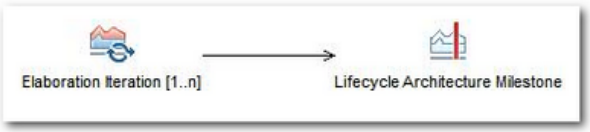
.

.

2.3 #phase# Elaboration Phase

- #iteration# Elaboration Iteration [1..n]
- 2.3.1 #activity# Plan and Manage Iteration
  - 2.2.2.1 #task# Plan Iteration
  - 2.2.2.2 #task# Manage Iteration
  - 2.3.1.3 #task# Outline Deployment Plan
  - 2.3.1.4 #task# Assess Results (same as 2.2.2.3)
- 2.3.2 #activity# Identify and Refine Requirements (same as 2.2.4)
  - 2.2.4.1 #task# Identify and Outline Requirements
  - 2.2.4.2 #task# Detail Use-Case Scenarios
  - 2.2.4.3 #task# Detail System-Wide Requirements
  - 2.2.4.4 #task# Create Test Cases
- 2.3.3 #activity# Develop the Architecture
  - 2.3.3.1 #activity# Develop Solution Increment (same as 2.3.4)
  - 2.3.3.2 #task# Refine the Architecture
- 2.3.4 #activity# Develop Solution Increment
  - 2.3.4.1 #task# Design the Solution
  - 2.3.4.2 #task# Implement Developer Tests
  - 2.3.4.3 #task# Implement Solution
  - 2.3.4.4 #task# Run Developer Tests
  - 2.3.4.5 #task# Integrate and Create Build
- 2.3.5 #activity# Test Solution
  - 2.3.5.1 #task# Implement Tests
  - 2.3.5.2 #task# Run Tests
- 2.3.6 #activity# Ongoing Tasks
  - 2.3.6.1 #task# Request Change
- 2.3.7 #activity# Prepare Environment (same as 2.2.3)
  - 2.2.3.1 #task# Tailor the Process
  - 2.2.3.2 #task# Set Up Tools
  - 2.2.3.3 #task# Verify Tool Configuration and Installation
  - 2.2.3.4 #task# Deploy the Process

WBS

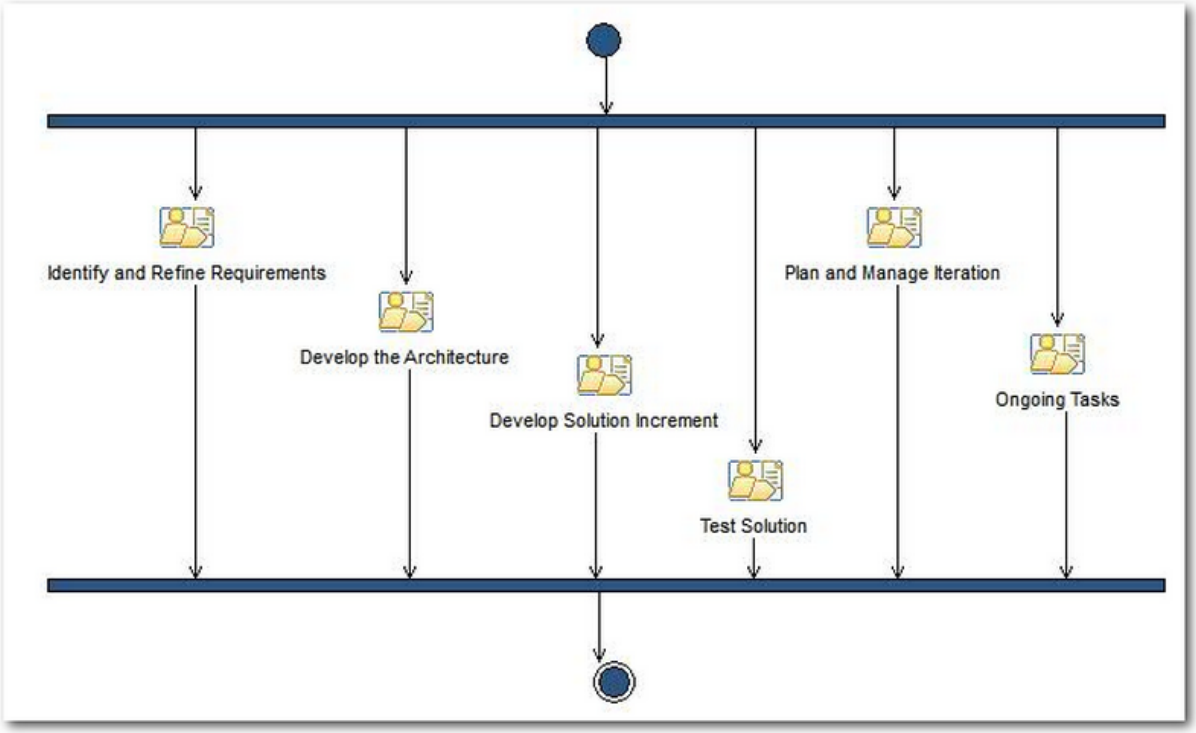


.

.

#iteration# Elaboration Iteration [1..n]

WBS



Visual Paragigm Online Link

Summary

This iteration template defines the activities (and associated roles and work products) performed in a typical iteration in the Elaboration phase.

Description
Most activities during a typical iteration in Elaboration phase happen in parallel. Essentially, the main objectives for Elaboration are related to better understanding the requirements, creating and establishing a baseline of the architecture for the system, and mitigating top-priority risks.

The following table summarizes the Elaboration phase objectives and what activities address each objective:

Phase objectives	Activities that address objectives
Get a more detailed understanding of the requirements	• **Identify and Refine Requirements**
Design, implement, validate, and baseline an architecture	• **Develop the Architecture** • **Develop Solution Increment** • **Test Solution**
Mitigate essential risks, and produce accurate schedule and cost estimates	• **Plan and Manage Iteration**

Alternatives
There will be iterations when projects risks are being addressed by creating software, but they may not be architecturally significant. In this case, Develop Solution Increment will be performed outside the context of the architecture. For

the most part, Develop Solution will be performed in the context of Develop the Architecture during the Elaboration phase.
.

.
2.3.1 #activity# Plan and Manage Iteration

Summary

Initiate the iteration and allow team members to sign up for development tasks. Monitor and communicate project status to external stakeholders. Identify and handle exceptions and problems.

Description
This activity is performed throughout the project lifecycle. The goal of this activity is to identify risks and issues early enough that they can be mitigated, to establish the goals for the iteration, and to support the development team in reaching these goals.

The project manager and the team launch the iteration. The prioritization of work for a given iteration takes place. The project manager, stakeholders, and team members agree on what is supposed to be developed during that iteration.

Team members sign up for the work items they will develop in that iteration. Each team member breaks down the work items into development tasks and estimates the effort. This provides a more accurate estimate of the amount of time that will be spent, and of what can be realistically achieved, in a given iteration.

As the iteration runs, the team meets regularly to report status of work completed, the work to do next, and issues blocking the progress. In some projects, this status checking occurs in daily meetings, which allows for a more precise understanding of how the work in an iteration is progressing. As necessary, the team makes corrections to achieve what was planned. The overall idea is that risks and issues are identified and managed throughout the iteration, and everyone knows the project status in a timely manner.

During iteration assessments, the key success criterion is the demonstration that planned functionality has been implemented. Lessons learned are captured in order to modify the project or improve the process. If the iteration end coincides with the phase end, make sure the objectives for that phase have been met (see Concept: Phase Milestones for more information).
.

.
2.3.1.3 #task# Outline Deployment Plan

Summary
If a deployment plan for the project already exists, update it to reflect the nature of this release. If this document does not exist, develop a deployment plan to indicate how this release will be rolled out to the production environment.

Purpose
The purpose of this task is to ensure that the various aspects of deploying a release to production are considered and understood between the development team and the deployment engineer.

Relationships.

.

Roles	Inputs	Outputs
Primary: • #role# Deployment Engineer	Mandatory: • None	• #workproduct# Deployment Plan
Additional: • #role# Developer	Optional: • None	
Assisting:	External: • None	

.

Main Description

Because a deployment engineer is responsible for accepting the results of one or more development team members and deploying those integrated releases into the production environment, it is important for all parties to agree on the details of a particular release. The deployment plan documents, in one place, all the information that will be consumed by the deployment engineer before and during the deployment to production of a particular release package.

Steps

.

- **Determine if deployment plan exists**

  Determine whether the development team has a deployment plan already written for a previous release. If so, part of that plan might be reusable. If this is the development team's first release, another development team with a similar feature set might have a plan that can be used as a starting point.

- **Develop the deployment plan (if applicable)**

  If a deployment plan does not exist, or one cannot be found to be used as a starting point, use the recommended format documented in the #workproduct# Deployment Plan and refer to the **#checklist# Deployment Plan** to start and develop a deployment plan.

- **Update the deployment plan (if applicable)**

  If a deployment plan does exist that can be used as a baseline, review that plan and update, add, or delete information as necessary. When the plan is done, it should reflect entirely the contents of the upcoming deployment only, not a release in the past or one in the future. In other words, the deployment plan should be specific only to **this release**.

More Information

- **#checklist# Deployment Plan**

.

.

2.3.3 #activity# Develop the Architecture

Summary
Develop the architecturally significant requirements prioritized for this iteration.

Description
This activity refines the initial high-level architecture into working software. The objective is to produce stable software that adequately addresses the technical risks in scope.

The architect and developers work together to:

- Refine the initial sketch of the architecture into **concrete design elements**.
- Ensure that the **architecture decisions** are adequately captured and communicated.
- Ensure that the team has enough information to enable software to be developed.
- Ensure that the requirements that were prioritized for the current iteration are adequately addressed in the software.

In an iterative project, the team should not attempt to develop the architecture for the entire project in a single pass. Rather, they should **focus on meeting the requirements in scope for the current iteration**, while **making decisions in the context of the wider project**.

.

.

2.3.3.2 #task# Refine the Architecture

Summary
Refine the architecture to an appropriate level of detail to support development.

Purpose
To make and document the architectural decisions necessary to support development.

Relationships.
.

Roles	Inputs	Outputs
Primary: • #role# Architect	Mandatory: • #workproduct# [Technical Specification] • #workproduct# Architecture Notebook	• #workproduct# Architecture Notebook
Additional: • #role# Developer • #role# Project Manager	Optional: • [Technical Design] #workproduct# • #workproduct# [Technical Implementation]	

Assisting:	External:	
•	• None	

.

Main Description

This task builds upon the outlined architecture and makes concrete and unambiguous architectural decisions to support development. It takes into account any design and implementation work products that have been developed so far. In other words, **the architecture evolves as the solution is designed and implemented, and the architecture documentation is updated to reflect any changes made during development. This is a key, since the actual implementation is the only real "proof" that the software architecture is viable and provides the definitive basis for validating the suitability of the architecture**. For more information, see **#concept# Executable Architecture**.

The results are captured for future reference and are communicated across the team.

Steps

- **Refine the architectural goals and architecturally-significant requirements**
  Work with the team, especially the stakeholders and the requirements team, to review the status of the Architectural Goals and Architecturally Significant Requirements and refine them as needed. It may be that some new architecturally-significant requirements have been introduced or your architectural goals and priorities may have changed.

  The development work performed so far will also inform the decisions and goals you've identified. Use information from designing and implementing the system so far to adjust and refined those decisions and goals.

  See **#concept# Architectural Goals** and **#concept# Architecturally Significant Requirements**

- **Identify architecturally significant design elements**
  Identify concrete design elements (such as **Components, classes and subsystems**) and provide at least a name and brief description for each.

  The following are some good sources for design elements:
  - **Architecturally Significant Requirements.** Highlight the areas of the architecture that participate in realizing, or implementing, the requirements.
  - **Key Abstractions.**
  - Components that encapsulate the **system's interface with external systems.** For more information, see **#guideline# Representing Interfaces to External Systems**
  - Components that implement the **Architectural Mechanisms.**
  - **Architectural and key design Patterns.** Apply the chosen patterns to define a new set of elements that conform to the patterns.

  Identifying components will help hide the complexity of the system and help you work at a higher level. Components need to be internally cohesive and to provide external services through a limited interface. At this point, interfaces do not need to be as detailed as a signature, but they do need to

document what the elements need, what they can use, and what they can depend on.

Component identification can be based on architectural layers, deployment choices, or key abstractions. Ask yourself these questions:

- What is logically or **functionally related** (same use case or service, for example)?
- What entities **provide services** to multiple others?
- What entities **depend on each other**? Strongly or weakly?
- What entities should you be able to **exchange independently** from others?
- What will run on the **same processor or network node**?
- What parts are constrained by **similar performance requirements**?

When you identify a component be sure to briefly describe the functionality that should be allocated to the components.

- **Refine architectural mechanisms**
  Refine the applicable Architectural Mechanisms, as needed to support the design. For example, refining an **analysis mechanism** into a **design mechanism** and/or refining a design mechanism into an **implementation mechanism.**

- **Define development architecture and test architecture**
  Ensure that the development and test architectures are defined. Note any architecturally significant differences between these environments and work with the team to devise strategies to mitigate any risks these may introduce.

- **Identify additional reuse opportunities**
  Continue to look for more opportunities to reuse existing assets. Where applicable, identify existing components that could be built to be reused.

  For more information, see **#guideline# Software Reuse.**

- **Validate the architecture**
  Make sure that the architecture supports the requirements and the needs of the team.

  Development work should be performed to produce just enough working software to show that the architecture is viable. This should provide the definitive basis for validating the suitability of the architecture. As the software should be developed iteratively, more than one increment of the build may be required to prove the architecture. During the early stages of the project it may be acceptable for the software to have a incomplete or prototypical feel, as it will be primarily concerned with baselining the architecture to provide a stable foundation for the remaining development work.

- **Map the software to the hardware**
  Map the architecturally significant design elements to the target deployment environment. Work with hardware and network specialists to ensure that the hardware is sufficient to meet the needs of the system; and that any new hardware is available in time.

- **Communicate decisions**
  Ensure that those who need to act upon the architectural work understand it and are able to work with it. Make sure that the description of the architecture clearly conveys not only the solution but also the motivation

and objectives related to the decisions that have been made in shaping the architecture. This will make it easier for others to understand the architecture and to adapt it over time.

Key Considerations
It is important to continue to reduce the complexity of the solution by raising the levels of abstraction. For more information, see **#guideline# Abstract Away Complexity.**

Continue the collaboration with the whole team on the refining of the architecture in order to promote consensus and a common understanding of the overall solution. **The architect should be working to coordinate and guide the technical activities of the team rather than doing all the work alone.** Place special emphasis on involving the developer(s) throughout this task since it's the developed solution that will prove out the architecture and may result in refinements to the architecture documentation.

Ensure that those who need to act upon the architectural work understand it and are able to work with it. **Make sure that the description of the architecture clearly conveys not only the solution but also the motivation and objectives related to the decisions that have been made in shaping the architecture. This will make it easier for others to understand the architecture and to adapt it over time.**

You can communicate your decisions as many ways as you wish. For example:
- Publication of reference source code
- Publication of reference models
- Publication of software architecture documentation
- Formal presentations of the material
- Informal walkthroughs of the architecture

As you evolve the architecture, you may wish to evolve your architectural models. For more information, see **#guideline# Modeling the Architecture.**
.

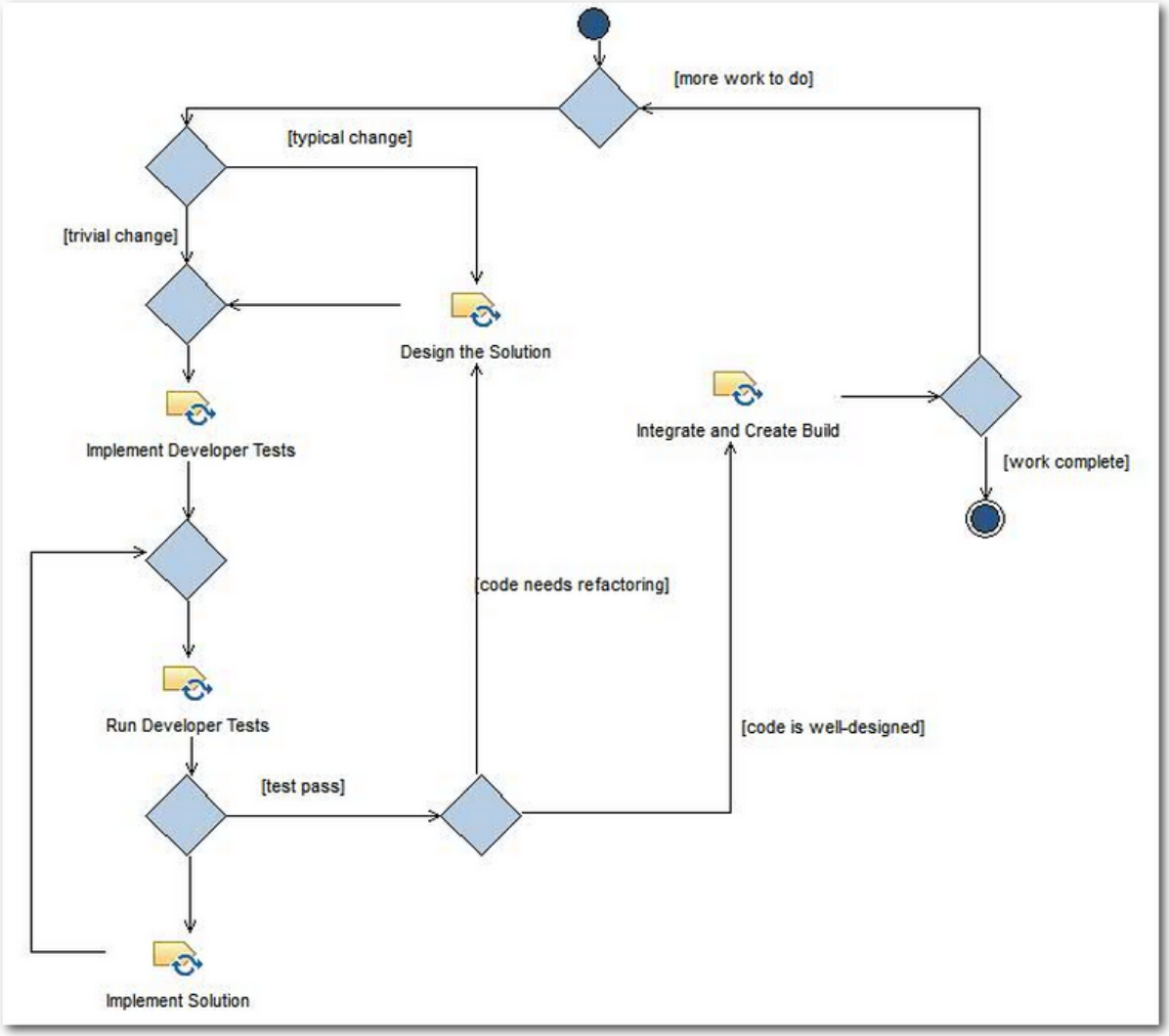
More Information

- **#concept# Architectural Constraints**
- **#concept# Architectural Goals**
- **#concept# Architecturally Significant Requirements**
- **#concept# Architectural Mechanism**
- **#concept# Component**
- **#concept# Executable Architecture**
- **#concept# Key Abstractions**
- **#guideline# Abstract Away Complexity**
- **#guideline# Modeling the Architecture**
- **#guideline# Representing Interfaces to External Systems**
- **#guideline# Software Reuse**

.

.
2.3.4 #activity# Develop Solution Increment

WBS



Summary
Design, implement, test, and integrate the solution for a requirement within a given context.

Purpose
For developers: To create a solution for the work item for which they are responsible
For project managers: To have a goal-based way of tracking project status

Description

- **Introduction**

  Run this activity as a way to perform goal-based planning and execution. Work is taken on by developers, and work progress is tracked based on the goals achieved using the designed, developer-tested, and integrated source code.

- **Context of what is being developed**

  A context can be specified when a requirement is assigned to be developed, thus specifying how broadly a requirement is to be developed in an iteration. Development may focus on a layer (such as the user interface, business logic, or database access), on a component, and so on.

Whether a context is specified or not, the developer's responsibility is to create a design and implementation for that requirement. The developer also writes and runs developer tests against the implementation to make sure that it works as designed, both as a unit and integrated into the code base.

- **Overview of workflow**

  Typical changes require some effort in designing the solution before moving into implementation, even if it is only a mental exercise that results in no long-term work product. The design for trivial changes to the existing implementation (to, for example, support some requirement) might be self-evident in the context of the existing architecture and design.

  Once the organization of the technical solution is clear, define developer tests that will verify the implementation. This test-driven approach ensures that design considerations have in fact taken place before the solution is coded. The tests are run up front and, if they fail, clearly define the criteria to determine if the implementation works as intended.

  Failed tests lead to an implementation of the solution, upon completion of which you run the tests again. This innermost loop of implementation and developer testing is repeated until the tests pass.

  Passing the tests does not necessarily mean that the solution is a high-quality, appropriate solution. It is proper to revisit the design at this point. That path loops back through the process, since any changes to the design could affect the developer tests and implementation.

  Once the tests pass and the design of the solution is appropriate, there is one more possible loopback. It is best to keep the test-driven, evolutionary design inner loops as tight as possible. Come up with some small-scale design solution for a part of the work item, define a test or two for the implementation of that one part of the solution, pass that test, verify the quality, and then continue on in a test-first manner until that part of the design is working. Then, in the outermost loop of the activity, go back to the work item and design another chunk to get closer to completion.

.

.

2.3.4.1 #task# Design the Solution

Summary
Identify the elements and devise the interactions, behavior, relations, and data necessary to realize some functionality.

Render the design visually to aid in solving the problem and communicating the solution.

Purpose
The purpose of this task is to describe the elements of the system so that they support the required behavior, are of high quality, and fit within the architecture.

Relationships.
.

Roles	Inputs	Outputs

Primary: • #role# Developer	Mandatory: • #workproduct# [Technical Architecture] • #workproduct# [Technical Specification]	• #workproduct# Design
Additional: • #role# Analyst • #role# Architect • #role# Stakeholder • #role# Tester	Optional: • #workproduct# Design	
Assisting:	External: • None	

.
Main Description
This task is about designing part of the system, not the whole system. It should be applied based upon some small subset of requirements. The requirements driving the design could be scenario-based functional requirements, non-functional requirements, or a combination.

This task can be applied in some specific context such as the database access elements required for some scenario. In this case the task might be applied again later to deal with a different context on the same requirements. Keep in mind that to actually build some functionality of value to the users, all contexts will typically need to be designed and implemented. For example, to actually utilize some system capability it will have to have been designed and implemented all its context such as user interface, business rules, database access, etc.

For cohesion and completeness, this task is described as an end-to-end pass of designing a scenario of system usage. In practice, this task will be revisited many times as the design is first considered, portions are implemented, more design is performed based on what was learned, etc. The healthiest application of this task is in very close proximity to the implementation.
If this task is being performed on an architecturally significant element the results of this design should be referenced by the #workproduct# [Technical Architecture].

Steps

- **Understand requirement details**

  Examine the relevant [Technical Specification] to understand the scope of the design task and the expectations on the Design. Work with the stakeholder and Analyst to clarify ambiguous or missing information.

  If the requirements are not represented in some sort of scenario form (for example a non-functional requirement might not have a scenario associated

with it), a scenario will have to be identified that appropriately exercises the requirements under consideration.

If the requirements are determined to be incomplete or incorrect, work with the analyst to get the requirements improved and possibly submit a change request against the requirements.

- **Understand the architecture**

  Review the **Architecture Notebook** to identify changes and additions to the architecture. See Guideline: Evolve the Design for more information. Work with the architect if there is any uncertainty on the understanding of relevant parts of the architecture or of the conformance of the design strategy.

  This step can be skipped if there were no changes to the architecture in the previous iteration

- **Identify design elements**

  Identify the elements that collaborate together to provide the required behavior. This can start with the key abstractions identified in the Architecture Notebook, design, domain analysis, and classical analysis of the requirements (noun filtering) to derive the elements that would be required to fulfill them. The **#guideline# Entity-Control-Boundary Pattern** provides a good start for identifying elements. Also see **#guideline# Analyze the Design.**

  Existing elements of the design should be examined to see if they should participate in the collaboration. It is a mistake to create all new elements in each execution of this task.

- **Determine how elements collaborate to realize the scenario**

  Walk through the scenario distributing **responsibilities** to the participating elements and ensuring that the elements have the **relationships** required to collaborate.

  These **responsibilities** can be simple statements of behavior assigned to elements; they need not be detailed operation specifications with parameters, etc. Similarly, the **relationships** can just be defined at this step. This step is about ensuring that a quality model is being created that is robust enough to support the requirements. See **#guideline# Analyze the Design.**

  Look to the architecture and previous design work to create a consistent collaboration. Work with the architect to understand the details and motivations of the architecture. **Look to reuse existing behavior and relations or to apply similar structure to simplify the design of the overall system.** For more information, see **#guideline# Software Reuse.**

- **Refine design decisions**

  Refine the design to an appropriate level of detail to drive implementation and to ensure that it fits into the architecture. In this step the design

can take into consideration the actual implementation language and other technical decisions. Revisit the identification of the elements and the collaborations that realize the scenarios if necessary as this refinement takes into consideration details at a lower level of abstraction. Discuss testability issues, such as design elements that are difficult to test or critical performance areas, with the tester and architect.

Evolve the design by examining recent changes in the larger context of the design and determine if refactoring and redesigning techniques will improve the robustness, flexibility, and understandability of the design.
See **#guideline# Evolve the Design** for guidance specific design decisions and on making design improvements just when they're needed.

Incorporate **#concept# Architectural Mechanisms** from the architecture. Apply consistent structure of the elements and organization of the behavior as in other areas of the design and use patterns identified in the architecture.

- **Design internals (for large or complex elements)**

  Design large or complex elements or some complex internal behavior in more detail. This might just involve devising an **algorithm** that could be performed to produce the desired behavior. Add additional **operations**, **attributes**, and **relationships** to support the expectations of an element.

  Design the state of the element over the course of its lifetime to ensure its proper behavior in various circumstances. It may be useful to describe a state machine for elements with complex states.

- **Communicate the design**

  Communicate the system's design to those who need to understand it. Though this is described here toward the end of the task, communication should be going on throughout the steps. Working collaboratively is always better than reviewing the work after it is complete. Here are some ways to communicate the design:

  - Formal models specified in **UML**.
  - Informal **diagrams** that render static structure and capture dynamic behavior.
  - Annotated code that communicates information about the static structure. This can be supplemented with textual descriptions of collaborative behavior across code modules.
  - **Data models** to describe the database schema.

  Here are some examples of individuals who will need to understand the design of the system:

  - Developers who will implement a solution based on the design.
  - Architects who can review the design to ensure that it conforms to the architecture or who might examine the design for opportunities to improve the architecture.
  - Other designers who can examine the design for applicability to other parts of the system.
  - Developers or other designers who will be working on other parts of the system that will depend on the elements designed in this task.

- Other reviewers who will review the design for quality and adherence to standards.

- **Evaluate the design**

  Evaluate the object design for coupling, cohesion, and other quality design measurements.

  Consider the design from various angles to ensure that it is a high-quality, communicable design. Work with other technical team members; an independent party can provide a fresh perspective. Use the tester and architect to provide perspectives on design quality and adherence to the architecture. However, when identifying potential reviewers keep in mind that if someone can add value by reviewing the design, then perhaps they could have added even more value by actively participating in the design effort itself. If design flaws are identified, improve the design.

  See **#concept# Design**, **#guideline# Analyze the Design**, and **#guideline# Evolve the Design** for more information.

Key Considerations

Each step in this task can cause all previous steps to be revisited in light of new information and decisions. For example, while determining how elements collaborate you might find a gap in the requirements that causes you to go back to the beginning after collaborating with the analyst, or when evaluating the design a reviewer could note that a reusable element being used doesn't work as expected and that could cause you to identify new elements to take its place. Consider the architecture while performing this task. All design work must be done while regarding the architecture within which the design exists. Furthermore, certain design elements will be deemed architecturally significant; those elements will require updates to the architecture.

This task will be applied numerous times. Design is best performed in small chunks.

Even when starting the design for a particular project it is expected that there will be existing frameworks and reusable elements. Every step of this task must give attention to the existing design and existing implementation, utilizing existing elements when possible and emulating or improving existing elements as appropriate while designing this portion of the solution.

Apply patterns throughout this task. Patterns represent proven designs and their usage promotes quality and consistency across the design.

.

More Information

- **#concept# Architectural Mechanism**
- **#concept# Design**
- **#concept# Design Mechanism**
- **#concept# Implementation Mechanism**
- **#concept# Pattern**
- **#concept# Requirements Realization**
- **#guideline# [Design Guidance]**
- **#guideline# Analyze the Design**
- **#guideline# Designing Visually**
- **#guideline# Entity-Control-Boundary Pattern**

- **#guideline# Evolve the Design**
- **#guideline# Software Reuse**

.

.

2.3.4.2 #task# Implement Developer Tests

Summary
Implement one or more tests to verify an implementation element.

Purpose
Prepare to validate an implementation element (e.g. an operation, a class, a stored procedure) through unit testing. The result is one or more new developer tests.

Relationships.
.

Roles	Inputs	Outputs
Primary: • #role# Developer	Mandatory: • #workproduct# [Technical Implementation]	• #workproduct# Developer Test
Additional: • #role# Tester	Optional: • #workproduct# [Technical Design]	
Assisting: •	External: • None	

Main Description
Developer testing is different from other forms of testing in that it is based on the expected behavior of code units rather than being directly based on the system requirements.

It is best to do this at a small scale, much smaller than the complete code base to be authored by a developer over the course of an iteration. This can be done for one operation, one field added to a user interface, one stored procedure, etc. As the code base is incrementally built, new tests will be authored and existing tests might be revisited to test additional behavior.

Steps

- **Refine scope and identify the test(s)**

  Select the increment of work to be tested and identify developer test(s) to verify that the software implementation being developed behaves correctly. One source for the expected behavior for an implementation element is the software design.

In identifying the tests or in any other part of this task, consider collaborating with a team member who is well-versed in the issues of testing.

- **Write the test setup**

  To successfully run a test the system must be in a known state so that the correct behavior can be defined. Implement the setup logic that must be performed as part of the developer test.

- **Define the expected results**

  Define the expected results of each test so that it can be verified.

  After a test runs, you need to be able to compare the results of running the test against what was expected to happen. The test is successful when the actual results match the expected results.

- **Write the test logic**

  Write the steps that perform the actual test(s).

- **Define the test response**

  Define the information the test(s) must produce to successfully indicate success or failure. Consider if a response of True or False is sufficient, or if a detailed message should be logged as well.

- **Write clean-up code**

  Identify, and then implement, the steps to be followed in order to restore the environment to the original state for each test. The goal is to ensure that there are no side effects from running the tests.

- **Test the test**

  Verify that each developer test works correctly. To do this:

  - Run the test(s), observe their behavior, and fix any defects in the tests.
  - Ensure that the expected results are defined properly and that they're being checked correctly.
  - Check the clean-up logic for each test.
  - Ensure that each developer test works within your test suite framework.

Key Considerations

Automate tests via a unit regression testing tool (for example, xUnit) so that tests may be run by developers whenever they make changes to the code.

Test to the risk of the implementation element. For example, the more critical an element is, the more important it is to test it thoroughly.

Pair with team members with testing skills in all steps of this task to gain insight on testing and quality considerations.

The [Project Work] is implicitly used in implementation tasks to manage which requirements or change requests are being realized in the code.
Back to top

Alternatives
Rely on acceptance tests to validate your software. This will likely be time consuming, late, and not as effective as developer testing in identifying bugs and finding their location in the code.

More Information

- **#concept# Developer Testing**

.

.
2.3.4.3 #task# Implement Solution

Summary
Implement source code to provide new functionality or fix defects.

Purpose
The purpose of this task is to produce an implementation for part of the solution (such as a class or component), or to fix one or more defects. The result is typically new or modified source code, which is referred to the implementation.

Relationships.
.

Roles	Inputs	Outputs
Primary: • #role# Developer	Mandatory: • #workproduct# [Technical Design] • #workproduct# [Technical Specification]	• #workproduct# Implementation
Additional: • #role# Stakeholder • #role# Tester	Optional: • #workproduct# [Technical Implementation] • #workproduct# Developer Test	
Assisting: •	External: • None	

Main Description
Usually, this task is focused on a specific implementation element, such as a class or component, but it does not need to be.

A portion of the design is implemented by performing this task. This task can be performed any number of times during an iteration. In fact it is best to do this task in as small a scope as possible to tighten the loop between it and related tasks involving developer testing and consideration of the design.

Steps

- **Determine a strategy**

  Determine a strategy based on the software design and developer tests for how you are going to implement the solution. The fundamental options are:

  - Apply existing, reusable assets.
  - Model the design in detail and generate the source code (by model transformation).
  - Write the source code.
  - Any combination of the above.

- **Identify opportunities for reuse**

  Identify existing code or other implementation elements that can be reused in the portion of the implementation that you are creating or changing. A comprehensive understanding of the overall design is helpful, because it is best to leverage reuse opportunities when you have a thorough understanding of the proposed solution.

- **Transform design into implementation**

  If you are using sophisticated modeling tools, you should be able to generate a portion of the required source code from the model. Note that programming is commonly required to complete the implementation after the design model has been transformed into code.

  Even without tools, there is typically some amount of code that can be created by rote by examining the design and developer tests.

- **Write source code**

  Write the source code to make the implementation conform to the design and expected behavior. You should strive to reuse and/or generate code wherever possible, but you will still need to do some programming. To do so, consider the following:

  - Examine the technical requirements. Because some requirement information does not translate directly into your design you should examine the requirement(s) to ensure that they are fully realized in the implementation.
  - Refactor your code to improve its design. Refactoring is a technique where you improve the quality of your code via small, safe changes.
  - Tune the results of the existing implementation by improving performance, the user interface, security, and other nonfunctional areas.
  - Add missing details, such as completing the logic of operations or adding supporting classes and data structures
  - Handle boundary conditions.
  - Deal with unusual circumstances or error states.
  - Restrict behavior (preventing users or client code from executing illegal flows, scenarios, or combinations of options).
  - Add critical sections for multi-threaded or re-entrant code.

Though many different considerations are listed here, there is one clear way to know when the source code is done. The solution has been implemented when it passes the developer tests. Any other considerations can be taken care of in a refactoring pass over the code to improve it once it is complete and correct.

- **Evaluate the implementation**

  Verify that the implementation is fit for its purpose. Examine the code for its suitability to perform its intended function. This is a quality assurance step that you perform in addition to testing which is described in other tasks. Consider these strategies:

  - Pair programming. By pairing to implement the code in the first place, you effectively evaluate the code as its being written.
  - Read through the code for common mistakes. Consider keeping a checklist of common mistakes that you make, as a reminder reference.
  - Use tools to check for implementation errors and inappropriate code. For example, use a static code rule checker or set the compiler to the most detailed warning level.
  - Use tools that can visualize the code. Code visualization, such as the UML visualizations in the Eclipse IDE, help developers identify issues such as excessive coupling or circular dependencies.
  - Perform informal, targeted code inspections. Ask colleagues to review small critical sections of code and code with significant churn. Avoid reviewing large sections of code.
  - Use a tester to ensure the implementation is testable and understandable to testing resources.

  Improve the implementation based on the results of these evaluations.

- **Communicate significant decisions**

  Communicate the impact of unexpected changes to the design and requirements.

  The issues and constraints that you uncover when you implement the system must be communicated to the team. The impact of issues discovered during implementation must be incorporated into future decisions. If appropriate, use the change management process to reflect ambiguities that you identified and resolved in the implementation so they can be tested and you can manage stakeholder expectations appropriately. Similarly, leverage the design process to update the design to reflect new constraints and issues uncovered during implementation to be sure that the new information is communicated to other developers.

  Usually, there is no need for a change request if the required change is small and the same person is designing and implementing the code element. That individual can make the design change directly. If the required change has a broad impact, it may be necessary to communicate that change to the other team members through a change request.

Key Considerations

It is best when developer tests already exist so there is an unambiguous definition of what behavior is considered correct. The implementation should be immediately tested.

The [Project Work] is implicitly used in implementation tasks to manage which requirements or change requests are being realized in the code.

More Information

- **#concept# Refactoring**
- **#guideline# Mapping from Design to Code**

.

.

2.3.4.4 #task# Run Developer Tests

Summary
Run tests against the individual implementation elements to verify that their internal structures work as specified.

Purpose
To verify that the implementation works as specified.

Relationships.
.

Roles	Inputs	Outputs
Primary: • #role# Developer	Mandatory: • #workproduct# [Technical Implementation] • #workproduct# Developer Test	• #workproduct# Test Log
Additional: •	Optional: • None	
Assisting: •	External: • None	

Steps

- **Run developer tests**

  Run the developer tests. The procedure will vary, depending on whether the test is manual or automated and whether additional test components are necessary, such as drivers or stubs.

  To run the tests, you need to make sure that you have initialized the test environment with all necessary elements, such as software, hardware, tools, data, and so on.

  Automated tests will often update a test results which you can evaluate to determine where your tests went wrong.

- **Evaluate test execution**

  Evaluate the test execution by analyzing the test run.

  Testing will complete either normally or abnormally. For correctly implemented tests, a normal completion represents a passed test, though it could warrant additional examination of the test results log to ensure the test ran as expected. Abnormal termination could be premature termination or just a test that does not complete as intended.

  Review the test log to understand any reported failures, warnings, or unexpected results. The cause of the problem(s) might be that the implementation element being tested is faulty, a problem with the developer tests, or a problem with the environment.

- **Respond to test results**

  Determine the appropriate corrective action to recover from a "failed" developer test run. If the implementation element under test is faulty, fix the problem if possible and rerun the tests. If the problem is serious and cannot be immediately addressed, report the defect. If the developer test is faulty fix the test and rerun the tests. If there was a problem with the environment, resolve it and then rerun the tests.

- **Promote changes for integration test**

  When the developer tests pass and no further work is need to complete the change set, promote the changes for integration test. If the passing of these tests represent completion of a requirement update the status of the work item.

Key Considerations
It is common to require that code pass all developer tests before it can be delivered in an integrated source code repository.

Pair with testing experts to gain insight on testing and quality considerations.

The [Project Work] is implicitly used in implementation tasks to manage which requirements or change requests are being realized in the code.

More Information

- **#concept# Developer Testing**

.

.
2.3.4.5 #task# Integrate and Create Build

Summary
This task describes how to integrate all changes made by developers into the code base and perform the minimal testing to validate the build.

Purpose
The purpose of this task is to integrate all changes made by all developers into the code base and perform the minimal testing on the system increment in order to

validate the build. The goal is to identify integration issues as soon as possible, so they can be corrected easily by the right person, at the right time.

Relationships.

.

Roles	Inputs	Outputs
Primary: • #role# Developer	Mandatory: • #workproduct# [Technical Implementation] • #workproduct# Test Script	• #workproduct# Build
Additional: •	Optional: • None	
Assisting: •	External: • None	

Steps

- **Integrate implemented elements**

  In the relevant **#concept# Workspace**, combine all completed **#concept# Change Sets** that are not in the latest baseline. Resolve any conflicting versions of the artifacts by either removing one of the change sets that created the conflict or by creating a new change set that includes merged versions of the conflicting artifacts.

- **Create build**

  Create the build. The details of this step depend upon the implementation language and development environment and may involve compiling and linking (in the case of compiled languages) and/or other processes that result in an executable increment of the system.
  Examples of these steps include:

  - Compiling and linking the source artifacts to create an executable
  - Loading binary objects on a test bench or simulator
  - Running a script to load/update database schemas
  - Packaging and deploying web applications

- **Test integrated elements**

  Re-run the developer tests against the integrated elements to verify that they behave the same as they did in isolation. Ensure that the scope of these tests is as broad as possible, which ensures that the latest change sets did not cause failing developer tests in other areas of the system.

- **Run "Smoke Tests"**

Several builds will be created in each iteration. For each build, this step is performed only when change sets have been delivered to satisfy the requirements of that build.

Execute a sub-set of the system tests to ensure that the build is suitable prior to committing resources to the full scope of system testing. While the level of testing will vary, focus on gaining confidence that the increment is of sufficient quality to establish a baseline for system testing.

- **Make changes available**

  When tests are successfully completed and the build is considered "good," the results are made available to the rest of the team by **#guideline# Promoting Changes**. The details of this step depend on the configuration management tools in use, but in general this involves committing a tested change set to the CM repository so that it serves as the basis of development for the next increment of the system. This is the essence of **#guideline# Continuous Integration**.

Key Considerations
In order to be effective at applying the practice of Continuous Integration, the time to integrate, build, and test the increment must be short enough that it can be performed several times per day. Changes should be broken down into relatively small Change Sets that can be implemented, integrated and tested quickly.

More Information

- **#concept# Change Set**
- **#concept# Workspace**
- **#guideline# Continuous Integration**
- **#guideline# Promoting Changes**

.

.
2.3.5 #activity# Test Solution

Summary
From a system perspective, test and evaluate the developed requirements.

Purpose
Develop and run test scripts to validate that the **system** satisfies the requirements.

Description
This activity is repeated throughout the project lifecycle. The main goal of this activity is to validate that the current build of the system satisfies the requirements allocated to it.
Throughout the iterations, your intent is to validate that the implemented requirements reflect a robust architecture, and that the remaining requirements are consistently implemented on top of that architecture. As developers implement the solution for the requirements in a given iteration, unit test the integrated source code. Then, a tester conducts system-level testing in parallel with development to make sure that the solution, which is continuously being integrated, satisfies the intent specified in the test cases. The tester defines what techniques to use, what the data input is, and what test suites to create.

As tests run, defects are identified and added to the work items list, so that they can be prioritized as part of the work that you will do during iterations.

Stakeholders and end-users also may also be involved in performing tests to accept the release.

.

.

2.3.5.1 #task# Implement Tests

Summary
Implement Test Scripts to validate a Build of the solution. Organize Test Scripts into suites, and collaborate to ensure appropriate depth and breadth of test feedback.

Purpose
To implement step-by-step Test Scripts that demonstrate the solution satisfies the requirements.

Relationships.
.

Roles	Inputs	Outputs
Primary: • #role# Tester	Mandatory: • #workproduct# Test Case	• #workproduct# Test Script
Additional: • #role# Analyst • #role# Developer • #role# Stakeholder	Optional: • #workproduct# [Technical Implementation] • #workproduct# Test Script	
Assisting: •	External: • None	

Steps

- **Select Test Cases to implement**

  Select a set of Test Cases to develop into detailed, executable Test Scripts.

  Work with project managers and developers to determine which Test Cases need detailed Test Scripts during the current iteration. At a minimum, select Test Cases for requirements that are planned in the current or next iteration.

  Perform each subsequent step in this task for each Test Script.

- **Design the Test Script**

Sketch an outline of the Test Script as a logical sequence of steps. Review the data requirements of the Test Case, and determine if existing data sets are sufficient, or if you need to develop new test data for this Test Script. Examine system-wide requirements that apply to this Test Script, and note where they affect the expected results of a step.

If available, review a build that implements the scenario, or demonstrates similar functionality.

Select an implementation technique for this design. At a minimum, determine if the Test Script will be manual or automated. If the Test Case is well understood, it's best to implement an automated Test Script without first writing a manual procedure. However, if the Test Case is new or novel, writing a manual Test Script can help validate the design of the test and aid collaboration with other team members. See **#guideline# Programming Automated Tests** for more details about this decision.

- **Implement the executable Test Script**

  Develop a detailed, procedural Test Script based on your design. Use a request-response style that declares an exact input, and expects an exact output.

  Explain the **pre-conditions** that must be met before running this Test Script. Use temporary test data or put parameters in your script for data values. Ensure that each **post-condition** in the Test Case is evaluated by steps in the Test Script.

- **Define specific test data**

  Specify data values that are specific to the Test Script or reference existing test data. For example, instead of specifying "a prime number", indicate an actual value such as "3." If the Test Script uses a dataset (such as a file or database), add the new test data to it and parameterize the Test Script to retrieve values from the dataset. Otherwise, add executable test data values to the steps of the Test Script. This applies to both manual and automated scripts.

  Identify and minimize dependencies between test data used or modified by other Test Scripts. Note dependencies in the Test Script.

  If necessary, create containers for your test data sets, and separate the production data from generated data.

- **Organize Test Scripts into suites**

  Collect tests into related groups. The grouping you use depends on your test environment. Since the system under test is undergoing its own evolution, create your test suites to facilitate regression testing, as well as system configuration identification. For help with test suite organization, see **#guideline# Test Suite**.

- **Verify Test implementation**

Run the Test Script to verify that it implements the Test Case correctly. For manual testing, conduct a walkthrough of the Test Script. For automated tests, verify that the Test Script executes correctly and produces the expected result. Verify that the Test Script meets the criteria in **#checklist# Test Script**. Add or update the Test Script(s) in configuration management.

- **Share and evaluate Test Scripts**

  Walk through the new or refined Test Scripts with the developers responsible for the related scenarios. Optionally, the analysts and the stakeholders also participate.

  Seek agreement that the Test Scripts correctly evaluate the expected results of the test, and that you understand the implementation of the requirements. If the scenario is already implemented (such as in a developer workspace), walk through a representative set of the Test Scripts using an implementation of the system.

More Information

- **#concept# Test Ideas**
- **#guideline# Maintaining Automated Test Suites**
- **#guideline# Programming Automated Tests**
- **#guideline# Test Ideas**

.

.

2.3.5.2 #task# Run Tests

Summary
Run the appropriate tests scripts, analyze results, articulate issues, and communicate test results to the team.

Purpose
To provide feedback to the team about how well a build satisfies the requirements.

Relationships.

.

Roles	Inputs	Outputs
Primary: • #role# Tester	Mandatory: • #workproduct# [Technical Implementation] • #workproduct# Test Script	• #workproduct# Test Log
Additional: •	Optional: • None	
Assisting: •	External: • None	

Steps

- **Review work items completed in the build**

  Review work items that were integrated into the build since the last test cycle. Focus on identifying any previously unimplemented or failing requirements are now expected to meet the conditions of satisfaction.

- **Select Test Scripts**

  Select test scripts related to work items completed in the build.

  Ideally, each test cycle should execute all test scripts, but some types of tests are too time-consuming to include in each test cycle. For manual or time-intensive tests, include test scripts that will provide the most useful feedback about the maturing solution based on the objectives of the iteration.

  Plan with test suites to simplify the process of selecting tests for each build (see **#guideline# Test Suite**).

- **Execute Test Scripts against the build**

  Run the tests using the step-by-step procedure in the Test Script.

  For automated test scripts, initiate the test execution. Automated test scripts should run in suites in the correct sequence, and collect results in the Test Log.

  To execute a manual test script, establish its preconditions, perform the steps while logging results in the Test Log, and perform any teardown steps.

- **Analyze and communicate test results**

  Post the test results in a conspicuous place that is accessible to the entire team, such as a white board or Wiki.

  For each failing test script, analyze the Test Log to identify the cause of the test failure. Begin with failing tests that you expected to begin passing against this build, which may indicate newly delivered work items that do not meet the conditions of satisfaction. Then review previously passing test scripts that are now failing, which may indicate regressive issues in the build.

  If a test failed because the solution does not meet the conditions of satisfaction for the test case, log the issue in the Work Items List. In the work item, clearly identify the observed behavior, the expected behavior, and steps to repeat the issue. Note which failing test initially discovered the issue.

  If a test failed because of a change in the system (such as a user-interface change), but the implementation still meets the conditions of satisfaction in the test case, update the test script to pass with the new implementation.

If a test failed because the test script is incorrect (a false negative result) or passed when it was expected to fail (a false positive result), update the test script to correctly implement the conditions of satisfaction in the test case. If the test case for a requirement is invalid, create a request change to modify the conditions of satisfaction for the requirement.

It's best to update test scripts as quickly and continuously as possible. If the change to the test script is trivial, update the test while analyzing the test results. If the change is a non-trivial task, submit it to the Work Items List so it can be prioritized against other tasks.

- **Provide feedback to the team**

  Summarize and provide feedback to the team about how well the build satisfies the requirements planned to the iteration. Focus on measuring progress in terms of passing tests. Explain the results for the test cycle in the context of overall trends:

  - How many tests were selected for the build, and what are their statuses (pass, fail, blocked, not run, etc.)?
  - How many issues were added to the Work Items List, and what are their statuses and severities?
  - For test scripts that were blocked or skipped, what are the main reasons (such as known issues)?

Key Considerations
Run all tests as frequently as possible. Ideally, run all test scripts against each build deployed to the test environment. If this is impractical, run regression tests for existing functionality, and focus the test cycle on work items completed in the new build.

Even test scripts that are expected to fail provide valuable feedback. However, once a test script is passing, it should not fail against subsequent builds of the solution.
.

.
2.3.6 #activity# Ongoing Tasks

Summary
Perform ongoing tasks that are not necessarily part of the project schedule

Description
This activity includes a task for requesting changes. This task may occur any time during the lifecycle in response to an observed defect, a desired enhancement, or a change request. It is not planned, which means that it does not appear as a scheduled activity on the project plan, iteration plan, or work items list. Nevertheless, it is a critical activity that must be performed to ensure project success, because your environment is not static.

Any role may perform this task, however the most common sources of **#concept# Change Requests** are stakeholders (enhancement requests and change requests) and testers (defect reports
.

.

2.3.6.1 #task# Request Change

Summary

Capture and record change requests

Relationships.

.

Roles	Inputs	Outputs
Primary: • #role# Any	Mandatory: •	• #workproduct# Work Items List
Additional: •	Optional: • None	
Assisting: •	External: • None	

Steps

- **Gather change request information**

  Gather the information required to describe the change request. This should include a description of the requested change, the reason for the change (defect or enhancement), the artifact affected (including the version), and the priority of the change. If possible, provide an estimate of the effort required to implement the change.

- **Update the Work Item List**

  Update the **#workproduct# Work Items List** to document the information that you gathered in the previous step.

.

.

.

.