A Holistic Life Cycle for Large-Scale Complex Software

OSMAN BALCI
Professor

Department of Computer Science
638 McBryde Hall, MC 0106
Virginia Polytechnic Institute and State University (Virginia Tech)
Blacksburg, Virginia 24061, U.S.A.

http://manta.cs.vt.edu/balci/
balci@vt.edu

August 18, 2016
Abstract

Conducting large-scale complex software engineering projects continues to pose significant challenges for software engineers, project managers, and sponsoring organizations. This paper presents a software life cycle called SECRET to alleviate the challenges. SECRET (holiStic lifE Cycle for laRge-scale complEx softWare) describes a framework for organization of the phases, processes, work products, quality assurance activities, and project management activities required to develop, use, maintain, and reuse a software application / system from birth to retirement. It provides guidance to software engineers, project managers, and sponsoring organizations for conducting the software engineering tasks in a structured manner. SECRET specifies the work products to be created by executing the corresponding processes together with the integrated verification, validation and quality assurance activities. SECRET is critically needed to modularize and structure complex software development. It provides valuable guidance for conducting a large-scale complex software engineering project.
1. Introduction

A life cycle is commonly used in software/system engineering under the names of system engineering process, software process, (software) process model, or (software) life cycle model [Pressman and Maxim 2015; Balci 2012]. We define a software life cycle as a framework for organization of the phases, processes, work products, quality assurance activities, and project management activities required to develop, use, maintain, and reuse a software application (app) / system from birth to retirement.

We use the term “process” to refer to a set of activities, actions, and tasks within the life cycle instead of using it to refer to the entire life cycle as it is sometimes done in the literature. Under our definition, a process (e.g., architecting) is executed to create a work product (e.g., architecture specification).

Many software life cycle models have been published [Pressman and Maxim 2015; Sommerville 2015], but, including the Spiral model, the V-model, and the Waterfall model, they (a) do not provide the critically important process ↔ work product viewpoint, (b) are not sufficiently holistic, (c) lack the essential Architecting process, which has become crucial with the emergence of cloud software engineering, and (d) lack the Problem Formulation process. The life cycle presented herein is created to remedy these deficiencies and to provide a comprehensive quality-centric framework for conducting large-scale complex software engineering projects.

The remainder of this paper is organized as follows. Section 2 introduces a strategy for software quality assurance throughout the entire software life cycle. Section 3 describes a holistic life cycle for large-scale complex software. Concluding remarks are given in Section 4.

2. A Strategy for Software Quality Assurance

As the saying goes “Quality is Job 1!” The ultimate goal of a software engineering project is to develop a software application (app) / system with sufficient quality characteristics such as accuracy, security, dependability, interoperability, performance, and usability. Software Quality Assurance (QA) refers to the planned and systematic activities that are established throughout the software life cycle to substantiate adequate confidence that a software application / system possesses a set of characteristics required for its intended uses.

Undoubtedly, accuracy is the most important quality characteristic of a software application / system, and is assessed by conducting Verification and Validation (V&V). The terms V&V are consistently defined for whatever entity they are applied to. Let $X$ be that entity such as a requirement, software design, user interface, executable software component, or data. Then, V&V can be defined generically as follows:

- $X$ Verification deals with the assessment of transformational accuracy of the $X$ and addresses the question of “Are we creating the $X$ right?”
- $X$ Validation deals with the assessment of behavioral or representational accuracy of the $X$ and addresses the question of “Are we creating the right $X$?”

For whatever entity to be subjected to V&V, substitute the entity name in place of $X$ above, the definitions will hold.

Transformational accuracy refers to the accuracy of transforming $X$ from one form into another, e.g., transforming a flow chart representation of an algorithm into Java programming language code, transforming requirements into a design specification, or transforming a software component design into an executable code.
Behavioral accuracy refers to the accuracy of how well $X$ mimics the behavior of what it represents. Representational accuracy refers to the accuracy of an entity that does not have behavior such as data. We validate data by assessing how well it represents what it is intended to represent since data does not have behavior.

V&V are typically conducted by way of testing. Testing refers to the activity of designing a test, specifying test conditions and data, and determining a procedure to follow for the purpose of judging transformational accuracy (verity) and/or representational/behavioral accuracy (validity).

Under the current state of the art, we are unable to claim a level of accuracy of a reasonably large and complex software application / system with 100% confidence due to many reasons including software complexity, lack of exhaustive testing, reliance on human judgment, qualitative measurements, and lack of data. Hence, software application / system V&V is viewed as a “confidence building” activity. For a reasonably large and complex software application / system, the “confidence building” activity must be performed by assessing not only the software application / system accuracy, but also other quality characteristics. Successful assessment of the overall software application / system quality increases our confidence in software application / system accuracy.

The more comprehensive and detailed the overall software application / system quality assessment is, the more confidently a certification decision can be reached for a software application / system. Four major perspectives or four Ps influence the software application / system quality as depicted in Figure 1.

![Figure 1. Four Ps Influencing the Software Application / System Quality](image)

Software application / system quality assessment can be approached from any one of the four Ps, but a combination of all four will provide the best balance and result in a much higher level of confidence in making a certification decision. Therefore, we advocate the following strategy.

The software application / system QA strategy should involve the measurement and integrated assessment of the following in a particular software life cycle stage:

- Output work product (or artifact),
- Process used in creating the output work product,
- Quality of the people employed in executing the process to create the work product, and
- Project characteristics (e.g., configuration management, risk management, project planning, project monitoring and control).
3. A Holistic Life Cycle for Large-Scale Complex Software (SECRET)

A holistic life cycle for large-scale complex software (SECRET) is presented in Figure 2 representing a framework for organization of the phases, processes, work products, QA/V&V activities, and project management activities required to develop, use, maintain, and reuse a software application/system from birth to retirement.

Legend:
- Process
- Work Product
- Iteration
- Quality Assurance (QA)
- Executable Software
- Maintenance

Figure 2. A Holistic Life Cycle for Large-Scale Complex Software (SECRET)
SECRET decomposes the life cycle framework into four phases: (1) requirements phase, (2) architecting and design phase, (3) programming and publishing phase, and (4) software use and feedback phase. The first three phases constitute the software development phases. A phase is decomposed into stages, e.g., the requirements phase is decomposed into problem formulation stage and requirements engineering stage. Each stage contains one process and is named after that process.

A process, represented by a double-line arrow, is executed to create a work product during the development phases (i.e., the first three phases). For example, we execute the process of Architecting to create the work product Architecture Specification.

The direction of the process double-line arrow is intended to show the workflow throughout the life cycle; however, SECRET should not be interpreted as strictly sequential or linear. SECRET is iterative in nature, as represented by the single-line back arrows, and we bounce back and forth between the work products until a work product is completed with acceptable quality characteristics. For example, an error identified during V&V of an executable software component may require changes in the requirements specification and therefore, reverse transition is expected.

The QA, shown in a rounded rectangle symbol in Figure 2 should not be interpreted as a distinct step during the development phases although it is specified under each work product. QA, V&V and testing are continuous activities conducted hand in hand as integrated with every software life cycle task.

SECRET does not dictate any particular development methodology; it can be used with any desired development methodology such as agile development, prototyping, incremental development, exploratory development, evolutionary development, reuse-based development, or top-down/bottom-up development.

SECRET advocates the four perspectives (Ps) of software development: Process, Product, Project, and People. These four Ps cannot be all included in a graphical representation of the life cycle, but they play a central role in SECRET’s framework. SECRET enables to view software development from the four Ps. It (a) specifies the work products to be created under the designated processes together with the integrated QA/V&V activities, (b) modularizes and structures software development and provides valuable guidance for project management, and (c) identifies areas of expertise in which to employ qualified people.

Table 1 lists the SECRET input-output transformations throughout the software development phases where a process takes a work product(s) as input and produces another work product as output.

<table>
<thead>
<tr>
<th>Input Work Product(s)</th>
<th>Life Cycle Process</th>
<th>Output Work Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Domain</td>
<td>Problem Formulation</td>
<td>Problem Specification</td>
</tr>
<tr>
<td>Problem Specification</td>
<td>Requirements Engineering</td>
<td>Requirements Specification</td>
</tr>
<tr>
<td>Design Specification</td>
<td>Programming</td>
<td>Software Components</td>
</tr>
<tr>
<td>Software Components</td>
<td>Integration</td>
<td>Software Application / System</td>
</tr>
<tr>
<td>Software Application / System</td>
<td>Publishing</td>
<td>Published, Delivered, or Deployed Software</td>
</tr>
</tbody>
</table>
3.1 Problem Formulation

Software engineering is a discipline for providing solutions to complex problems by way of engineering a software product or a software system. Software engineers are commonly referred to as “Solution Providers”. Many different categories of software-based solutions exist such as cloud, embedded, mobile, real-time, or standalone. Each category of solution has its own set of unique characteristics and brings its own technical challenges.

Problem Formulation (a.k.a. Problem Structuring or Problem Definition) is the process by which the problem domain is analyzed to create a Problem Specification, which is sufficiently well defined to enable specific action. This process takes the problem domain as input and produces the work product Problem Specification as output as depicted in Figure 3. Problem Formulation is the first process of every problem solving methodology as well as of a software life cycle.

The communicated problem, its boundary, and the problem context are often vague, unclear, ambiguous, and carries the subjective human perception and judgment as represented by the dashed irregular lines in Figure 3. The process of problem formulation is conducted to transform the communicated problem into a well-defined problem specification, which is expected to contain the entire actual problem. If the problem specification does not completely contain the actual problem, Type III Error occurs. Type III Error is the error of solving the wrong problem and its probability of occurrence must be minimized. [Balci 2010, 2012; Balci and Nance 1985]

Once a problem specification is produced, it must be analyzed to determine the best solution strategy in an unbiased manner without being under the influence of a solution technique. The analysis is conducted by (a) estimating the benefits/cost ratios of alternative potential solution strategies, and (b) selecting the strategy with the highest benefits/cost ratio. If the result of this analysis identifies a software-based solution, then we transition to the next process, requirements engineering, in the software life cycle.

The problem specification is required to possess sufficient credibility, which is the degree to which the problem specification is believable. Following the QA strategy based on the four Ps outlined in section 2, the problem specification credibility can be assessed by way of integrative assessments of the (a) quality of the problem specification as a work product, (b) problem formulation process used in creating the problem specification, (c) quality of the people employed for problem formulation, and (d) project characteristics such as planning, resources, and documentation as related to this life cycle stage. Balci [2012] provides a set of indicators for assessing the problem specification credibility.
3.2 Requirements Engineering

Requirements Engineering is the process of elicitation of requirements based on the problem specification and the problem domain, and specification of the requirements in an authoritative manner. This process takes the Problem Specification and the problem domain knowledge as input and produces a software Requirements Specification Document (RSD) as the output work product. The software RSD becomes part of a legal contract between the software developer and the software sponsor in those cases where software-based solution is provided under a contract.

Many approaches exist for executing a variety of activities in the requirements engineering process [Pressman and Maxim 2015; Sommerville 2015]. Use Case-based Requirements Engineering is considered the best practice for the following reasons:

(a) A Use Case represents a small amount of work the software system is required to perform. Thus, decomposing large-scale complex software system functionality into Use Cases produces the modularization much needed to overcome the complexity.

(b) Identifying the “real” functional requirements is always challenging for large-scale complex software systems. However, “real” functional requirements can be more successfully identified by employing Use Cases because a small amount of software functionality described by a Use Case is understandable and manageable in complexity.

(c) Listing requirements one after the other, even in different categories, does not provide any help for transitioning from requirements to software system design. However, Use Cases turn themselves into classes in an object-oriented design and significantly facilitate the transition.

(d) Associating functional requirements with a Use Case enables better life-cycle traceability: A requirement is traced to a use case, a use case is traced to a class in software design, which in turn is traced to a class in code. This forward life-cycle traceability can also be performed backward, all the way from code to a requirement.

Although a requirement is specified authoritatively using “shall” as a sentence or a paragraph in a natural language such as English, it must not be viewed as a natural language specification. A requirement must be viewed as a product and must be engineered as a product to possess a number of desirable quality characteristics such as the ones shown in Figure 4.

![Figure 4. A Set of Quality Characteristics of a Requirement](image-url)
• Requirement Accuracy is the degree to which the requirement possesses sufficient transformational (verity) and representational (validity) correctness. The accuracy is assessed by performing V&V.
  o Requirement Verification is substantiating that a requirement is transformed from higher levels of abstraction into its current form with sufficient accuracy.
  o Requirement Validation is substantiating that a requirement represents a real need with sufficient accuracy. Rapid Prototyping is known to be an effective approach for requirements validation.
• Requirement Clarity is the degree to which the requirement is unambiguous and understandable.
  o Requirement Unambiguity is the degree to which each statement of the requirement can only be interpreted one way.
  o Requirement Understandability is the degree to which the meaning of each statement of the requirement is easily comprehended by all of its readers. A requirement should be specified using active (direct) voice as opposed to passive (indirect) voice to achieve clarity.
• Requirement Completeness is the degree to which all parts of a requirement are specified with no missing information, i.e., each requirement is self-contained.
• Requirement Consistency is the degree to which the requirement is specified using uniform notation, terminology, and symbology, and any one requirement does not conflict with any other.
• Requirement Feasibility is the degree of difficulty of implementing a single requirement, and simultaneously meeting competing requirements. Sometimes it may be possible to achieve a requirement by itself, but it may not be possible to achieve a number of them simultaneously.
• Requirement Modifiability is the degree to which the requirement can easily be changed.
• Requirement Testability is the degree to which the requirement can easily be tested. A testable requirement is the one that is specified in such a way that pass/fail or assessment criteria can be derived from its specification.
• Requirement Traceability is the degree to which the requirements related to a particular requirement can easily be identified.

Following the QA strategy based on the four Ps outlined in section 2, Requirements Engineering QA should be conducted by way of integrative assessments of the (a) quality of the software RSD as a work product, (b) requirements engineering process quality, (c) quality of the people employed in requirements engineering, and (d) project characteristics as related to this life cycle stage.

3.3 Architecting

Architecting is the process of creating and specifying an architecture for a network-centric / cloud software system. The process of architecting takes the Problem Specification and software RSD as input and produces a network-centric / cloud software system’s Architecture Specification as the output work product. Chigani and Balci [2012] provide a CMMI-like process description [SEI 2010] of architecting, which can be used for conducting the process of architecting.

The output work product, Architecture Specification, refers to the fundamental organization of software system components that interoperate over a network (e.g., Internet, virtual private network, wireless network), relationships among the software system components, and the principles and guidelines governing the design and evolution of those components [IEEE 2011]. The Architecture Specification is produced by typically using the DoD Architecture Framework (DoDAF), which provides 52 models (diagrams) for representing an architecture under the following viewpoints [DoDAF 2009a, b, c]:

1. All Viewpoint (AV), 2 models
2. Capability Viewpoint (CV), 7 models
3. Data and Information Viewpoint (DIV), 3 models
4. Operational Viewpoint (OV), 9 models
5. Project Viewpoint (PV), 3 models
6. Services Viewpoint (SvcV), 13 models
7. Standard Viewpoint (StdV), 2 models
8. Systems Viewpoint (SV), 13 models

Major architectures that have been in use include Client-Server Architecture [Berson 1996], Distributed Objects Architecture [Balen 2000], Peer-to-Peer Architecture [Kwok 2011], and Service-Oriented Architecture [Erl 2005]. Client-server and service-oriented architectures are the most commonly used ones.

A \textit{Client-Server Architecture} (CSA) based on the Java platform, Enterprise Edition (Java EE) [Oracle 2015a] is shown in Figure 5 with five tiers: (1) client tier, (2) web tier, (3) business tier, (4) data mapping tier, and (5) data source tier.

![Figure 5. A Java EE-based Client-Server Architecture](image)

A \textit{Service-Oriented Architecture} (SOA) based on web services is shown in Figure 6 with six conceptual layers [Chigani and Balci 2012; Glass 2008]. SOA is considered to be the architecture of choice for providing the best interoperability among network-centric systems of systems.

A distinct difference exists between an architecture and design. A \textit{design} is an instantiation from an \textit{architecture} similar to how an \textit{object} is an instantiation from a \textit{class}. Many designs can be created from a given architecture specification. For example, Figure 5 shows the Java EE-based design instantiated from the CSA. Similarly, a design based on the Microsoft .NET framework [Microsoft 2015a] can also be instantiated from the CSA.

The archit ecting process can be conducted by (a) selecting a known architecture (e.g., CSA, SOA), (b) composing an architecture from a set of known architectures (e.g., CSA combined with SOA on the back end), or (c) creating a new architecture. [Chigani and Balci 2012]
Following the QA strategy based on the four Ps outlined in section 2, architecting QA should be conducted by way of integrative assessments of the (a) quality of the Architecture Specification as a work product, (b) architecting process quality, (c) quality of the people employed in architecting, and (d) project characteristics as related to this life cycle stage.

### 3.4 Design

The process of Design deals with the instantiation (creation) of a design specification of a software application / system from the Architecture Specification given. The design process takes the software RSD and Architecture Specification as input and produces a Design Specification as the output work product.

The design process must be carried out with respect to the software category and the architecture specification given. Major software categories include the following:
1. **Cloud Software** is multi-tier distributed software that runs on a server computer typically under the control of an Application Server software product (e.g., GlassFish, WebSphere, WildFly) and used by a user usually with a web browser over the Internet on a network-connected computer such as desktop, laptop or handheld computer (e.g., smartphone, tablet). Many cloud software development platforms and frameworks exist including the following [Schutt and Balci 2015]:

   - Java platform, Enterprise Edition (Java EE) [Oracle 2015a]
   - Microsoft platform, .NET Framework [Microsoft 2015a]
   - Ruby on Rails Framework
   - Zend Framework (PHP)
   - Node.js Platform (JavaScript)
   - Python Framework

2. **Embedded Software** is the software that is incorporated within a hardware system such as airplane, automobile, guided missile, Navy ship, or satellite. The software becomes part of the operation of the hardware system it is embedded in. [Sommerville 2015]

3. **Mobile Software** is the software that runs on a handheld computer (e.g., smartphone, tablet) or wrist computer (e.g., Apple Watch). Different platforms exist for mobile software development such as Apple iOS [Apple 2015a], Google Android [Google 2015], and Microsoft Windows Phone [Microsoft 2015b].

4. **Real-Time Software** is the software that is required to respond to events in real time [Cooling 2002]. Producing a response before a time deadline dictated by the requirements stands out to be a critical technical challenge in real-time software development. Typical examples of real-time software-based systems include air traffic control system, military command and control system, and nuclear power plant.

5. **Standalone Software** is the traditional software that does not require network connection to provide its functionality. It is installed on one personal computer such as workstation, desktop, or laptop and it is intended for use typically by a single user.

Each software category possesses its own unique characteristics, poses its own technical challenges, and has its own design strategies, methodologies, and design patterns. The design process must be defined and executed based on its software category.

The process of design can be conducted in one or more stages depending on the design complexity under the software category. Typically, we conduct the design process in two stages: high-level design and detailed design. For large-scale and complex software systems, the design process is conducted in three stages: preliminary design, interim design, and critical design.

Following the QA strategy based on the four Ps outlined in section 2, software application / system Design QA should be conducted by way of integrative assessments of the (a) quality of the Design Specification as a work product, (b) design process quality, (c) quality of the people employed in design, and (d) project characteristics as related to this life cycle stage.
3.5 Programming

The process of programming is typically executed by employing an Integrated Development Environment (IDE) such as

- Eclipse [Eclipse Foundation 2015]
- Rational Application Developer for WebSphere Software [IBM 2015]
- NetBeans [Oracle 2015b]
- Visual Studio [Microsoft 2015c]
- Xcode [Apple 2015b]

An IDE is an integrated set of software tools intended to provide computer-aided assistance for the development of executable software. An IDE typically provides source code editor, compiler, debugger with breakpoints support, graphical user interface (GUI) builder, an extensive library of reusable classes, build and run automation tools, automatic code formatting, refactoring and code snippet insertion, and syntax highlighting and code completion.

Given a problem description, launching an IDE and starting coding by skipping the processes of problem formulation, requirements engineering, architecting, and design is known as the build-and-fix approach and must never be used.

Following the QA strategy based on the four Ps outlined in section 2, programming QA should be conducted by way of integrative assessments of the (a) quality of the executable Software Components as the work product, (b) programming process quality, (c) quality of the people employed in programming, and (d) project characteristics as related to this life cycle stage. Dozens of dynamic software verification, validation, and testing techniques are available [Pressman and Maxim 2015] for assessing the accuracy quality characteristic of executable software components.

3.6 Integration

Integration is the process of bringing together the software components developed separately by different teams or subcontracting companies. This process takes the executable software components as input and produces the integrated software application / system as a finished product.

The process of integration should be well planned and executed when components of a large-scale software system are contracted out to different companies or teams for development. It is considered best practice to establish an Integrated Product Team (IPT) with members from subcontractors or teams to oversee the planning and execution of the integration process.

In the current era of system of systems engineering, the key to success for integration is to employ the Open Systems Paradigm (OSP). Traditionally, a network-centric (cloud) software system is engineered to provide and receive services at the system level. Under OSP, each software system component is engineered to provide and receive services directly. OSP mandates that the interface specification of a component (a) is fully and explicitly defined, (b) is widely used and available to the public (i.e., non-proprietary interface), and (c) uses standards which are developed / adopted by industrially recognized standards bodies. OSP not only facilitates integration, but also it is essential to achieve interoperability.

Following the QA strategy based on the four Ps outlined in section 2, integration QA should be conducted by incorporating the assessments of the (a) quality of the completed software application / system as the
work product, (b) integration process quality, (c) quality of the people employed in integration, and (d) project characteristics as related to this life cycle stage.

3.7 Publishing

The process of publishing the finished software application or software system includes:

1. Distributing the software application (e.g., mobile software, desktop/laptop/server software, operating system software) on an App Store (e.g., Apple App Store, Google Play, Windows Phone Store) or on a website for users to download from,
2. Providing the software application or software system in the cloud (i.e., on a server computer) for users to subscribe to it and use it over the Internet (Software as a Service – SaaS),
3. Delivering the software application as a product in a shrink-wrapped box to the user or allowing the user to download it from a website as a product (Software as a Product – SaaP), or
4. Deploying the software system (e.g., cloud software, embedded software, real-time software) by incorporating it into a larger system such as emergency response management system, banking system, air traffic control system, or naval warfare system.

If the software application / system is developed under contract, acceptance testing is performed on the delivered / deployed software either by the customer itself or by a third party hired by the customer. The developer performs alpha testing and beta testing to prepare the software for the acceptance testing.

3.8 Software Application / System Use

As the saying goes “The only exhaustive software testing there is, is so much testing that the tester is exhausted!” Exhaustive (complete) software testing requires that the software be tested under all possible millions of logical paths in the execution of a large and complex software application / system. Due to time, budgetary, and technical constraints, it is impossible to test the accuracy of millions of logical paths. Therefore, it is essential to collect diagnostic data (e.g., errors, crashes, freezes) during the process of software application / system use so that corrective maintenance can be conducted to improve the software accuracy during its lifetime.

A software application / system provides many features. For the purpose of improving the software quality during its lifetime, it is necessary to know which of these features are most commonly used so that the maintenance efforts can be focused on those features.

Software vendors / developers typically instrument their software to automatically collect diagnostic and usage data and receive that data over the Internet after obtaining the user’s permission.

3.9 Feedback

Every successful software application / system evolves during its lifetime. Feedback obtained from the user community is essential for the evolution of the software and improvement of the software quality.

The process of feedback can be established in many different ways including (a) providing the “Send Feedback” menu item in the software for the user to use to send feedback, (b) user surveys, (c) user reviews, and (d) user interviews and observations.
3.10 Maintenance

The maintenance process, designated by a circle in the center of the software life cycle in Figure 2, starts right after the software is published and continues during the software’s lifetime until the software is retired. Every successful software evolves during its lifetime and new versions are published periodically.

The process of maintenance deals with modifications of the software application / system after being published to (a) cope with changes in the software’s external environment, (b) correct faults, (c) improve quality by satisfying new or changed user requirements, or (d) prevent problems in the future.

Four types of maintenance exist:

1. Adaptive maintenance: adaptations required as changes occur in the software’s external software or hardware (operational) environment (e.g., operating system, graphics engine, communication protocol, display size and resolution)
2. Corrective maintenance: fixing errors reported by the users and discovered by the developer.
3. Perfective maintenance: adding new capabilities, improving existing ones, or making functional enhancements identified by the developer or based on feedback received from the users.
4. Preventive maintenance: making modifications or reengineering so as to prevent potential future problems.

Regression testing plays an important role during maintenance. It seeks to ensure that correcting errors and/or making changes to the software do not create other errors. It is carried out by retesting the modified software with a set of test data, test cases, and test procedures that were used during the software development. Assertion checking is also known to be an effective testing technique to guard against errors created by software modifications.

3.11 Other Processes

In addition to the ten major processes of the software life cycle described in the previous sections, there exist many others including:

- Configuration Management
- Measurement and Analysis
- Organizational Training
- Project Management
- Project Monitoring and Control
- Project Planning
- Quality Assurance (QA)
- Risk Management
- Verification and Validation (V&V)

For a description of these and other processes, the reader is referred to Capability Maturity Model Integration for Development (CMMI-DEV) [SEI 2010]. Figure 7 shows the CMMI-DEV processes at five maturity levels in staged representation.
4. Concluding Remarks

A blueprint (detailed outline or plan of action) is commonly used for engineering a complex product. Since a software application / system is also very complex, a blueprint is required for its development. Such a blueprint is presented in this paper under a holistic framework in the form of a life cycle to provide guidance not only to software developers (engineers), but also to software project managers, software sponsoring organization, and others involved in the development of the software.

Modularization or decomposition stands out to be the best approach to employ when faced with significant complexity. The software life cycle presented herein is critically needed to modularize and structure the development of a large-scale complex software application / system. The life cycle enables the software project managers to decompose the overall complex work in terms of processes and work products and to identify areas of expertise in which to employ qualified people. Effective use of the software life cycle presented herein significantly increases the probability of success in a large-scale complex software application / system development project.

A large-scale complex software application / system development requires many areas of expertise as illustrated by the software life cycle. Many people can be employed with job titles such as analyst, problem formulation engineer, software requirements engineer, software architect, software design
engineer, software programmer, software integration engineer, software QA engineer, software V&V engineer, software project manager, and software configuration manager.

The software life cycle presented herein is applicable for software engineering projects of all sizes. A small software development project may skip some of the processes or conduct them in a shorter period of time. The presented life cycle framework is intended to provide guidance regardless of the software engineering project size. It is up to the user of the life cycle to decide how much of the guidance provided by the life cycle to employ.

References


Cooling, J. (2002), Software Engineering for Real-Time Systems, Addison-Wesley, Boston, MA.


