LUMINARY SOFTWARE

SOFTWARE DEVELOPMENT PROCESS

THE PROCESS BY WHICH
LUMINARY SOFTWARE
DEVELOPS SOFTWARE SYSTEMS

SOFTWARE DEVELOPMENT PROCESS

THE PROCESS BY WHICH LUMINARY SOFTWARE DEVELOPS SOFTWARE SYSTEMS

TABLE OF CONTENTS

DOCUMENT PURPOSE AND USE STATEMENT

This document is designed to communicate the software development process, which will organize and coordinate the efforts and interactions of all persons involved with developing a single software system. The process is written and presented as a series of phases, with each phase being comprehensively described. Supporting diagrams showing relationships and interactions are included along with any additional process forms or documents. This process was created particularly for smaller projects and therefore requires less formal by-products (see by-product section within each process phase description).

PROCESS SCOPE

This process begins once a *CONCEPTUALIZER*, having an idea or concept supported by a *CUSTOMER*, approaches a software development organization. This process concludes once a *CUSTOMER* is satisfied with the actual product produced by the efforts of the software development organization. This is a software development process. It is not a product planning process, a maintenance process nor a management process. Any development organization will have a network of interacting processes that guide the various activities required in an organization.

PROCESS OBJECTIVES

Through coordination and organization this process provides a predetermined path for the software development team, allowing the team to focus more completely on the task at hand, developing the software system.

PROCESS PARTICIPANT ROLES

The idea of having roles for different responsibilities required throughout the process provides a general structure for how different people will interact within the process. A single person or multiple persons can hold a role and a person can simultaneously hold many different roles. The following list of roles will be used to describe responsibilities in the different phases of the software development process. Since the person fulfilling the role at any given point may vary, no names are associated or recorded in this document.

- *CONCEPTUALIZER* – Person who has the original concept or idea for the project
- *CUSTOMER* – Person who will fund the project
- *USER* – Person who will utilize the system
- *DOMAIN EXPERT* – Person who is knowledgeable about the domain
- *MANAGER* – Person who is responsible for coordination and facilitation of the project
- *SYSTEMS ENGINEER* – Person who owns the requirements
- *ARCHITECT* – Person who owns the architecture
- *DEVELOPER* – Person who performs some set of analysis and development activities in the process
- *TESTER* – Person who performs some set of validation activities in the process

1. REQUIREMENTS SCOPING AND DEFINITION

1.1. Description

To capture and understand from a variety of perspectives "what" the application is that we are trying to create. Requirements aid in defining the scope of analysis activities and are the standards to which the final implementation is held accountable. The requirements, once clearly elaborated provide a solid foundation to build the software application. Requirements provide the needed insight regarding what the customer wants. The benefits of providing a complete set of requirements is the potential for fewer mistakes and less time spent developing the application. The requirements should be as complete as possible before beginning any of the other phases of the software development process.

1.2. Responsibility

- *CONCEPTUALIZER* needs to be available for clarification purposes.
- *CUSTOMER* needs to be informed about the usage of funds.
- *USER* needs to be available to clarify how all the users will utilize the application.
- *MANAGER* needs to be informed to be able to produce schedules, resource allocations, and the software development plan.
- *SYSTEM ENGINEER* takes the lead in soliciting customer and user input and is the producer and owner of the requirements.
- *ARCHITECT* needs to be involved to gather an understanding of the requirements that are placed on the architecture.
- *TESTER* guides assessment of the requirements ensuring they are testable.

1.3. Input

1.3.1. Actual

- Product idea or concept from the *CONCEPTUALIZER*
- Better understanding of product idea or concept from previous iterations, if applicable

1.3.2. By-product

- Business plan describing the concept or idea
- Revisions to the business plan based on experience from previous iterations, if applicable

1.4. Entry Criteria

Contractual obligation

1.5. Activities

- (Optional) Exploration and scoping of concept – Should result in sufficient knowledge to complete the second activity and may result in executable prototypes.
- Requirements writing – Use the Luminary Software use case template to capture the requirements.

1.6. Output

1.6.1. Actual

- An understanding of what the system should be

1.6.2. By-product

- Requirements

1.7. Exit Criteria

1.7.1. Proceed to next phase

- If an independent team of *SYSTEM ENGINEERS* can not identify any additional requirements

1.7.2. Iterate back to process

- PLANNING: If there is confusion about the scope of the requirements

1.8. Metrics

1.8.1. Raw data

- Number of actors
- Number of use cases

1.8.2. Derived measures

- Number of dependencies per use case
- Average depth of "extends" relations

2. DOMAIN ANALYSIS

2.1. Description

To capture the concepts and relationships within the bodies of knowledge that underlie the basic problem to be solved by the application. Operating within the scope defined by requirements, domain analysis provides a superset of concepts and relationships needed for application analysis.

2.2. Responsibility

- *CONCEPTUALIZER* assists in determining the domain boundaries and serves as a domain expert.
- *DOMAIN EXPERT* provides concepts and connections between the concepts.
- *MANAGER* reviews the output of this phase to be able to produce schedules, resource allocations, and the software development plan.
- *SYSTEM ENGINEER* provides validation during the assessment activity.
- *ARCHITECT* gains an understanding of the domain to construct the appropriate architecture.
- *DEVELOPER* takes the lead in soliciting domain expert input and defining and modeling the domain.
- *TESTER* guides the assessment of the domain analysis by-products ensuring that they are testable and non-contradictory.

2.3. Input

2.3.1. Actual

- An understanding of what the system should be
- The collective domain knowledge of those participating in the analysis

2.3.2. By-product

- Business plan describing the concept or idea and the requirements

2.4. Entry Criteria

Agreement has been reached that the requirements are sufficiently scoped to begin this phase.

2.5. Activities

- Domain Modeling using UML models.
- Guided inspection of the domain model.

2.6. Output

2.6.1. Actual

- An understanding of specific domains related to the problem being solved.

2.6.2. By-product

- A UML model capturing the concepts and relationships
- Mapping between the domain model concepts and the use case model

2.7. Exit Criteria

2.7.1. Proceed to next phase

- If the results of the guided inspection show the domain model is complete, correct, and consistent
- If the requirements model is still sufficiently complete

2.7.2. Iterate back to previous phases

- REQUIREMENTS: If domain concepts have been discovered that lead to the discovery of new uses of the system

2.8. Metrics

2.8.1. Raw data

- Number of concepts (classes) captured in model

2.8.2. Derived measures

- Percentage of use cases covered by concepts in the domain model

3. APPLICATION ANALYSIS

3.1. Description

To create an understanding of what this specific application will become using the ongoing concepts and relationships from domain analysis. The set of concepts and relationships identified is the basis for further elaboration during the architecture and application design phases.

3.2. Responsibility

- *CONCEPTUALIZER* provides validation during the assessment activity.
- *DOMAIN EXPERT* continues to provide concepts and connections between the concepts.
- *MANAGER* reviews the output of this phase to be able to produce schedules, resource allocations, and the software development plan.
- *SYSTEMS ENGINEER* provides validation during the assessment activity.

- *DEVELOPER* takes the lead in soliciting domain expert input and defining and modeling the application.
- *TESTER* guides the assessment of the application analysis by-products ensuring that they are testable and non-contradictory.

3.3. Input

3.3.1. Actual

- An understanding of the domains in which the application resides and a basic idea of the requirements for the application

3.3.2. By-product

- Domain analysis UML model
- Requirements model

3.4. Entry Criteria

Requirements model verified and domain analysis model stable but not necessarily verified

3.5. Activities

- Domain model is pruned to remove classes not needed by the current application
- Multiple domain models are integrated
- Class specifications are made more detailed

3.6. Output

3.6.1. Actual

- An understanding of the problem to be solved and how the software is expected to solve that problem

3.6.2. By-products

- Application analysis UML model
- Mapping between concepts and use cases

3.7. Exit Criteria

3.7.1. Proceed to next phase

- If the results of the guided inspection show the application model is complete, correct, and consistent
- If only minor changes are required for the use cases, such as new alternative paths

3.7.2. Iterate back to the previous phases

- REQUIREMENTS: if examining the specific problem revealed significant incompleteness in the use cases for the system
- DOMAIN ANALYSIS: if a specific problem feature is found that relates to a more general domain concept that has not been included in the domain model

3.8. Metrics

3.8.1. Raw data

- Number of classes in model

3.8.2. Derived measures

- Average number of relationships between classes (coupling)
- Average cohesion within classes

4. ARCHITECTURE

4.1. Description

To determine the basic structure of the application components. This structure must be compatible with the structure of knowledge as revealed in domain analysis and must define sufficient functionality to allow the application to satisfy the requirements. The structure of the architecture provides interfaces to which the individual components must conform.

4.2. Responsibility

- *MANAGER* reviews the output of this phase to be able to produce schedules, resource allocations, and the software development plan.
- *ARCHITECT* takes the lead in defining and modeling the architecture.
- *DEVELOPER* provides validation during the assessment activity to ensure that the architecture is able to be implemented.
- *TESTER* guides the assessment of the architecture ensuring that it completely meets the requirements.

4.3. Input

4.3.1. Actual

- Understanding of the general problem to be solved and the environment within which the application will be sited

4.3.2. By-products

- Application analysis model
- Hardware system descriptions

4.4. Entry Criteria

Requirements are basically known but not necessarily verified

4.5. Activities

- Constraints are identified
- Functional subsystems are identified
- Standard architectural patterns are searched and the most appropriate ones are selected
- Models of the architecture are created and analyzed

4.6. Output

4.6.1. Actual

- A basic idea of how the fundamental structure of the application will connect the pieces of the system to each other

4.6.2. By-products

- A UML model describing the architectural structure
- Text document describing constraints on architecture

4.7. Exit Criteria

4.7.1. Proceed to next phase

- If the results of the guided inspection show the architecture completely satisfies the system qualities specified in the production plan, correctly represents the concepts

identified in the analysis model and is consistent with the constraints imposed by the domain model and use cases

4.7.2. Iterate back to previous phases

- REQUIREMENTS: if examining the structure of the system revealed significant new uses of the system.
- DOMAIN ANALYSIS: if developing the high-level system design identified the need for a conceptual basis for an interface
- APPLICATION ANALYSIS: if creating the system structure raised questions about the specific problem to be solved or if the structure required concepts not identified in the analysis model

4.8. Metrics

4.8.1. Raw data

- Individual measures required by the application

4.8.2. Derived measures

- An analytic model of the architecture that presents the architecture in a quantified form

5. APPLICATION DESIGN

5.1. Description

To describe in detail the internals of each of the application components. This design must be compatible with the interfaces defined in the architecture. The component designs should be sufficiently detailed to support immediate implementation.

5.2. Responsibility

- *MANAGER* reviews the output of this phase to be able to produce schedules, resource allocations, and the software development plan.
- *ARCHITECT* provides validation during the assessment activity verifying that the architecture is followed.
- *DEVELOPER* takes the lead in defining and modeling the detailed application design.
- *TESTER* guides the assessment of the detailed application design ensuring that it incorporates the application analysis.

5.3. Input

5.3.1. Actual

- A basic structure for the system
- An understanding of the problem
- An understanding of what the application must be able to do

5.3.2. By-products

- Architectural, application analysis and requirements models
- Standard patterns of object interactions

5.4. Entry Criteria

Architectural model stable but not necessarily verified, requirements model verified

5.5. Activities

- Standard class design patterns are searched and the appropriate ones selected

- Each interface defined in the architecture is elaborated

5.6. Output

5.6.1. Actual

- An understanding of the responsibilities of each class
- An understanding of how the relationships between classes provide the basic algorithms

5.6.2. By-products

- A UML model that combines the architectural model and the application analysis model and supplements it with classes that support the chosen solution

5.7. Exit Criteria

5.7.1. Proceed to the next phase

- If the results of the guided inspection show the detailed design completes the skeleton provided by the architecture and is consistent with the requirements specified in the use cases and correctly solves the problem specified in the analysis model

5.7.2. Iterate back to previous phases

- REQUIREMENTS: if developing the details of the design revealed new uses of the system.
- DOMAIN ANALYSIS: if developing the detailed design identified the presence of a previously unidentified concept, often meta-level concepts are identified at this level.
- APPLICATION ANALYSIS: if creating the detailed design raised questions about the specific problem to be solved such as specific limits on attributes.
- ARCHITECTURE: if filling in the details of the architecture revealed contradictions between constraints, incorrect relationships or inconsistent treatment of similar constructs.

5.8. Metrics

5.8.1. Raw data

- Class counts
- Dependencies between classes
- Number of design patterns used

5.8.2. Derived measures

- Coupling and cohesion of model, similar to the analysis model

6. PROGRAM IMPLEMENTATION

6.1. Description

To translate information and concepts from domain analysis, architecture, and application design into a machine-executable form. The executable machine code is run using a variety of inputs to determine its correctness.

6.2. Responsibility

- *MANAGER* reviews the output of this phase to be able to produce schedules, resource allocations, and the software development plan.
- *DEVELOPER* takes the lead in implementing the detailed application design.
- *TESTER* guides the assessment of the implementation ensuring that it meets the specifications described in the detailed application design.

6.3. Input

6.3.1. Actual

- An understanding of the details of the specified interfaces

6.3.2. By-product

- Application design model
- Standard language idioms

6.4. Entry Criteria

Detailed design model verified

6.5. Activities

- Language idioms are applied to standard pieces of design

6.6. Output

6.6.1. Actual

- An understanding of the performance and other implementation issues

6.6.2. By-products

- Working code for the application

6.7. Exit Criteria

6.7.1. Proceed to next phase

- If the results of the class tests show a set of classes whose implementations are consistent with the designs and that produce correct answers

6.7.2. Iterate back to previous phases

- REQUIREMENTS: if writing the code revealed new uses of the system.
- DOMAIN ANALYSIS: if developing the implementation of a class led to the identification of variants of a domain concept.
- APPLICATION ANALYSIS: if creating class implementations identified special cases that should be broken into separate classes
- ARCHITECTURE: if writing the code identified incomplete interfaces or inconsistencies between interfaces.
- DETAILED DESIGN: if the class specifications are incomplete or writing the code has identified inconsistencies between related method specifications.

6.8. Metrics

6.8.1. Raw data

- Lines of code

6.8.2. Derived measures

- Program complexity measures

7. INTEGRATION

7.1. Description

To verify program implementation in the context of domain analysis, architecture, and application design, by clearly identifying interface ambiguities and implementation errors. After the integration phase, the program implementation can be further judged against the requirements during the system test phase.

7.2. Responsibility

- *MANAGER* schedules increments and their integration
- *DEVELOPER* makes adjustments to code in new and existing increments to make them interoperate
- *TESTER* executes tests of the functionality of the newly integrated functionality

7.3. Input

7.3.1. Actual

- A view of how the increments should fit together

7.3.2. By-products

- Application code
- Increment plan

7.4. Entry Criteria

A second increment has been verified

7.5. Activities

- Classes in the increment are added to the system build
- Minor problems are fixed and code retested

7.6. Output

7.6.1. Actual

- A view of dependencies between subsystems

7.6.2. By-products

- A larger integration of code

7.7. Exit Criteria

7.7.1. Proceed to next phase

- If the results of the integration tests show a set of subsystems whose behaviors are consistent with the designs and that produce correct answers

7.7.2. Iterate back to previous phases

- *REQUIREMENTS*: if integrating the subsystems revealed new ways in which the system can be used.
- *DOMAIN ANALYSIS*: if integrating the subsystems identified inconsistencies with domain concepts
- *APPLICATION ANALYSIS*: if writing the expected results for integration tests identified incorrect underlying assumptions about the problem
- *ARCHITECTURE*: if failures of the integration tests identified incomplete interfaces or inconsistencies between interfaces
- *DETAILED DESIGN*: if the integration reveals inconsistencies between class specifications
- *PROGRAM IMPLEMENTATION*: if failures of integration tests identify failures of specific component implementations

7.8. Metrics

7.8.1. Raw data

- Test failures

7.8.2. Derived measures

- Defect yield

8. SYSTEM TEST

8.1. Description

To verify that the program as assembled meets the requirements. The implementation has been previously tested during the integration of multiple increments. If all of the requirements are adequately satisfied, the application is ready to be placed in service.

8.2. Responsibility

- *DOMAIN EXPERT* validates the expected results listed in test cases.
- *MANAGER* sets levels of quality that are the stopping criteria for testing.
- *SYSTEMS ENGINEER* responds to requests for clarifications from testers.
- *DEVELOPER* responds to error reports for minor fixes.
- *TESTER* creates and executes tests based on requirements.

8.3. Input

8.3.1. Actual

- A strategy for determining what is most important to test

8.3.2. By-products

- A completely integrated set of code; requirements model

8.4. Entry Criteria

A sufficient set of increments have been integrated to provide end user functionality

8.5. Activities

- Test cases are derived from the requirements model
- Data is collected on the types of defects identified during testing
- Recommendations are made for improving the development process

8.6. Output

8.6.1. Actual

- An identified set of areas that need repair; a set of commonly made errors

8.6.2. By-products

- Tested code

8.7. Exit Criteria

8.7.1. Proceed to next phase

- If the results of the system tests show system uses are completely supported

8.7.2. Iterate back to previous phases

- REQUIREMENTS: if writing system tests revealed contradictions between the ways in which the system can be used or incomplete definitions of uses so that tests can not be written.
- DOMAIN ANALYSIS: if writing system tests identified incomplete definitions of domain concepts

- APPLICATION ANALYSIS: if writing the expected results in system tests identified incorrect underlying assumptions about the problem
- ARCHITECTURE: if failures of the system tests identified incomplete interfaces or inconsistencies between interfaces
- DETAILED DESIGN: if the system tests reveal inconsistencies between class specifications
- PROGRAM IMPLEMENTATION: if failures of system tests identify incorrect results that differ from the specifications
- INTEGRATION: if there is a significant correlation between system test failures and a specific set of subsystems

8.8. Metrics

8.8.1. Raw data

- Defect counts
- Severity levels
- Test cases per use case

8.8.2. Derived measures

- Defect yield

9. RELEASE

9.1. Description

To checkpoint the files that have passed system test and to package all of the information required for deployment of the product.

9.2. Responsibility

- *MANAGER* determines the platforms to target.
- *DEVELOPER* uses an installation package to package the system.
- *TESTER* tests the deployment package.

9.3. Input

9.3.1. Actual

- A product concept

9.3.2. By-products

- Tested system

9.4. Entry Criteria

Identified set of requirements has been implemented and tested

9.5. Activities

- Deployment testing
- Acceptance testing
- Document clean up and finalization

9.6. Output

9.6.1. Actual

- A completed deployment product

9.6.2. By-product

- An automatic deployment package

9.7. Exit Criteria

9.7.1. Proceed to next phase

- If the results of the deployment tests, alpha and beta test sites contain no "show stopper" failures

9.7.2. Iterate back to previous phases

- REQUIREMENTS: if deployment revealed new ways in which the system can be used.
- DOMAIN ANALYSIS: if users have identified domain concepts whose definitions are too narrow
- ARCHITECTURE: if failures in the field identified incomplete interfaces; inconsistencies between interfaces; failures to meet quality standards
- DETAILED DESIGN: if a significant number of failures in the field identify failure of specific component implementations to produce appropriate answers
- PROGRAM IMPLEMENTATION: if a significant number of failures in the field identify failure of specific component implementations to produce correct answers
- INTEGRATION: if failures in the field can be traced to mismatched interfaces
- SYSTEM TEST: if sufficient deployment tests fail, analyze the system test suite to determine need for additional test cases

9.8. Metrics

9.8.1. Raw data

- Defects identified in the field

9.8.2. Derived measures

- Reliability measure
- Availability measure

PROCESS CONTEXT

The Product Development Process is the macro process primarily concerned with the interactions between each of the micro processes. This allows the smaller internal processes to focus on a specific task. The Software Development Process is just one of several micro processes where the development team participates. This document is focused on defining the Software Development Process.

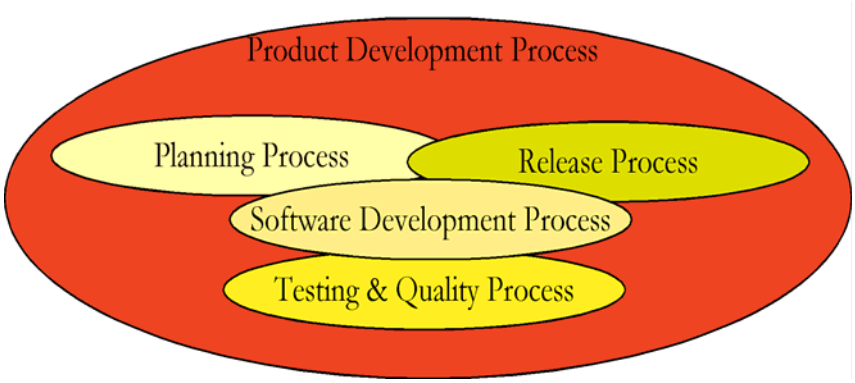


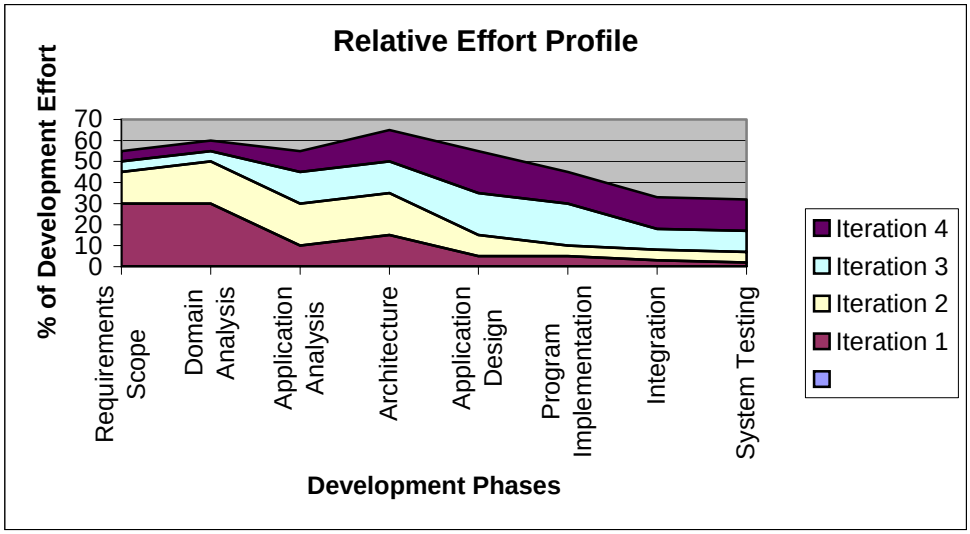
Figure 1Process Context

The typical increments for a project include:

- Infrastructure increment
- Most widely "used" use cases and abstract use cases
- Specific concrete use cases, "uses" use cases required by the concrete use cases (may be more than one increment)
- Remaining "extends" use cases for already implemented concrete use cases (may be more than one increment)

The typical iterations for each increment is

- Planning and requirements prototyping
- Implement "sunny day" scenarios (may be more than one iteration)
- Implement "rainy day" scenarios (may be more than one iteration)
- Final increment testing and repair



The time spent in each phase of the process changes from one iteration to the next. The cumulative effort, as evidenced by the height of the top line shows the importance of the architecture phase. The shift in emphasis from coding and system test to analysis and design can also be observed.

SOFTWARE DEVELOPMENT PROCESS FLOW - DEPENDENCIES AMONG PHASES