Object Thinking

(*For* Extreme Programmers and Agile Developers)

Dr. David M. West

2003

Preface

Extreme Programming (XP) and the various Agile Methods (Agile) focus on the practice of software development. XP and Agile are most often discussed as if they were methods but it would be more appropriate to speak of them in terms of attitude, behaviors and culture.[1] To the extent that XP and Agile are methods, ***they are methods for producing better developers*** – in direct contrast to our normal understanding of method as ways for developers to make better products.

What makes a better developer? Why do we need better developers?

The Need for Better Developers

I'll answer the second question first. Traditional methods (focused on helping developers make better products) and traditional process improvement programs (CMM, RUP) implicitly promise superior outcomes using ‘average’ human resources. For forty years the software industry has attempted to apply, at least implicitly and often explicitly, the principles of Taylorism (“scientific management”) to software development. For the most part, these efforts have failed miserably.

There are instances of success of course. Project ‘A’ using method ‘X’ did, in fact, achieve notable success, but - industry statistics as a whole have failed to improve much since 1968 when ‘software engineering’ and ‘scientific management’ were introduced as means for resolving the ‘software crisis.’ Abandoned projects, cost/time overruns, and bloated buggy software still dominate the landscape.

Even the most ardent advocates (e.g. Dykstra and Parnas) of software engineering as well as those (Yourdon and Martin) most responsible for popularizing software engineering among the corporate masses recognized the limits of formal approaches. Parnas wrote, “A Rational Design Process: How and Why to Fake It,”[2] that acknowledged the fact that highly skilled developers did not “do” development the way that so-called rational methods suggested they should. Martin suggested that the best way to obtain high quality software, that met real business needs, on-time and within budget was the use of special “SWAT” teams comprised of highly skilled, and greatly rewarded, developers doing their work with minimal managerial intervention.

The only consistently reliable approach to software development is, “good people.” So why has so much attention and effort been devoted to process and method? There are at least three contributing reasons:

- A widespread belief, partially justified, that there were not enough good people available to accomplish the amount of development that needed to be done.

- An unspoken, but just as widely held, belief that really good developers were not to be trusted – they could not be "managed" they all seemed to be "flaky" to some degree, and they did not exhibit the loyalty to company and paycheck of "normal" employees. Really "good" developers tended to be "artists" and art was (is) not a good word in the context of software development.
- A suspicion, probably justified, that we did not (do not) know how to "upgrade" average developers to superior developers – except by giving them lots of experience and hoping for the best.

It is no longer possible to believe that "method" and "process" are adequate substitutes for "good people." There is a resurgence of interest – spurred mostly by XP and the core practices in other Agile methods – in how to improve the human developer. This interest takes many forms: ranging from XP itself; to redefinition of software development as a craft (McBreen[3]) and a "calling" (West[4], Denning[5]) instead of a "career;" to software apprenticeship (Auer[6]).

Why do we need better developers? Because increasing the supply of highly skilled people – not methods, not process - is the only way to resolve the "software crisis."

Better Developers

What makes a better developer? The traditional answer, experience. Most textbooks on software engineering and formal process contain a caveat to the effect that extensive real-world experience is required before the tools and techniques provided by the method/process can be effectively utilized. In this context, experience is just a code word for those aspects of development – philosophy, attitude, practices, mistakes, and even emotions – that cannot be reduced to syntactic representation and cookbook formulation in a textbook.

Today's practitioners, and some theorists, are intensely interested in teasing apart and understanding the complex elements behind the label "experience." There is intense interest in understanding what developers actually "do" as opposed to some *a priori* theory of what is appropriate for them to do. Models are seen as communication devices – unique to a community of human communicators – instead of 'vessels of objective truth' with a value that transcends the context of their immediate use. Development is seen as a human activity – hence a fallible activity – that must accommodate and ameliorate imperfection and mistakes rather than seek to eliminate them. Communalism is valued over individualism. Systems are seen as complex instead of just complicated, necessitating a different approach and different insights than were required when software developers merely produced "artifacts that met specification."

All of this intense interest is producing results. Although the total picture is still emerging, some facets of 'the better developer' are coming into focus.

XP provides a foundation by identifying a set of discrete practices that can be 'practiced.' They are things that any developer can actually do, and by simply doing them, become a better practitioner. They offer no grand theory, nor are they derived from any theory. The justification for the XP approach is based on two simple empirical observations: "We have seen master developers do these things;" and, "We have seen less proficient developers do these things and become better." Although XP does make claims to improve both product and process, these are side effects (one is tempted to say mere side effects) of the improvement in the human developers.

Although XP (and the other Agile methods) provide a solid foundation, it is just the foundation. What "Xgilistas"[7] do is but part of what makes them master developers. What they think, and how they think is critically important as well. There is a value system behind XP and other Agile Methods, one that is occasionally made overt but more often is implicit. There is history. There is context. There is oral tradition that has never been reduced to ink and paper (and maybe cannot be so reduced). Aspiring Xgilistas must become conversant with all of this before they can attain true "master" status.

The intent of this book is to provide one small contribution to help those following the Xgilista Path. Specifically, a contribution in the area of "object thinking."

Object Thinking

Thirty plus years have passed since Alan Kay coined the term "object-oriented." Almost all contemporary software developers describe their work using object vocabulary and use languages and specification tools that lay claim to the object label. The ubiquity of object terminology does not mean, however that everyone has mastered object thinking. Nor does the popularity of Java. Nor does the *de facto* standardization of object modeling embodied in UML. A prediction made by T. Rentsch (cited by Booch in 1991[8]) remains an accurate description of today's development and developers:

> *"My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be favor of it. Every manufacturer will promote his products*

as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

- Booch 1991

In fact, the situation may be worse than Rentsch predicted. An argument can be made that the contemporary mainstream understanding of objects is but a pale shadow of the original idea. Further, it might be argued that the mainstream understanding of objects is, in practice, antithetical to the original intent. Alan Kay certainly seems to think this is true.[9]

Clearly the "behavioral approach" to understanding objects has almost disappeared.[10] This fact is important for our immediate purposes primarily because two of the leading advocates behind XP were also the leading advocates of "behavioral objects." Kent Beck and Ward Cunningham invented the CRC card approach to finding and defining objects – the most popular of the "behavioral methods." Others deeply involved in the Agile Alliance were also identified with "object behavioralism" and with Smalltalk – the programming language the came closest to embodying behavioral objects.

It is not unreasonable to assume that the behavioral approach to understanding objects dominates the "object thinking" of many of the best XP practitioners, in part because it was almost necessarily part of the oral tradition passed on by Kent Beck as he initiated others into the XP culture.

It is also reasonable to assume that behavioral object thinking is only implicit in the XP / Agile culture because so few books or texts were ever written in support of this approach. Neither Kent nor Ward ever wrote such a book. Rebecca Wirfs-Brock and her co-authors have yet to update their 1991 offering on CRC cards and Nancy Wilkerson's CRC book appears to be the last effort in this area.

This is particularly unfortunate because the CRC card "method" as described in the early 1990s did not incorporate all aspects of object thinking. In fact, I believe, that "object thinking" transcends the notion of "method" just as it transcended programming languages. (You can do good object programming in almost any language even though some languages offer you more support than others and actually reinforce object thinking.)

Object thinking requires more than an understanding of CRC cards as presented circa 1990. It also requires understanding some of the history and some of the philosophical presuppositions behind object behavioralism, CRC cards, and languages like Smalltalk. It also requires an understanding of the metaphors that assist in good object thinking and extension of the CRC card metaphor, in particular, to include more than object identification and responsibility assignment.

It is my hope that this book will promote such an understanding by capturing at least part of the oral tradition of behavioral objects and making it explicit.

A Different (and Possibly Controversial) Kind of Software Book

This book will be deliberately different from almost any other object / component / method / XP book you may have encountered. It is also likely to be controversial which is not intended but is, perhaps, inevitable.

Several factors contribute to the differences and potential controversy.

- The reader will be asked to read and digest a lot of history and philosophy before embarking on the more pragmatic aspects of object thinking.
- The author will present an unabashed, adamant, and consistent advocacy of “behavior” as the key concept for discovering, describing, and designing objects and components.
- The centrality of CRC (Class-Responsibility-Collaborator) cards as an object thinking device or tool will likely be viewed as anachronistic and irrelevant, since UML (Unified Modeling Language) has achieved the status of *de facto* standard.
- The apparent indifference, or even antagonism, towards formalism and formal approaches to software specification that is intrinsic to the behavioral approach will concern some readers, especially those trained in computer science departments at research universities.
- The emphasis on analysis and conceptualization – thinking – instead of implementation detail might strike some readers as too academic or ethereal.

It will take the rest of the book to explain why these differences are considered important but the motivation behind them is found in another quote from Grady Booch:

> *"Let there be no doubt that object-oriented design is fundamentally different than traditional structured design approaches: it requires different ways of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture."*
>
> *- Booch 1991*

"Different ways of thinking" is the key phrase in the Booch quote and the word "culture" is equally important. The history of objects in software development is characterized by the mistaken notion that the object difference was to be found in a computer language or a software module specification. But objects are fundamentally different because they reflect different ideas – particularly about decomposition – and because they reflect a different set of values and worldview (i.e. a culture) than traditional software development. Understanding objects requires understanding the philosophical roots and historical milestones from which objects emerged.

The first chapter of this book summarizes that context. Two philosophical traditions, one (Formalism) supporting mainstream development practices and the other (Hermeneutics-Postmodernism) supporting alternatives like objects, patterns, and XP are briefly introduced. A highly focused examination of a few programming languages – and their history – shows how philosophy becomes tangible. And finally, these roots are tied together in a discussion of the "object culture" as a kind of ground or foundation for understanding object thinking.

Chapter two explores the role of metaphor and vocabulary in shaping object thinking. Metaphors are essential for bridging between the familiar and the unfamiliar and selection of the "right" metaphors is critical for further development of fundamental ideas. Vocabulary provides us with one of the essential tools for thinking and communicating. It is important to understand why object advocates elected to use a different vocabulary for object thinking and why it is inappropriate to project old vocabulary onto new concepts.

Chapter three presents a brief discussion of methods and models, both of which are presented as lacking in any intrinsic value. It is suggested that methods are little more than helpful reminders of "things to think about." Models, it is suggested, are nothing more than graphical vocabulary, a kind of external short term memory, and useful only in the context of a particular group of people at a particular point in time. These biases are linked to XP "method" in specific and Agile method in general. The "Object Cube" is introduced as a specific model for thinking about objects and the utility of some traditional models, like sequence diagrams, for supporting object thinking are discussed.

Chapters four and five attempt to bring together the ideas introduced in early chapters and outline how objecting thinking is made manifest. Chapter four focuses on objecting thinking as a foundation for discovery and definition of objects while chapter five deals with elaborating objects and thinking about their design and implementation.

Chapter six provides a collection of examples of object thinking and design consistent with the ideas presented in the main portion of the book. Some of the examples provide a bridge to related areas of software development thinking like patterns and architecture.

However focused on a particular topic each section of this book might be, there is an overarching bias or perspective that colors the discussion – i.e. "Behavioralism." If a single word could be used to capture the "… *fundamentally different ways of thinking about decomposition*" noted by Booch, it would be *behavior*. The centrality of behavior is most evident when considering and comparing various approaches to defining object model syntax, object application models, and methods for object development.

A majority of the published books on objects mention using behavior as the criteria for conceptualizing and defining objects. Many claim to present behavioral approaches to object development. However, with one or two exceptions[11] they provide object definitions and specification models that owe more to traditional concepts of data and function than to behavior. Even when behavior is used to discover candidate objects, that criteria is rapidly set aside in favor of detailed discussion of *data* attributes, member *functions*, and *class/entity* relationships.

This focus on behavior, and all its aspects and implications, is consistent, I believe, with both the origins of object ideas and the design tradition assumed by XP and to a lesser extent the other Agile approaches.

Material in the book is presented in a matter-of-fact style as if everything stated was unequivocally correct. Alternative ideas and approaches are cited but there is little effort to incorporate those alternatives or to discuss them in detail. This is not because the author is unaware of other viewpoints nor that he is dismissive of those viewpoints. Alternative ideas are best expressed by their advocates, and, to the extent that this book is used as a classroom text, discussion of those alternatives is best conducted by an instructor.

Intended Audience

A conscious effort has been made to make this book useful to a wide audience that includes professional developers and post-secondary students as well as anyone with an interest in understanding object-oriented thinking.

The contents of this book derive from material used in graduate software engineering classes where 90% of the students are employed as software developers and in several different commercial seminars[12]. It generally assumes that the students know a great deal about software development but little or nothing about behavioral object thinking. This book is, in part, a response to requests to consolidate material currently provided only in lectures and discussions.

A profile of the typical student in those courses and seminars best defines the target audience for the book. They are professional software developers, averaging eleven years of experience, or managers overseeing the software development process. They may be pursuing formal education at the graduate level but, if so, will likely be enrolled in applied programs of software engineering or business rather than theoretical computer science programs. Most will have encountered object concepts and many will have taken at least one other object class, usually a method or programming oriented commercial seminar.

Three caveats help to further define the audience for this book.

- This is not a programming text. Some limited examples of pseudo-code (reflecting Smalltalk syntactic conventions) are be presented when they can illuminate a concept or principle of development.
- The book, however, is expressly intended for programmers, especially those using Java and C++. It is hoped this book will facilitate their work, not by providing tricks and compiler insights, but by providing conceptual foundations. Languages like C++ and

Java require significant programmer discipline if they are to be used to create "true" objects and object applications. That discipline must, in turn, be grounded in the kind of thinking presented in this book.

- Although this book stresses the philosophy and history of objects/components, the primary focus remains on assisting people to develop pragmatic skills.

The reader is encouraged to engage the book in the order presented. It is possible to skip chapter one if the reader is prepared to take on faith the assertions made about objects and object thinking in subsequent chapters.

If this text is used in an academic course (undergraduate or graduate) roughly forty percent of the available class time should be devoted to workshop type activities. Software development of any kind is learned through experience but objects, because they are new and different, require even greater amounts of reflection and practice. Another characteristic of an ideal course is interaction and discussion of multiple viewpoints. This book is best used in conjunction with other method books (particularly any of the excellent books on UML or RUP) that can present alternative viewpoints and, of course, at least one of the XP or Agile texts to add depth to what is presented here.

About the Author

Currently Dr. West is a professor in the School of Business at New Mexico Highlands University where he is developing an object-based curriculum in software architectures, business engineering, and management information systems. He also teaches at the University of New Mexico and is engaged in developing a software development track for students in graduate and undergraduate computer science at that school.

Prior to joining the faculty at NMHU he was an associate professor in the Graduate Programs in Sofware at the University of St. Thomas and a consultant/trainer to several *Fortune 500* companies. He has taught courses in object-oriented development ranging from three-hour introductory sessions for managers, to multi-day technical seminars for professional developers, and semester long courses at both the graduate and undergraduate level.

He founded and served as the Director of the Object Lab at the University of St. Thomas. The Object Lab was a cooperative effort with local corporations dedicated to researching and promoting object technology.

He was a co-founder of the Object Technology User Group (the original, not the Rational sponsored group), first editor of its monthly newsletter, and principal organizer and chair of two regional conferences – attracting over 500 paid attendees – sponsored by OTUG.

Digitalk's *Methods* (the first incarnation of Smalltalk for the personal computer – later named Smalltalk/V) was his first object development environment – used to construct an "automated cultural informant," a teaching tool for cultural anthropologists learning ethnographic fieldwork techniques. His object experience is complemented by over twenty years of software development work, ranging from assembler programmer to executive management.

His undergraduate education at Macalester College (oriental philosophy and East Asian history) was capped with an MS in computer science (artificial intelligence) and an a MA in cultural anthropology followed by a Ph. D. in cognitive anthropology. All the graduate degrees were earned at the University of Wisconsin at Madison.

[1] West, David. "Enculturating Extreme Programmers," Proceedings …..
[2] Parnas, David Lorge and Paul C. Clements. "A rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineeering.* Vol. SE-12, No. 2, February 1986.
[3] McBreen, Peter. *Software Craftsmanship.*
[4] West, David. "Enculturating Extreme Programmers" and "Educating Xgilistas"
[5] Denning, Peter. ACM editorial.
[6] Ken Auer, www.rolemodelsoft.com.
[7] A neologism – useful for labeling those that embody one or more of the approaches that fall under the 'Agile' label.
[8] Booch, Grady. Object-Oriented Design. 1991.
[9] Kay, Alan. "The Object Revolution Hasn't Happened Yet." 1998?.
[10] Yes, UML does allow for a "responsibility" segment in their class diagram and description. But this hardly offsets the dominant trend in UML to treat objects as if they were "animated data entities" or "miniature COBOL programs."
[11] Wirfs-Brock's, Weiner's, and Wilkerson's**, *Objected Oriented Design*** offers a behavior based method, as does Wilkinson's ***Using CRC Cards*.**

[12] Much of the material has also been used successfully in undergraduate courses where students are limited in their software experience but have had numerous courses in traditional software development or computer science. It has also been used with remarkable success with audiences having absolutely no computer background!