CHAPTER 10

SOFTWARE ENGINEERING MODELS AND METHODS

ACRONYMS

3GL	3^{RD} GENERATION LANGUAGE
BNF	BACKUS-NAUR FORM
FDD	FEATURE-DRIVEN DEVELOPMENT
IDE	INTEGRATED DEVELOPMENT ENVIRONMENT
KA	KNOWLEDGE AREA
PBI	PRODUCT BACKLOG ITEM
RAD	RAPID APPLICATION DEVELOPMENT
UML	UNIFIED MODELING LANGUAGE
XP	EXTREME PROGRAMMING

INTRODUCTION

Software engineering models and methods impose structure on software engineering with the goal of making that activity systematic, repeatable, and ultimately more success-oriented. Using models provides an approach to problem solving, a notation, and procedures for model construction and analysis. Methods provide an approach to the systematic specification, design, construction, test, and verification of the end-item software and associated work products.

Software engineering models and methods vary widely in scope—from addressing a single software life-cycle phase to covering the complete software life cycle. The emphasis in this Knowledge Area (KA) is on software engineering models and methods that encompass multiple software life-cycle phases, since methods specific for single life-cycle phases are covered by other KAs.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MODELS AND METHODS

This chapter on software engineering models and methods is divided into four main topic areas:

- *Modeling* – discusses the general practice of modeling and presents topics in modeling principles; properties and expression of models; modeling syntax, semantics, and pragmatics; and pre-conditions, post-conditions, and invariants.
- *Types of Models* – briefly discusses models and aggregation of sub-models and provides some general characteristics of model types commonly found in the software engineering practice.
- *Analysis of Models* – presents some of the common analysis techniques used in modeling to verify completeness, consistency, correctness, traceability, and interaction.
- *Software Engineering Methods* – presents a brief summary of commonly used software engineering methods. The discussion guides the reader through a summary of heuristic methods, formal methods, prototyping, and agile methods.

The breakdown of topics for the Software Engineering Models and Methods KA is shown in Figure 1.

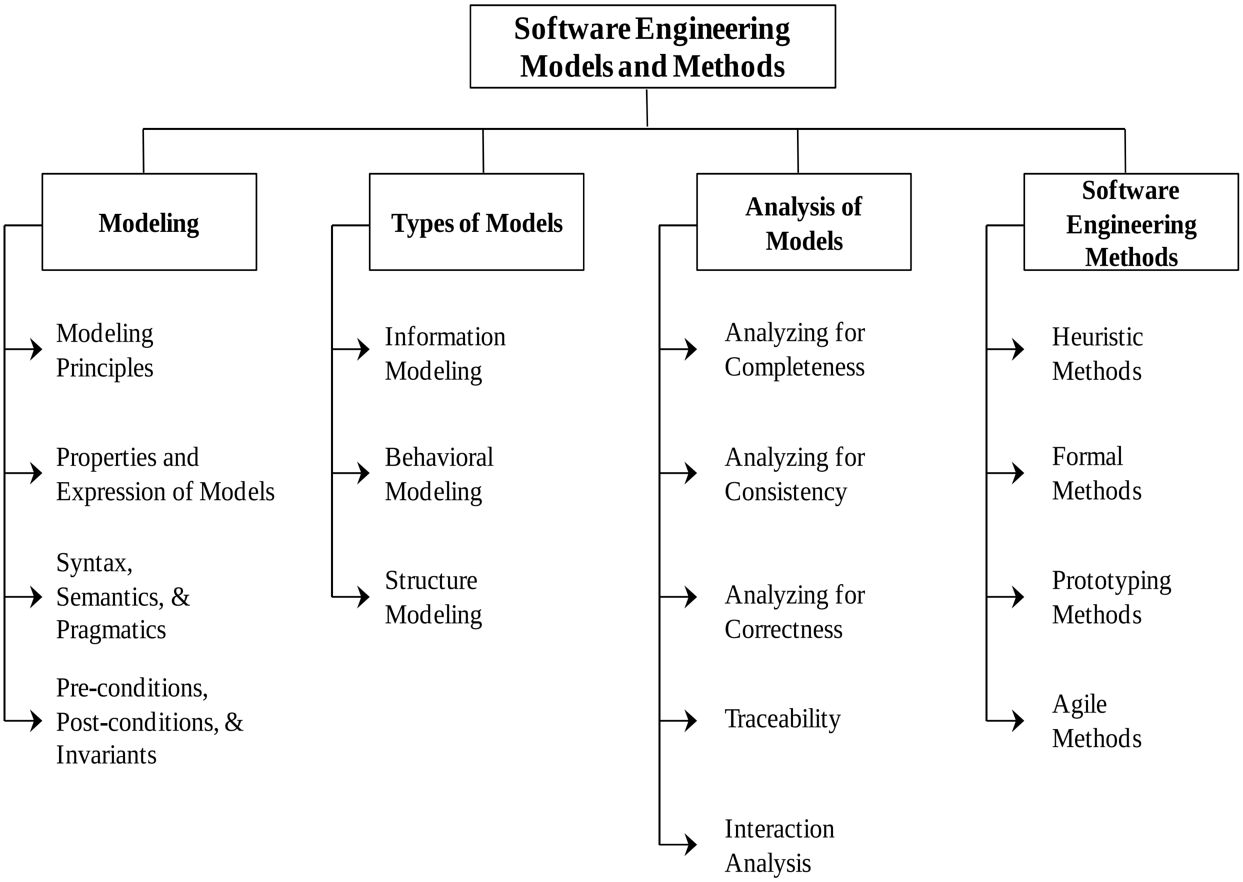

Figure 1 Breakdown of topics for the Software Engineering Models and Methods KA

1 Modeling

Modeling of software is becoming a pervasive technique to help software engineers understand, engineer, and communicate aspects of the software to appropriate stakeholders. Stakeholders are those persons or parties who have a stated or implied interest in the software (for example, user, buyer, supplier, architect, certifying authority, evaluator, developer, software engineer, and perhaps others).

While there are many modeling languages, notations, techniques, and tools in the literature and in practice, there are unifying general concepts that apply in some form to them all. The following sections provide background on these general concepts.

1.1 Modeling Principles

[1 c2s2, c5s1, c5s2; 2 c2s2; 3 c8s0]

Modeling provides the software engineer with an organized and systematic approach for representing significant aspects of the software under study, facilitating decision-making about the software or elements of it, and communicating those significant decisions to others in the stakeholder communities. There are three general principles guiding such modeling activities:

- *Model the Essentials* – good models do not usually represent every aspect or feature of the software under every possible condition. Modeling typically involves developing only those aspects or features of the software that need specific answers, abstracting away any non-essential information. This approach keeps the models manageable and useful.

- *Provide Perspective* – modeling provides views of the software under study using a defined set of rules for expression of the model within each view. This perspective-driven approach provides dimensionality to the model (for example, a structural view, behavioral view, temporal view, and other views as relevant). Organizing information into views focuses the software modeling efforts on specific concerns relevant to that view using the appropriate notation, vocabulary, methods, and tools.
- *Enable Effective Communications* – modeling employs the application domain vocabulary of the software, a modeling language, and semantic expression (in other words, meaning within context). When used rigorously and systematically, this modeling results in a reporting approach that facilitates effective communication of software information to project stakeholders.

A model is an *abstraction* or simplification of a software component. A consequence of using abstraction is that no single abstraction completely describes a software component. Rather, the model of the software is represented as an aggregation of abstractions, which—when taken together—describe only selected aspects, perspectives, or views—only those that are needed to make informed decisions and respond to the reasons for creating the model in the first place. This simplification leads to a set of assumptions about the context within which the model is placed that should also be captured in the model. Then, when reusing the model, these assumptions can be validated first to establish the relevancy of the reused model within its new use and context.

1.2 Properties and Expression of Models

[1 c5s2, c5s3; 3 c6s1.1p7, c7s3p2, c8s0p4]

Properties of models are those distinguishing features of a particular model used to characterize its completeness, consistency, and correctness within the chosen modeling notation and tooling used. Properties of models include the following:

- *Completeness* – the degree to which all requirements have been implemented and verified within the model.
- *Consistency* – the degree to which the model contains no conflicting requirements, assertions, constraints, functions, or component descriptions.
- *Correctness* – the degree to which the model satisfies its requirements and design specifications and is free of defects.

Models are constructed to represent real-world objects and their behaviors to answer specific questions about how the software is expected to operate. Interrogating the models—either through exploration, simulation, or review—may expose areas of uncertainty within the model and the software to which the model refers. These uncertainties or unanswered questions regarding the requirements, design, and/or implementation can then be handled appropriately.

The primary expression element of a model is an *entity*. An entity may represent concrete artifacts (for example, processors, sensors, or robots) or abstract artifacts (for example, software modules or communication protocols). Model entities are connected to other entities using relations (in other words, lines or textual operators on target entities). Expression of model entities may be accomplished using textual or graphical modeling languages; both modeling language types connect model entities through specific language constructs. The meaning of an entity may be represented by its shape, textual attributes, or both. Generally, textual information adheres to language-specific syntactic structure. The precise meanings related to the modeling of context, structure, or behavior using these entities and relations is dependent on the modeling language used, the design rigor applied to the modeling effort, the specific view being constructed, and the entity to which the specific notation element may be attached. Multiple views of the model may be required to capture the needed semantics of the software.

When using models supported with automation, models may be checked for completeness and consistency. The usefulness of these checks depends greatly on the level of semantic and syntactic rigor applied to the modeling effort in addition to explicit tool support. Correctness is typically checked through simulation and/or review.

1.3 Syntax, Semantics, and Pragmatics

[2 c2s2.2.2p6; 3 c8s0, c8s5]

Models can be surprisingly deceptive. The fact that a model is an abstraction with missing information can lead one into a false sense of completely understanding the software from a single model. A complete model (“complete” being relative to the modeling effort) may be a union of multiple sub-models and any special function models. So, examination and decision-making relative to a single model within this collection of sub-models may be problematic.

Understanding the precise meanings of modeling constructs can also be difficult. Modeling languages are defined by syntactic and semantic rules. For textual languages, syntax is defined using a notation grammar that defines valid language constructs (for example, Backus-Naur Form (BNF)). For graphical languages, syntax is defined using graphical models called meta-models. As with BNF, meta-models define the valid syntactical constructs of a graphical modeling language; the meta-model defines how these constructs can be composed to produce valid models.

Semantics for modeling languages specify the meaning attached to the entities and relations captured within the model. For example, a simple diagram of two boxes connected by a line is open to a variety of interpretations. Knowing that the diagram on which the boxes are placed and connected is an object diagram or an activity diagram can assist in the interpretation of this model.

As a practical matter, there is usually a good understanding of the semantics of a specific software model due to the modeling language selected, how that modeling language is used to express entities and relations within that model, the experience base of the modeler(s), and the context within which the modeling has been undertaken and so represented. Meaning is communicated through the model even in the presence of incomplete information through abstraction; pragmatics explains how meaning is embodied in the model and its context and communicated effectively to other software engineers.

There are still instances, however, where caution is needed regarding modeling and semantics. For example, any model parts imported from another model or library must be examined for semantic assumptions that conflict in the new modeling environment; this may not be obvious. Check the documented model assumptions. While modeling syntax may be identical, the model may mean something quite different in the new environment, which is a different context. Also, consider that as software matures and changes are made, semantic discord can be introduced. With many software engineers working on a model part over time coupled with tool updates and perhaps new requirements, there are opportunities for the unchanged portions of the model to represent something different from the original author's intent and initial model context.

1.4 Pre-conditions, Post-conditions, and Invariants

[2 c4s4; 4 c10s4p2, c10s5p2p4]

When modeling functions or methods, the software engineer typically starts with a set of assumptions about the state of the software prior to, during, and after the function or method executes. These assumptions are essential to the correct operation of the function or method and are grouped, for discussion, as a set of pre-conditions, post-conditions, and invariants.

- *Pre-conditions* – a set of conditions that must be satisfied prior to execution of the function or method. If these pre-conditions do not hold prior to execution of the function or method, the function or method may produce erroneous results.
- *Post-conditions* – a set of conditions that is guaranteed to be true after the function or method has executed successfully. Typically, the post-conditions represent how the state of the software has changed, how parameters passed to the function or method have changed, or how the return value has been affected.
- *Invariants* – a set of conditions within the operational environment that persist (in other words, do not change) during execution of the function or method. These invariants are relevant and necessary to the software and the correct operation of the function or method.

2 Types of Models

A typical model consists of an aggregation of sub-models. Each sub-model is a partial description and is created for a specific purpose; it may be comprised of one or more diagrams. The collection of sub-models may employ multiple modeling languages or a single modeling language. The Unified Modeling Language (UML) recognizes a rich collection of modeling diagrams. Use of these diagrams, along with the modeling language constructs, brings about three broad model types commonly used: information models, behavioral models, and structure models.

2.1 Information Modeling

[1 c7s2.2; 3 c8s3]

Information models provide a central focus on data and information. An information model is an abstract representation that identifies and defines a set of concepts, properties, relations, and constraints on data entities. The semantic or conceptual information model is often used to provide some formalism and context to the software being modeled as viewed from the problem perspective, without concern for how this model is actually mapped to the implementation of the software. The semantic or conceptual information model is an abstraction and, as such, includes only the concepts, properties, relations, and constraints needed to conceptualize the real- world view of the information. Subsequent transformations of the semantic or conceptual information model lead to the elaboration of logical and then physical data models as implemented in the software.

2.2 Behavioral Modeling

[1 c7s2.1, c7s2.3, c7s2.4; 2 c9s2; 3 c8s2, c8s4.3]

Behavioral models identify and define the functions of the software being modeled. Behavior models generally take three basic forms: state machines, control-flow models, and data-flow models. State machines provide a model of the software as a collection of defined states, events, and transitions. The software transitions from one state to the next by way of a guarded or unguarded triggering event that occurs in the modeled environment. Control-flow models depict how a sequence of events causes processes to be activated or deactivated. Data-flow behavior is typified as a sequence of steps where data moves through processes toward data stores or data sinks.

2.3 Structure Modeling

[1 c7s2.5, c7s3.1, c7s3.2; 3 c8s4; 4 c4]

Structure models illustrate the physical or logical composition of software from its various component parts. Structure modeling establishes the defined boundary between the software being implemented or modeled and the environment in which it is to operate. Some common structural constructs used in structure modeling are composition, decomposition, generalization, and specialization of entities; identification of relevant relations and cardinality between entities; and the definition of process or functional interfaces. Typical structure diagrams provided by the UML for structure modeling, for example, include class, component, object, deployment, and packaging diagrams.

3 Analysis of Models

The development of models affords the software engineer an opportunity to study and understand the structure, function, and assembly considerations associated with software. Analysis of constructed models is needed to ensure that these models are complete, consistent, and correct enough to serve their intended purpose for the stakeholders.

The sections that follow briefly describe the analysis techniques generally used with software models to ensure that the software engineer and other relevant stakeholders gain appropriate value from the development and use of models.

3.1 Analyzing for Completeness

[3 c6s1.1p7, c7s3; 5 pp8–11]

In order to have software that fully meets the needs of the stakeholders, completeness is critical—from the requirements capture process to code implementation. Completeness is the degree to which all of the specified requirements have been implemented and verified. Models may be checked for completeness by a modeling tool that uses techniques such as structural analysis and state-space reachability analysis (which ensure that all paths in the state models are reached by some set of correct inputs); models may also be checked for completeness manually by using inspections or other review techniques. Errors and warnings generated by these analysis tools and found by inspection or review indicate probable needed corrective actions to ensure completeness of the models.

3.2 Analyzing for Consistency

[3 c6s1.1p7, c7s3, c20s1; 5 pp8–11]

Consistency is the degree to which models contain no conflicting requirements, assertions, constraints, functions, or component descriptions. Typically, consistency checking is accomplished with the modeling tool using an automated analysis function; models may also be checked for consistency manually using inspections or other review techniques. As with completeness, errors and warnings generated by these analysis tools and found by inspection or review indicate the need for corrective action.

3.3 Analyzing for Correctness

[5 pp8–11]

Correctness is the degree to which a model satisfies its requirements and design specifications, is free of defects, and ultimately meets the stakeholders' needs. To analyze a model for correctness, one analyzes it—either manually (for example, through the use of inspections or other review techniques) or automatically (using the modeling tool)—searching for possible defects (for example, syntax, function, or data errors) and then removing or repairing confirmed defects before the software is released for use.

3.4 Traceability

[3 c7s3.1p3, c7s4.2pp3–6]

Developing software typically involves the use, creation, and modification of many work products such as, for example, planning documents, process specifications, requirements specifications, diagrams, designs and pseudo-code, handwritten and tool-generated code, manual and automated test cases and reports, and files and data. These work products may be related through various dependency relationships (for example, uses, implements, tests). As software is being developed, managed, maintained, or extended, there is a need to map and control these traceability relationships to maintain requirements consistency with the overall software end-item(s) and work products. Use of traceability typically improves the management of software work products and software process quality. Traceability also enables change analysis once the software is developed and released, since relationships to software work products can easily be traversed to assess change impact. Modeling tools typically provide some automated or manual means to specify and manage traceability links between requirements, design, code, and/or test entities as may be represented in the models and to other software work products.

3.5 Interaction Analysis

[2 c10, c11; 3 c16s1.1, c16s5; 4 c5]

Interaction analysis focuses on the communications or control flow relations between entities used to accomplish a specific task or function within the software model. This analysis examines the dynamic behavior of the interactions between different portions of the software model, including other software layers (such as the operating system, middleware, and applications). It may also be important for some software applications to examine interactions between the computer software application and the user interface software. Some software modeling environments provide simulation facilities to study aspects of the dynamic behavior of modeled systems. Stepping through the simulation provides an analysis option for the software engineer to review the interaction design and verify that the different parts of the software work together to provide the intended functions.

4 Software Engineering Methods

Software engineering methods provide an organized and systematic approach to developing software for a target computer. There are numerous methods from which to choose and it is important for the software engineer to choose an appropriate method or methods for the software development task at hand; this choice can have a dramatic effect on the success of the software project. Use of these software engineering methods coupled with people of the right skill set and tools enables the software engineers to visualize the details of the software and ultimately transform the representation into a working set of code and data.

Selected software engineering methods are discussed below. The topic areas are organized into discussions of Heuristic Methods, Formal Methods, Prototyping Methods, and Agile Methods.

4.1 Heuristic Methods

[1 c13, c15, c16; 3 c8s5, c14, c17s3]

Heuristic methods are those experience-based software engineering methods that have been and are fairly widely practiced in the software industry. This topic area contains three broad discussion categories: structured analysis and design methods, data modeling methods, and object-oriented analysis and design methods.

- *Structured Analysis and Design Methods* – The software model is developed primarily from a functional or behavioral viewpoint, starting from a high-level view of the software (including data and control elements) and then progressively decomposing or refining the model components through increasingly detailed designs. The detailed design eventually converges to very specific details or specifications of the software that must be built (in other words, coded) and verified.
- *Data Modeling Methods* – The data model is constructed from the viewpoint of the data or information used. Data tables and relationships define the data models. This data modeling method is used primarily for defining and analyzing data requirements supporting database designs or data repositories typically found in business software, where data is actively managed as a business systems resource or asset.
- *Object-Oriented Analysis and Design Methods* – The object-oriented model is represented as a collection of objects that encapsulate data and relationships and interact with other objects through methods. Objects may be real-world items or virtual items. The software model is constructed using diagrams to constitute selected views of the software. Progressive refinement of the software models lead to a detailed design. The detailed design is then either evolved through successive iteration or transformed (using some mechanism) into the implementation view of the model,

where the code and packaging approach for eventual software product release and deployment is expressed.

4.2 Formal Methods

[1 Cc18; 3 c10; 5 pp8–24]

Formal methods are software engineering methods used to specify, develop, and verify the software through application of a rigorous mathematically based notation and language. Through use of the specification language, the software model can be checked for consistency (in other words, lack of ambiguity), completeness, and correctness in a systematic and automated or semi-automated fashion.

This section addresses specification languages, program refinement and derivation, formal verification, and logical inference.

- *Specification Languages* – Specification languages provide the mathematical basis for a formal method; specification languages are formal, higher-level computer languages (in other words, not a classic 3rd Generation Language (3GL) programming language) used during the software specification, requirements analysis, and/or design phases of the software development project to describe specific input/output behavior. Specification languages are not directly executable languages; they are typically comprised of a notation and syntax, semantics for use of the notation, and a set of allowed relations for objects.

- *Program Refinement and Derivation* – Program refinement is the process of creating a lower-level (or more detailed) specification using a series of transformations. It is through successive transformations that the software engineer derives an executable representation of a program. Specifications may be refined, adding details until the model can be formulated in a 3GL programming language or in an executable portion of the chosen specification language. This specification refinement is made possible by defining specifications with precise semantic properties; the specifications must set out not only the relationships between entities but also the exact run-time meanings of those relationships and operations.

- *Formal Verification* – Model checking is a formal verification method; it typically involves performing a state-space exploration or reachability analysis to demonstrate that the represented software design has or preserves certain model properties of interest. An example of model checking is an analysis that verifies correct program behavior under all possible interleaving of event or message arrivals. The use of formal verification requires a rigorously specified model of the software and its operational environment; this model often takes the form of a finite state machine or other formally defined automaton.

- *Logical Inference* – Logical inference is a method of designing software that involves specifying pre-conditions and post-conditions around each significant block of the design, and—using mathematical logic—developing the proof that those pre-conditions and post-conditions must hold under all inputs. This provides a way for the software engineer to predict software behavior without having to execute the software. Some Integrated Development Environments (IDEs) include ways to represent these proofs along with the design or code.

4.3 Prototyping Methods

[1 c12s2; 3 c17s4; 6 c7s3p5]

Software prototyping is an activity that generally creates incomplete or minimally functional versions of a software application, usually for trying out specific new features, soliciting feedback on requirements or user interfaces, further exploring requirements, design, or implementation options, and/or gaining some other useful insight into the software. The software engineer selects a prototyping method to understand the least understood aspects or components of the software first; this approach is in contrast with other development methods which usually begin development with the most understood portions first. Typically, the prototyped product does not become the final software product without extensive development rework or refactoring.

This section discusses prototyping styles, targets, and evaluation techniques in brief.

- *Prototyping Style* – This addresses the various approaches to developing prototypes. Prototypes can be developed as throwaway code or paper products, as an evolution of a working design, or as an executable specification. Different prototyping life-cycle processes are typically used for each style. The style chosen is based on the type of results the project needs, the quality of the results needed, and the urgency of the results.

- *Prototyping Target* – The target of the prototype activity is the specific product being served by the prototyping effort. Examples of prototyping targets include a requirements specification, an architectural design element or component, an algorithm, or a human-machine user interface.

- *Prototyping Evaluation Techniques* – A prototype may be used or evaluated in a number of ways by the software engineer or other project stakeholders, driven primarily by the underlying reasons that led to prototype development in the first place. Prototypes may be evaluated or tested against the actual implemented software or against a target set of requirements (for example, a requirements prototype); the prototype may also serve as a model for a future software development effort (for example, as in a user interface specification).

4.4 Agile Methods

[3 c17s1, c17s2, c17s3; 6 c7s3p7; 7 c6, App. A]

Agile methods were born in the 1990s from the need to reduce the apparent large overhead associated with heavyweight, plan-based development methods used in large-scale software-development projects. Agile methods are considered lightweight methods in that they are characterized by short, iterative development cycles, self-organizing teams, simpler designs, code refactoring, test-driven development, frequent customer involvement, and an emphasis on creating a demonstrative working product with each development cycle.

Many agile methods are available in the literature; some of the more popular approaches, which are discussed here in brief, include pair programming, Rapid

Application Development (RAD), eXtreme Programming (XP), scrum, and Feature-Driven Development (FDD).

- *Pair Programming* – Software engineers work in pairs, sitting at one workstation, to design and program the software. The pair programming process promotes collective ownership of the software, informal code review, and continuous code refactoring.
- *RAD* – Rapid software development methods are used primarily in data-intensive, business-systems application development. The RAD method is enabled with special-purpose database development tools used by software engineers to quickly develop, test, and deploy new or modified business applications.
- *XP* – This approach uses stories or scenarios for requirements, develops tests first, has direct customer involvement on the team (typically defining acceptance tests), uses pair programming, and provides for continuous code refactoring and integration. Stories are decomposed into tasks, prioritized, estimated, developed, and tested. Each increment of software is tested with automated and manual tests; an increment may be released every couple of weeks or so.
- *Scrum* – This agile approach is more project-management friendly than the others. The scrum master manages the activities within the project increment; each increment is called a sprint and lasts about 30 days. A Product Backlog Item (PBI) list is developed from which tasks are identified, defined, prioritized, and estimated. A working version of the software is tested and released in each increment. Daily scrum meetings ensure work is managed to plan.
- *FDD* – This is a model-driven, short, iterative software development approach using a five-phase process: (1) develop a product model to scope the breadth of the domain, (2) create the list of needs or features, (3) build the feature development plan, (4) develop designs for iteration-specific features, and (5) code, test, and then integrate the features. FDD is similar to an incremental software development approach; it is also similar to XP, except that code ownership is assigned to individuals rather than the team. FDD emphasizes an overall architectural approach to the software, which promotes building the feature correctly the first time rather than emphasizing continual refactoring.

There are many more variations of agile methods in the literature and in practice. Note that there will always be a place for heavyweight, plan-based software engineering methods as well as places where agile methods shine. There are new methods arising from combinations of agile and plan-based methods where practitioners are defining new methods that balance the features needed in both heavyweight and lightweight methods based primarily on prevailing organizational business needs. These business needs, as typically represented by some of the project stakeholders, should and do drive the choice in using one software engineering method over another or in constructing a new method from the best features of a combination of software engineering methods.

RECOMMENDED REFERENCES FOR SOFTWARE ENGINEERING MODELS AND METHODS

[1] D. Budgen, *Software Design*, 2nd ed., New York: Addison-Wesley, 2003.

[2] S.J. Mellor and M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed., Boston: Addison-Wesley, 2002.

[3] I. Sommerville, *Software Engineering*, 8th ed., New York: Addison-Wesley, 2006.

[4] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, 1st ed., Reading, MA: Addison-Wesley, 1999.

[5] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, IEEE CS Press, vol. 23, no. 3, pp. 8, 10 – 23, 1990.

[6] J.G. Brookshear, *Computer Science: An Overview*, 10th ed., Boston: Addison-Wesley, 2008.

[7] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Boston: Addison-Wesley, 2003.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Boehm	Brookshear	Budgen	Mellor	Page-Jones	Sommerville	Wing
1. Modeling							
1.1 Modeling Principles			C2S2, C5S1, C5S2	C2S2		C8S0	
1.2 Properties and Expression of Models			C5S2, C5S3			C6S1.1P7, C7S3P2, C8S0P4	
1.3 Syntax, Semantics, and Pragmatics				C2S2.2.2P6		C8S0, C8S5	
1.4 Pre-conditions, Post-conditions, and Invariants				C4S4	C10S4P2, C10S5P2P4		
2. Types of Models							
2.1 Information Modeling			C7S2.2			C8S3	
2.2 Behavioral Modeling			C7S2.1, C7S2.3, C7S2.4	C9S2		C8S2, C8S4.3	
2.3 Structure Modeling			C7S2.5, C7S3.1, C7S3.2		C4	C8S4	
3. Analysis of Models							
3.1 Analyzing for Completeness						C6S1.1P7, C7S3	PP8-11
3.2 Analyzing for Consistency						C6S1.1P7, C7S3, C20S1	PP8-11
3.3 Analyzing for Correctness							PP8-11
3.4 Traceability						C7S3.1P3, C7S4.2P3 – 6	
3.5 Interaction Analysis				C10, C11	C5	C16S1.1, C16S5	
4. Software Engineering Methods							
4.1 Heuristic Methods			C13, C15, C16			C8S5, C14, C17S3	
4.2 Formal Methods			C18			C10	PP8-24
4.3 Prototyping Methods		C7S3P5	C12S2			C17S4	
4.4 Agile Methods	C6, App. A	C7S3P7				C17S1, C17S2, C17S3	

List of Further Readings

None.