JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering **©JOT, 2002**

Vol. 1, No. 2, July-August 2002

Notes on a Practical Guide and Thoughts on Software Development

A Practical Guide to Extreme Programming
David Astels, Granville Miller, Miroslav Novak The Coad Series, Prentice Hall, 2002, pp. 347 ISBN 0-13-067482-6

Reviewed by Dr. Ognian Pishev, Ocean Informatics Pty. Ltd.

This year's Design Tools and Processes Jolt Award was awarded to the Rational Unified Process (RUP)[1]. It had to withstand the assault of an old competitor – the index card (nominated by several developers). The index card is returning in stacks and decks to the development scene as the principal modelling tool as advocated by eXtreme programming and other agile methods. And yet RUP is still recognised as the market leader. Rational's suites of complex design and product management tools and the humble index card represent the two extremes of the wide spectrum of development methods. Will the evolution of software engineering be diverted by this resurgence of house-card building?

1 THE BACKLASH - EXTREME PROGRAMMING

The heavy dependence on its rigidities, its high cost and steep learning curve have produced a backlash against the Unified Process whose most notable example is the rapid emergence of eXtreme programming as the flavour of the day. It could be argued that the rapid dissemination of ideas of lightweight, human-centred and creative programming is a sociological manifestation of the constant competition and cooperation between self-organization and regulation as the forces shaping the nature of social activity. It can also be argued that this is a modern-day manifestation of the latent Luddite tendencies that rebel against the constraints of complex engineering environments.

The Practical Guide that is the subject of this review is representative of the new thinking that in a guerrilla-like fashion is beginning to spread in the software development community.

[1] http://www.sdmagazine.com/jolts/press_release_4-26-02.htm.

The godfather of extreme programming is Kent Beck, whose seminal work was followed by contributions from Martin Fowler, Scott Ambler and others. In the last couple of years various strands of rapid, lightweight agile development methods started converging towards XP, whose message is more humanistic than technological in nature. These methods include Ken Schwaber's Scrum, Peter Coad's Feature-Driven Development, and Jim Highsmith's Adaptive Software Development. Their roots go back to the start of the 90s. According to Jim Highsmith, the problem domain in which agile development improves the odds on success is what he has labelled as *exploratory projects*. He includes in the "ecosystem" the three characteristics that define agile development: a "chaordic" (chaos + order) perspective, "collaborative" values and principles and "barely sufficient" methodology[2].

XP is a method that is really extreme in its emphasis on the frequent delivery of tangible results. It is test-driven with the minimum required of design and modelling. It combines a number of innovations such as pair programming, short time-boxed iterations, refactoring, continuous testing and simplicity as a leading goal of software development. In his foreword to the book Scott Ambler states that:

> *"XP is ruthlessly focused on the creation of high-quality software, abandoning any sort of process overhead that does not directly support this goal. XP is explicitly people-oriented, going against common management wisdom that people are interchangeable cogs within your software process machine. XP is also based on the concept that software professionals can be successful simply by adopting and then following a collection of synergistic principles and practices, rebelling against the prescriptive process paradigm that defines procedural "cookbooks" for building systems."*

The conclusion that follows is that "XP is clearly different."[3] If we look, however at the best practices for software development processes adopted and adapted for XP, we will recognise many of the well-established principles of software engineering. These practices are as follows:

- Modular – the process is broken into discrete activities
- Iterative
- Incremental
- Time bounded – each increment has a fixed duration.
- Parsimonious – agile processes require a minimal number of activities necessary to mitigate risks and achieve their goals.
- Adaptive – new risks may require new activities, existing activities are modified.

[2] Highsmith, Jim (2002), *Does Agility Work?* Software Development Magazine, June.
[3] David Astels, Granville Miller, Miroslav Novak (2002), *A Practical Guide to eXtreme Programming*. Prentice Hall, p. xvii

- Convergent – all task are executed in such a way that after each increment the system is closer to its final goal. Developers are converging on a solution, architecture and design being the two keys in maintaining convergence.
- People oriented – Agile processes work best with small teams

- Collaborative
- Complementing[4]

According to the *Practical Guide,* XP is fluid. Each area (conceptualisation, planning, development, and delivery) is like a colour in a kaleidoscope. XP declares as its most extreme principle the insistence on having a real customer working directly with the project.

Metaphors are suggested as a powerful way of creating the *big picture view* of the system and the problems that it is to solve. They give a conceptual framework and a source of terminology as well as define the overall conceptual architecture of the system[5].

If you have metaphors, perhaps you don't need planning? XP admits that you must do *some* planning. Planning decisions are made by **those who know**. The customer makes decisions about the business: *scope, priority, release content and dates. Technical people decide on estimates, consequences* (technical results from decisions affecting tools, hardware, subsystems, etc.), process (team organisation, culture, facilities), *and detailed scheduling*.

Testing is another central element of XP philosophy. Unit tests are written before the code that runs them. Accordingly, tests define and document desired behaviour; they give you the feedback on the full and correct implementation. The logic that leads to the extreme conclusion that test cases are used as semantic modelling tools is turned on its head. Is there a better way of describing the semantics (behaviour) of various software elements (routines and classes) before they are implemented? Shouldn't the logic be reversed with test procedures following from the formal specification (design by contract)? I think that the answer to both questions is yes.

Simplicity is the motto of eXtreme programming. Even design has to be deferred until it is required ("*just in time design*"). Refactoring is the tool that makes code nimble and malleable.

Collective code ownership is another revolutionary idea, which is based not on programming work divisibility but on the relative ease of replication of software artefacts that poses additional coordination problems.

Releasing in small increments helps avoid complications: it is often said that you should *release early, release often, "frequent, tangible, early results."*

[4] Ibid, p. xxvi-xxviii.

[5] If we don't understand metaphors, we can use allegories. The example from the now mythical Chrysler Comprehensive Compensation project describes how the decision to use a manufacturing metaphor led to the overall success of the project. The concepts of lines, parts, bins, jobs and stations were used to describe a rich accounting domain model and the project team was able to quickly learn and understand an extremely complex domain.
It is hard to understand how accounting felt adopting the metaphor of an assembly line to describe posting rules in a payroll application; in the end accountants do not assemble cars and may feel more comfortable using metaphors from banking or baseball, for example.

Staying flexible is the name of the game played by two cooperating teams, the customers and the developers. Both have their rights that have to be preserved throughout the execution of a project.

Storytelling is the principal form of requirements gathering, the principal tool is the index card. However you MUST not write your unit tests on the back of the index card with a story.

The two teams are aligned through the process of conceptualisation. And this is a difficult process since one has to decide precisely what to build.

> *"Trying to create a complete picture of the system up front sets up a dynamic that ensures problems for nontrivial systems. Moreover, setting this picture in concrete due to a cost-of-change curve created by the development process itself just makes the possibility of failure that much greater. XP has shown that you can operate without complete knowledge and that you can delay the need for 100 percent understanding through customer involvement and small releases."*[6]

One simple solution (that will probably be discarded after the end of the planning and estimating exercise) needs to be formulated as a means of estimating work and unifying the software development team. XP however is not looking for architecture in conceptualisation. Kent Beck defines architecture as all of the design decisions that will not change. Baselining the initial architectural vision does not contradict the statement that on XP projects, architecture is an emergent phenomenon.

We throw away our simple solution, as we only need it to perform the estimates of the user stories[7]. "To successfully deliver applications that will delight out customers requires that we have an expert understanding of their domain."

Metaphorically speaking, planning is a little like taking multiple trips to a grocery store (K.Beck). During each trip we have a fixed budget for groceries. I am almost ready to accept the analogy, being a proponent of the shopping list approach (B.Meyer) to class definition[8]. And of course, planning is regarded as an art, not a finely tuned science.

Estimation is made in story points[9] (quite different from function points, you may get more points for a more interesting story or an *epic*, not to mention the *sequel*). Velocity, or the number of ideal weeks or story points that a team can complete in a fixed amount of time (called an iteration), helps us make our estimates more realistic.

[6] A Practical Guide… p. 33.

[7] User stories are the itinerary of our journey (with the customer, the storyteller, as our guide through a foreign country). "They are simple descriptions of a single aspect of our system." We stack the index cards describing those user stories, hold them together with a rubber band and then produce the deck that describes the entire journey. "A user story" is the smallest amount of information (a step) necessary to allow the customer to define a path through the system."

[8] In (1997*) Object-Oriented Software Construction*, Bertrand Meyer advocates the shopping list approach: the realisation that it does not hurt to add features to a class if they are conceptually relevant to it. (p. 771-772).

[9] *Story points* are ideal weeks of development time, where 40 hours are dedicated to programming.

Iterations, according to the *Guide*, are an acknowledgement that we do things wrong before we do them right. In normal human activity, iterations or approximations are a way of discovering the truth. There are no rules that guarantee the success of knowledge discovery but we can still base our approach on scientific methods and experience that help us avoid costly mistakes. If we expect to be wrong with the first iteration, which, according to the book "defines the core system on which all other iterations will be built", we will have difficulties getting the project off the ground.

Design is one of the controversial areas of eXtreme programming. Quite often, the need for design activities is dismissed. XP, however, is about continuous design. What XP is against is *big upfront design*.

The evolutionary approach to design is not new and has been implemented in a number of development environments through the implementation of abstraction. In Eiffel for example (and in C++), abstract classes may define routines whose implementation is deferred (virtual) and left to concrete classes that inherit general definitions but elect specific implementation. The difference between the Eiffel method and all other development methods lies in the fact that it eliminates the separation of software activity into analysis, design and implementation thus transforming programming into a continuous roundtrip engineering that is seamless and reversible.

The XP process instead relies on simple designs, automatic tests, and aggressive refactoring. Design is needed upfront only to establish (just enough, but not too much) a metaphor and overall shape. One of the most controversial positions adopted by eXtreme programming affect the role of architecture.

> *"In XP, there isn't always something you can point to and label as the architecture. Architecture is partly defined by he metaphor. In this capacity, the metaphor gives an overall context and shape to the system. The aggressive Xpers (Beck, Jeffries, Martin, and the rest) are of the opinion that any upfront architectural design is to be avoided. Refactoring will move the design in the required direction… By keeping the code clean, I can always add complexity later when it has been proven to be required."*[10]

This radical statement completely overlooks the fact that architecture-driven development reduces complexity by creating a modular design, simplifying teamwork and organising concurrent development of relatively independent clusters thus achieving the agility of flexibility advocated by XP. Refactoring is a useful technique of improving an object-oriented programme and quite often deals with implementation choices. We need however to have a clean design that is inconceivable through the constant use of refactoring.

Instead of developing its own version, XP has adopted agile modelling (AM) as formulated by Scott Ambler. AM is a collection of values, principles and practices for modelling software that can be applied in an effective and lightweight manner. Its

[10] A Guide… p.134.

principles include the importance of *assuming simplicity,* recognizing *incremental change*, the recognition that *content is more important than representation*. Its values include communication, honesty, simplicity, feedback, courage and humility[11].

Having lost the battle in a marketplace dominated by Rational Rose, Scott Ambler, like many other gurus has switched to guerrilla warfare. He supports Alistair Cockburn in advocating the revival of the Class Responsibility Collaborator modelling[12] and the return of the index card exemplifies for him the trend toward simple tools[13]. He is even ready to foretell impending doom for CASE tool vendors, particularly for those that are unable to support agile methods.

2 WHAT ARE THE ALTERNATIVES?

We looked at the two extremes of the development method spectrum: the orderly full-blown engineering approach of the Rational Unified Process and the humanistic flexible and agile eXtreme programming. We might be tempted to suggest that the next step would be the return to the centre. But this where problems begin as Goethe, the natural scientist, liked to say. Are we left without any alternatives outside the mainstream activity and its guerrilla opposition? My conviction is that the current debate is neglecting powerful ideas that provide revolutionary exit from the rather limited framework of the current debate. The first one is Design by contract as defined and realised by Bertrand Meyer, the second is Archetype methodology as implemented within the OpenEHR initiative[14].

Design by Contract

There are several persistent themes in software engineering literature. One of them is Design Patterns, Design by Contract is another.

> *"Design patterns can be considered reusable microarchitectures that contribute to an overall system architecture. Ideally they capture the intent behind a design by identifying the component objects, their collaborations, and the distribution of responsibilities. The purpose… is to show how this intention can be made* explicit, *based on the notion of* contract. *A software contract captures mutual obligations and*

[11] See *A Guide*…pp. 134-145. The book gives the example of TogetherSoft whose tool provides continuous synchronisation of model and code or as the eXtremos say, "The design is the code." But this is the principle adopted by Eiffel since its appearance back in 1985. Source code text and graphic models are two representations of the same thing and within the development environment of Eiffel Studio 5.1 it is possible to constantly switch from one view to another thus providing the best of both worlds.

[12] Ward Cunningham and Kent Beck were CRC proponents. See their forewords to David Bellin, Susan Suchman Simone (1997) *The CRC Card Book*. Addison Wesley.

[13] Scott Ambler (2002) *Easy Does It*. Software Development Magazine, March.

[14] The OpenEHR Foundation was established with the goal of producing an open-source model of an electronic health record. See: http://www.openehr.org.

> *benefits among stakeholder components… Contracts strengthen and deepen interface specifications. Along the lines of abstract data type theory, a common way of specifying software contracts is to use Boolean assertions called pre- and postconditions for each service offered, as well as class invariants for defining general consistency properties."*[15]

Using formal approaches to program validation, Bertrand Meyer formulated Design by Contract as the disciplined use of assertions to define the semantics of each class. The analogy (or should I say, the metaphor?) between business and software contracting is almost perfect. The prerequisites and resulting behaviour of each operation (called a routine in Eiffel) are specified through two types of assertions, *pre-* and *postconditions,* while overall class consistency is expressed through the *class invariant*. Having defined the semantics of each operation, we are then capable of formulating the contracts between *supplier* classes and all classes using their operations, or the *clients*. The entire software system is thus represented as a network of cooperating clients and suppliers whose exchange of requests and services are channelled not by a central planning authority but through decentralised contracts.

Design by contract makes possible the early definition of semantics expressed in terms of software contracts without the need to resort to premature commitment and early implementation decisions.

Instead of using tests to document and define behaviour, we reverse the logic of extreme programming. The consistent use of pre- and postconditions formalised through the use of first order predicate logic provides the same service while offering the assertions mechanisms as the basis for testing and disciplined exception handling[16].

BON (Business Object Notation) is the development approach based on the strengths of Eiffel and Design by Contract. It provides a systematic description of the general method, the BON process and standard activities.

It is not the topic of this paper to describe BON in detail and I would refer the reader to the book by Kim Waldén and Jean-Marc Nerson[17]. A number of books on Design by Contract have already been published but we are still waiting to see the new book by Bertrand Meyer on the same topic.

The general contribution of the Eiffel development method, which is fully supported by state-of-the-art tools, is:

- The elimination of the artificial separation between the software development activities: analysis, design and implementation

[15] Jezequel, Jean-Marc, Michel Train, Christine Mingins (2000) Design Patterns and Contracts, Addison Wesley.
[16] The best exposition of Design by Contract is provided in Chapter 11 of Bertrand Meyer (1997) *Object-Oriented Software Construction*, pp. 331-410, while broken contracts are covered in the following chapter on exception handling, pp. 411-438.
[17] (1995) Seamless Object-Oriented Software Architecture, Prentice Hall.

- The unification of coding and graphic modelling within a single programming paradigm (Eiffel Studio 5.1 provides this advanced tool support)
- The consistent application of object-oriented principles and mechanisms to architectural definition and decomposition

It is possible to build constantly evolving testable and executable models with clearly defined semantics ascending from the abstract to the concrete, as Hegel and Marx would have put it. Eiffel and Design by Contract give some of the best examples of component-based development within a single object-oriented architecture. The single model approach however has inherent limitations and they are addressed by the archetype development method.

Archetypes

The principal source of information on this exciting new development is the paper published on the Web by Thomas Beale. Its full title is *Archetypes: Constraint-Based Domain Models for Future-Proof Information Systems*[18]. The intellectual argument in this paper has as a starting point the simple statement that the purpose of any information system is

"The creation and processing of instances of concepts." (p. 11)

The Single-model methodology that is the dominant form of development process can be briefly described as follows: iterative writing of use cases (or user stories), finding classes, that is, defining the modular architecture, and building a model that will eventually become software. In the majority of cases, semantic concepts are invariably hard-coded.

No matter how agile and flexible you are, the classical modelling approach has a number of well-documented problems. In addressing them, the archetype method is led by concerns similar to eXtreme programming:

- The model encodes *only the requirements found during the current development*, along with guesses about future ones. Iterative requirements engineering or delaying the explicit formulation of evolving requirements do not answer the problem.
- *Models containing both generic and domain concepts in the same inheritance hierarchy are problematic*: general and specific concepts are mixed together; generalisation as advocated by the BON method is effectively prevented. Reusability across multiple domains suffers as a result
- *Technical problems* such as the "fragile base class"[19] have to be managed during construction and maintenance.

[18] http://www.deepthought.com.au, the most recent version is dated 21 August 2001.
[19] Mikhajlov, Leonid and Emil Sekerinski (1997) The Fragile Base Class Problem and its Solution. http://www.tucs.fi/Publications/techreports/tMiSe97.php. TUCS Technical Report No. 117, May.

- If the number of domain concepts is large and ongoing requirements gathering leads to an explosion of domain knowledge, *it is difficult to complete models satisfactorily*. In complex domains such as healthcare software system completion can be retarded significantly.
- There may be a *problem of semantic fitness*, which is expressed in the inability to model domain concepts directly in typical object formalisms such as classes, methods and attributes. Domain concepts are characterised by significant variability, and constraint definitions using first order predicate logic are required to complete their definition.
- The two types of specialists involved in domain modelling – domain experts and software developers – have to join forces, which *is logistically difficult to manage*. Business users are forced to adopt software description languages such as UML or help transform their story directly into code, while software developers are faced with the often impossible (and unnecessary task of becoming domain experts themselves).
- *New concepts are introduced at the cost of software changes, rebuilding, testing and redeployment*, which is expensive and risky no matter how fast iterations are.
- *Interoperability is difficult to achieve*: single-model systems have to conform to an interface standard (IDL), be connected to other systems built in the same language environment (Java) or proprietary technology (Microsoft, etc.) or correspond to exactly the same information model.
- *Standardisation is difficult to achieve (pp. 13-14).*

An answer to these problems is the design of a knowledge modelling approach. The dual-model methodology, in which standard domain terminology and concept models are used to drive the runtime functioning of software information systems is essentially the *interoperable knowledge methodology,* developed by Thomas Beale for the needs of the Good Electronic Health Record (http://www.gehr.org).

> *"In this approach, the single, hard-coded software model has become a small* reference object model *(ROM), while domain concepts are expressed in a separate formalism and maintained in a concept library. Accordingly,* software development can proceed separately from domain modelling. *Since the software is an implementation of the ROM, it has no direct dependency on domain concepts, i.e.*, if new concept models are introduced, the system does not need to be altered.*" (p. 17).*

Designing domain ontologies and creating an archetype language specific to the business domain thus break the dual dependence between business experts and software developers. This approach is in direct contrast to the early XP statement that, "To successfully deliver applications that will delight out customers requires that we have an expert understanding of their domain." It is much better to provide our customers with the tools to formalise their own understanding of the business. Archetypes defined as constraint-based concept definitions serve that purpose. The concept model, that eXtreme programming is struggling with to the point that it is ready to abandon the idea of it, becomes *structural constraint definitions*, i.e. the set of constraints, which together define the set of instances, which can still be called the same concept.

> *"Our reconception of information systems is now one in which domain concepts are formally modelled as archetypes, which are then* used *by the system at run time. The "software" as we previously knew it is now* an implementation of a reference object model. An archetype effectively corresponds to a constellation of valid combinations of ROM objects for a particular domain concept. " (p. 28)

While XP is stating that business users and software developers "are aligned through the process of conceptualisation", it does not provide a suitable conceptualisation mechanism. In focusing on knowledge modelling, the archetype method establishes solid foundations that can be used both in established areas such as healthcare and finance and in "exploratory projects" that have to create their own language.

The rigorous implementation of the best software engineering practices embodied in Design by Contract and the BON method and the dual-model approach advocated by the archetype methodology produce a powerful combination that is far superior to ad hoc agile modelling or heavy-duty process. Its requirements to the development method are qualitatively different to market leaders and their guerrilla opponents and are yet to be described with the necessary detail.

Books like *The Practical Guide* inadvertently create the feeling of inevitability and lack of other viable alternatives. In contrast to Unified Processes and eXtreme programming, this survey has highlighted ideas that break out of existing process thinking.

Dr. Ognian Pishev, Ocean Informatics Pty. Ltd., e-mail: pishev@vitosha.com