**Question 1**

The Capability Maturity Model Integration (CMMI) is a process model, an organized, comprehensive collection of best practices for improving processes used in product development and maintenance. It was created by US Department of Defense-funded Software Engineering Institute at Carnegie Mellon University, and is an integration of several domain-specific CMMs previously used in software and systems engineering. 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]. The current purpose of the CMMI is two-fold. First, it provides an engineering organization with guidance for improving *process maturity* – the efficacy, repeatability, and measurability of a process – in all areas of product development. Second, it provides customers with the capability to assess the level of process maturity between different organizations [CMMI-DEV, p. 2].

The CMMI can be useful for many reasons. Because it incorporates large amounts of industry-tested workflow and concepts, it may be useful for troubleshooting existing processes and helping managers identify areas of improvement within an organization. CMMI process model descriptions are comprehensive, detailed, and consistent, and they provide a shared vocabulary for discussing the current state of a process. They exist as generic-but-complete templates that a product team can use to construct a new process or customize an existing one.

To get a better idea of the way CMMI can be applied, imagine a young company of just a few dozen employees, which we will call Sioch Industries. They had previously developed a software product that made several million dollars of profit. The company has expanded to work on a second version of this successful product. In the planning phase, it was discovered that many of the features in the first version are superfluous, and that no current employee knows why they exist. The project manager has so far been unable to find any requirement documents or specifications for the first version. Being somewhat familiar with CMMI, she suspects a breakdown in one of the process areas.

The 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]. Collectively, these process areas cover all aspects of product development, including communication, planning, modeling, construction, and deployment [Pressman05, p. 24]. It also includes several process areas that deal explicitly with improving process through measurement and analysis. Each area is broken up into specific goals, and each specific goal is accomplished by following one or more associated practices, which are the actions that must occur in order to satisfy the goal.

Let’s return to Sioch Industries, where our CMMI-trained project manager has identified what appears to be a deficiency in the process area of Requirements Development. In further conversations with some of the employees, she discovers that a functional specification had, at some point, been created, but employees remember it as being incomplete and no one can locate it now. The evidence suggests that some of the specific goals of the Requirements Development process area (customer requirements, product requirements, analysis, and validation) were followed, but the current state of the product indicates that there may have been some problems along the way.

In addition to specific goals, the CMMI defines generic goals and practices. Whereas specific goals define the components 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]. (See Figure 1 for the relationship between process areas, specific goals, and generic goals.) For measuring the maturity of an individual process area, the CMMI contains a “continuous” model with 6 capability levels, 0-5 (no process, initial, managed, defined, quantitatively defined, and optimizing) [CMMI-DEV, p. 45]. These levels describe the extent to which the work artifacts and processes of a process area are reviewed, standardized, customized, measured, and continuously improved. For measuring overall organizational process maturity, the CMMI also provides a “staged” model that groups process areas into overall maturity levels 1-5. Maturity level definitions map 1:1 to capability level definitions, minus level 0.

Back at Sioch Industries, our project manager needs to convince her boss, the CEO, that the capability level of the company’s Requirements Development process is low and that the process is in need of overhaul. She has also observed a need for improvement in the product architecture and in the process used to track code defects. The improvements will require a significant investment in tools and employee training; to successfully make her case for a slice of the budget, she will need hard evidence.

The Standard CMMI Appraisal Method for Process Improvement (SCAMPI) is an assessment process designed to “provide benchmark quality ratings relative to CMMI models” [SEI07]. A SCAMPI appraisal is the implementation of Watts Humphrey’s original vision of a substantive process for comparing different software companies by analyzing internal processes. 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. There are three levels of appraisal, C, B, and A, representing an increasingly thorough level of organizational coverage and evidential rigor. At the highest level, (A), work artifacts must be presented to demonstrate overall capability level; at the lowest level, interviews with employees suffice. A SCAMPI assessment can be a valuable window into an organization’s processes and process mentality. It provides a systematic way to analyze the level of process maturity and provides customers with useful datapoints for comparing competing organizations. It also provides the organizations themselves with valuable information useful in modifying or augmenting existing processes.

Our Sioch Industries project manager decides that she needs a SCAMPI appraisal. She finds an SEI-affiliated organization and sets it up. After completing the appraisal, the appraisal team unsurprisingly indicates a capability level of 1 in multiple process areas, including Requirements Development, and a low overall maturity level as a result. The project manager decides to recommend a concentrated process improvement campaign in just a few specific areas to her managers. The plan includes action items such as establishing a requirements document formatting standard, speaking with a minimum number of customers about all existing features of the product, and performing specification reviews after preliminary requirements have been identified. It also includes plans for improving bug tracking and design reviews.

When used properly, CMMI claims to reduce overall costs and schedule variance, and increase productivity, quality, customer satisfaction, and return on investment (CMMI-DEV, p. 2). When used improperly, critics claim that CMMI causes product teams to emphasize process over product quality and can introduce layers of bureaucracy that can greatly hinder the overall speed of product development [WIKI07a]. Ultimately, CMMI is a tool. Different organizations will find that it fits their business culture to varying degrees, and they will make use of its elements accordingly. At its worst, it stands as perhaps the most complete body of reference material ever compiled regarding software and systems engineering processes, and provides a small amount of useful material to everyone. At its best, it is a proven guidebook on which an organization can model its entire engineering structure, with a significant chance of success.

At Sioch Industries, our project manager’s efforts have shown some fruit. As a result of implementing a few key process areas identified by the SCAMPI audit, the capability level of the Requirements Development process area was raised to 3, and several other missing process areas were adopted and standardized at level 2. Requirements for the second version were carefully identified, using a standard company-wide requirement template, tailored to the specific product. Useless or obscure features were removed or retooled to be more useful. Requirements documents were archived for later quantitative analysis. Despite delivering many more features than the first version, schedule slips were dramatically reduced from the previous version, and the second version was delivered only a few months late. The project stayed within budget. Overall, employees reported higher satisfaction with their jobs, knowing that they had a predictable process in place as a framework. When the product finally reached customers, there were three times fewer support calls compared to the previous version, despite an expanded customer base. For at least one organization, CMMI was a success.

**References:**

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.

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

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/>

**Question 2a**

The waterfall model is a classic software development lifecycle (SDLC) model, first proposed in 1970 by W. W. Royce, [Wiki07b]. In its purest form, it is linear, sequential, and prescriptive. Each phase of the process ends before the next phase begins, and the output (specs, documents, meeting notes, etc) of the previous phase feeds the next phase, very much the way a mass of water moves from one ledge to the next in a waterfall (see Figure 2). 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 grouping of all the Requirements together with all the Design at the beginning of the waterfall model is not an arbitrary decision. It is an example of Big Design Up Front (BDUF), a philosophy driven by the observation that bugs found in the requirements phase are significantly less expensive to fix than bugs found further “downstream”, i.e., later in the project [WIKI07b]. By investing in up-front design, waterfall devotees hope to avoid writing unnecessary code and introducing unnecessary bugs. BDUF is a document-heavy philosophy. In order to adequately capture the requirements gathering and architectural design that happens in these phases, it is necessary to create extremely detailed records of the work. Ideally, the documents allow a detailed analysis of the requirements, prior to writing any code that might then be wasted when the design changes. Additionally, they keep track of the information relevant to the implementers in the large gap between requirements and implementation. They act as the “blueprint” for the remainder of the project, and live on as historical records for maintenance programmers.

To understand the benefits and drawbacks of the Waterfall model, it may be helpful to examine the development of two separate products. Company A, Brawn LLC, builds software that runs critical systems inside of tanks, and has a multi-year, cost-plus-fixed-fee contract with the US government. Company B, Brains Ltd., builds software add-ons for Microsoft Office, selling copies to individuals for a few dollars per copy on its website. Each company has a need for a software development lifecycle.

For Brawn LLC, completing all the requirements documentation and design specs prior to writing a single line of code comes naturally. As a highly mature organization of two thousand people working in government contracting, they are used to large amounts of process. Tank specifications such as engine characteristics and vehicle weight come from the military and are highly likely to be locked down by the time Brawn sees them. Additional tank features such as potential on-board weapons systems, navigational packages, and weight distribution caused by armor type are selected by the customer and have likely already been purchased. The development timeline is also quite lengthy – Brawn has contracted to deliver a completed product within four years, and part of their work item commitment is a comprehensive set of documentation detailing every requirement, and the location of every line of code responsible for fulfilling that requirement. To fulfill this requirement, Brawn has hired 5 full-time technical writers and a full-time editor. Brawn also has long-term quality bonuses built into their contract. If the number of reported flaws in the system stays below a certain threshold, Brawn is entitled to several more million dollars. Additionally, the upper management of Brawn is fully aware of the seriousness of code defects in the tank software. If something goes wrong, people will die.

The advantages of the waterfall model for Brawn and their customer are clear. Brawn gets to compete in the highly process-oriented world of military contracting, and they receive the exhaustive investigation into customer requirements provided by BDUF. The customer receives extensive documentation, which is extremely useful for systems likely to be deployed for decades. Brawn reduces its potential culpability for deaths in the field and increases its chances of a performance bonus by focusing on rigorous design and quality. Brawn’s rate of software production may not be as high as it could possibly be, but the business structure of its contract does not require this optimization for the company to be successful.

Brains LTD faces a very different set of challenges. They are a small company of about 40 people, started by a college dropout and his roommate. Their customer base is constantly shifting, depending on which customers are using Microsoft Office and what their specific needs are. Most development ideas come from user feedback, not hierarchically-organized military committees. Brains is heavily influenced by emerging trends on the Internet; one of their most popular add-ins in recent times allowed a user to embed YouTube videos in Word documents. Additionally, they have many more customers to satisfy, given the low price of each of their products. For documentation and user support, Brains relies on its user community, which makes use of company-hosted message boards to answer common questions, and they employ a single webmasters/message board moderator for this purpose. Bugs in their products cause inconveniences, but are unlikely to cause serious injury or death.

For Brain, potential benefits of the waterfall model are much harder to see. Increasing Brain’s documentation requirements would cost them valuable internal resources, and requiring two phases to be entirely complete before any construction could begin would likely idle an significant portion of their small company. Taken together, these elements could potentially cost Brain some number of lost customers, due to the heavy competition in the Office add-on market. Brain’s success is also dependent on being highly response to customer feedback. Waiting an entire milestone to apply customer requirements to the design could also mean losing a niche to a competitor.

The weaknesses of the waterfall model are not applicable only to smaller companies for whom lighter-weight processes might be a better fit. Let us return to Brawn LLC, where the US Government has returned with new requirements, half way through the Design phase for the new tank software. A new administration has just been inaugurated, and they have shifted the focus away from heavy battle armor to more agile transports. While fortunate to still retain the contract, Brawn must now retool all the tank-specific requirements to make them appropriate for the next-generation vehicle. They must also create an entirely new set of specifications for the new required features. The same documentation requirements apply now as before, and Brawn has lost perhaps a year’s worth of effort. If implementation has begun, it must be stopped and the entire development team idled while the new design documentation is generated.

On the other hand, the waterfall model, especially in its modified forms such as “sashimi” (waterfall with overlapping phases) does provide a significant amount of basic structure in what can otherwise be an ad-hoc development environment [WIKI07b]. Back at Brains LTD, a small team has been investigating adding a web server inside an Office application, to address a customer requirement for an alternate vector to remotely retrieve data from an application. This is a more complicated undertaking with multiple dependencies. If the development team chooses to launch into it without investigating the alternatives, they may find that they have chosen to rely on an unsupported software library or external tool. A formal up-front requirements process in this scenario might cost some time, but the investment is likely to be worthwhile, especially if prototyping and architectural design can proceed before the requirements and the design document phases are complete.

Both Brains and Brawn are deliberately simplified examples of potential software development scenarios, but each contains the basic elements that need to be considered when choosing a model. What product is actually being 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? After evaluating many of these questions, many modern development organizations find that basic waterfall is unsuitable to their needs for many reasons, due partially to the existence of other models that seem more nuanced or efficient. In some ways, the waterfall model is the “bubble sort” of the SDLC world – simple, predictable, easy to teach, and, frequently, the least efficient software development process of all. Like bubble sort, however, it contains useful lessons, and some significant strengths.

**References:**

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

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

**Question 2b**

Software processes do not develop in a vacuum. Their popularity rises and falls along with the tools, concepts, and pressing development needs of the eras in which they are born. As much as any process, this is true for the Unified Process, also known by its most popular trademarked variant, the Rational Unified Process. Developed in the 80s and 90s by the Rational Software corporation as object-oriented languages were reaching prominence, the Unified Process, while not restricted in scope to object-oriented techniques, strongly reflects its OO roots. The UP is an implementation of Barry Boehm’s spiral model [WIKI07d]. It is iterative, incremental, driven by user scenarios (“use cases”), architecture-centric, and informed by risk [JAC99]. To eschew the involuntary obfuscation indigenous to any buzzword, an example may be useful.

Imagine a company that has decided to implement a new client for a new Internet file-sharing protocol known as ByteFlood. The protocol supports tit-for-tat bandwidth throttling, multi-user chat, and many other complex nuances that enable support for an equitable file-sharing community. These features can be exposed (or not) by the client as the developer sees fit. The company has decided to gamble a relatively small number of resources on developing this client, and they want to optimize their return on investment. While the basic vision is clear, the overall implementation and even the initial set of requirements are currently unclear. Furthermore, the key project decision-makers expect the initial requirements to change, and to continuously receive a stream of secondary requirements throughout the project as increasing numbers of users consume the alpha and beta releases. This team needs a software development lifecycle model that will support their development efforts in this type of rapidly-changing environment.

The Unified Process development lifecycle is split up into five phases: Inception, Elaboration, Construction, Transition, and Production. These roughly map to Pressman’s generic phases of communication, planning, modeling, construction, and deployment. Each UP phase is broken up into one or more iterations, defined as an interval of time that results in a release of a stable set of “work artifacts” or “work products”, which can be executable code, prototypes, system features, or documents of various sorts. This is in contrast to SDLCs like the Waterfall Model that do not permit coding until the requirements are complete [Pressman05, p. 64]. In the Unified Process, all development activities, including coding, happen simultaneously in each iteration and in all four phases. Only the relative weight of the development activity varies, depending on the phase. (See Figure 3 for estimated activity levels in the various disciplines.) This addresses the problem of unallocated human resources, one of the key deficiencies of the Waterfall Model. It also provides a team with the ongoing capability to address changing requirements. If the Waterfall Model represents the eponymous waterfall, the Unified Process model can be seen as a fishing net, strung around a whale. Each iteration of each phase incrementally gathers up the net, until the final product is fully wrapped and completely rendered.

Except in extremely complicated development scenarios, Inception is a single-iteration phase used to enumerate business goals, communicate the high-level vision, generate rough architectural diagrams, plan the total number of iterations, and identify preliminary “use-cases” [Pressman05, pp. 64-65]. 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]. In contrast to requirements, use-cases are customer-centric, and may exist in a many-to-one or one-to-many relationship with requirements. This evolutionary shift from requirements to use-cases helps enable the process of incorporating frequent customer feedback into subsequent iterations. It puts product team members into a customer mindset and forces project managers to create supporting vocabulary that enables communication between customers and programmers and supporting processes that enable mapping between use-cases and requirements. Stakeholder feedback, including customer feedback, is taken at the end of each iteration and incorporated into the next iteration.

Having selected Unified Process, our ByteFlood product team begins talking to customers and researching existing products. They identify a set of 7 primary use-cases to support high-level file-swapping scenarios, assemble a preliminary architectural design, create an overall schedule and identify risks. Despite the incomplete nature of these starting activities, the ByteFlood team is practicing Big Design Up Front. This is because the Unified Process is architecture-centric, a fact which stems partially from a tool that predates the UP, Unified Modeling Language. UML is a flowchart-esque notation used to model object-oriented designs, and the Unified Process was created to support UML-driven modeling. UP design philosophy states that designs may be iterated, and development may be incremental, but the basis of everything must be sound architecture, designed from the very beginning [WIKI07d].

On exiting the Inception phase, the ByteFlood team begins Elaboration. This involves further ongoing customer conversations, possibly after having demoed the preliminary prototype. As a result of incorporating stakeholder feedback, the number of use-cases is greatly expanded from the initial set. The product team looks carefully for system-related and technical risks in this phase. Each iteration is an opportunity to change course or to abort the project completely, and identifying risks and mitigations early is vital to the continued success or the clean termination of the project. Multiple different models are generated, including a detailed architectural model that relies on software design patterns to validate its design, prior to actual implementation. A rough cut of the entire system is implemented [Pressman05, p. 64]. The first Quality Assurance efforts begin in this phase, and the first unit tests (simple tests that exercise a small piece of functionality) are developed to ensure ongoing quality [WIKI07d].

Up to this point, it is still possible to abort or redesign the project. Some resources have been consumed, but the costliest investments are in the upcoming phase: Construction. At this point, design models are finalized and major coding begins. Test actively engages with development and unit tests are created to keep pace with the development of new features. Stability and quality are ongoing goals through each iteration of the Construction phase, and integration and scenario tests as well as unit tests are added to the automated and manual test scenarios. The iterative and incremental development tenant of UP is seen clearly in this phase; each iteration produces a stable, functional prototype that builds incrementally on the previous iteration.

The Construction phase terminates at the Initial Operational Capability Milestone, a checkpoint at which all the use-cases required for the release have been implemented [WIKI07d]. At this point, the product moves into the Transition phase, which involves beta-testing, bug-fixing, and final quality signoff. Multiple Transition iterations may be required to reach the final sign-off state, at which point, the product is complete and moves into Production mode.

And what of the ByteFlood client? The Unified Process gave the product team multiple opportunities to solicit feedback from the stakeholders. It provided a strong architectural grounding from which to drive incremental redesigns, and it encouraged tighter collaboration between the customers, the product managers, and the development team. The final released product matched the original vision, but contained use-cases and workflow that were significantly modified from the ones in the Inception phase. This should be seen as a strength of the model –changes that might have been much costlier to make later in the product cycle were made earlier, thanks to the increased rate of feedback and the existence of processes within the model set up to receive and incorporate it.

Despite its strengths, there are many circumstances for which the Unified Process might not be the ideal choice. Although it is an iterative and incremental development process, UP is still very document-heavy. Many software teams, especially smaller organizations, chafe under these types of document requirements, and they may also be overkill for systems that are natively self-documenting (i.e., properly commented Java or C#). Additionally, each phase still has a set of deliverables that includes specific documents, which can lead to an increased management focus on documentation, rather than on working product. UP is also still Big Design Up Front, which can cause a major loss of time and resources if the customer scenarios change so drastically that the underlying architecture is no longer appropriate.

Despite these shortcomings, UP is still an excellent, well-rounded SDLC. It encapsulates a solid set of common-sense project management principles that give product teams multiple opportunities to change course and align with customer requirements, while providing enough flexibility that many other process improvement principles such as Agile methodologies or CMMI can be incorporated into its framework. It brings along with it a mature object-modeling language in UML and a wealth of industry-tested techniques based on failures from the past. Overall, it is an excellent choice for teams with a strong commitment to architecture and requirements development that still have the need for some flexibility in the final product.

**References:**

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

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>

**Appendix – Referenced Figures**



Figure 1: Relationships between CMMI process area model components   
Source: CMMI for Development, 1.2



Figure 2: Basic structure of the Waterfall Model.   
Source: Wikipedia, <http://en.wikipedia.org/wiki/Image:Waterfall_model.png>

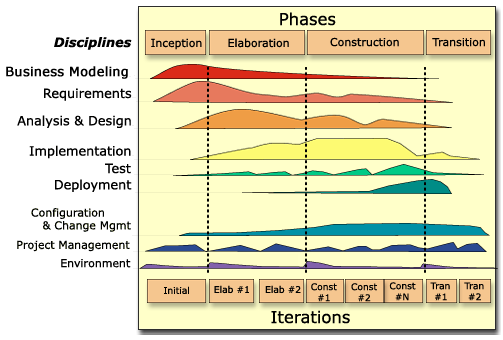


Figure 3: Rational Unified Process Lifecycle  
Source: <http://en.wikipedia.org/wiki/Image:RationalUnifiedProcess.png>