6

Method, Process, and Models

For forty years, software development theorists focused on method, process, and modeling as the key means for improving practice. All software problems could be resolved if developers would give up their idiosyncratic, imprecise, and careless ways. Formally defined methods would incorporate rigorous process and modeling requirements. Following the dictates of formal methods would eliminate all vestiges of ad hoc and subjective "art." Software development would be transformed; if not into a science like mathematics and physics, then at least into a solid engineering discipline solidly grounded on a science of computing.

Models would have precisely defined syntax (preferably based on a kind of predicate calculus) and all semantics would derive from formal transformations of that syntax. Properly constructed models would contain unambiguous truth and one model (a Data Flow Diagram, perhaps) could be mechanically transformed into another (source code, for example).

A process could be defined that would describe in precise detail each step required to move from vague idea to functional solution. At each step in the process, the developer would know exactly what to do, how to elicit required information, and how to express that information in syntactically correct models.

Methods of this sort could be expressed as automated tools. It would be possible to remove significant numbers of human developers from the process – especially those pesky and annoying programmers. Computer Assisted Software Engineering (CASE) tools would allow a properly trained analyst to construct precise models which would then be transformed into error free code at the touch of a button (click of a mouse). It would, in fact, be possible to build a "Repository," an ultimate CASE tool that would allow the automatic generation of computer programs and systems in response to a change in business requirements – without the intervention of analysts or programmers.

The vision of the theorists was compelling – and highly salable to managers who rushed to adopt the latest advances, buy the latest tools, and pay for audits that would document their compliance with the latest process standards.

Practitioners lacked such grand vision. Instead they had heuristics and practices, both grounded in experience rather than theory. The "art" and "craft" followed by expert developers was transmitted via oral tradition, mentoring, and informal mimicry. Real improvements in software development were realized – but "non self-consciously[1]" in the form of a shared culture and oral traditions.

Formal methodologists adopted many of these practitioner innovations and recast them in terms of their pet theories (sometimes ignoring the origins of their 'insights'). This allowed those methods to give the appearance of supporting new innovations without initiating any substantive change. In fact, innovations were all too often redefined by the methodologists as being nothing more than some feature already present in the method. "We have been doing objects since 1960, only we called them ________:" this, and similar statements, were expressed by innumerable mainstream developers and methodologists in response to innovations.

XP and the other Agile Methods represent the first attempt by practitioners to systematize (not formalize) practice in such a way that it could be perceived as a legitimate alternative to formal method. So legitimate in fact, that the backlash from mainstream formal methodologists is intense. "XP is nothing more than good software engineering." "RUP and CMM are just as agile as anything coming out of the Agile Alliance." "XP only works in certain niches (which are not really important anyway) while formal methods are required for most development." "XP is nothing more than a fad, a way for unemployed Smalltalk programmers to get jobs again."

The jibe about Smalltalk programmers alludes to important relationships among XP, methodology (in general), and object thinking. To fully see the connection it is necessary to make one last historical foray – a brief recap of object-oriented methods and models that have been advanced over the past twenty years.

Two Decades of Object Methodology

Scores of object development methods have been advanced since Adele Goldberg and Ken Rubin published their paper, "Object Behavior Analysis" while part of the same Xerox PARC project where the Smalltalk object development environment and programming language was invented. Their method clearly reflected the same presuppositions and philosophy that provided foundations for Smalltalk itself. Object Behavior Analysis (OBA) was described in published papers but it was never truly marketed as a method. An automated support product for OBA was developed but not, at that time, sold[2].

Behind the Quotes

Adele Goldberg

Adele Goldberg was a central figure in the Smalltalk world from the time the language was conceived at Xerox PARC until it was eclipsed by Java. Her concerns with education helped shape the evolution of Smalltalk and she contributed to its leaving PARC and becoming a commercial language. When PARC spun off ParcPlace Systems she became its president. With Ken Rubin, she authored the first behavioral object analysis method (Object Behavior Analysis or OBA) and later a definitive guide to object oriented project management – *Succeeding With Objects*. While at Parc Place Systems, the main competitor version of Smalltalk – from a company called Digitalk – was acquired and efforts were made to merge the languages and their customer bases. That effort failed, stopping the momentum of Smalltalk and allowing Java (a pale imitation of Smalltalk at that time) to succeed. She left ParcPlace and founded a company and product called LearningWorks that used the Smalltalk environment to teach computing and computer programming. For a time it was adopted by the Open University in England as a foundation for its programming courses.

1991 was a watershed year for object methods in terms of books published. Booch, Coad and Yourdon, Jacobson, Rumbaugh, Schlaer and Mellor, and Wirfs-Brock published books describing various methods. All of these methods could be characterized as "first generation." They were all produced quasi-independently and exhibited significant syntactic variation.

A great deal of argument ensued. Which method was best? Much of that argument coincided with arguments about implementation languages. It was common for methods to be closely associated with languages and praised or dammed based on that association. For example: Booch's method (which was originally written to support Ada development) was preferred in the C++ community, while Wirfs-Brock was more popular with the Smalltalk crowd.

In addition to arguing with each other, methodologists took note of each other's work and recognized omissions in their own. Points of concurrence were noted. Complements were noted. Booch, for instance, specifically noted the value of CRC cards for object discovery and preliminary definition in the second edition of his method book. Other efforts were made to extend the first generation methods to support specific types of problems. Real time systems and distributed systems, for example.

At the same time, "The Market" demanded a single "right answer" as to which method to adopt. Booch, Rumbaugh, and Jacobson, coming together under the Rational corporate umbrella were quick to consolidate their methods and models in order to present a "unified method." Given the sobriquet, "The Three Amigos," they advanced their cause. Their tool sets and methods became the foundation for UML (Unified Modeling Language) and the RUP (Rational Unified Process) – a tool and a method that dominate 'official[3]' practice today.

A review of all of the methods advanced in the nineties is not possible in this space. It is possible to categorize the plethora of methods into three general categories (data driven, software engineering, and behavioral) and look at exemplars of each category in order to outline the main points of divergence.

Data-driven approaches are exemplified by the work of Schlaer and Mellor, the Object Modeling Technique (OMT) of Rumbaugh, et. al., and the joint work of Coad[4] and Yourdon. The central focus of this category of methods is data and the distribution of that data, via the classical rules of normalization, across a set of objects. Objects in this instance are equivalent to Entities in classical data models. Schlaer and Mellor, for example describe the process for discovering objects in a manner that is indistinguishable from data modeling techniques.

OMT syntax and models allow for more flexible description of relationships among objects and does a better job of capturing interaction among objects in non-data specific ways, but still defines objects in terms of data attributes. Coad's and Yourdon's Object-oriented Analysis (OOA) method essentially advocates the creation of an entity diagram, placing functions with the distributed data attributes, and adding message passing to the data model.

Data driven methods do not make the object paradigm shift. Instead they carry forward a legacy abstraction, data, and use it as the basis for decomposing the world. Followers of this type of method "think like data" or "think like a (relational) database." This obviously has the appeal of familiarity and consistency with legacy systems and provides the appearance of a quick path to objects. It is also quite consistent with the definition of an object as a package of data and methods.

It is not consistent with decomposition of the world in a natural way because the world is not composed of computationally efficient data structures. There are no "natural joins" in the domain that map to normalized entities. This disjunction is evident in the world of data modeling itself – just consider the differences between a conceptual data model (models data as understood in the domain) and a logical data model (normalized for implementation).

Data driven methods tend to create objects with more frequent and tighter coupling than other methods. Class hierarchies are therefore more brittle and changes in class definitions tend to have greater impact on other class definitions than desirable. (**Figure 6.1**) Distributing data across a set of objects in accordance with normalization rules and by following the dictum that a class must be created if a class (entity really) has at least one attribute different from any existing class needlessly multiplies the number of classes required to model a domain. (**Figure 6.2**)

F06xx01

Figure 6.1 - DMV Example (patterned after Coad and Yourdon 1991)
Data modeling rules lead the designer to place the fee attribute and, therefore, the calculate fee method in the LegalDocument class. This mandates the need for a case statement to handle fee calculations and couples LegalDocument to the entire Vehicle class hierarchy.

F06xx02

Figure 6.1 – Customer and Credit Customer
The addition of a single new attribute requires the creation of a new class – allowing some Customers to buy on credit and therefore have a 'creditLimit' attribute, for example.

Booch, and Jacobson exemplify **Software Engineering** methods. They openly acknowledge an intellectual debt to classical structured development and claim to be an appropriate response to the demand by management and government for a documented (formal) process for software development. They are characterized by multiple models that, when (and if) integrated present a comprehensive specification of a desired software artifact. Supporting code generation is a frequently stated motive for these methods.

Most software engineering methods claim to be (partly) behavior driven. Jacobson's Use Cases are essentially identical, in fact, to the scenarios of OBA, a prototypical behavioral method. Because they are focused on the artifact - the software - they tend to fall into thinking about an object as encapsulated data and procedures – thinking like a computer. Nothing in these methods compels a design to reflect computer thinking.

However, replication of modeling syntax from structured development, and the vocabulary employed in discussion, (e.g. attributes and operations), reflect the "artifact centric" point of view borrowed from structural development. Developer focus remains on the artifact and its design to meet specification, rather than understanding and modeling the problem domain.[5]

Software engineering methods (with the exception of Booch's 1991 book) fail to stress the paradigmatic differences[6] between objects and traditional modules and data structures. Even Use Cases, as developed by Jacobson, are less about responsibility discovery and assignment than they are about requirements gathering.

Behavioral methods, despite being first and most closely allied with the development of object thinking, have always received less attention that data and engineering methods. Proponents of other methods are quick to assert that this lack of popularity reflects fundamental flaws or lack of utility in behavioral approaches. They are wrong – other factors account for the failure of behavioralism to capture "market share."

First, behavioralism was not promoted, as were competing methods. The originators of behavioral ideas – Goldberg and Rubin at PARC and Beck and Cunningham (inventors of the CRC card) never published any kind of "method book." At a time when management almost demanded an automated tool to accompany and support any new method:

- PARC chose not to market the tool they had created. Although they did provide the tool to those taking object analysis and design seminars from them.
- Beck and Cunningham actively lobbied against the creation of any kind of automated CRC tool. Their object was quite sound, philosophically, but disastrous in terms of marketing.

The first, and for a long time the only, book promoting the behavioral approach to objects was that of Wirfs-Brock, Weiner and Wilkerson. Nancy Wilkinson published a small volume outlining classical CRC cards a few years later - long after the "method war" had been all but won by the opposition.

Second, and more importantly, behavioral approaches were the least developed in terms of expressiveness of models and making the transition from analysis to design and implementation in a traceable fashion. It would not be misleading to characterize behavioral methods, circa 1991, as "CRC cards - then Smalltalk."

Wilkinson, in fact, suggests the primary value of CRC cards and behavioralism is to provide an informal and rapid way to obtain input for more formal software engineering methods. (Booch, OMT (Rumbaugh's Object Modeling Technique), and OOSE (Jacobson's Object Oriented Software Engineering) were the formal software engineering methods she seemed to have had in mind.

Third, behavioral approaches are the most alien to established software developers – even today. Thinking like an object is very different from traditional conceptualizations of the software development process. It requires, at least at the beginning, constant diligence to avoid falling into old mental habits. It is hard. Most people are unwilling to engage in this kind of hard cognitive work without a compelling argument as to why it is worth their while. That argument was never made – except in a kind of oral tradition shared by a very small community.

In fact, behavioralism had the same reputation as XP does today – as being "anti-method." There is a small element of truth in this assertion. Both OO and XP proponents oppose the use of methods as traditionally defined. Especially comprehensive, highly formalized, and labor intensive methods that prevent developers from immediate engagement with "code" – with programming.

This does not mean that object thinking lacks anything resembling a method. (The same is true of XP.) It merely means that the rigor and the systematization of work that accompanies object thinking is quite different from what most developers have come to associate with the term method.

To understand the relationship between method and XP it is necessary to take a few moments and reflect on how method provides value to software developers.

Purpose and Use of Method

Methods come in many guises. The simplest is nothing more than the adage, “If at first you don’t succeed, try and try again.” Hacking, trial and error, and rapid prototyping are examples of methods based on this precept.

Other methods are more prescriptive, instructing the developer as to what to do and when to do it. The simplest form of prescriptive method would be a ‘checklist; like that used by pilots for each phase of flight. Such checklists consist, essentially, of a set of questions of the form, “did you remember to do this action?” Actions that need to be done in a particular order simply have their reminder questions occur in the necessary sequence.

Prescriptive methods can become highly complex and comprehensive as the checklist expands to include documentation that must be produced to confirm that each action did in fact take place, and syntax that specifies the exact form of the documentation so that it is known the action was done correctly. (I have seen one such method that filled almost twenty volumes defining each step, document or artifact produced, and syntax for determining correctness.)

Between hacking and prescriptive overkill is the realm of formal, informal, and aformal methods.

Formal methods tend to be prescriptive, characterized by a large number of models each of which requiring a reasonably complex syntax, filled with ‘rules’ and ‘techniques’ that assure proper use of the method, and fairly strict in terms of activity sequencing. Most offer a promise of removing human developers, to some degree, from the development process – via code generation, for example.

Informal methods require fewer models with simpler syntax, demand fewer activities, and offer heuristics instead of rules and techniques. Most pick a single activity – programming for example – and elevate it to primary status while suggesting that all other activities are valuable only in so far as they support the primary.

An **aformal method**[7] would reject the idea that any task, model, syntax, rule, technique, or heuristic has any intrinsic value. Such things are valuable only to the extent that they support or assist the innate capabilities of the human developer(s) engaged in a particular task at a particular moment in time. Both formal and informal methods are seen as a kind of 'exoskeleton' that might offer some protection, but at the cost of severely limiting the being "enjoying" that protection. Improving the innate character and capabilities of the human developer is the alternative to defining an external method, for an aformalist.

Methods themselves are less important than the culture shared by those that embrace a method. Software engineers, for example, represent a culture enamored of formal methods. Those methods are considered necessary and desirable because the method expresses the set of presuppositions, values, and worldview shared by the members of the culture. Formalists like formal methods – an obvious truism.

Managers, academics, software engineers, computer scientists, proponents of UML, RUP, and CMM all tend to be formalists. Practitioners, as documented by Parnas and Glass among many others, generally are informalists. XP and object thinkers are aformalist.

Methodological conflicts, therefore, are really cultural conflicts. When an advocate of RUP asserts the claim that "RUP is Agile," they are making an irrelevant statement. Of course RUP can be agile – it is merely a tool after all and the practitioner has the choice of how, when, and why to use that tool. But, someone from the culture that values RUP cannot be agile. (OK, technically, will not be agile. OK, OK, is very unlikely to be agile.) The same, mostly non-conscious, worldview that leads to adoption of RUP and UML will prevent the use of that method in any but a formal way - unless a constant, conscious, and deliberate effort is made to adopt the Agile worldview and cultural values at every step of development.

Formalists will advocate methods that are focused on the production of computer artifacts (software programs). Such methods will liberally employ terms like software *engineering* and will stress correctness and both syntactic and semantic integrity. It is held that such methods are capable of producing a "correct" solution to an unambiguous requirement, prior to the expensive process of committing it to code. Code generation is then a happy by-product of formal methods. The major drawback, of course, is that the effort required to produce the formally correct models is essentially the same as required to code and test. This makes for a steep learning and overhead curve for those using this kind of method. They will also suffer from some degree of mismatch between the real world and the formal world they attempt to describe. The real world is far fuzzier, flexible, and illogical than any formal method can accommodate.

Informalists are more likely to focus on methods that support negotiated understanding; the informal communication oriented methods like JAD and CRC. The goal of these methods is to produce documentation of a consensus understanding of both the problem and the desired solution – in a context defined by the problem domain. It is clearly recognized that these solutions are bound in terms of time and composition of the group[8]. The primary problem with these methods is the lack of technique for translating the human understanding of a solution into a "computer understanding." The computer, after all, is a formal machine and can only operate on formally defined products. This explains the tendency of informal methods (XP, rapid prototyping, open source, etc.) to focus on programming – the actual interface between formal machine and informal developer.

Aformalist appear to reject method entirely. Observation, however, reveals patterned behavior and tasks performed in a sequential manner. Models are constructed (on whiteboards, Post-It Notes, and 4x6 cards more often than with the use of an automated tool) giving those models a relatively short half-life. Aformalists do not reject method so much, as they reject the idea that methods and models have any intrinsic value apart from those using them in a very specific context.

In one of the *Alien* movies, Sigourney Weaver's character fought the alien monster using a mechanical exoskeleton. This exoskeleton greatly increased her human strength by amplifying her human movements with electro-mechanical means. Formalists and a lot of semi-formalists see method as a similar kind of exoskeleton, one to be wrapped around a human developer in order to amplify their skills.

An aformalist viewing this scene would be mildly impressed by the magnification potential of the exoskeleton but be completely appalled at the cost required to gain that advantage. The human becomes a mere part in the machine and is severely restricted to a range of actions determined by the exoskeleton and its designers. They would note that continued use of such mechanical strength enhancement devices would perpetuate the physical weakness of the human and, likely, increase muscle atrophy over time.

Object thinkers and extreme programmers reject the "method as exoskeleton" approach in favor of a "method as weight room" idea. A human uses mechanical weights and machines in a gym to increase their innate capabilities – to make their own muscles stronger and more reliable. Using a more cerebral (and therefore more appropriate for object thinking) metaphor, Kent Beck suggests using method and even XP practices as if they were etudes (musical exercises designed to help the musician internalize certain skills and techniques). Etudes are used for practice, to increase and discipline the innate capabilities of the musician so that she can then go on stage and perform music.[9]

Object thinkers and extreme or agile developers find value in method only to the extent that it helps them become better practitioners. The method itself should "wither away" as developer skills increase until it becomes the merest vestige required to act as a reminder or trigger for the human developer to apply his or her enhanced skills when and as appropriate.

This is not a new idea. The south-Asian philosophy of karma yoga is grounded in the idea that right actions allow you to minimize the consequences of your actions sufficiently that you can maximize your changes of enlightenment. Aristotle suggested, "wear the mask of a good man long enough and you become that good man." David Parnas suggested some very good reasons for 'faking" a rational design process even when you could not actually follow such a thing. Christopher Alexander stressed repeatedly that a Pattern Language is but a Gate that must be passed through and left behind before you could actually practice the Timeless Way of Building.

For object thinkers, method is never an end in and of itself. There is no intrinsic value in either method or process. Both are useful only to the extent that they provide practice in the use of, and a means of enhancement for, innate human capabilities. This worldview cannot be overemphasized.

Within the context of this worldview, it is possible to ask if some methods, some tools, some models, are more suited to promoting object thinking than others? And the answer is yes. To determine which are the most efficacious tools, the following criteria can be used:

- First, the method, tool, or process cannot substitute its own goals in place of those articulated in object (or XP) philosophy.
- Second, each task advocated by the method and each model created using the method must contribute to the realization of basic goals. Exactly how must be clearly articulated.
- Third, the models, vocabulary, and syntax associated with the method and its models must value expressiveness over correctness. Models must evoke knowledge in the head of the developer instead of making pretensions to unambiguous representation of that knowledge.
- Fourth, methods and models useful for the enhancement of object thinking must also:
    - Provide support for "natural" decomposition and abstraction of problem (enterprise) domains. This requirement refers back to the simulation notion behind SIMULA and the object paradigm.
    - Recognize that objects require two complementary processes for development: domain modeling that yields a set of classes fully capable of simulating the domain in question; and, assembly and scripting of objects from those classes into applications that actually produce desired results.
    - Heuristics for discovery and evaluation. Heuristics should be grounded in appropriate metaphors to facilitate learning..
    - Heuristics or metrics that allow you to measure progress and 'goodness."

A Syncretic Approach

The preceding discussion emphasizes the differences between formal and aformal approaches to software development. It might lead one to suspect the gulf between the two philosophies is unbridgeable – especially when considering method.

Even hard-core object purists recognize that there are many things of value, even in the most formal of methods. Blending methods and approaches presents a significant challenge.[10] Incorporating valuable ideas from formal methods in such a way that the worldview and values of object thinking are preserved is equally hard.

Nevertheless, it is desirable to find some kind of common ground. Two prerequisites are required. First, the term 'method' will be abandoned in favor of 'approach.' This is purely a political move – to eliminate distractions caused by the "M-word" itself. Second, we will borrow a concept – syncretism – from anthropology and religious studies as a metaphorical title for developing a "syncretic approach"

Syncretism is a term encountered in anthropology and religious studies. It refers to a particular type of blending of traditions and cultures. For example, a visitor to a Catholic church in the Caribbean or South America will almost certainly find representations (icons, statues, relics, feast days, etc.) Catholic saints that are indistinguishable from "pagan" deities. And, attendees at Voudon ceremonies might be surprised to encounter rituals borrowed from the Catholic mass.

A Syncretic approach to software development would be characterized by a smooth transition from behavioral decomposition and analysis into representations sufficiently formal that they can be implemented as computer software. Along the way the user would see many things "borrowed" from other sources, things that might even be redefined in various subtle ways to maintain consistency of expression.

Using the label Syncretic *approach* is deliberate. Although many will take what is advocated in this book as YAM (yet another method) the intent is to create an approach to thinking about objects that maintains consistency with the object paradigm while complementing - not replacing - existing behavioral and software engineering methods. It will not be possible, however, to truly complement data-driven methods because they are almost the antithesis of what is advocated here.

Succeeding chapters will develop the Syncretic approach more fully but the remainder of this chapter will provide a summary of the process and a quick definition of the models (all of which are adapted from existing methods) that will be employed.

Paradigms, metaphors, and concept definitions provide the foundation for understanding object orientation. They do not however, tell us much about how-to-do object development. "How-to" requires:

- The enumeration of specific actions that may be taken in order to reach certain goals.
- Agreement as to a vocabulary (both verbal and graphic) and grammar (a language) for assuring mutual communication and understanding. Style, idiom and "standards" are important aspects of this common "language."
- Criteria by which we can evaluate ourselves, our progress, and our products.

Object thinking imposes four additional expectations that must be addressed by a syncretic approach to development. Implicit in these four expectations is the demand that two separate but complementary processes (as discussed in Chapter 4) be respected – object discovery and specification (Lego block construction); and, application assembly and scripting.

1. Decomposition based on discovering the "natural joins" in the domain. What goes on inside of a computer, the implementation context of the software to be created, is not part of the domain.
2. Responsibility assignment based on expected behavior in the domain as observed or inferred in the domain. Preserving existing channels and means of communication, even if there is intent to replace some existing objects with software simulacra, is essential. Implementation, even fore-shadowing of implementation, should not play any role in decomposition!
3. Aggregation of objects into communities capable of interaction and collective solution of a task or tasks. Not only must you identify what must be done by a group of objects acting in concert, you must also identify the constraints under which they must work and any qualitative characteristics that will be used to evaluate their work. Interaction of both human and automated objects must be taken into account.
4. A reasonably direct, metaphor preserving, means of adding design and implementation detail to objects based on their responsibilities.

The most tangible elements of a syncretic approach satisfying these requirements will be the set of models suggested for use by developers. This chapter will conclude with a brief introduction of a set of models and their syntax. Subsequent chapters will explore the actual use of those models.

Models

Models are valuable only to the extent that they facilitate communication among human beings. This implies that the effort required to construct the model must be less than the communication value that arises from its use. The dramatic failure of CASE (Computer Assisted Software Engineering) can be traced almost entirely to the fact that the effort required to learn and use the tools far exceeded the ability of resulting models to facilitate communication.

Unfortunately, there will always be a learning curve and some degree of overhead associated with the use of any set of models and any automated tool[11] supporting the construction of those models. It is the balance between benefit and cost that is critical in determining whether or not models and tools will be useful[12]. In that spirit, none of the following are essential in the sense that you cannot do object thinking without them. None of the models have fixed syntax or content – what is presented is suggestive. All of the models can and should be simplified as the developer internalizes object thinking skills – until, like the Cheshire Cat, only the model's "smile" remains to evoke the knowledge in the head of the developer and of the development team.

Semantic Net

A semantic net consists of little more than nodes, arcs, and text labels, Figure 6.3, with nodes representing 'things' and arcs representing 'relationships' between those things. Nodes are labeled with a noun or a noun phrase and relationship arcs with a verb or verb phrase.

F06xx03

Figure 6.3 – Semantic Net
A brainstorming tool for discovery of potential objects in a domain

A semantic net can be useful to "jump start" a discussion of objects in the same way that underlining nouns and verb phrases in a domain or problem description. Its primary utility is as a brainstorming tool that can be used by a group of people simultaneously. (A good sized whiteboard is very useful.) The model produced need not persist beyond the point when the objects, responsibilities, and constraints identified have been transferred to other models (like the Object Cube). The LIMT sidebar illustrates a typical use of a semantic net.

LIMT – Low-Income Mortgage Trust

The Legislature has created a $2 billion dollar fund to assist young couples to acquire a home they ordinarily could not afford – the LIMT. Mortgages are granted to qualified couples, subsidizes monthly mortgage payments when necessary, collects payments, invests whatever capital is not tied up in mortgages, and otherwise manages its assets.

To qualify for a mortgage:

1. A couple must have been married at least one year but not more than ten years.
2. Both spouses must be gainfully employed. At the time of application proof must be submitted of full-time employment for at least 48 of the previous 52 weeks.

3. The couple meets the NMMT definition of financial need by meeting one or both of the following requirements.

    a. Installments on a fixed rate, 30 year, 90% mortgage for a qualified home exceed 28% of their combined gross income.

    b. They lack sufficient savings to pay 10% of the cost of the qualifying home plus $7,000 (the estimated closing costs).

A mortgage may be issued to qualified couples if:

1. The price of the home to be purchased is below the published median price of homes in its area for the past twelve months.
2. The price of the home to be purchased is less than $150,000.
3. NMMT has sufficient funds to purchase the home.

NMMT calculates weekly its pool of available mortgage funds as follows:

1. Expected annual income from investments is calculated and divided by 52.
2. Expected annual operating expenses are calculated and divided by 52.
3. Total income from weekly mortgage payments is calculated.
4. Total of expected Payment Grants is calculated.
5. Pool amount is then equal to item 1 plus item 3, less item 2 plus item 4.
6. If the cost of homes to be purchased is less than the amount calculated in line five, LIMT is deemed to have sufficient funds.
7. Unspent funds each week are invested.

If a mortgage is granted, the couple must:

1. File an annual copy of their income tax form to confirm preceding year income.
2. Submit copies of pay slips to confirm monthly gross income each week.
3. Make weekly payments on the mortgage as calculated. Payment amount may vary from week to week.

Weekly payments are calculated as follows:

1. Capital repayment is 1/1560 of the purchase price of the home.
2. Interest payment is 1/52nd of the 4% of the current mortgage balance.
3. Escrow payment is 1.52nd of the sum of the annual property tax and homeowner's insurance premium.
4. Total mortgage payment is the sum of lines 1-3.
5. The couple will pay a maximum of 28% of their weekly gross income. If the mortgage payment calculated exceeds the 28% limitation a Payment Grant for the amount of the difference between the mortgage payment and the income limitation is provided by NMMT. This amount is a grant as is not added to the outstanding debt.

Processing Requirements:

1. All data requirements must be updatable.
2. Annual return on investments is updated frequently by the brokerage firm.
3. Expected expenses are updated quarterly.
4. Available funds are computed every week. A report is printed showing the computation details and results.

5. A listing of all or a selected subset of investments is printed on demand.
6. A listing of all or a selected subset of mortgages is printed on demand.
7. Weekly mortgage payments (with Payment Grants where applicable) are calculated and sent to morgagees.

Required Reports / Queries

1. List of Mortgages by specified price range.
2. Past Due Mortgage Payments with indication of which are less than 7 days, 8-14 days, 15-21 days, 22-28 days, more than 29 days late.
3. Total Grant amounts by month.
4. Status of specified account.
5. Combined details of mortgage application, mortgage information, and payment details.

F06xx04

Figure 6.4 – Semantic Net for LIMT example

Object Cube

An object cube is derived from the CRC card as invented by Beck and Cunningham and elaborated by Wirfs-Brock (particularly stereotypes). Object Cubes offer a single, consistent, metaphor-preserving, model objects. The model is used to aid thinking about decomposition as well as design and some aspects of implementation. Each of the six sides of the cube captures one critical aspect of the conceptualization of an object.

- Side One – (class, responsibilities, collaborations, the classic CRC card) contains the name of the class to which the object being modeled is an instance, a list of its responsibilities, and identification of any required collaborators.
- Side Two- includes a text description of the nature of the objects represented by the class. Stereotypes (as defined by Wirfs-Brock) should be included on this side and any notations helpful for implementation.
- Side Three – provides a list of "contracts" and the responsibilities (later the messages) associated with each contract. Contracts as a reminder to programmers of the intended senders for certain methods (making them public or private, for example).
- Side Four – identifies the discrete pieces of information that an object will require if it is to fulfill its assigned responsibilities. Each piece of information is given a descriptive label, a code indicating the source of the knowledge, and the name of the class that will embody or contain that knowledge.
- Side Five - lists of all the messages that an object will respond to. The message signature will identify the selector (the actual message name), the classes of an arguments, and the class of the object returned when the method associated with the message is executed.
- Side Six - Names and describes any events (changes in the objects state) that other objects might wish to be made aware of.

F06xx05

Figure 6.5 – Object Cube (side one)
The classic CRC card view of an object (LIMT example)

F06xx06

Figure 6.6 – Object Cube (side two)
Object description and stereotype (LIMT example)

F06xx07

Figure 6.7 – Object Cube (side three)
Contracts - categories of methods or responsibilities (LIMT example)

F06xx08

Figure 6.8 – Object Cube (side four)
Knowledge required by the object (LIMT example)

F06xx09

Figure 6.9 Object Cube (side five)
Message protocol

F06x10

Figure 6.10 – Object Cube (side six)
Events generated by object

Interaction Diagram

Interaction diagrams are very versatile – used to aid discovery as well as to capture implementation specifications. They model the communication among a group of objects engaged in a particular task. Most behavioral object thinkers employ the term "scenario" for interaction diagrams – or a nested set of such diagrams. Each scenario is bounded by a precondition (what must be true and what message must be sent to initiate the scenario) and a post condition (what is to be returned when the scenario completes normally).

F06x11

Figure 6.11 – Interaction Diagram Symbol Set
Symbols whose labels appear in italics are additions to standard UML

Use Cases, as advocated by Jacobson, are a nested set of scenarios (interaction diagrams) whose pre and post condition are defined by a single exchange between a "user role" and the target system.

F06x12

Figure 6.12 – Interaction Diagram (LIMT example)
Conversation required to fill-in and validate a mortgage application

A text equivalent of an interaction diagram is a simple table with the pre and post conditions noted as headers and columns for Client, Request, Server, Object Returned, and Comments. The columns for Client, Server, and Object Returned contain text labels for objects. The Request column contains a text description of the request. It will eventually map to a message sent to the Server by the Client. The comments column contains any explanatory text or commentary found to be useful.

F06x13

Figure 6.13 – Text-only Scenario Model
Text-only equivalent of previously modeled interaction diagram

A still simpler form can capture the same information as a top-level Use Case diagram – the requests and responses between a User Role and the System. These might also be used to exhaustively list all requests that might be made by one object of exactly one other object. The model has a header that identifies the Client object and the Server object. Columns are available for listing the requests that the Client makes, the Object Returned as a result of that request, and Comments.

F06x14

Figure 6.14 – Binary Use Case Model (text-only)
This text-only model captures same information as top level Use Case Diagram

These models are useful in defining the high level responsibilities of an application or system and when it is important to make sure that an object's responsibilities have been exhaustively determined. The concatenation of responsibilities on all Binary Use cases involving a particular object as server represents the entire list of responsibilities for that object. These models are used only occasionally.

Static Relation Diagram

Probably the most familiar to any software developer of all the models presented here. Static relation diagrams depict relationships, relatively static in nature, among a group of classes. Three common examples and their use:

· Class Hierarchy – a very simple diagram showing a single relationship - "is-a-kind-of." By convention the root (topmost class) of the hierarchy is labeled 'Object.'

F06x15

Figure 6.15 – Class Hierarchy Diagram (static model)
Showing single inheritance only

· Gestalt Map – the most common use of such diagrams – intended to provide an overview or gestalt of a system or application while minimizing the amount of detail required of a complete application or system without incorporating all of the details. A semantic net is a very simple example of a static relation diagram. Gestalt maps provide a convenient shared reference point that reminds developers of the objects involved in an application and important ways in which they are related. Some very common (they often have their own graphical syntax representation instead of requiring a text label) types of relationships captured on a gestalt map include: Is-a-Part-Of, Is-a-Collection-of, Is-Composed-Of, Uses, Coordinates, Associates, and Is-a-Kind-Of. Note the absence of labels like "Manages" or "Controls," relationships that should not exist from an object thinking point of view. Decorations and annotations added to a Gestalt Map enable the capture of constraints or business rules. This is a very common use, but in a later chapter we will argue that rules and constraints should be elevated to first-class objects in their own right and modeled as instances of "Self Evaluating Rules."

F06x16

Figure 6.16 – Gestalt Map
Provides an overview of the objects and their static relationships

· Collective Memory Map a specialized static relation that shows how knowledge is distributed among a community of objects and provides structural and definitional metadata relevant to that knowledge. Specifically it notes the distribution of objects – stored in instance variables of other objects - whose primary purpose is to represent some piece of data (strings, numbers, dates, etc.). Special purpose structures – e.g. valueHolders – are composed entirely of these 'data representation' objects. Any constraints and relationships that exist among this subset of objects are noted on this diagram. The most typical examples of such constraints or relationships are analogous to the key-foreign key relationships found in a standard relational data model and class (type) or value domains typically defined in data dictionaries or schemas.

F06x17

Figure 6.17 – Memory Map
A static view of the "data" in a system

Object State Chart

David Harel revolutionized state modeling with his "StateCharts." Almost all object modelers now use at least a subset of his notation to capture state dependent information about objects.

Properly defined and designed most objects will have exceedingly simple state. Therefore, the use of this kind of diagram to model object state is seldom required. The diagrams are extremely useful for modeling state driven interactions among objects – the typical GUI, for example.

F06x18

Figure 6.18 – Harel State Chart Syntax (subset)
The most commonly used symbols borrowed from Harel

I wish to conclude this outline of typically used object modeling tools by repeating the caveat that no tool has any value other than to assist in object thinking and to provide a means for inter-personal communication. If you can model your objects and your scenarios in your head while engaged in writing code – and if those mental models are consistent with object thinking – great! No need to write them down. If you and your colleagues use a visual model on a whiteboard as an aid to talking about, and clarifying your collecting thinking about, and erase the board when you are done meeting – also great! If your models are crudely drawn and use only a subset of the syntax defined here (or a completely different syntax that you and your colleagues collectively agree upon) – still great!

Model what you must and only what you must.

[1] Christopher Alexander differentiated between self-conscious and non self-conscious processes in architecture. The latter arise only in affluent societies with universities and give rise to architectural theories whose merits are based on abstract meta-arguments rather than any connection with practice or actual manifestations. Alexander, Christopher. *Notes on the Synthesis of Form*. 1968.

[2] Very late in the life of ParcPlace Systems (the spin-off that marketed the Smalltalk developed at PARC) the OBA tool was sold – under the name "MethodWorks." But by that time it had no hope of capturing any share of the development tools market. MethodWorks provided a set of forms, consistent with the OBA models containing textual documentation about the nature and relationships of objects.

[3] There remains a huge discrepancy between what developers officially use and what the actually use, both in terms of method and model. Licenses sold still does not equate to actual use by actual developers.

[4] Coad's later and independent work was far more focused on behavior and patterns of behavior among groups of objects.

[5] Only nominal attention is paid to the process of developing a domain class hierarchy as a tangible product in most software engineering methods.

[6] Booch is a notable exception to this rule. He clearly recognizes that a fundamental shift in thinking is required for object development. He also clearly recognizes the need to decompose the world based on the domain expert's point of view. He parallels Parnas's views on domain (design) centric decomposition. His method, however, does not develop nor directly support this aspect of object development. Instead his models and syntax are directly focused on what is needed to construct the software artifact.

[7] Aformal method is almost, but not quite, an oxymoron. Some familiar aspects of method remain even in the most aformal approaches to software development.

[8] Which is why the composition of the group must include users, managers, analysts, implementers, documenters, maintainers, etc. etc.

[9] It is true that some etudes are themselves sufficiently beautiful that they get played in concert, just like any other kind of music composition. This does not distract from the intent behind their creation.

[10] UML and RUP, it has been reported, took a lot of compromise and negotiation before it was hammered out, and the three methods synthesized there were basically similar. The elements of behavioralism in Booch were essentially discarded to make the synthesis possible. It proved much harder to combine behavioralist and software engineering approaches, although a product called *Fusion* from Hewlett-Packard made a valiant attempt.

[11] Most of the models are illustrated with examples from a tool, *Behavior!*, developed by the author and Kevin Johnson. *Behavior!* began as a graduate project but was developed almost to the point of being marketed (a late beta version) before other activities precluded completing the tool.

[12] Booch clearly recognized that developers had a upper limit on the amount of time they were willing to expend learning a method or syntax. He advocated the use of a subset of his full modeling set, called whimsically "Booch Lite," as the kernel that was most useful most of the time.