's 'special place' in OOP counter that while inheritance may be a form of delegation it is much broader and higher-level concept compared to delegation; delegation is really an implementation idea. They also plead for inheritance to be used with prudence and pragmatism: *"Inheritance is not a general programming solution, and all the OOP books saying so are doing OOP a great disservice. ... If inheritance causes problems, simply do not use it!"* (Appendix, W&C 3).

The rising importance of interface inheritance is noted in the online discussions. However, participants lament the ongoing confusion over the concepts of implementation inheritance and interface inheritance. *"It is important to understand and to be able to distinguish between Interface inheritance and implementation inheritance."* (Appendix, LTU 2). The generally accepted notion of interface inheritance in the online discussion is that the derived class only inherits the interface of the base class; the implementation can therefore be different for each derived class that inherits the base class interface. This is contrasted with implementation inheritance where implementation of the base class is also inherited. Many languages are said to make mistakes in their implementation of OOP by confusing the two types of inheritance: *"Inheritance can be used to connect abstract interfaces with possibly multiple implementations. It can also be used for code reuse (implementation inheritance) and for incremental implementation (extendibility). However the programming language has no way to tell which of these semantic interpretations apply in each case."* (Appendix, LTU 15).

The relative importance of interface inheritance and implementation inheritance is vigorously contested. *"The general feeling appears that inheritance of interface is preferable over inheritance of implementation. One comes across this sentiment so often that one wonders whether implementation inheritance should be included in modern OOP"*. (Appendix, LTU 13). Implementation inheritance is said to be associated with the problems of the fragile base class, unwieldy monolithic class hierarchies and multiple

inheritance. Inheritance of implementation is apparently grossly overrated compared to interface inheritance, which allows polymorphism without the associated implementation inheritance problems: *"Inheritance of interface is the key to separation of interface and implementation giving rise to polymorphism."* (Appendix, LTU 15).

However, there is still support for implementation inheritance. It is argued that it is really the reckless use of the concept by programmers that have given implementation inheritance its bad name. The concept offers an easy mechanism for code reuse: *"Inheritance of implementation is easily abused. There are lots of examples where it is poorly done and a few where it is well thought-out and really gives the developer leverage."* (Appendix, LTU 13). This abuse is also promoted by the abundance of poor literature on OOP: *"When programmers first learn an OO language what they learn about OO is in the introductory chapter of a programming language book that mostly focuses on inheritance."* (Appendix, ART 6). Implementation inheritance is said to work nicely in small components. According to one developer, code sharing via inheritance is still very important in OOP: *"Inheritance of implementation is important for code sharing. If two classes have implementations in common then they should probably have a common superclass where the common implementation actually resides."* (Appendix, LTU 15).

Composition is supposedly a better choice than implementation inheritance; today's OO practitioners apparently favour composition over inheritance. Participants regard a compositional relationship as the link established between two objects because one object creates and uses the other object within its implementation instead of the one object inheriting from the other object. Advocates of composition claim that it yields code that is easier to change however its disadvantages are also acknowledged: *"Although composition in general yields more flexible code than inheritance, composition yielding code that's easier to change, but not necessarily easier to* understand." (Appendix, ART 5). Detractors of composition argue that it '*munges*' the '*is* a' and '*has a*' relationships that exists between objects. There is also the reuse debate among OOP practitioners themselves, some argue in favour of composition because it said to offer better reuse over inheritance.

Inheritance is apparently not unique to OOP: *"Inheritance is not the important concept in OOP if one were to consider inheritance purely as code sharing and something different to subtyping with interfaces. Inheritance can be used in functional programming just as easily as in stateful programming."* (Appendix, LTU 8). Some participants regard this as a red-herring for distinguishing between OOP and other programming paradigms like functional programming. It is argued that separating interface from implementation is a basic design strategy not unique to OOP.

Perhaps the harshest criticism is reserved for the use of inheritance to model taxonomies in the real world: *"Hierarchical taxonomies of just about anything are usually problematic, except for the trivial perhaps. I think software engineering has gotten carried away with trees."* (Appendix, W&C 13). Critics say that even in the life sciences taxonomies tend to be somewhat arbitrary because bacteria and other organisms can grab DNA and stick them in a far-off organism, busting the tree. While hierarchies are good for front-end navigation, detractors say that they should not serve as the only organization mechanism, especially internally. It is argued that classification hierarchies are extremely subjective and dependent on whom the '*observer*' is.

The arguments made about the impact of inheritance on software quality, namely, maintainability, coupling and reuse have been dealt with in the section on software quality.

Salient points made under the 'Key Concepts - Inheritance' theme:

- Inheritance is a core OOP concept.
- Inheritance enables reuse.
- Inheritance is used to model real world taxonomies.
- The confusion between interface inheritance and implementation inheritance must be cleared.

- Interface inheritance is more prominent than implementation inheritance because it creates polymorphic code without the maintenance issues.
- Implementation inheritance is abused by programmers because it seems to offer an easy code reuse mechanism that negatively impacts software quality.
- Composition is better than implementation inheritance because it creates more maintainable code.
- Polymorphism is dependent on interface inheritance in OOP.
- Interface inheritance is not unique to OOP.
- The use of inheritance to model real world taxonomies leads to inflexible designs.
- Composition offers better reuse compared to inheritance.
- OOP without inheritance is just traditional programming with abstract data types.
- The diminishing importance of inheritance in OOP is an 'admission of guilt' that inheritance was initially oversold.
- Delegation is the real intent behind inheritance.
- Inheritance is a broader and higher-level concept compared to delegation; delegation is really an implementation idea.

5.6.4 Polymorphism

There is ample support for polymorphism's claim to key OOP concept status: *"The key aspect of OOP is polymorphism, which is the ability to respond to a message differently depending on the type."* (Appendix, LTU 15). Some contributors accept that polymorphism is not unique to OOP and can be achieved without inheritance; generic functions, for example, provide a form of polymorphism however they supposedly aren't the answer in many scenarios. However, polymorphism is supposed to work well with OOP's interface inheritance: *"I never claimed polymorphism was tied to abstract data*

types, just that they are easier to work with when tied together for most cases. Simply put... the object provides much needed scope to the list of available messages." (Appendix, W&C 19). While some accept the argument that polymorphism can be done without inheritance, they claim that the only purpose of inheritance should be to provide polymorphism: *"...I will accept that polymorphism can be done without inheritance (although I would be interested in hearing the details). I would suggest, however, the only purpose for using inheritance is to provide polymorphism."* (Appendix, W&C 26).

Polymorphism is apparently the least understood tenet of OOP: *"Objects can present multiple views of itself through polymorphism which is probably the least understood tenet and probably the most powerful for creating good OO designs."* (Appendix, ART 3). Polymorphism is said to improve the structure of code: "*Polymorphism is an alternative to putting everything into if statements*". (Appendix, LTU 20).

Salient points made under the 'Key Concepts - Polymorphism' theme:

- Polymorphism a key OOP concept.
- Polymorphism is the least understood of the OOP concepts.
- Polymorphism improve software quality because it provides an alternative to putting everything into if statements.
- Polymorphism is dependent on inheritance in OOP.
- The only real benefit of inheritance is polymorphism.
- Polymorphism is not unique to OOP.

5.6.5 Messaging

It is argued that the fundamental idea of messaging is captured by the function call so there is no need for 'Smalltalk' style messaging in OOP. Smalltalk style messaging proponents claim that Smalltalk style messaging is central to OOP: "*OOP essential advantage is being able to send an arbitrary message to a 'thing' and have it respond.*" (Appendix, LTU 21). Other OOP supporters see the function call as being sufficient for

OOP therefore Smalltalk style messages is not core to OO. Smalltalk messaging proponents however insist on the privileged status of messaging in OOP: "*Objects act just like servers. One sends them messages - they respond. Messages need not even return (continuations). They can be forwarded. They can be filtered and selectively thrown on the floor.*" (Appendix, ART 3). While the messaging concepts of continuations and forwarding are regarded as '*great things*' in their own right it still doesn't define OOP, according to some supporters of the function call semantics.

Salient points made in the 'Key Concepts – Messaging' theme:

- Smalltalk style messaging is a defining OOP concept.
- Function call semantics are sufficient for OOP; the messaging semantics are great but they are not necessary for OOP.

5.6.6 Objects

Fortunately there is no dispute across the entire online discussion that the 'object' is a key concept of OOP however OOP detractors claim that the term 'object' is just another word for older ideas that already existed prior to OOP. These claims were largely dealt with earlier in the analysis, under the theme 'The meaning of OOP', but some of these claims are reiterated here for the sake of completeness. One participant regards objects as purely procedural entities; they are seen as glorified data structures with methods that help hide the implementation details. Similar comparisons are drawn between the object concept and older traditional programming concepts like the abstract data types. Another regards objects as simply a way of abstracting information: "*In some sense, objects are simply a way of abstracting information by joining the underlying state of the information with the code that manipulates the information.*" (Appendix, ART 28).

The relative strengths of the two methods of instantiating objects, the class method and the prototyping method are debated. Supporters of 'prototyping' consider prototypical objects to be more intuitive because this is closer to how objects are created in the real world. Supporters of class-based objects make similar claims: "*The main issue with prototype based object systems is that the prototypical parent object exists outside the*

client object. It is worrisome to have an abstract 'Animal' object floating around the system masquerading as concrete." (Appendix, LTU 13).

Some OOP supporters argue for the incorporation of stronger semantics for objects. Apparently, the creation of objects should be more structured: *"There should be a mechanism in each class for the instantiation of other classes in a certain way and not everywhere in every method."* (Appendix, LTU 14). They also ask for OOP to manage inter-object links in a consistent and robust way.

While 'objects' may be the one undisputed key concept of OOP, one OO educator asks why then are classes most often presented as the foundation of OOP instead of objects. He believes that it is easier to teach the interface inheritance concept if we place objects at the centre of OOP. The various interfaces of an object are the various roles that the object plays. A given object can take on several roles at once. A Circle object as an example: can ask it to yield its circumference, location, or area; it can be asked to move, rotate or warp: these owe to a role that might be called Shape. The Circle object may respond to requests to draw itself on some output medium, or to change its color: these it owes to a role called Drawable. An image file object might also respond to all the responsibilities of Drawable, though that may be all that images have in common with Shapes (Appendix, ART 23).

Salient points made in the 'Key Concepts - Objects' theme:

- The object is a key OOP concept.
- Objects are just another name for older concepts from traditional programming.
- The instantiation of objects through classes is better than prototype-based OOP since a prototypical parent object is abstract but masquerades as a concrete object.
- The instantiation of objects through prototype-based OOP is better than class-based OOP because this is closer to how objects are created in the real world.
- Objects require stronger semantics that prescribe best practice.

- Objects rather than classes should be at the centre of OOP since this make OOP easier to teach and understand.

6 Discussion

Neither the preunderstandings of the subject matter of OOP appropriated by this study from the literature nor the empathetic understanding of OOP that emerged from the text during the analysis can hold privileged positions. Each must be subject to modification in the knowledge of the other resulting in a fusion of horizons of understanding. A deeper insight into the online OOP discussion is appropriated by allowing the meanings 'in' the online discussion and the preunderstandings from the literature to encounter one another.

6.1 The Meanings 'Beneath' the Text

While the analysis uncovers the meanings 'in' the online OOP discussion this section uncovers the meanings 'beneath' the claims and counter-claims made in the online OOP discussion. It seeks to uncover underlying interpretations of OOP from a fusion of meanings in the literature with the meanings in the text in order to arrive at a deeper understanding of the OOP discourse. This deeper understanding also offers a form of explanation of the contesting claims and counter-claims. The underlying interpretations of OOP will not only help to explain the arguments made in the online discussion but also the entire OOP discourse in general.

The fusion of meanings in the literature with the meanings in the text reveals 3 different interpretations of OOP in the OOP discourse:

- The Revolutionary Perspective
- The Evolutionary Perspective
- The Traditionalist Perspective

The revolutionary perspective of OOP is the prevailing perception of OOP. It is the theory of OOP emphasized in the literature, shaping the perception of OOP that exists 'out there' on the street and in the popular media. The evolutionary perspective of OOP is constituted in the actual application of OOP. While it may very well be the prevailing practice of OOP it is not well articulated in the literature and only expresses itself incoherently in the OOP debate. The traditionalist perspective of OOP is forged in the

counter-reaction to the revolutionary perspective of OOP. It stages its vociferous critique against OOP from its basis in traditional programming also largely ignoring or unaware of the evolutionary perspective.

6.1.1 The Revolutionary Perspective

The revolutionary perspective is the viewpoint that OOP is a revolution in software development, quite apart from and different to traditional programming. It is the 'theory' of OOP emphasized in the literature shaping the popular perception of OOP. Normark (2007) considers OOP to be one of the four main programming paradigms in his classification and definition of the main programming paradigms. It is considered to be distinct from the imperative programming paradigm or traditional programming (Normark, 2007). The 'invented' version of the origin of OOP emphasizes the dramatic departure from traditional programming. Pawson (2004), states that OOP was invented in Simula in the 1960s and refined in Smalltalk in the 1970s. Simula was created to simulated real-world phenomena such as industrial processes, engineering problems and disease epidemics by building systems out of objects. Software objects model the properties and behaviours of the real-world entity. Simula broke away from tradition by challenging the practice of explicitly separating software into procedure and data.

Any link with traditional programming is largely ignored and only vaguely acknowledged. It is the difference from traditional programming that is emphasized. Meyer (1988) believes that OOP is not only different from but incompatible with the software design methods used in traditional programming and generally taught. From this perspective OOP uses objects instead of algorithms as the fundamental logical building blocks (Booch, 1994).

The revolutionary perspective views the concept of the simulation or modeling of reality as fundamental to OOP and OO in general. It emphasizes the building of real-world models, using an object-oriented view of the world (Booch, 1994). Taylor (1998) states that the most fundamental idea behind object technology is that real-world objects or objects of the problem domain are represented in software systems through logical units called objects. Object-Oriented analysis and design also emphasized a revolutionary

perspective of OOP. Since OO designs are an operational model of some aspect of the world, the objects are there for the picking (Meyer, 1988). Object-Oriented analysis is a method that examines the requirements from the perspective of the classes and objects found in the vocabulary of the problem domain. The objects in the software simulate, model and reflect these external objects therefore an OOP program is a simulation of reality (Booch, 1994).

The revolutionary perspective challenges the tradition of separating data and process. Objects subsume both data and process throughout the software system from analysis and design through to programming and object storage. The concept of process operating on data is extinguished at the conceptual stage during analysis and design. In contrast traditional methods of development consistently treat process and data as separate elements throughout the analysis, design, programming and storage stages. In the revolutionary perspective of OOP neither data nor process can exist independently of the object abstraction. Objects identified in reality during analysis become software artifacts in the program and are persisted in storage as objects. The OO program code is seen as an interaction of objects rather than the more traditional view of process acting algorithmically on data.

6.1.2 The Meaning of OOP

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- The simulation or modeling of reality is fundamental to OOP; this is reason why OOP came be in the first place.
- Simply using the concepts of classes, inheritance and polymorphism in the program does not make it OOP.
- We must take our definitions of OOP from the founding fathers.
- The 'street' does not define what OOP means.
- The programming language Smalltalk defines OOP.

The revolutionary perspective of OOP is the 'theory' of OOP emphasized in the OOP literature by the OOP experts. Participants in the online discussion who share the revolutionary perspective take their lead from the OOP experts. The meaning of OOP for them is quite clear.

6.1.3 Hype

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- OOP benefits like its 'naturalness' are obvious so they do not require proof.
- OOP is now popular among software developers because it fulfills genuine needs.
- OOP detractors are mainly uninformed amateurs therefore these detractors should accept the benefits touted by the OOP experts because the experts know better.

The many claimed benefits of OOP follow directly from the fundamental understandings of the revolutionary perspective of OOP. This perspective supports the claim that the problem domain is effectively and easily conceptualized as interacting objects. It derives from the belief that the 'object reality' is universal and a tacit acceptance that the problem domain and the nature of reality are the same. Even if the 'object reality' is indeed universal little attention is given to the possibility that the problem domain is different from the 'object reality' in which it exists. Therefore the possibility that conceptualizing the problem domain as interacting objects may not be effective is ignored from the revolutionary perspective. Participants in the online discussion with a revolutionary perspective will not perceive OOP to be hype.

6.1.4 Purity

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- OOP Purity is so vital to the OOP paradigm that there can be no talk of OOP without the concept of purity.

- OO programs can only be effectively developed in a pure OO language since hybrid languages simply hack in OO features rather than design then into the core of the language.

- Hybrid languages are more complex and require a steeper learning; it's simply not worth the effort various the benefits one gains from the hybrids.

- The quest for the ubiquity of OO is not fanatical because non-OO technologies like the relational paradigm need to be re-invented in pure OO for sound reasons.

From a revolutionary perspective OOP purity is vital. If reality consists only of objects and the software must model this reality then everything in the software should also be an object. There should be no need for data or process to exist independently of the object. The ubiquity of OO is important to maintaining the object from conceptualization in the problem domain to software and storage. Relational databases that only store data in a different organization to the object model will create difficulties. Participants in the online discussion with the revolutionary perspective will insist on the purity and ubiquity of OO.

6.1.5 Naturalness

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- The simulation or modelling of reality is fundamental to OOP; this is reason why OOP came be in the first place.

- Humans find it psychologically easier to translate the conceptual design to a software implementation when using OOP.

- Inheritance enables us to naturally model taxonomies found in the real world.

- OOP requires minimal training because of its naturalness however mindsets evolved for other programming paradigms confuses the issue since it can indeed make the transition to OOP more difficult.

For the revolutionary perspective the 'naturalness' of OOP follows quite obviously from the very idea that OO is fundamentally about the simulation or modeling of reality by software. This 'object reality' is the same for all people and all problem domains from a revolutionary perspective. Participants in the online discussion with the revolutionary perspective acknowledge the obvious naturalness of OOP.

6.1.6 Software Quality

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- It is easier to create correct code in OOP from the very start (however OOP concepts must be applied correctly).
- The very quest for quality in large software systems has led us to OOP.
- OOP is unfairly blamed for project failures and poor quality software when bad project management is really the real culprit.
- If OOP has not achieved reuse in practice then it's only because OOP tool implementations gives developers too much choice currently; they need to be straitjacketed into a particular OO framework that enforces the right way of doing things.

From a revolutionary perspective the positive impact on software quality derives from the belief that software models an object reality more effectively. Since the main issues with software quality derives from the difficulties of conceptualizing the problem domain correctly (Brooks, 1987) it would be easier to create quality software because modeling the problem domain as objects is 'natural' from this perspective. Revolutionary OOP failures are seen to be largely due to bad project management and poor OOP tool implementations. Participants in the online discussion with a revolutionary perspective will accept that software quality is dramatically improved.

6.1.7 Key Concepts

Participants in the online discussion who share the revolutionary interpretation will support the following salient points in the online discussion:

- The class is a key defining concept of OOP.
- Inheritance is a core OOP concept.
- Inheritance enables reuse.
- Inheritance is used to model real world taxonomies.
- Polymorphism a key OOP concept.
- The object is a key OOP concept.
- Smalltalk style messaging is a defining OOP concept.
- Polymorphism a key OOP concept.
- Polymorphism is not the only real benefit of inheritance.

There may be some support within revolutionary OOP for the following salient points:

- The instantiation of objects through classes is better than prototype-based OOP since a prototypical parent object is abstract but masquerades as a concrete object.
- The instantiation of objects through prototype-based OOP is better than class-based OOP because this is closer to how objects are created in the real world.
- Objects require stronger semantics that prescribe best practice.
- Objects rather than classes should be at the centre of OOP since this make OOP easier to teach and understand.
- Inheritance is a broader and higher-level concept compared to delegation; delegation is really an implementation idea.

From a revolutionary perspective the key OOP concepts of encapsulation, objects, classes, inheritance, polymorphism and messaging applied to conceptualization, programming and storage forge a unique software paradigm distinct from other software paradigms. They originate in the problem domain as conceptual ideas modeling the object reality before translating into implementation and storage concepts. Therefore from this perspective inheritance is a way of modeling objects in reality and not just a means to create reusable program code or polymorphic program code. The concept of 'messaging passing' models the interaction between objects in the real world more accurately than 'function calling' therefore 'messaging passing' will have more support than 'function calling'. Prototype-based OOP does not challenge the fundamental assumption of an object reality therefore it may find support within the revolutionary perspective even though class-based OOP is firmly entrenched in this perspective. Participants in the online discussion with a revolutionary perspective will view the key OOP concepts as being distinct and different from their application in other software paradigms.

6.2 The Evolutionary Perspective

The evolutionary perspective of OOP is of the view that OOP is an evolutionary step in traditional programming. It introduces a different approach to organizing process in traditional programming forging a new programming paradigm within imperative programming. The evolutionary perspective of OOP is constituted in the practice of OOP and not well articulated in the literature. In contrast with the revolutionary perspective this viewpoint emphasizes OOP's inextricable link with traditional programming.

While the evolutionary perspective of OOP does not represent the prevailing viewpoint in the literature there is some support for the notion that OOP concepts like encapsulation, inheritance and polymorphism build on basic programming principles incrementally advancing traditional programming. According to Sircar (2001), fundamental programming logic (decision and loop statements) is an essential aspect of both traditional and OO programming. Lewis (2000) states that mutual exclusivity of OOP and procedural programming is a myth. OOP does not abandon the concepts that we admire

in a procedural approach; it augments and strengthens them. Solving a problem by dissection into multiple and manageable pieces, modularity and encapsulation have been known for years. The best procedural programs, according to Lewis (2000), are the ones that are the most object-oriented. According to Fraser et al. (2005) it is not correct to say that algorithmic abstractions have been superseded by object-oriented ones. The beginnings of object abstractions are in structured methods while object-oriented methods echo algorithmic abstractions. Both structured and object-oriented techniques play a role in the construction of contemporary systems (Fraser et al., 2005).

The evolutionary perspective of OOP continues to support the traditional separateness of process and data in imperative programming. Traditional methods of development have consistently treated process and data as separate elements throughout the software, during analysis and design, programming and storage. The objects in an evolutionary OO program are software artifacts that organize process differently to the top-down, functional decomposition of process in traditional programming. The evolutionary OOP organization of process identifies and organizes the software objects required before algorithmic composition. Software objects are conceptually meaningful groupings of related functions and their program state. Program state is different from data, it may hold data from the problem data but it is not synonymous with the data from the problem domain. Unlike revolutionary OOP software objects do not subsume process and data, neither do they necessarily map to objects in reality. The functions in the software objects are still implemented as algorithmic compositions of process acting on data using traditional programming techniques.

The online OOP discussion indicates that the evolutionary perspective has substantial support in practice and it may indeed be the prevailing practice of OOP. Salient points emerge from across the themes of discussion supporting this interpretation of OOP:

- OOP is a way of organization a traditional program to a higher level of abstraction therefore it is an evolutionary step in traditional programming.
- OOP is a procedural programming organized in a different way.

- OOP provides a way of fully implementing the process of abstraction in procedural programming.
- Much of the backlash is against the revolutionary portrayal of OOP.
- OOP in practice is different from OOP in theory.
- The quest for the ubiquity of OO is fanatical because elegant non-OO technologies like the relational paradigm are re-invented in pure OO without sound reasons.
- Programming paradigms are not really inherently separate or mutually exclusive since they borrow and build on ideas from each other.
- OOP behaviours should be built on top of a relational model.
- OOP and the relational paradigm need not be mutually exclusive.
- The naturalness of OOP is the most abused and misleading notion of OOP.
- The OOP concepts of classes and inheritance and OOP are not applied to model the real world rather these concepts are used to modularize and parameterize code in order to facilitate reuse in practice.
- While the vocabulary and the relationships of the problem domain are represented in the OOP software system we do not model reality in OOP.
- Collection classes, singleton classes and many of the classes discussed in OOP design patterns are presented as examples of classes that are purely software constructs.
- OOP codifies the best practices in procedural programming; the best procedural programs are really OO programs.

- OO is in fact just a set of techniques or programming language constructs that allow for the organization of process into separate components with localized scope.

- Polymorphism improve software quality because it provides an alternative to putting everything into if statements.

- Function call semantics are sufficient for OOP; the messaging semantics are great but they are not necessary for OOP.

In addition to the support for evolutionary OOP in the online discussions the slow uptake of UML (Grossman et al., 2004) despite the popularity of OO programming may be another indication of the prevalence of evolutionary OOP in practice. Evolutionary OOP will be challenged to find real value in OO analysis and design because it maintains the separateness of data and process. The continued popularity of relational databases (Bloor, 2004) despite the availability of OO databases may also indicate that the traditional separation of process and data is being maintained in prevailing OOP practice. Relational databases stores and model the data while OOP models the process.

6.2.1 The Meaning of OOP

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- The simulation or modeling of reality is not fundamental to OOP; this is not the reason why OOP came be in the first place.

- OOP is a way of organization a traditional program to a higher level of abstraction therefore it is an evolutionary step in traditional programming.

- The meaning of OOP is so confused that any definition will please only a fraction of the software development community.

- The lack of clear definition of OOP is used by detractors to discredit OOP.

- We can only have a meaningful dialogue about the advantages and disadvantages of OOP if we have a precise definition of OOP.
- OOP is too young to have a sound theoretical framework.
- We cannot take our definitions of OOP from the founding fathers.
- The 'street' defines what OOP means.
- The programming language Smalltalk does not define OOP.
- OOP is a procedural programming organized in a different way.
- OOP provides a way of fully implementing the process of abstraction in procedural programming

The evolutionary interpretation of OOP presents an alternative to the prevailing revolutionary perception of OOP that exists in the literature and the popular media. While there is support for evolutionary OOP it is not a well articulated, coherent school of thought in the literature. However, it expresses itself in the actual practice of OOP, its critique of the revolutionary perspective's 'object reality' and the ubiquity of objects. Supporters of the evolutionary OOP lament the confusion around the OOP phenomenon and yearn for a deeper theoretical understanding of their perspective of OOP. The online discussion as a whole indicates that for a large number of participants the evolutionary interpretation of OOP represents the way they actually apply OOP in practice.

6.2.2 Hype

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- OOP benefits like its 'naturalness' are not obvious so they do require 'proof'.
- OOP is now popular among software developers because it fulfills genuine needs.

- OOP detractors are not mainly uninformed amateurs. The benefits touted by the OOP experts are not all true.
- OOP may be oversold to a certain extent but it cannot be dismissed as a mere fad because there are real benefits to OOP underneath all the hype.
- OOP is sold and advertised for all kinds of purposes, far exceeding its real scope of application.
- Much of the backlash is against the revolutionary portrayal of OOP.
- OOP in practice is different from OOP in theory.

The evolutionary perspective accepts the value of OOP as a different way of organizing imperative code. It believes that objects organize process better compared to functional decomposition therefore OOP is not all hype. However, it also values the traditional separation of process and data throughout the software, in the conceptual modeling, the program, and storage. Therefore it refutes the revolutionary interpretation of OOP. Participants in the online discussion with an evolutionary perspective will consider OOP to be valuable in the way they actually practice it yet hyped in theory.

6.2.3 Purity

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- OOP Purity is not vital to the OOP paradigm.
- OOP can be effectively developed in hybrid languages.
- The quest for the ubiquity of OO is fanatical because elegant non-OO technologies like the relational paradigm are re-invented in pure OO without sound reasons.
- A single programming paradigm that excludes other ideas is not good for the advancement of software development in general.

- Programming paradigms are not really inherently separate or mutually exclusive since they borrow and build on ideas from each other.

- A program does not need to be OO pure in order to be considered OO; it is the use of concepts such as concepts encapsulation, inheritance and polymorphism that make it OOP.

- OOP behaviours should be built on top of a relational model.

- OOP and the relational paradigm need not be mutually exclusive.

An evolutionary perspective supports the pervasive application of OO to organize process therefore it makes sense that all method code should reside in classes however its support for the separateness of data and process implies its support for the existence of other data types alongside objects. Therefore it supports one dimension of purity yet not the other. The continuation of the traditional separateness of data and process obviates the need for the object concept to be ubiquitous. Therefore OO analysis & design, entrenched in revolutionary OOP, does not derive measurable benefit from this perspective while relational databases will continue to work elegantly with evolutionary OOP. Participants in the online discussion with an evolutionary perspective will largely refute the need for purity and the ubiquity of OO.

6.2.4 Naturalness

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- The simulation or modeling of reality is not fundamental to OOP; this is not the reason why OOP came be in the first place.

- Humans do not find it psychologically easier to translate the conceptual design to a software implementation when using OOP.

- Inheritance does not enable us to naturally model taxonomies found in the real world.

- The naturalness of OOP is the most abused and misleading notion of OOP.
- The OOP concepts of classes and inheritance and OOP are not applied to model the real world rather these concepts are used to modularize and parameterize code in order to facilitate reuse in practice.
- While the vocabulary and the relationships of the problem domain are represented in the OOP software system we do not model reality in OOP.
- Philosophical assumptions of the real world made by naturalness supporters are false since not everything in the real world is an object.
- Collection classes, singleton classes and many of the classes discussed in OOP design patterns are presented as examples of classes that are purely software constructs.

The evolutionary perspective is inherently against any notion that objects model reality better. Process and data are conceptualized separately in evolutionary OOP. Process is seen as acting on data. The naturalness claim from a revolutionary perspective is rejected on this basis. It is more correct to say that the vocabulary and relationships of the problem domain are represented in software objects that are entirely software artifacts and not that the software objects map to reality. Participants in the online discussion with an evolutionary perspective will refute the claim of naturalness from a revolutionary perspective yet support the idea that an OO organization of the process in the imperative program makes software less complex and easier to create.

6.2.5 Software Quality

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- OOP improves maintenance by solving problems like the global variable problem (if the class concept is applied correctly).
- OOP class and inheritance concepts help improve reuse.

- It is easier to create correct code in OOP from the very start (however OOP concepts must be applied correctly).
- OOP codifies the best practices in procedural programming; the best procedural programs are really OO programs.
- The very quest for quality in large software systems has led us to OOP.
- Simply applying OOP concepts does not seem to guarantee software quality since OOP provides the tools but it does not tell us how to use them correctly.
- OOP does not degrade software quality because the concept of inheritance does not necessarily increases coupling.

The evolutionary perspective recognizes the benefits of OOP that follow from the introduction of objects into traditional programming. Reusable objects may be more difficult to create yet they are potentially more powerful than reusable functions since objects are regarded as conceptually more powerful. While recognizing that the imperative paradigm is inherently 'stateful' classes can help group state and functions meaningfully thus helping to solve the 'global variable problem'. Correct code can be created and maintained more easily if the OO organization of process organizes process more effectively. The need to correctly apply OOP concepts is acknowledged since OOP concepts alone does not guarantee quality. Inheritance does not increase coupling if applied correctly yet has the opposite effect it applied incorrectly. Participants in the online discussion with an evolutionary perspective will support the view that OOP from an evolutionary perspective does not guarantee quality but it will improve software quality if applied correctly.

6.2.6 Key Concepts

Participants in the online discussion who share the evolutionary interpretation will support the following salient points in the online discussion:

- Encapsulation is a key defining concept of OOP since the class is an organization of code and data into separate components with localized scope.

- Encapsulation is fundamental to programming language evolution as a whole defining each stage of the evolution of programming.
- OO is in fact just a set of techniques or programming language constructs that allow for the organization of code and data into separate components with localized scope.
- Classes can be too encapsulated rendering them less useful because too much of the implementation is hidden and not directly accessible.
- The class is a key defining concept of OOP.
- Inheritance is a core OOP concept.
- Inheritance enables reuse.
- Inheritance is not used to model real world taxonomies.
- Implementation inheritance is abused by programmers because it seems to offer an easy code reuse mechanism that negatively impacts software quality.
- Polymorphism is dependent on interface inheritance in OOP.
- The use of inheritance to model real world taxonomies leads to inflexible designs.
- Polymorphism a key OOP concept.
- Polymorphism improve software quality because it provides an alternative to putting everything into if statements.
- Polymorphism can improve code structure by providing an alternative to 'if' statements.
- Function call semantics are sufficient for OOP; the messaging semantics are great but they are not necessary for OOP.

- The object is a key OOP concept.

There is some support within evolutionary OOP for the following salient points:

- Too many OOP semantics are associated with the class mechanism making it a confusing concept.
- Interface inheritance is more prominent than implementation inheritance because it creates polymorphic code without the maintenance issues.
- Polymorphism is the least understood of the OOP concepts.
- The confusion between interface inheritance and implementation inheritance must be cleared.
- Composition is better than implementation inheritance because it creates more maintainable code.
- Composition offers better reuse compared to inheritance.
- The diminishing importance of inheritance in OOP is an 'admission of guilt' that inheritance was initially oversold.
- Delegation is the real intent behind inheritance.
- The only real benefit of inheritance is polymorphism.
- Objects require stronger semantics that prescribe best practice.
- Objects rather than classes should be at the centre of OOP since this make OOP easier to teach and understand.

There may be some support within evolutionary OOP for the following salient points:

- The class does not necessarily define OOP since OOP using prototypes does not require OOP.

- The instantiation of objects through classes is better than prototype-based OOP since a prototypical parent object is abstract but masquerades as a concrete object.

- The instantiation of objects through prototype-based OOP is better than class-based OOP because this is closer to how objects are created in the real world.

The evolutionary perspective supports the use of objects, encapsulation, classes, inheritance and polymorphism in structuring and organizing process in software rather than modeling reality. This perspective acknowledges that OOP is a further evolution of imperative programming converging and merging a number of previous encapsulation mechanisms. The evolutionary nature of this perspective suggests that it will be accommodating of prototype-based OOP as an alternative approach to class-based OOP for organizing process within imperative programming. The semantics of traditional function calling are sufficient for a perspective that uses OOP to organize program process rather than model objects in reality. There is no unanimous agreement in evolutionary OOP about the relative value and correct application of OOP concepts. However, the evolutionary nature of this perspective is accommodating of the differences in interpretation of the value of implementation inheritance, interface inheritance, polymorphism, and composition. The ongoing evolution of the key concepts can be accommodated within the evolutionary perspective. Despite their differences in opinion, participants in the online discussion with an evolutionary perspective will be unanimous in their support for the view that the key concepts of OOP evolved within traditional programming helping to create a distinct paradigm distinct within imperative programming.

6.3 The Traditionalist Perspective

The traditionalist perspective of OOP represents the viewpoint of the extreme OOP skeptics who views the entire OOP paradigm with distain and distrust. From this perspective OOP is nothing more than a new name for old ideas that already existed in traditional programming. The traditionalist perspective is formulated mainly in the negative critique of the revolutionary perspective of OOP from its basis in traditional programming. Benefits like reusability are not regarded as a particular and unique

advantage of OO; it is the essential characteristics of abstraction and encapsulation found in modularity that already exists in traditional programming (Ambler, 2001). Inheritance on the other hand may actually discourage reuse (Bieman & Karunanithi, 1995) because it can create strong module coupling that adversely affects maintenance and testing. According to Hatton (1998), the complexity of an OO application is much greater than a procedural application in practice.

The traditionalist perspective refutes the idea that the simulation or modeling of reality as an interaction of objects is natural. In contrast the procedural organization of code is regarded as being more natural. According to Neubauer and Strong (2002), the procedural paradigm is more "natural" than OO for various reasons. Students learn mathematics as procedural processes applied to data. Encapsulating methods into objects implies that those objects are animated in some form yet most objects we encounter in real life are not animated. A recipe is an algorithm that is applied to a set of ingredients. We view our world as containing many inanimate objects that we control and manipulate. In sports like baseball, the ball does not pitch itself neither does the bat swing itself.

The traditionalists regard the traditional separation of data and process to be superior to OO. The concept of the object subsuming process and data is rejected while the evolutionary use of OOP to organize only process in software is perhaps not well understood and therefore largely ignored in the traditionalist consideration of OOP. From the traditionalist perspective objects do not pervasively structure or organize software systems. Objects have localized use in a software organization that is still largely procedural.

6.3.1 The Meaning of OOP

Traditionalists in the online discussion will support the following salient points in the online discussion:

- OOP is nothing new it is just a re-invention or new name for old concepts like namespaces and modules.

- Simply using the concepts of classes, inheritance and polymorphism in a program makes it OOP.

The traditionalists reject the notion of OOP as a distinct programming paradigm. OOP concepts only have localized application within a procedural organization of imperative code. For the extreme OOP detractors in the online discussion OOP is nothing more than new names for old concepts.

6.3.2 Hype

Traditionalists in the online discussion will support the following salient points in the online discussion:

- OOP's popularity and acceptance is driven mainly by hype; claims are accepted without conclusive proof.
- There is no real empirical evidence to support OOP claims; empirical studies in support of OOP thus far are inconclusive because they contradict one another.
- OOP detractors are not mainly uninformed amateurs; the experts do not know better.

For the traditionalists, OOP is mainly conceived in hype. The localized application of OOP concepts in traditional programming does not make a substantial difference from what was before. It is the premise of this study that the nature of OOP and programming in general make real empirical evidence conclusively proving the claims of OOP impossible. Therefore traditionalists will never get the proof that they are looking for but neither will they be able to prove their own counter-claims concerning a procedural organization of code. The general acceptance that the early claims of OO from the revolutionary perspective have not been realized in practice feeds the hype theory.

6.3.3 Purity

Traditionalists in the online discussion will support the following salient points in the online discussion:

- The quest for the ubiquity of OO is fanatical because elegant non-OO technologies like the relational paradigm are re-invented in pure OO without sound reasons.

- A single programming paradigm that excludes other ideas is not good for the advancement of software development in general.

- The multi-paradigm approach is more flexible and adaptable to the problem at hand since they do not prescribe a uniform approach to all problems.

- Programming paradigms are not really inherently separate or mutually exclusive since they borrow and build on ideas from each other.

The concept of purity seems ridiculous from this perspective since OOP concepts have only localized use in traditional programming. OOP concepts and also other programming paradigms are best appropriated for local use within traditional programming.

6.3.4 Naturalness

Traditionalists in the online discussion will support the following salient points in the online discussion:

- Humans do not find it psychologically easier to translate the conceptual design to a software implementation when using OOP.

- Philosophical assumptions of the real world made by naturalness supporters are false since not everything in the real world is an object.

The traditionalists may counter the claims of naturalness from revolutionary OOP by proclaiming the naturalness of procedural programming instead. The separation of data and process and the idea that reality is modeled better from the perspective that process acts on data is fundamental to traditional programming and the traditionalists. The traditionalists ignore the value of evolutionary OOP in organizing only process. The naturalness of OOP is just hype for the traditionalist.

6.3.5 Software Quality

Traditionalists in the online discussion will support the following salient points in the online discussion:

- OOP degrades software quality because the concept of inheritance increases coupling and the reuse that has been claimed has not materialized in practice.
- If takes just as much effort to achieve a steady (correct) OOP system compared to other types of systems since the problem of correctness cannot be solved by OOP because it is fundamentally a mathematical issue.
- OOP cannot lay special claim to improving reuse since what is really required is a reuse strategy, which will also lead to improved reuse in other paradigms, not just OOP.
- OOP does not solve issues like the global variable problem because instance variables are still really global to the class.

The traditionalist perspective is that OOP does not contribute anything to the improvement of software quality. The quality of software will indeed suffer if OOP from a revolutionary perspective fails to conceptualize the problem domain effectively since the issues of software quality arise mainly from the conceptualization of the problem domain (Brooks, 1987). Traditionalist arguments emphasize the abuse of OO concepts. Inheritance can indeed increase coupling if used to borrow code randomly. The global variable issue is not solved if classes are not created as meaningful higher-level abstractions. Reuse is not unique to OOP and it has not materialized in revolutionary OOP. The evolutionary OOP perspective on reuse is ignored. The focus of the traditionalists on the failures of revolutionary OOP, the abuse of OOP concepts and its ignorance of evolutionary OOP leads traditionalists to conclude that OOP is not beneficial and may actually lead to software of poorer quality.

6.3.6 Key Concepts

Traditionalists in the online discussion will support the following salient points in the online discussion:

- Encapsulation is not a concept unique to OOP; the function for example, is also a form of encapsulation.
- Implementation inheritance is abused by programmers because it seems to offer an easy code reuse mechanism that negatively impacts software quality.
- Interface inheritance is not unique to OOP.
- The use of inheritance to model real world taxonomies leads to inflexible designs.
- Polymorphism is not unique to OOP.
- Objects are just another name for older concepts from traditional programming.

The concepts of objects, encapsulation and polymorphism are not regarded as being uniquely OOP from this perspective. The procedural abstractions of namespaces, modules and abstract data types are also encapsulating mechanisms. Polymorphic behaviour can also be done at the function level using function pointers. While inheritance is considered to be a uniquely OOP concept traditionalists view inheritance negatively focusing on the frequent abuse of inheritance as a reuse mechanism and its shortcomings in modeling real world taxonomies. The key OOP concepts are merely old ideas that have only local applicability in traditional programming from this perspective.

6.4 The Meanings 'Beyond' the Text

While the previous part of the discussion sought to appropriate the deeper meanings lying 'beneath' the text by uncovering the underlying interpretations of OOP that give rise to the OOP arguments this discussion section seeks to find the meanings that lie 'beyond' the text. Through the principle of distanciation a critical and explanatory stance is taken. Distanciation from newly appropriated understandings of OOP opens the door to critique

and a further explanation of the OOP discourse in general. A deeper understanding of the OOP discourse is appropriated through the dialectic of appropriation and distanciation.

Hermeneutic philosophy excludes any possibility of finding one correct interpretation of OOP. It challenges the notion of an objective understanding of reality independent of the subject observing that reality. While an objective reality may exist we cannot directly know this reality; we can only have a shared understanding of reality forged in the interaction between subject and object. This excluded the possibility of proving why one interpretation of OOP is more correct than the other by scientific means.

This study reveals three different interpretations of OOP. Each interpretation of OOP is a particular shared understanding of OOP that is not scientifically falsifiable. However, the OOP discourse is open to critique that does not apply atemporal definitions of OOP and casual explanations. Hermeneutic philosophy and the hermeneutic principles of understanding form the basis for critique and explanation.

While such a quest is fundamentally futile both the online discussion and the literature reflect a general desire to continue to strive for a single correct interpretation of OOP so that the confusion around OOP may be cleared. According to Armstrong (2006), a clear understanding of what concepts characterize OO is of paramount importance to both practitioners in the midst of transitioning to the OO approach and researchers studying the transition to OO development. This studies most telling critique of the prevailing OOP discourse is the persistent quest for a single, causal and atemporal definition of OOP. OOP can only be deeply understood it we accept a hermeneutic understanding of OOP that accommodates multiple valid interpretations of OOP. The 3 perspectives of OOP uncovered in this section each reflect different understandings of OOP.

Revolutionary OOP is the interpretation of OOP that exists in the literature and 'out there' on the street. It is based on the belief that an object reality is true for all people and applicable to most problem domains. Therefore software can easily and effectively simulate or model the problem domain and this object reality. While the benefits of naturalness and improved software quality are plausible while revolutionary OOP's fundamental belief hold it dissolves into hype if the problem domain cannot be

conceptualized effectively using OO. A reasoned quest for purity will then seem fanatical. Even if we accept that the 'object reality' is universal it does not mean that the problem domain and reality are equivalent. The counter-claims against revolutionary OOP in the OOP discourse and the existence of a different interpretation of OOP in practice suggests that revolutionary OOP does indeed fail for many problem domains.

The traditionalist perspective of OOP is constituted in the vociferous counter-reaction to revolutionary OOP and the failure of the revolutionary perspective to fulfill its early promises; it clings to traditional programming also ignoring the possibilities presented by the evolutionary perspective. It supports the view that the problem domain and software is best understood as process acting on data each separately organized in a procedural organization of code. OOP concepts have only localized use within traditional programming from this perspective. However, the many persistent problems in software development that have not been solved by traditional programming (Brook, 1997) indicate that the traditionalist perspective is not the final answer to the software crisis.

The evolutionary perspective of OOP is formulated in the evolving practice of traditional programming. It applies OOP to the object organization of process in programming while still maintaining the traditional separateness of process and data. The support for this perspective in the online discussions, the slow uptake of OO analysis and design and the continued dominance of relational databases in OOP suggests that this may be the prevailing tacit interpretation of OOP in practice perhaps accounting for the current popularity of OOP in mainstream programming languages, which are still largely imperative. While reflecting a willingness to integrate ideas from both revolutionary OOP and traditional programming it rejects the view that objects model reality better. It also departs from the functionally decomposed organization of process found in traditional structured design. These indicate definitive approaches to conceptualizing the problem domain and the organization of process in programs. While evolutionary OOP may have wide applicability it may still not be suitable for all people in very problem domain.

Each perspective of OOP makes definite decisions about the nature of reality and the organization of process and data. The arguments made in the OOP discourse arise from

the unwillingness to acknowledge that there are multiple perspectives of OOP and the lack of understandings of the different perspectives. The literature and OOP experts focus largely on developing the revolutionary perspective of OOP, dismissive of critique and largely ignoring or unaware of differences in the practice of OOP. The traditionalist perspective of OOP focuses on critique of the early failures of revolutionary OOP ignoring evolutionary OOP. The nature of OOP and software development in general precludes the possibility of conclusive 'proof' of OOP that the traditionalists seek but neither would the traditionalists be able to prove their own counter-claims concerning procedural programming. The evolutionary perspective of OOP has not developed into a coherent school of thought that makes it clearly distinguishable from revolutionary OOP and indeed traditional programming. Therefore it has remained obscure and vague, only finding its voice in the debate between revolutionary OOP and the traditionalist perspective without a willingness to assert its identity. A hermeneutic understanding of OOP allows us to clarify the evolutionary perspective of OOP obscured in the OOP discourse yet representative of the tacit meanings guiding the prevailing practice of OOP.

A hermeneutic understanding of the multiple interpretations of OOP opens the door for more effect application of OOP in practice. It allows us to appropriate the many meanings of OOP, to neither blindly adopt OOP theory nor cling to tradition in the act of programming but to deeply understanding their meanings and the tacit understandings that arise in the practice of OOP. There are informed choices to be made. The 3 interpretations of OOP are neither universally true nor untrue. Each interpretation may have advantages for particular people in particular problem domains. It requires us to be more critical of the particular OOP approach we take to solve the problem at hand. Some of the failures of OOP in practice may be due to the application of the incorrect interpretation of OOP or the unsuitability of the interpretation of OOP to the people. A conceptualization of the problem at odds with our shared understandings of the problem domain will make the software construction feel unnatural and impact the quality of the software negatively.

The future evolution of OOP will also benefit from a hermeneutic understanding of OOP. The three interpretations of OOP revealed by this study do not exclude the possibility of

new interpretations of OOP developing in the future. OOP and software development in general will only prosper if our general understanding of programming is accommodation of multiple interpretations. OOP concepts have been appropriated in different way taking on different meanings through the course of time. New perspectives of OOP will not displace existing ideas; they will be able to coexist in a broader hermeneutic understanding of OOP. A hermeneutic understanding of OOP will accommodate the different meanings that OOP may take on in the future allowing it to grow and prosper without confusion.

7 Conclusion

The online discussions confirm that OOP is indeed a subject matter of intense debate. It reflects not just an exchange of exalted claims and vicious counter-claims between OOP evangelists and rabid OOP detractors each with their own self-serving interests but a genuine desire to make sense of the confusion surrounding OOP. The online discussion reveals a commitment and willingness to hold meaningful dialogue despite the seemingly treacherous nature of the topic of OOP and software development in general. The OOP discussions taking place in three online software development communities 'Lambda the Ultimate', 'Artima' and 'Ward and Cunningham' provides a rich source of text because these online discussions do not represent expert opinion only; they capture the meanings of OOP as informed or uninformed as these understandings may be. This serves the study well for the premise of this study is that OOP and programming concepts in general are shared understandings that exist out there in the broad software development community rather than only expert opinion. We can only develop a deep understanding of OOP if we seek to understand the meanings of OOP in the software development community in general.

The following limitations of this study should be noted. While the Internet provides a rich and broad sample of the OOP discussion taking place among software developers the online text should be not taken to completely represent every aspect of the OOP debate. Hitherto unknown OOP issues raised in face-to-face workshops, seminars and other media may not have made their way onto the Internet as yet. For practical considerations only 3 online communities where chosen for in-depth analysis and study because of the prevalence of the type of text relevant to this study. However this may have excluded relevant discussions taking place in online developer communities outside the 3 chosen discussion forums.

Through hermeneutic ontology, epistemology and principles of understanding we uncovered the meanings 'in' the text, the meanings 'beneath' the text and the meanings 'beyond' the text. Hermeneutics allows us to uncover the obscure, confused and hidden meaning of OOP in the online discussion and the broader OOP discourse. The ontological and epistemological basis of this study premises this study in the belief we exist in a

shared understanding of reality. It is not possible to know reality directly therefore there can be no objective knowledge of reality only understandings of reality that we share with one another. The subject matter of OOP is a shared understanding that we came to deeply 'understand' and 'explain' through the hermeneutic principles of the circular nature of understanding, the principle of prejudgement, the principle of appropriation and distanciation and the principle of multiple interpretations.

This study revealed and acknowledged its initial preunderstandings of OOP appropriated from the literature. The analysis of the online discussion then appropriated the meanings 'in' the online discussion. The principle of circular understanding is an over-arching principle for the other hermeneutic principles however in its most explicit application broad themes of discussion emerged from the individual online arguments. The themes of 'The meaning of OOP', 'Hype', 'Purity', 'Naturalness', 'Software quality' and 'Key Concepts', and the salient points made under each theme allowed a deeper understanding to develop. The discussion revealed the meanings 'beneath' the arguments made in the online discussion as the preunderstandings of OOP revealed in the literature encountered the meanings appropriated from the online OOP discussions. A fusion of meanings in the text with the meanings in the literature resulted in a deeper understanding of the broader OOP discourse. Three different underlying interpretations of OOP emerged. The revolutionary perspective, the evolutionary perspective and the traditionalist perspective form the underlying basis for the various claims and counter-claims made in the online discussion and the OOP discourse in general. Through the principle of distanciation a critical and explanatory stance sought the meanings 'beyond' the text and the prevailing OOP discourse. The general unwillingness of the prevailing OOP discourse to accommodate and acknowledge multiple interpretations of OOP is at the heart of this critique. The conclusion is that a deep understanding of OOP requires a hermeneutic understanding of OOP.

This study reveals the hermeneutic nature of the OOP phenomena both in its understanding and explanation of the OOP discourse, and its critique of the prevailing OOP discourse. The challenge for the software development community is to overthrow its culture of scientism that OOP and the subject matter of software development in

general is mired in. The pursuit of casual and atemporal definitions of OOP and other software concepts are futile however determined they may be. We must reexamine our philosophical underpinnings of OOP, programming paradigms and software development in general. It is only in the acknowledgement of the deeply hermeneutic nature of software programming that a deeper understanding of OOP can be forged. It is only in the realization that OOP and other software concepts are shared understandings that we will properly apply OOP and software development concepts to maximum effect.

The particular research approach followed in this study of OOP can also be applied to other areas of software development. The subject matter of software development in general requires a hermeneutic understanding. Hermeneutics recognizes the historicity of meanings of software development because our understanding of software development is inherently temporal and coloured by our prejudgements. Hermeneutics is accommodating of multiple interpretations that develop as concepts are critically appropriated for different uses over time, far beyond their original definitions. Software development in general will greatly benefit from hermeneutic studies that promote multiple interpretations weaving the 'text' of theory and the 'text' of practice into a cohesive, meaningful whole.

References

Ambler, S. (2001). *A realistic look at object-oriented reuse*. Retrieved March 1, 2007 from http://www.ddj.com/dept/architect/184415594?cid=Ambysoft

Armstrong, D.J. (2006). The quarks of object-oriented development. *Communications of the ACM*, 49(2), 123-128.

Arnold, S.J. & Fischer, E. (1994). Hermeneutics and consumer research. *The Journal of Consumer Research*, 21(1), 55-70.

Berntsen, K.E, Sampson J. & Osterlie, T. (2004). *Interpretive research methods in computer science*. Retrieved December 1, 2008 from Norwegian University of Science and Technology, Dept. of Computer and Information Science Website http://www.idi.ntnu.no/~thomasos/paper/interpretive.pdf

Bezroukov, N. (2007). A *skeptical view on the object-oriented programming*. Retrieved June 19, 2007 from http://www.softpanorama.org/SE/anti_oo.shtml#Problem-cause

Bieman, J.M. & Karunanithi, S. (1995). Measurement of language-supported reuse in object-oriented and object-based software. *Journal of Systems Software*, 30, 271-293.

Bloor, B. (2004). *The failure of relational database, the rise of object technology and the need for the hybrid database*. Retrieved December 1, 2008 from http://www.intersystems.com/cache/whitepapers/pdf/baroudi_bloor.pdf

Booch, G. (1994). *Object-oriented analysis and design with applications* (2nd ed.). Redwood City, CA: The Benjamin/Cummings Publishing.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The unified modeling language user guide*. Reading, MA: Addison-Wesley.

Brooks, F.P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4), 10-19.

Capretz, L.F. (2003). A brief history of the object-oriented approach. *Software Engineering Notes*, 28(2), 1-10.

Cox, B.J. (1987). *Object-oriented programming: An evolutionary approach*. Reading, CA: Addison-Wesley.

Demeterio, F.P.A. (2001). Introduction to hermeneutics. *Diwatao*, 1(1). Retrieved December 1, 2008 from http://www.geocities.com/philodept/diwatao/introduction_to_hermeneutics.htm

Deng, P. & Fuhr, C.L. (1995). Using an object-oriented approach to the development of a relational database application system. *Information & Management*, 29, 107-121.

Elmasri, R. & Navathe, S.B. (1989). *Fundamental of database systems*. Redwood City, MA: The Benjamin/ Cummings Publishing.

Fowler, M. (2005). *The new methodology*. Retrieved June 20, 2007 from http://www.martinfowler.com/articles/newMethodology.html

Fraser, S., Beck, K., Booch, G., Constantine L., Henderson-Sellers, B., McConnell S., Wirfs-Brock, R. & Yourdon, E. (2005). Echoes? Structured design and modern software practices, *OOPSLA '05, San Diego, California,* pp. 383-386.

Gabriel, R.P. (2002). Objects have failed, opening remarks. *OOPSLA 2002, Seattle, Washington*. Retrieved March 1, 2007 from http://www.dreamsongs.com/ObjectsHaveFailedNarrative.html

Gosling, J., Joy, B., Steele, G. & Bracha, G (1996). *The Java language specification*. Boston, MA: Addison-Wesley (3rd ed.).

Graham, P. (2003). *The hundred year language*. Retrieved March 1st, 2007 from http://www.paulgraham.com/hundred.html

Grossman, M., Aronson, J.E. & McCarthy, R.V. (2004). Does UML make the grade? Insignts from the software development community, *Information and Software Technology*, 47, 383-397.

Hatton, L. (1998), Does OO sync with how we think?' *IEEE Software*, 15(3), 46-54.

Jacobs, B. (2006). *Object oriented programming oversold!* Retrieved March 1, 2007 from http://www.geocities.com/tablizer/oopbad.htm

Jacobson, I., Christerson, M., Jonsson, P. & Overgaard, G. (1992). *Object-oriented software engineering*. Wokingham: Addison-Wesley.

Johnson, R.A., Hardgrave, B.C. & Doke, E.R. (1999). An industry analysis of the developer beliefs about object-oriented systems development. *The Database for Advances in Information Systems*, 30(1), 47-63.

Jorgensen, P., Fernandez, D., Fischer, A., Greco, M., Hussey, B., Kuchta, S., et al. (2002). *Has the object-oriented paradigm kept its promise?* Allendale, MI: Grand Valley State University, Dept. of Computer Sc. and IS. Retrieved March 1, 2007 from http://www.informatikdidaktik.de/HyFISCH/Informieren/Programmiersprachen/OOPromisesAndReality.pdf

Klein, H. K. & Myers, M. D. (1999). A set of principles for conducting and evaluating interpretive field studies in Information Systems, *MIS Quarterly,* 23(1), 67-94.

Lewis, J. (2000). Myths about object-orientation and its pedagogy, *SIGCSE '00 Proceedings,* pp. 245-249.

Lieberman, H. (1986). Using prototypical objects to implement shared behaviour in object-oriented systems, *OOPSLA '86 Proceedings*, pp. 214-223.

Linge, D. E. (Ed.) (1976). *Philosophical hermeneutics*. Berkeley: University of California Press.

Madsen, O. L. (1996). Strategic research direction in object-oriented programming, *ACM Computing Surveys,* 28(4).

Meyer, B. (1988). *Object-oriented software construction*, Hertfordshire: Prentice-Hall.

Microsoft Developer Network Library. (n.d.). Retrieved June 1, 2008 from http://msdn.microsoft.com/en-us/library/default.aspx

Mitchell, J. C. (2002). *Concepts in programming languages*, Cambridge: Cambridge University Press.

Myers, M. D. (1997). Qualitative research in information systems, *MIS Quarterly*, 21(2), 241-242.

Neubauer, B.J. & Strong, D.D. (2002). The object-oriented paradigm: More natural or less familiar? *Journal of Computing Sciences in Colleges*, 18(1), 280-289.

Normark, K. (n.d.). *Programming paradigms*. Retrieved March 1, 2007 from Aalborg University, Denmark, Dept. of Comp. Sc. Website http://www.cs.auc.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html

Ormiston, G.L. & Schrift, A. D. (1990). *The hermeneutic tradition: From Ast to Ricoeur*. Albany: State University of New York Press.

Palmer, R. E. (1969). *Hermeneutics*. Evanston, IL: Northwestern University Press.

Pawson, R. (2004). *Naked Objects* (Doctoral dissertation, University of Dublin, Trinity College, 2004). Retrieved March 1, 2007 from http://www.nakedobjects.org/downloads/Pawson thesis.pdf

Post, G. & Kagan, A. (2000). OO-CASE tools: An evaluation of Rose. *Information and Software Technology*, 42, 383-388.

Power, M. J., Cheney, P. H. & Crow, G. (1990). *Structured Systems Development* (2nd ed.). Boston, MA: Boyd & Fraser Publishing.

Quatrani, T. (1998). *Visual modeling with Rational Rose and UML.* Reading, MA: Addison Wesley.

Raccoon, L.B.S. (1997). Fifty tears of progress in software engineering. *Software Engineering Notes*, 22(1), 88-104.

Ricoeur, P. (1981). *Hermeneutics and the human sciences.* Thompson, J.B. (Ed. & Trans.), Cambridge: Cambridge University Press.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice-Hall.

Sebesta, R.W. (1996). *Concepts of programming languages* (3rd ed.). Menlo Park, CA: Addison-Wesley.

Schach, S.R. (1996). *Classical and object-oriented software engineering* (3rd ed.). Boston, MA: McGraw-Hill.

Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kimura, I., London, R.L., Jonathan, R., et al. (1978). Validating the utility of abstraction techniques. *ACM Annual Conference Proceedings*, pp. 106-110.

Sklenar, J. (1997), *Introduction to OOP in Simula.* Retrieved December 15, 2007 from http://staff.um.edu.mt/jskl1/talk.html

Sircar, S., Nerur, S.P. & Mahapatra, R. (2001). Revolution or evolution? A comparison of object-oriented and structured systems development methods. *MIS Quarterly*, 25(4), 457-471.

Smalltalk dot org. (n.d.). Retrieved December 15, 2007 from http://www.smalltalk.org/main/

Stroustrup, B. (2007), Evolving a language in and for the real world: C++ 1991-2006. *HOPL III, June 2007.*

Taylor, D.A. (1998). The keys to object technology. In Zamir, S. (Ed.), *Handbook of Object Technology* (pp. 1-12), Florida: CRC Press.

Walsham, G. (2006). Doing interpretive research. *European Journal of Information Systems*, 15, 320-330.

West, D. (1997). Hermeneutic Computer Science. *Communications of the ACM*, 40(4), 115-116.

8 Appendix

8.1 Online Discussion Threads

8.1.1 LTU (Lambda the Ultimate)

1. *A Good OOP Critique?* http://lambda-the-ultimate.org/classic/message4857.html
2. *A software engineering problem: how would functional programming solve it?* http://lambda-the-ultimate.org/node/865
3. *A survey of object-oriented concepts* http://lambda-the-ultimate.org/classic/message9895.html
4. *An Interview with A Stepanov* http://lambda-the-ultimate.org/classic/message185.html
5. *Animism An Essential Concept in Programming* http://lambda-the-ultimate.org/classic/message3659.html
6. *Are Frames and Slots anything more that OO with a different name?* http://lambda-the-ultimate.org/node/1962
7. *Classification according to type vs function - An anecdote* http://lambda-the-ultimate.org/node/1106
8. *Concepts Techniques and Models of Computer Programming* http://lambda-the-ultimate.org/classic/message7355.html
9. *DP-COOL 2003 Proceedings* http://lambda-the-ultimate.org/classic/message8641.html
10. *Growing a language* http://lambda-the-ultimate.org/classic/message12028.html
11. *History of Logic Programming: What went wrong* http://lambda-the-ultimate.org/node/2803
12. *HOPL III: Evolving a language in and for the real world: C++ 1991-2006* http://lambda-the-ultimate.org/node/2283
13. *Implementation Inheritance* http://lambda-the-ultimate.org/node/2103
14. *Instantiation of classes in wrong place leads to wrong structures* http://lambda-the-ultimate.org/node/2359
15. *Is Inheritance a Pillar of OO?*

http://lambda-the-ultimate.org/classic/message6069.html

16. *Is "post OO" just over?*
http://lambda-the-ultimate.org/node/1741

17. *Joel Spolsky views on CS education*
http://lambda-the-ultimate.org/node/1204

18. *MultiMethods*
http://lambda-the-ultimate.org/classic/message7074.html

19. *Multi-Paradigm Design and Generic Programming*
http://lambda-the-ultimate.org/classic/message4953.html

20. *Objective scientific proof of OOP's validity? Don't need no stinkun' proof.*
http://lambda-the-ultimate.org/node/893

21. *OOP Is Much Better in Theory Than in Practice*
http://lambda-the-ultimate.org/node/489

22. *Paul Graham - Revenge of the Nerds*
http://lambda-the-ultimate.org/classic/message3404.html

23. *"Popular vs Good" in Programming Languages*
http://lambda-the-ultimate.org/node/497

24. *Return of the Global Variables?*
http://lambda-the-ultimate.org/node/1206

25. *Ruby vs Python*
http://lambda-the-ultimate.org/node/1480

26. *Singleton classes really that bad?*
http://lambda-the-ultimate.org/node/1219

27. *The Case for First Class Messages*
http://lambda-the-ultimate.org/classic/message12290.html

28. *The Next Move in Programming (Livschitz interview)*
http://lambda-the-ultimate.org/classic/message11172.html

29. *What do you believe about programming languages (that you can't prove (yet))?*
http://lambda-the-ultimate.org/node/1439

30. *Why do they program in C++*
http://lambda-the-ultimate.org/node/663

8.2 ART (Artima)

1. *ThingsGrady Booch has learned about Complex software* http://www.artima.com/forums/flat.jsp?forum=270&thread=231785
2. *Are Programmers People? And If So, What to Do About It?* http://www.artima.com/forums/flat.jsp?forum=106&thread=4414&start=0&msRange=15
3. *Back at OOPSLA* http://www.artima.com/forums/flat.jsp?forum=106&thread=182002&start=0&msRange=15
4. *Coupling is not necessarily a bad thing* http://www.artima.com/forums/flat.jsp?forum=106&thread=107338&start=0&msRange=15
5. *Designing with Interfaces* http://www.artima.com/designtechniques/interfacesP.html
6. *Encapsulation Violation* http://www.artima.com/forums/flat.jsp?forum=106&thread=132358
7. *For Now, Virtual Methods are a better default* http://www.artima.com/forums/flat.jsp?forum=106&thread=14374
8. *Gavin King: In Defence of the RDBMS* http://www.artima.com/forums/flat.jsp?forum=276&thread=206671
9. *Generics* http://www.artima.com/forums/flat.jsp?forum=106&thread=117200
10. *Going all in* http://www.artima.com/forums/flat.jsp?forum=106&thread=108879
11. *How Many Hello Worlds are left* http://www.artima.com/forums/flat.jsp?forum=106&thread=14196
12. *I want know the Main Disadvantages using Inheritance* http://www.artima.com/forums/flat.jsp?forum=1&thread=112752
13. *In Defense of Pattern Matching* http://www.artima.com/forums/flat.jsp?forum=106&thread=166742
14. *Modelling the real world* http://www.artima.com/forums/flat.jsp?forum=106&thread=37870
15. *Modern C++ Style* http://www.artima.com/forums/flat.jsp?forum=226&thread=22582

16. *More on languages and objects*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=105346
17. *Namespacey Programming*
    http://www.artima.com/forums/flat.jsp?forum=270&thread=216587
18. *Object Design*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=115101&start=0&msRange=15
19. *Oh No! DTO!*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=36312
20. *On the Tension Between Object-Oriented and Generic Programming in C++*
    http://www.artima.com/forums/flat.jsp?forum=226&thread=216972
21. *Signs of the Next Paradigm Shift*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=167119
22. *Software Reusability: Myth or Reality?*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=187331
23. *Teaching OO: Putting the Object back into OOD*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=6771
24. *The Harry Potter Theory of programming Language Design*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=123234
25. *The Myth of Paradigms and TMTOWTDI*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=175991
26. *The Problem with Programming*
    http://www.artima.com/forums/flat.jsp?forum=269&thread=186478&start=165&msRange=15
27. *Think of Objects as Machines*
    http://www.artima.com/forums/flat.jsp?forum=32&thread=4771
28. *Thinking about Objects*
    http://www.artima.com/forums/flat.jsp?forum=106&thread=85308&start=0&msRange=15

8.3 W&C (Ward and Cunningham)

1. *Acceptable Criticism of OO On wiki*
   http://c2.com/cgi/wiki?AcceptableCriticismOfOoOnWiki
2. *Alan Kays Definition of Object Oriented*

http://c2.com/cgi/wiki?AlanKaysDefinitionOfObjectOriented
3. *Arguments against OOP* http://c2.com/cgi/wiki?ArgumentsAgainstOop
4. *Benefits of OO* http://c2.com/cgi/wiki?BenefitsOfOo
5. *Benefits of OO Original Discussion* http://c2.com/cgi/wiki?BenefitsOfOoOriginalDiscussion
6. *Coupling and Cohesion* http://c2.com/cgi/wiki?CouplingAndCohesion
7. *Definitions For OO* http://c2.com/cgi/wiki?DefinitionsForOo
8. *Delegation Is Inheritance* http://c2.com/cgi/wiki?DelegationIsInheritance
9. *Encapsulation Definition* http://c2.com/cgi/wiki?EncapsulationDefinition
10. *Internal Polymorphism* http://c2.com/cgi/wiki?InternalPolymorphism
11. *Is CeePlusPlus Object Oriented* http://c2.com/cgi/wiki?IsCeePlusPlusObjectOriented
12. *Is Java Object Oriented* http://c2.com/cgi/wiki?IsJavaObjectOriented
13. *Limits of Hierarchies* http://c2.com/cgi/wiki?LimitsOfHierarchies
14. *Mixing Paradigms* http://c2.com/cgi/wiki?MixingParadigms
15. *Nobody Agrees On What OO Is* http://c2.com/cgi/wiki?NobodyAgreesOnWhatOoIs
16. *NonPolymorphic Inheritance* http://c2.com/cgi/wiki?NonPolymorphicInheritance
17. *Nygaard Classification* http://c2.com/cgi/wiki?NygaardClassification
18. *Object Oriented Design is Difficult* http://c2.com/cgi/wiki?ObjectOrientedDesignIsDifficult
19. *Object Oriented Programming*

http://c2.com/cgi/wiki?ObjectOrientedProgramming

20. *Object Relational Psychological Mismatch*
http://c2.com/cgi/wiki?ObjectRelationalPsychologicalMismatch

21. *Object vs Model*
http://c2.com/cgi/wiki?ObjectVsModel

22. *Object Oriented*
http://c2.com/cgi/wiki?ObjectOriented

23. *OO lacks Consistent Discussion*
http://c2.com/cgi/wiki?OoLacksConsistencyDiscussion

24. *OO Lacks Math Arguments*
http://c2.com/cgi/wiki?OoLacksMathArgument

25. *Polymorphism And Inheritance*
http://c2.com/cgi/wiki?PolymorphismAndInheritance

26. *Polymorphism Encapsulation Inheritance*
http://c2.com/cgi/wiki?PolymorphismEncapsulationInheritance

27. *Principles of Object Oriented Design*
http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

28. *Programming is in the mind*
http://c2.com/cgi/wiki?ProgrammingIsInTheMind

29. *Structured Programming*
http://c2.com/cgi/wiki?StructuredProgramming

30. *The Structure of Scientific Revolutions*
http://c2.com/cgi/wiki?TheStructureOfScientificRevolutions

31. *What is an object*
http://c2.com/cgi/wiki?TheStructureOfScientificRevolutions

32. *What Is Delegation*
http://c2.com/cgi/wiki?WhatIsDelegation

33. *When to Use What Paradigm*
http://c2.com/cgi/wiki?WhenToUseWhatParadigm