|  |
| --- |
| What is software engineering? |
| A practitioner’s guide |
| Ethan Crawford |

Table of Contents

[Introduction 3](#_Toc168731692)

[Process Models 4](#_Toc168731693)

[CMMI 4](#_Toc168731694)

[Software Development Lifecycles 5](#_Toc168731695)

[Waterfall 5](#_Toc168731696)

[Spiral 6](#_Toc168731697)

[Unified Process 7](#_Toc168731698)

[Agile 8](#_Toc168731699)

[Scrum 8](#_Toc168731700)

[Extreme Programming 8](#_Toc168731701)

[Requirements Engineering 8](#_Toc168731702)

[Flow-Oriented Models 9](#_Toc168731703)

[Data-flow diagrams 9](#_Toc168731704)

[Scenario-based Models 9](#_Toc168731705)

[Use-cases 10](#_Toc168731706)

[User stories 10](#_Toc168731707)

[UML Activity Diagrams 10](#_Toc168731708)

[UML swim-lane diagrams 11](#_Toc168731709)

[Class-based 11](#_Toc168731710)

[UML class diagrams 12](#_Toc168731711)

[Class-Responsibility-Collaborator cards 12](#_Toc168731712)

[Behavioral 12](#_Toc168731713)

[UML Sequence Diagrams 13](#_Toc168731714)

[Design Analysis 13](#_Toc168731715)

[Testing and Quality Assurance 14](#_Toc168731716)

[Unit Testing 14](#_Toc168731717)

[Integration Testing 15](#_Toc168731718)

[Validation Testing 15](#_Toc168731719)

[System Testing 16](#_Toc168731720)

[White-box Testing 16](#_Toc168731721)

[Black-box Testing 17](#_Toc168731722)

[Project Management 18](#_Toc168731723)

[Summary 19](#_Toc168731724)

# Introduction

Software engineering is the set of practices and processes that enable a development team to build a useful software product in a timely fashion at a predictable cost. In this definition, a “useful product” is one that meets the main requirements of most of its customers, a “timely fashion” is soon enough to make money on the product, and “predictable cost” is a known amount of resources required to complete the project. Software engineering becomes more significant as the complexity of the project rises; each activity has a built-in associated cost, and becomes useful when the benefits outweigh the costs.

While software engineering encompasses the code-related disciplines of design and implementation, computer science is but one of the fields within it. Human resources, process improvement, project management, systems engineering, hardware, mathematics, law, social sciences, arts and humanities all play a role within software engineering. If software development is the star actor on a stage, software engineering is the “agents, the catering crew, the lawyers, the janitors, the accountants, the producer and the director” [Crawford07a]. To address all the various aspects of software engineering, most large product teams consist of systems analysts, project managers, architects, developers, testers, marketers, salespeople, usability engineers, designers, support engineers, maintenance programmers, and managers of various types.

In this paper, we present a few key areas of software engineering. At a minimum, one technique from each of the areas is likely to be required to successfully complete a complex project. For each area, we address the core issues that created a need for the area and discuss a limited set of techniques used to implement the area. The key areas are process models, software development life cycles, requirements engineering (including elicitation and requirements analysis), testing and quality assurance, and project management. In conjunction with design and implementation, these areas cover most of the major aspects of software engineering.

Each key area represents a major part of the entire process of software engineering. Process models provide guidance on how a team should approach their daily tasks. Software development lifecycles lend a skeleton to the overall development effort, breaking it up into phases and helping the entire team shift their collective focus. Requirements engineering provides a large number of techniques for gathering and analyzing information, used to determine what should be built, and why. Design engineering transforms requirements into implementable designs. During and after design and implementation, testing and quality assurance provides many mechanisms for measuring the quality of what was built. Over the entire process, project management provides multiple ways to manage people, track progress, and facilitate communication among multiple stakeholders.

Over the years, many of the central concerns of software engineering have evolved. When computers were slow and memory was expensive, RAM footprint and performance were extremely important. As machines improved, the focus shifted to integrating complex modules together and making them usable. As networked machines grew in prominence, rapid development and interoperability became extremely important. As the concerns evolved, so did the tools and techniques used to address them. This paper describes a small selection of techniques from each of these eras that have shown themselves valuable.

# Process Models

No software-producing organization exists in a vacuum. Products are usually created for monetary reasons, and other organizations compete for the same customers. Not all organizations are created equal, however. An organization’s ability to produce a high-quality product is directly influenced by the health and maturity of its internal processes. A well-financed customer in need of a specific software solution requires a way of comparing multiple organizations, in order to determine which to patronize. Process models facilitate this goal.

A process model is an organized, comprehensive collection of best practices for improving processes used in software development and maintenance. It is a meta-model used to describe the health and maturity of an individual process. Its purpose is to provide customers wishing to employ the services of a software vendor with the ability to measure the likelihood that the vendor will successfully deliver a product on time and within budget, and to provide vendors with the ability to improve the overall maturity of their processes in order to successfully attract customers. The main process model in use today is the Capability Maturity Model Integrated, CMMI.

## CMMI

CMMI was created by the Software Engineering Institute at Carnegie Mellon University. Originally, the model to was intended to provide “guidelines and procedures for assessing the ability of potential DoD contractors to develop software in accordance with modern engineering methods” [Humphrey87]. CMMI continues to serve this purpose, but is also actively used by organizations to improve their *process maturity*, the efficacy, repeatability, and measurability of a process. [CMMI-DEV, p. 2].

CMMI is useful for many reasons. It incorporates large amounts of industry-tested concepts, making it useful as a comparison tool for identifying areas of improvement within an organization. The detailed nature of CMMI process model descriptions give key stakeholders a common vocabulary for discussing a product vision and specifics. Process model descriptions strive to be comprehensive, enabling them to provide generic templates that can be used to construct a new process or customize an existing one.

CMMI defines twenty-two process areas, “cluster[s] of related processes in an area that, when implemented collectively, satisfy a set of goals considered important for making improvement in that area” [CMMI-DEV, p. 28]. Taken together, these process areas cover all aspects of generic product development, including communication, planning, modeling, construction, and deployment [Pressman05, p. 24]. In addition to product development, CMMI includes several process areas that focus exclusively on process improvement through continuous iteration, measurement, and analysis. These “processes to improve processes” are used to gauge and improve an organization’s overall process maturity.

CMMI defines both process-area-specific and generic goals. Goals are used to quantify the specific items to be accomplished in the process area. Whereas specific goals enumerate the elements of a process area, generic goals provide a way to measure the overall maturity of that process area; they are the “how,” not the “what” [Shrum99]. CMMI provides two different ways of measuring process maturity, “continuous” and “staged”, and provides a number rating from 0-5 or 1-5 [CMMI-DEV, p. 45]. The separate representations are intended to meet the needs of two types of organizations: those desiring improvement in a specific area (continuous), and those desiring a greater level of comprehensive process maturity (staged). In either form, the ratings describe the extent to which the work artifacts and processes of a process area are reviewed, standardized, customized, measured, and continuously improved.

Any model for improving processes needs a way of appraising process maturity. The Standard CMMI Appraisal Method for Process Improvement (SCAMPI) is an assessment process designed to “provide benchmark quality ratings relative to CMMI models” [SEI07]. The end-result of an appraisal is a maturity rating from 1-5, which can be used as a marketing point for the company. Appraisals are performed by small teams whose mission is to search for objective evidence of implementation of the practices of a process area that satisfy a specific capability level.

While many software development organizations will find that not every aspect of CMMI matches their internal culture, most will find that is a useful tool for driving change in targeted areas. CMMI makes the compelling case that process matters, and that monitoring and measuring processes will positively impact the capacity of an organization to deliver products in a structured, predictable, affordable manner.

# Software Development Lifecycles

A software development lifecycle is a process that directs the evolution of a software product from idea to shipping product. The purpose of an SDLC is to provide a structural framework to support a product development effort. The choice of SDLC is perhaps the most important one that a team can make, because it influences many other aspects of the project throughout the entire lifecycle, including requirements analysis techniques, development strategies, test planning, and product release assumptions.

When selecting an SDLC, a number of basic questions must be considered. What product needs to be built? How likely are the requirements to change, and how often? How frequently does the customer require feedback? What effect will changing requirements have on the business model, and on the likelihood of completing the product? Who is the customer, what is their personality and working style, and how much connection would they like to have with the product while it is being developed? What is the system in which the product will live? What other stakeholders besides the paying customer exist for the product? Each of these questions may affect the choice of SDLC and should be considered carefully.

Because the end goal of all lifecycles is high-quality, stable software, all SDLCs share common elements. Every SDLC contains one or more phases for gathering and analyzing requirements, designing, building, and testing software, and releasing it to customers for feedback. Besides those basic commonalities, there are many differences between them. This paper summarizes the waterfall model (the classic SDLC, often used as a teaching example), the spiral model and the Unified Process (both iterative and incremental), and Extreme Programming and Scrum, two main practices of the Agile methodology.

## Waterfall

The waterfall model, the “bubble sort” of the SDLC universe, is a classic software development lifecycle (SDLC) model. It was first proposed in 1970 by W. W. Royce, [Wiki07b]. It is linear and sequential, and one phase of the process ends before the next begins. The work output of the previous phase feeds the next phase, the way water flows down ledges in a waterfall (see Figure 1). The five phases of the waterfall are Requirements, Design, Implementation, Verification, and Maintenance. The Requirements phase, in which all the features to be built are identified, always comes first, and must be complete before the Design phase can begin. Similarly, the design for the entire project must be complete before any Implementation can start, and so on.

The waterfall model is an example of a document-heavy, Big Design Up Front (BDUF) philosophy. By investing in up-front design, waterfall practitioners strive to avoid writing unnecessary code and introducing unnecessary bugs. Because only one phase is active at a time, detailed documentation must be generated to keep track of relevant information in the large gap between requirements gathering and implementation.

The waterfall model has both significant strengths and weaknesses. On the plus side, it is simple to implement and easy to understand. Although it requires a significant investment in documentation and design, that level of document output may be appropriate for products with a significant lifespan. The biggest weaknesses of the waterfall model lie are its lack of ability to gracefully handle changing requirements combined with the relatively long periods of time that pass between customer feedback opportunities. Additionally, a “pure” waterfall implementation suffers from wasted resources; because people usually specialize in a single software engineering discipline, a portion of the team is likely to be idle while another team is executing one of the phases. Waterfall is simple, predictable, easy to teach, and, frequently, the least efficient software development process of all.

## Spiral

The spiral model is a cyclical, incremental and evolutionary SDLC, first proposed by Barry Boehm in 1988. The spiral model originated with the observation that the dimensions of a complete system are not known at the inception, and that some products fail due to unforeseen risks [Boehm88]. To address this, the spiral model provides for a multi-stage, iterative approach to product development. Each ‘revolution of the spiral’ incorporates requirements analysis, design, risk analysis, and development of a stable, functional prototype (see figure 2). In the first revolution, the requirements and prototyping activities usually result in a system containing a subset of the final feature set. Later iterations incorporate customer feedback gathered at the end of the previous revolution.

Prototyping is an important spiral model concept. The first prototype contains elements of the final product, but is lacking the scope and detail of the finished product. The second prototype, derived from the first, involves a risk analysis on various aspects of the first, followed by an expansion of the requirement set and the development of a new set of stable features. The final “prototype”, at whatever iteration, becomes the complete system.

The spiral model is driven by risk. The structure of the spiral enables an iterative investment in the work artifacts, which, in turns, enables an environment in which the customer can abort a project. The customer might choose to do this for multiple reasons, among them the projected cost of completing the system, a lack of supporting infrastructure or technologies, or a change in the market in which the product was to compete. This allows the customer to engage in “hedging”, i.e., investing a moderate amount of resources now to avoid losing a much larger amount later. At the end of each prototype, the customer can either choose to continue (with greater confidence that the product under development is the right one) or abort (with fewer resources lost than if the wrong product had been fully developed).

The spiral model has a large number of strengths. Because the scope of future work is better understood after each revolution of the spiral, it becomes progressively easier to create accurate work item estimates and predict the project end-date. Iterative development combined with multiple stable prototypes provides an effective mechanism for dealing with changing customer requirements [WIKI07g]. It also enables future design phases by providing working implementations on which developers can build.

## Unified Process

The Unified Process is a use-case driven, architecture-centric, document-heavy, big-design-up-front, iterative and incremental SDLC [JAC99]. First developed in the 80s and 90s by the Rational Software corporation, the RUP was created to support Unified Modeling Language, a set of symbols used to describe object-oriented systems. The Unified Process was designed to address a series of problems: poor requirements management, unclear communication, inflexible architectures, “over-engineering”, and lack of risk analysis and testing [WIKI07d].

The Unified Process development lifecycle is split up into five phases: Inception, Elaboration, Construction, Transition, and Production. Each phase is broken up into one or more iterations, an interval of time in which a set of work artifacts are created. In each iteration, the work artifacts can be executable code or different types of documents. Unlike other SDLCs (such as the waterfall model), even the first iteration can contain executable prototypes [Pressman05, p. 64]. All development activities, including coding, happen simultaneously in every iteration and in all four phases. However, the relative quantity each activity varies, depending on the phase. The first iterations are heavier in project planning and design, and later phases are heavier in construction. (See Figure 3 for estimated activities in each iteration.) This addresses the problem of wasting resources due to portions of the team going idle in different phases.

Managing change is an important part of the UP philosophy. The end of each iteration includes time used to receive customer feedback. Much like the spiral model, the output of each UP iteration is a working prototype, and this working code can be used to provide demonstrations to customers, who can then provide further requirements for future iterations.

The primary UP mechanism for requirements management is “use-cases”. A use-case is a sequence of events performed by an “actor” (user or other code consumer) that “lead to the system doing something useful” [WIKI07f]. Use-cases are actor-centric, not system-centric, and exist in a many-to-one or one-to-many relationship with requirements. The shift from requirements to use-cases enables the process of incorporating frequent customer feedback into subsequent iterations and puts developers into the shoes of the customer.

One of the primary focuses of the Unified Process is an ‘architecture-centric’ approach, the blurring of the line between requirements management, design, and implementation. Most of the modeling techniques used for requirements analysis can be extended, with very little modification, into actual software designs. Use-cases provide a grammar to construct an object model, from which class diagrams, communication and responsibility cards, and many other design descriptions are derived. This process continues in every iteration of every phase, until the entire product is complete in the Initial Operational Capability Milestone. This is a checkpoint at which all the use-cases required for the release have been implemented [WIKI07d].

## Agile

Agile software development is a collection of document-light, iterative and incremental software development techniques. Developed in the 90s as a response to document-heavy, big-design-up-front SDLCs, Agile development primarily emphasizes the role of the developer and attempts to optimize its processes in ways that more closely mirror the way developers actually write code. Agile development produces very little documentation; instead, it focuses on working software as the primary measure of progress [WIKI07h]. In contrast to most other SDLCs, agile iteration cycles, known as “sprints”, are extremely short; most last between one and four weeks. Each sprint aims to add a “feature” (a working unit of code) to the software product. Customer feedback is taken very frequently, with much of the feedback ideally provided in the form of acceptance tests, used to sign off on the final version.

One tenant of The Agile Manifesto emphasizes the value of “responding to change over following a plan” [AGILE01]. The principles of the on-site customer, code restructuring (“refactoring”), and face-to-face communication facilitate this tenant, as do both Scrum and Extreme Programming.

### Scrum

Scrum is a project management technique used to coordinate and organize small development teams. It exists to address the problems of inter-team communication, accountability, and agility [WIKI07i]. Prior to the beginning of the development effort, project planners create a feature list, known as a “product backlog”. Each feature is broken up into a “sprint” backlog, and consists of enough work to last one to three weeks. As the development effort begins, short, daily meetings (“scrum meetings”) are held.

In the meetings, everyone answers 3 questions: “What have you done since the last meeting?”, “What is blocking you?”,and “What do you plan to do next?”. A meeting facilitator (the “scrum master”) is responsible for keeping the meetings short, maintaining work artifacts to track progress, and getting answers to blocking issues. Follow-up meetings can happen immediately after scrum, with the intent of passing on detailed information that affects a person’s work for the day.

### Extreme Programming

Extreme Programming (XP) is an Agile software development methodology developed by Kent Beck, Ron Jeffries, and Ward Cunningham in 1996 [WIKI07j]. The purpose of the practice is to facilitate developer communication, design simplicity, module testability, and incremental change within a codebase. It consists of “pair-programming” (two developers working on the same module at the same time), “test-driven development” (creating unit tests before creating the design), “refactoring” (incorporating code review feedback to change the organization of the code), continuous feedback from customers and other programmers, and an emphasis on good up-front designs that meet today’s needs.

# Requirements Engineering

Requirements engineering is the art of figuring out what software product should be built, and what components should be used to build it. The two parts of requirements engineering are elicitation and analysis. Elicitation is gathering and capturing stakeholder input, and analysis is the process of transforming raw requirements into all the forms required to enter the design phase. This section focuses mainly on analysis, although some of the techniques are applicable to elicitation. Analysis is conducted using models, which are frequently diagrammatic in nature.

Why use models for requirements analysis? There are many reasons. Refining raw requirements into models frequently exposes previously-unknown user scenarios and system components. Models help explain complicated systems to interested stakeholders. They “enumerate required functionality, help organize that functionality into sensible groups, differentiate implementation details from functional requirements, show the flow of data through a system, capture and categorize user experiences, segregate external from internal entities, and establish responsibilities and relationships between objects” [Crawford07b]. In short, they act as the bridge between requirements and design, providing the analysis required to make the transition smooth.

There are four main approaches to modeling requirements: flow-oriented, class-based, scenario-based, and behavioral. Each approach has its own emphasis and set of strengths, and the techniques are used together to communicate all relevant dimensions. The following section describes each of the four approaches and some of the techniques that demonstrate the approach.

## Flow-Oriented Models

Flow-oriented models exist to show the flow of data within a system. They take an “input-process-output” perspective [Pressman05, p. 194], modeling raw data, transformations, and output vectors. The primary technique in this category is the data flow diagram.

### Data-flow diagrams

Data flow diagrams are system-centric, hierarchically-ordered diagram collections. They utilize a simple library of shapes and arrows to represent “entities” (data inputs or outputs), “data stores” (persistent data storage), and “systems” (functional software modules or products) [WIKI07a]. The diagram hierarchy is top-down; level 0 or “context-level” diagrams establish system-level inputs and outputs. (See figure FOO for an example of a level 0 DFD.) Subsequent diagrams are more detailed in scope, but preserve the input/output context. A complete diagram hierarchy permits as many levels of detail as required to model the system down to the individual calls that transform data from one state to another. The structure is driven by system services, i.e., functional requirements, not user scenarios. Users are merely input and output vectors, and specific user characteristics determine individual data attributes.

DFDs model only data flow and contain no information regarding implementation, class hierarchies, or user scenarios. Many modern systems are developed from an object-oriented perspective and place a heavy emphasis on the customer perspective. As such, it is unlikely that this modeling technique alone provides enough information to allow a smooth transition into the design phase. However, DFDs provide a valuable service in making it possible to distill complex transactions into a simple set of diagrams, making the technique ideal as a supplement to other modeling techniques.

## Scenario-based Models

Rather than revolving around system services or architectural structure, scenario-based models revolve around the user. The basis of each of these techniques involves observing real or imaginary users interact with the system and using those observations to derive system requirements, key “actors”, and communication relationships. All of these modeling techniques are driven by the philosophical perspective that the most enduring measure of product success is customer satisfaction [Pressman05, p. 186]. The main techniques used in scenario-based modeling are use-cases and their Agile counterpart, user stories, activity diagrams, and swim-lane diagrams. Most of the cases are modeling techniques used as part of the Unified Process.

### Use-cases

A use-case is a text description of a user scenario, written using a formal, structured grammar, from the perspective of an “actor”, a human or other external entity that interacts with the software system. Use-cases are the backbone of Unified Process requirements analysis activities and play an important part in requirements modeling, design and quality assurance.

A use-case is constructed by first writing down a requirement. The requirement is matched to an actor, and a simple description of each of the functional system steps required to fulfill the requirement is added. A preliminary user interface input vector is chosen, and alternate inputs and actors that might consume the requirement are enumerated. A list of error conditions that can occur at every step are created, and the use-case is given a set of metadata to help project manager track and prioritize the case [Pressman05, pp 190-191]. Figure 5 shows a sample use-case containing these elements.

Use-cases are useful for multiple reasons. Because the language within them is consistent, they can be used to identify possible classes, data attributes, and responsibility/collaboration relationships. The same language enables the formation of activity and sequence diagrams. During testing, they can be used as “test cases”, with the tester performing the role of the primary actor. After testing is complete, they exist as a record for quality assurance, to check the implemented codebase against for compliance with requirements.

### User stories

A user story is an Agile software development technique used to elicit and analyze requirements. It consists of a short written description of an action the user needs to be able to accomplish using the system, one or more captured “conversations” about the story that provide details, and a set of acceptance tests that verify when the user story has been fulfilled.

Unlike a use-case, a user story description is extremely short. This allows Agile teams to provide to customers high-level system descriptions prior to locking down the features needed to accomplish those goals. It also permits teams to spend relatively less time documenting requirements, compared to other techniques. User stories also cover relatively fewer system features and are more disposable [Cohn04]. Frequently, teams will destroy the user stories after the product is complete. This is possible because of the acceptance test suite developed in parallel with the product; the tests document the user stories.

### UML Activity Diagrams

Activity diagrams exist to describe use-cases in graphical form. Essentially, they are flow-charts that model the individual use-case steps taken by an actor and the functional steps performed, in response, by the system. Activity diagrams use standard flow-chart shapes: rounded rectangles for a specific action or state, diamonds for a decision point, and lines for loops or activity flow. See figure 6 for an example of an activity diagram.

Like data flow diagrams, activity diagrams are at their most useful as a collection. One top-level diagram models the entire system, and multiple subsequent activity diagrams model individual use-cases. Unlike data flow diagrams, the activity diagram hierarchy is much flatter than data flow diagrams (only a few levels instead of many). Additionally, the diagram organizational structure is dictated by user scenario, not by system service or functionality.

UML activity diagrams enhance text-based use-cases by making connections, retries, branches, failure paths, and other implied work flow details explicit. They also allow a visual analysis of the work flow, helping to identify logic or workflow errors that may not be as evident in the text steps.

### UML swim-lane diagrams

Swim-lane diagrams are a specific type of activity diagram. Functionally, they are identical to activity diagrams, but the data presentation is slightly different. Each actor involved in the use-case is assigned a vertical column, and the actor’s actions and decisions are moved to that column. Figure 7 shows the same diagram as figure 6, with added swim-lanes for Registered User and User Interface.

The primary purpose of swim-lane diagrams is to disambiguate the relationships and interactions between primary actors in a use-case, placing them in context within the work flow. This adds information to the activity diagram format, which otherwise does not provide a mechanism to accomplish this.

## Class-based

Class-based modeling is a “component-driven” approach, where “component” is roughly defined as a noun associated with a system description. It deals with the structure of components, the operations associated with them, and the relationships between them. and relationships between components, Class-based models go hand-in-hand with use-cases; by describing the objects required to perform the use-case, they provide a mapping between user requirements and the underlying structure of the software. Although the techniques of class-based modeling borrow heavily from the concepts of object-oriented programming, it is important to note that class-based models address a wider set of concerns than software designs, and as such, contain a distinct set of semantic differences from design classes. Nevertheless, the similarities outweigh the differences, and class-based models act as a stepping-stone to future architectural designs.

The mapping process begins with a “grammatical parse” of the use-cases [Pressman05, p. 201]. Nouns are categorized into data attributes, data objects (collections of data attributes), collections of data objects, external systems, sensors, devices, people, organizations, internal roles, locations, and events are categorized. Anything in the categorization that is not a data attribute is a potential “analysis class”. Each potential class is reviewed for inclusion in the model using a decision-making process that very closely resembles object-oriented design (e.g., the item contains multiple data attributes and a reusable set of verbs apply to those attributes). After a class is identified and attributes specific to the class are added to it, related verbs are organized into ‘responsibilities’ (private operations on attributes) and ‘collaborations’ (public associations between classes) [Pressman05, p. 207].

Class diagrams and Class-Responsibility-Collaborator (CRC) cards facilitate each step of this process, as well as providing visualization of more complex class relationships. Each one approaches the problem from a slightly different perspective.

### UML class diagrams

UML class diagrams model the attributes and operations of a class. They also represent different types of relationships between classes. Classes are represented as simple rectangles containing simple lists of text terms, with lines and arrows of different types used to indicate relationships. UML class diagrams are used for both requirements analysis and design analysis. In the requirements analysis phase, no private operations are modeled. Additionally, many actors which will never be implemented in code are represented in the model. Figure 8 shows a sample UML class diagram containing attributes, operations, and relationships.

Many different types of relationships can exist between classes. Classes can contain other classes (composition), inherit from other classes (inheritance/generalization), contain many instances of another class (aggregation), or implement a standard interface (realization) [Martin00]. Additionally, classes can have a one-to-one, many-to-one, or many-to-many relationship with each other. UML class diagram notation provides different types of lines and arrows, along with text annotations for the arrows, to represent these relationships.

In object-oriented software development efforts, UML class diagrams are useful for delineating attributes, responsibilities, and structural relationships between classes. They help identify responsibilities inside and outside the system, and inside and outside individual classes. They also provide another dimension of the use-case, because of the structured parse used to create the diagrams.

### Class-Responsibility-Collaborator cards

Class Responsibility-Collaboration cards are a series of real or virtual index cards used to describe classes. Each card consists of a class name, a list of internal responsibilities, and a list of the classes with which the class will communicate or collaborate. This modeling technique provides multiple features. First, it assists in identifying functionality for an individual class. Next, it assists in the process of grouping the functionality according to internal functionality (responsibilities) and external functionality consumed by other classes (collaborations). Classes that interact with each other are called collaborators [Pressman05, pp. 208-210]. See figure 9 for an example of the model-view-controller design patter, rendered on CRC cards.

One of the primary benefits of completing a set of CRC cards is having an easy way to measure the system coupling level - long collaborator lists on multiple CRC cards may be a indication of too-tight coupling. Additionally, CRC cards can be used during testing and quality assurance to check the implementation against the original requirements. Constructing CRC cards also helps to identify missing operations and attributes and to clarify class relationships.

## Behavioral

Behavioral modeling shows the dynamic elements, the “behavior”, of a system. In a behavioral model, specific information exchanges (also known as events) drive a sequence of interaction between the system and an external entity [Pressman05, p. 217]. Although this entity is very frequently a user, it can also be an external system or internal software module. Behavioral modeling blurs the line between analysis and design; it is sometimes impossible to model system state without referencing specific functionality that influences the state.

Behavioral models contain two different state representations, the internal state of the system and the observable state of the UI. Internal system state consists of both the values stored within attributes and a more abstract concept that varies based on the individual functionality of a class. For example, classes responsible for authenticating a user have “not authenticated”, “authenticating”, and “authenticated” states [Pressman05, p. 218].

Behavioral modeling techniques are used as a counterpart to object models and other static models; they are the “when?” to a static model’s “what?”. While multiple behavioral modeling techniques exist, this discussion covers only UML sequence diagrams.

### UML Sequence Diagrams

UML Sequence diagrams model the flow of events between objects over time [Pressman05, p. 220]. Key classes or actors are rendered as labeled rectangular boxes, with vertical dashed lines indicating the lifespan or “lifeline” of the object. Events are shown by horizontal lines and arrows. Vertical rectangles show time spent in processing within a specific module. System states are displayed as comments along the object lifeline. Figure 10 shows a simple sequence diagram with these elements.

UML sequence diagrams are useful for observing the conversation between objects and the corresponding system state. They help to model complex object collaborations, and they provide a visualization of the passage of time along with the estimated relative computational cost within each module.

# Design Analysis

The purpose of design analysis is to transform analysis models into design models. This is an evolutionary process that extends each of the analysis models. Class creation involves parsing analysis classes in UML class models. In this process, actors are removed or converted to design classes, structs are separated from classes, and class relationships are finalized. Architectural design emerges from class-based and flow-oriented models, essentially via a merge of the two. Interface (public method) design emerges naturally from use-cases and their associated actors. Individual component design naturally comes from the architectural design. Structural elements are transformed into procedural descriptions, relying on all the other analysis models for guidance.

After a set of design models have been created, it is desirable that the quality be measureable. Hewlett-Packard proposed the FURPS Quality Factors as a way of measuring the different dimensions of design quality. This acronym stands for Functionality (does the system fulfill its requirements?), Usability (Are the features exposed by the system easily accessible by all users?), Reliability (is the system available when, and for as long as, users need it?), Performance (does the system fulfill the requirements in a short-enough timespan, as defined by the stakeholders?), and Supportability, which includes extensibility, adaptability, and serviceability; all of which represent maintainability.

Implementation, the next phase following design analysis, is deliberately excluded from this overview. It is covered extensively in other courses.

# Testing and Quality Assurance

Testing is the process of exercising a software product with the intent of finding flaws in the product. It is commonly conducted by an independent test group, which helps to avoid potential bias and conflicts of interest caused by programmers testing their own code. Testing (the act of finding bugs) should not be confused with debugging (the act of fixing bugs) – they are separate and distinct activities. Testing is also not a substitute for quality. Assuring quality is a team-wide responsibility, and, programmers, testers and other members of the development team must work together throughout the development lifecycle to uphold the quality level.

Along with professional testers, programmers share most of the testing responsibilities. Ideally, the relationship between the test group and the programming group is an ongoing collaboration throughout the design and implementation phases. Testing (the act of finding bugs in today’s product) is also a separate discipline from that of quality assurance (the act of preventing bugs in tomorrow’s product).

Testing has two primary purposes: verification and validation. Verification tests answer the question: “Did we build the product correctly?” while validation tests answer the question: “Did we build the correct product?” These two questions span multiple dimensions. Tests that verify that the product was built correctly can measure functionality, error-handling, stability, performance, or load-handling. Testing to verify that the correct product was built involves more customer-related testing activities such as alpha and beta tests, as well as a thorough pass to verify adherence to requirements.

Quality assurance, although related to testing, is not the same activity. Testing focuses on keeping the quality of the current product high during design and implementation by finding bugs after they are introduced. Quality assurance is more concerned with defect prevention, i.e., the adoption of processes and techniques that prevent bugs from being introduced in the first place. It is closely tied up with process improvement and ongoing organizational process maturity, in the vein of CMMI.

The testing process tends to follow, or even mirror, the development cycle. As such, it progresses from the small to the large. The first tests to be performed are usually unit tests. Unit tests exercise the functionality of individual modules or classes. Integration tests are then created, along with stubs and drivers, to check for communication and collaboration errors as modules are integrated together. Finally end-to-end, system-level tests are run to prove the system, and validation tests run to measure quality in several dimensions.

All four types of tests can be executed both as white-box tests, code-driven tests that are informed and structured based on product execution flow, and black-box tests, functionality-driven tests that exercise features instead of underlying code.

### Unit Testing

Unit testing is a form of testing that exercises specific execution paths. Its purpose is to detect bugs in individual modules, components, or classes. Unit testing focuses on validating the flow of information within a module, verifying functionality, identifying logical or procedural errors, confirming functional fidelity at array and other data boundaries, and checking for structural errors in data objects. For products created using structured programming methodologies, unit tests should be written to exercise each subroutine. Object-oriented products should provide unit tests for all public methods (private functionality will be implicitly tested). The primary focus is on granularity of scope – each test should exercise a small functional unit.

Some examples of unit tests are regression and, less commonly, smoke tests. Regression tests are suites of units tests, maintained and extended as functionality is added to a module. Every test must pass as a prerequisite to any new features being added to the product. This allows regression tests to act as the first line of defense in the enduring battle to maintain a continuously high level of product quality, even during active development. Smoke tests are less comprehensive. Designed to be run quickly and to exercise the most common scenarios, smoke tests allow a team to rapidly declare a product fit for further testing. They are designed to find high-priority issues (segmentation faults and major functionality breaks), and very little else.

Unit tests are typically the first tests to be created, as they can test incompletely-implemented software modules prior to wiring them together. As modules begin to communicate with each other, integration tests are required.

### Integration Testing

Integration testing is the process of exercising the functionality of multiple modules, interacting with each other. Its purpose is to find bugs when components interact. In structured programming, this is simply modules calling each other’s subroutines. In object-oriented programming, this is the concept of communication and collaboration. Integration testing is heavily dependent on the implementation of “stubs” and “drivers”. Stub code shares an identical interface with a fully-implemented module, but does very little but return, when called. Driver code is simple high-level code that exercises implemented modules without exposing a user interface. Interface testing can be done either incrementally (adding one module at a time and retesting) or “big bang”, adding all implemented modules at once and seeing what happens. For ease of debugging, incremental integration testing is far more common, as it enables the scenario of making a single change and confirming that it works, prior to making other changes.

There are two main types of incremental integration testing techniques: top-down, and bottom-up. Top-down integration testing begins with the implementation of the main driver classes, adding stubs for all dependent modules. The system is implemented, module by module, either depth-first or breath-first, and integration tests are re-run as each stub is replaced with fully-functional code. Bottom-up performs a similar process in reverse, with simple driver code calling dependent modules until the entire system is implemented and tested. In both cases, a full automated test pass is run as each module is added, to flush out integration-related bugs. When a full pass cannot be completed for time reasons, integration smoke tests, targeted tests to find high-priority issues, can be substituted.

### Validation Testing

Validation testing is the process of validating the functional, behavioral, and performance requirements of the implemented system against the requirements specified during requirements engineering. Its purpose is to find previously-hidden product defects prior to deploying the software. As a result, this phase of testing includes extensive customer interaction, and incorporating customer feedback and gauging customer satisfaction is a significant part of the process. All the work artifacts of requirements analysis are leveraged heavily in this type of testing. From a structural programming perspective, this might include data flow diagrams and data models. Object-oriented designs would rely on use-cases, UML class diagrams, sequence diagrams, and other documentation.

Acceptance tests are a specific type of validation tests. More commonly known as alpha or beta tests, this type of testing involves giving an incomplete software product to prospective customers and providing a mechanism to solicit their feedback. The focus on this phase is on addressing high-priority, late-breaking customer requirements that could have a detrimental impact on the product later on, and clear communication with the customer is vital

During acceptance testing, a configuration audit may sometimes be performed. This involves a thorough comparison of existing requirements and other supporting documentation against the implemented product. It may also involve the compilation of a test kit and other support materials to enable maintenance programmers to support the product through its predicted lifespan.

### System Testing

The focus of system testing is on the quality characteristics of a software product as it interacts in a system. Its purpose is to ensure that the other dimensions of quality that may not have been explicitly referenced in development lifecycle receive some attention. These dimensions include error-handling, in which errors are injected to simulate hostile real-world conditions, performance testing, in which the product is exercised using system data, security testing, to check for attack hardness, load testing, to verify data integrity at the upper end of the specified limit, and UI testing, to confirm that system interaction points are functioning as expected. A structured programming approach would probably involve generating large or hostile data objects; Object-oriented system tests might involve an analysis of use-cases to determine vulnerable points.

### White-box Testing

White-box testing is a code-based testing technique that relies on knowledge of the actual code used to implement a module. White-box tests focus on code defects found by inspecting unique paths through code, branching statements, loops, and other coding constructs. Effective white-box tests check for loop or data structure boundary issues, invalid or uninitialized data, and unexecuted code blocks.

One type of white-box testing is basis path testing. It relies on procedure description language (PDL), which describes coding constructs in a language-neutral way, and program flow graphs, which provide symbols to represent loops and conditionals. Using these work artifacts as guide-posts, the tester annotates PDL with sequential integers and identifies independent paths through the code. Each of these independent paths becomes part of a “basis set” of independent paths, which can be used to structure a suite of test cases that are guaranteed to execute each code block at least once.

Other types of white-box tests are control structure testing, which involves testing each statement independently and using truth tables to keep track of internal execution state, data flow testing, which identifies where a function is implemented and where it is referenced, and loop testing, which involves examining the entry and exit conditions of loops, checking for unstructured or faulty conditions.

White-box testing is very similar from either a structured or an object-oriented programming perspective; the scope of the testing is low enough that the control structures used to implement the overall architecture are identical, or nearly so.

### Black-box Testing

Black-box testing verifies system functionality without examining the source code. The purpose of black-box testing is to exercise the system from the perspective, and through the vectors, of a user, with the rationale that most bugs that matter will manifest themselves in this way. Black-box tests exercise product functionality, valid and invalid inputs, sensitive or boundary-related inputs, system capacity for handling large amounts of input, and overall user experience.

Many black-box testing techniques involve graphs. Each graph format looks similar, but models a slightly different semantic. Transaction-flow testing models the system states during a transaction. Each link in the transaction graph represents a step to be tested. State models are similar, except that each achievable system state is drawn as a node on a graph and the expected state transitions are links. Data flow test cases can also be modeled with graphs. Once the locations for the data transformations are known, cases can be created to generate and input the correct type of data.

An important part of black-box testing is equivalence partitioning, the process of reducing the test case set to it minimum number, without hurting the test suite coverage. Each input type is assigned a domain, and valid and invalid values are selected for that domain. Inputs that take a range of values get 3 cases, 1 valid and 2 invalid. Specific values get two (valid and invalid) and set members and boolean values each get two. In addition to selecting valid and invalid values, it is useful to also select boundary values. These are values right on, or just before and after an internal range of values understood by the software. Many common software bugs are often seen at such boundaries.

Orthogonal array testing is another technique used to narrow down functionally-equivalent test cases. An array of inputs applicable to a specific system state is identified. Each input is narrowed down depending on input class, as above. From that table of inputs, a small set of representative cases is identified. The set is influenced by the consequences of one, two, and many invalid values, as provided to the system. If each invalid value causes a different error-handling state, the number of required cases is relatively more complex than if only a few failure paths exist.

Although black-box testing applies to both structured programming and object-oriented systems, it lends itself far more readily to object-oriented systems, due to the common work artifacts used in developing such systems. Use-cases translate almost without modification into test cases, and class diagrams and CRC cards quickly help to identify equivalence classes. There are several ways to group black-box test cases when exercising object-oriented systems. Random testing, in which input values are selected at random, is very useful for stability and load testing. Partition testing involves organizing cases into those that cause a state transition and those that do not. Attribute testing involves organizing cases into that that modify internal data and those that do not. Category testing involves grouping cases according to the type of action they exercise (initialization, computation, query, and termination.)

# Project Management

Project management is the discipline involved with keeping projects on track. It is an activity required at all phases of the development lifecycle, across all disciplines, from every member of a product team. It is required because the process of building complex software is an enormously complex thing, and the complexity would quickly overwhelm any project not prepared to deal with it. Teams without effective project management develop multiple problems. A poorly managed schedule can cause team members to allocate work hours inappropriately, rushing to finish assignments. A lack of in-depth research into supporting tools can have a ripple effect over the entire team. A lack of recognition or a work item misassignment can damage employee morale and lead to valuable employees leaving the team.

Project managers must be effective communicators. They are responsible for explaining project progress to managers higher in the organization, and facilitating communication between engineering teams and outside partners and customers. They also create schedules, vision documents, human resource job postings, and almost any other form of written communication required by a project. Project managers are also the keepers of the various processes used by the product team to do their job. They adopt, drop, and customize processes to facilitate the workflow of the entire team. Additionally, they identify risky features, help determine areas of emphasis for evaluating software quality, hold meetings, and do any number of other required tasks.

The principal elements requiring attention in any successful project management effort are project, process, product, and people. To address these elements, managers in each role are required. Project managers track ongoing progress as the development cycle progresses. Process managers build schedules, plan deliverables, and partition milestones. Product managers identify high-level product vision, establish product goals, restrict scope, and plot the course for multiple releases while factoring in environmental and resource constraints. People managers handle recruiting, staffing, training, employee performance and morale.

People managers are the most traditional of managers, those that handle the day-to-day interactions with individual contributors. Fundamentally, they are responsible for building effective teams. In practice, this involves allocating work items, identifying and solving work item/human resource mismatches, ensuring that employees receive adequate training, determining compensation levels, interviewing and hiring employees, and firing them, when necessary. Effective people managers are efficient at determining and leveraging an individual’s motivations, unique working style and skill set.

Teams can be structured in different ways to enhance organizational efficiency. The decision of which structure to use is based on many factors, including the complexity of the product, the length of the development lifecycle, the individual strengths and personality traits of the team members, and the organizational culture. Some possible structures are “top-down” or closed, in which managers drive most of the day-to-day activity, “committee-driven” or open, in which group consensus is required to proceed, “merit-driven” or random, in which different individual contributors lead the team at different times, depending on their individual strengths, and “synchronous” or compartmentalized, in which teams work parallel to each other with very little communication.

All project managers are concerned with avoiding project failure. A small set of universal principles helps ensure that projects succeed. Starting strong with a clear set of process guidelines and expectations may be the most important step. Expectations are set early, and early assumptions are very hard to change. Maintaining momentum from that point on is vital – consistency in metrics-tracking, progress-tracking and reporting lends assurance to people unfamiliar with the effort. Continued smart decisions throughout the project cycle help to build respect among the team. The “W5HH principle” helps to guide these efforts and ensure their consistency. Answering the 5Ws and the 2 Hs for each product sub-group provides a common focus around which teams can rally: “Why is the system being developed? What will be done? When will it be accomplished? Who is responsible for a function? Where are they organizationally located? How will the job be done technically and managerially? How much of each resource is needed?”

# Summary

The modern life is dominated by software; it is no exaggeration to state that our very lives depend on it. Building successful, large-scale software products is an enormously complicated undertaking, and its sheer complexity lends weight to the argument that “Software Engineering”, while not as mature an engineering discipline as civil or electrical, is at least not misnamed. The practices and patterns described in this paper are but a partial part of the overall discipline, but they work together to provide an effective starting place for any team attempting to think about software engineering for the first time.

Appendix – Referenced Figures



Figure 1: Basic structure of the Waterfall Model.   
Source: Wikipedia, <http://en.wikipedia.org/wiki/Image:Waterfall_model.png>

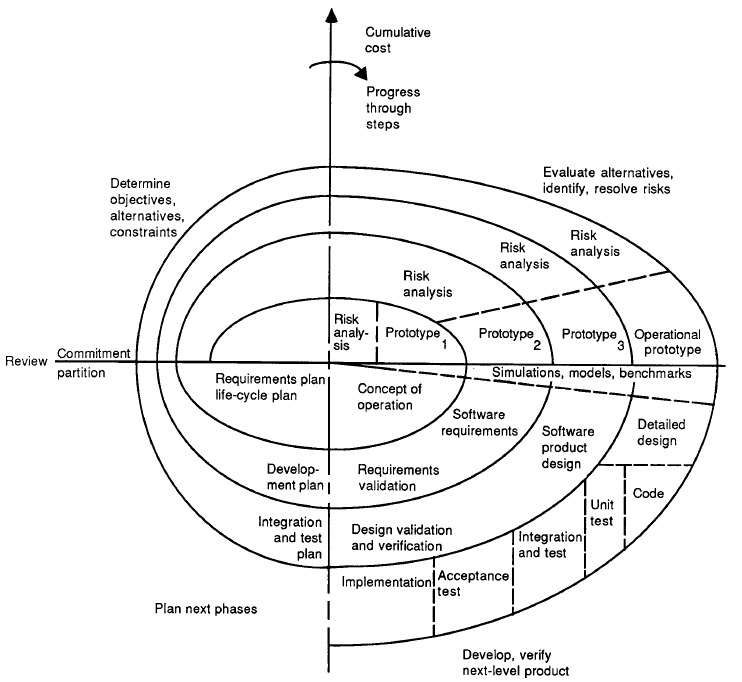


Figure 2: Barry Boehm’s spiral model

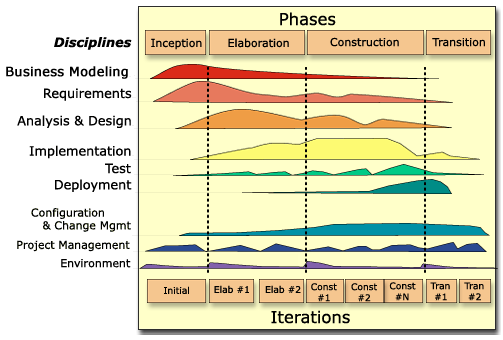


Figure 3: Rational Unified Process Lifecycle  
Source: <http://en.wikipedia.org/wiki/Image:RationalUnifiedProcess.png>



*Figure 4: System Context (level 0) DFD*

|  |
| --- |
| **Use-case:** Access location information about friends on a Buddy List  **Primary Actor:** Registered user  **Goal:** Provide a way for an FPB user to physically locate his or her friends  **Preconditions:** Registered user provides correct credentials; user has more than 0 friends added to the Buddy List; user’s friends have not blocked access to their information.  **Trigger:** A registered user decides to see where his or her friends are, at that moment.  **Scenario:**   1. The registered user enters their credentials on the login page 2. The system displays a customized home page for the registered user 3. The registered user clicks the “Buddy List” tab 4. The system displays a table of buddies, a zoomable map, and a set of icons on the map representing each Buddy 5. The registered user selects one of the Buddy icons and clicks “Details” 6. The system displays up-to-date location information for the buddy, including major landmarks, time spent at that location, and previous locations.   **Exceptions:**   1. Invalid credentials when logging into website 2. No Buddies configured 3. No Buddies found within given latitude/longitude parameters 4. Buddy has blocked the registered user from seeing location information or details   **Priority:** Moderate (priority 2)  **When available:** Second iteration  **Frequency of Use:** Frequent  **Channel to actor:** Via PC-based web browser and mobile device application  **Secondary actors:** System administrator, Buddy Beanies  **Channel to secondary actors**: Via PC-based web browser, server-side administrator application, cellular data connection and GPS  **Open issues:**   1. What are the infrastructure requirements to support this scenario? 2. What is the best way to handle latency issues? 3. How much tracking data should be cached, in the event that connection is lost? |

*Figure 5: Sample use-case*



Figure 6: UML Activity Diagram

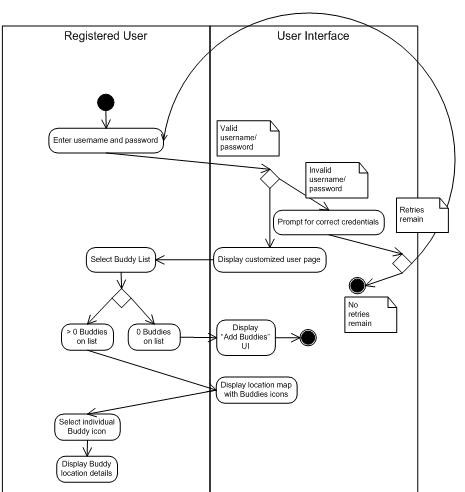


Figure 7: UML Swimlane diagram

Figure 8: UML class diagram

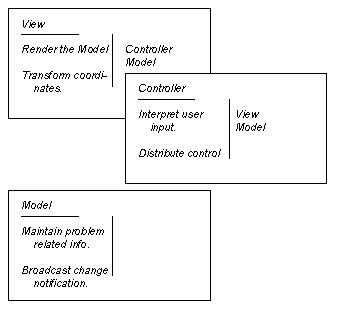


Figure 9: CRC cards



Figure 10: Sequence diagram

References:

Crawford07a: Ethan Crawford, *What is Software Engineering?* (Bothell: CSS 360, 2007)

Crawford07b: Ethan Crawford, *Requirements Analysis Modeling Approaches* (Bothell: CSS 360, 2007)

CMMI-DEV: Carnegie Mellon Software Engineering Institute, *CMMI for Development, Version 1.2*, <http://www.sei.cmu.edu/cmmi/models/CMMI-DEV-v1.2.doc> (Pittsburgh, 2006)

JAC99: Ivar Jacobson, Grady Booch, and James Rumbauch, *The Unified Software Development Process* (New York: Addison-Wesley, 1999)

Humphrey87: Watts S. Humphrey, *A Method for Assessing the Software Engineering Capability of Contractors*, CMU/SEI-87-TR-23, 1987.

Schrum99: Sandy Shrum, *Continuous and Staged, a Choice of CMMI Representations* <http://www.sei.cmu.edu/news-at-sei/features/1999/december/Spotlight.dec99.htm>

SEI07: *CMMI Appraisals*, <http://www.sei.cmu.edu/cmmi/appraisals/>

Pressman05: Roger S. Pressman, *Software Engineering: A Practitioner’s Approach, 6th Ed.* (New York: McGraw-Hill, 2005)

WIKI07c: Wikipedia, *Unified Process*, <http://en.wikipedia.org/wiki/Unified_Process>

WIKI07d: Wikipedia, *Rational Unified Process*, <http://en.wikipedia.org/wiki/Rational_Unified_Process>

WIKI07e: Wikipedia, *Iterative and Incremental Development*, <http://en.wikipedia.org/wiki/Iterative_and_incremental_development>

WIKI07f: Wikipedia, *Use Cases*, <http://en.wikipedia.org/wiki/Use_cases>

WIKI07g: Wikipedia, *Spiral Model*, <http://en.wikipedia.org/wiki/Spiral_model>

Boehm88: Barry Boehm, “A Spiral Model of Software Development and Enhancement”, *Computer*, May 1988, 61-72.

WIKI07b: Wikipedia, *Waterfall Model*, <http://en.wikipedia.org/wiki/Waterfall_model>

WIKI07h: Wikipedia, *Agile Software Development,* <http://en.wikipedia.org/wiki/Agile_software_development>

AGILE01: *Manifesto for Agile Software Development,* <http://agilemanifesto.org/>

WIKI07i: Wikipedia, *Scrum*, <http://en.wikipedia.org/wiki/Scrum_%28development%29>

WIKI07j: Wikipedia, *Extreme Programming,* <http://en.wikipedia.org/wiki/Extreme_Programming>

WIKI07k: *Data Flow Diagrams*, <http://en.wikipedia.org/wiki/Data_flow_diagram>

Martin00: Robert C. Martin, *UML Class Diagrams*, (ObjectMentor: <http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf>)

CRC cards example: <http://c2.com/doc/oopsla89/fig2.gif>

Cohn04: Mike Cohn, *Advantages of User Stories for Requirements,* (Prentice-Hall, New York: <http://www.awprofessional.com/articles/article.asp?p=342885&rl=1>)