Software Plan

PLAN-001, Rev. 1

Innolitics, LLC.

February 19, 2021

Contents

1 Purpose

This document describes a set of activities which will be used during software risk management, development, and maintenance of PROJECT. It is written primarily for software developers.

PROJECT is assigned a Class C software safety class, which means death or serious injury could occur if the software fails [62304:4.3.a]. See `risk.yml` for details. All of the software items that compose the software system are also presumed to have the same Class C safety class [62304:4.3.c 62304:4.3.d 62304:4.3.d 62304:4.3.e 62304:4.3.f 62304:4.3.g]. The primary purpose of this document is to help developers ensure PROJECT is safe and useful while also allowing developers to be productive. The secondary purpose is to comply with IEC62304:2006+A1:2015.

[In order to assist auditors and regulators, we have included section references to IEC62304:2006+A1:2015 as well as occasional comments throughout this document. These references and comments are always placed inside square brackets, and they are not present in the software-developer version of the document. Other than these comments, the software-developer version is identical to the auditor version of this document.]

[FDA-CPSSCMD:dev-environment]

2 Overview

2.1 Definitions

[Most of these definitions are very similar to the IEC62304:2006+A1:2015 definitions, however, they have been modified as appropriate for a better understanding by software developers.]

Activity A set of one or more interrelated or interacting tasks. An activity has an input, an output, and often an explicit verification task. Records of activity outputs must be available in case of an audit.

Change Request A documented specification of a change to be made to the software system.

Known Anomaly A problem report that we do not plan on addressing.

Problem Report A record of actual or potential behavior of a software product that a user or other interested person believes to be unsafe, inappropriate for the intended use or contrary to specification.

Record A special type of document that states the results achieved or provides evidence that activities were performed. Unlike other documents---such as this software plan---records are not usually revised after being approved.

SOUP Software of unknown provenance, also known as "off-the-shelf software", is a **software item** that has not been developed for the purpose of being incorporated into the medical device and for which adequate records of the development processes are not available.

Software Item A module within the software system which may further decomposed into smaller software items. The **software system** is itself a software item, and thus the software items for a hierarchy. See the software design specification for a description of how the software system is decomposed into smaller software items.

Software Requirement A particular function that the software must support, or some other constraint that the software must fulfill. Requirements describe the *what*, while specifications and designs describe the *how*.

Software System All of the software in the project. The software system is decomposed into **software items**.

Software Unit A software item which is not further subdivided.

2.2 Development Life Cycle Model

PROJECT will be developed using an agile software development life cycle model. The agile strategy develops the software system using a sequence of builds. Customer needs and software system requirements are partially defined up front, then are refined in each succeeding build [62304:5.1.1; by "agile" we mean a combined evolutionary and incremental life cycle model].

2.3 Roles and Responsibilities

The activities described in this document are designed for a team composed of a project lead and one to eight software developers. One of the software developers shall be assigned the role of the project lead. The project lead, working on behalf of the manufacturer, is responsible for the safety and utility of the software system built by the team.

At least one team member must be trained in risk management [14971:4.3].

2.4 Related Documents

[This section fulfills 62304:5.1.8.a, 62304:5.1.8.b, and 62304:5.1.8.c]

The **level of concern** document is written by the project lead, in conjunction with the manufacturer, up front. Its purpose is to help guide the risk analysis and inform the FDA. It may be updated as part of risk assessment activity. It is reviewed by the project lead during the release activity.

The **software description** is not a separate document, but is included within the SDS.

The **software requirements specification** (or **SRS**) describes what the software needs to accomplish. It is largely written by the project lead during the requirements analysis activity, and is reviewed by the project lead during the release activity. Software developers may clarify and extend the document slightly during the unit implementation and testing activity.

The **software architecture chart** is not a separate document, but is included within the SDS.

The **software design specification** (or **SDS**) describes how the software accomplishes what the SRS requires. A significant portion is written by the project lead during the architectural design activity, but many details and specifics are added by software developers during the unit implementation and testing activity. It is reviewed for consistency by the project lead during the release activity.

A **release history** includes a list of change requests and problem reports addressed within a release. It also includes a record of the implemented changes and their verification and a list of known anomalies. The content of the document is slowly built up by software developers during the unit implementation and testing activity. It is reviewed by the project lead during the release activity.

A **test record** describes a set of tests which were run, when, and by who. It also must include enough details to reproduce the testing setup.

A **release record** describes the verifications steps performed by the project lead during the release activity.

2.5 Development Standards

TODO: The project lead should keep an up-to-date list of development standards here (e.g., PEP8 on a Python project).

If the software system's safety classification is not level C (the highest), you may delete this section.

[This section fulfills 62304:5.1.4.a]

2.6 Development Methods

[62304:5.1.4.b]

TODO: The project lead should keep an up-to-date list of development methods here (e.g., Test Driven Development) if relevant.

If the software system's safety classification is not level C (the highest), you may delete this section.

2.7 Development Tools

[62304:5.1.4.c]

TODO: The project lead should keep an up-to-date list of development tools here, such as linters and versions.

If the software system's safety classification is not level C (the highest), you may delete this section.

To the extent possible, checking against these standards should be performed in an automated fashion (e.g., using a linter which is run on a Git-commit hook) [62304:5.1.4].

2.8 Testing Plan

[62304:5.5.2]

All final tests must include the Git hash or other objective reference that can be used to identify the exact software tested [62304:5.1.11].

TODO: Write out a testing plan for PROJECT.

This plan should include a pass/fail criteria for the entire test suite. E.g., you require that all unit tests pass and that all integration tests pass or the cause of the failure is understood and justified [62304:5.7.1.a]

2.9 Quality Assurance

The activities below are designed to meet ISO 13506 quality control standard [62304:4.1].

2.10 Risk Management

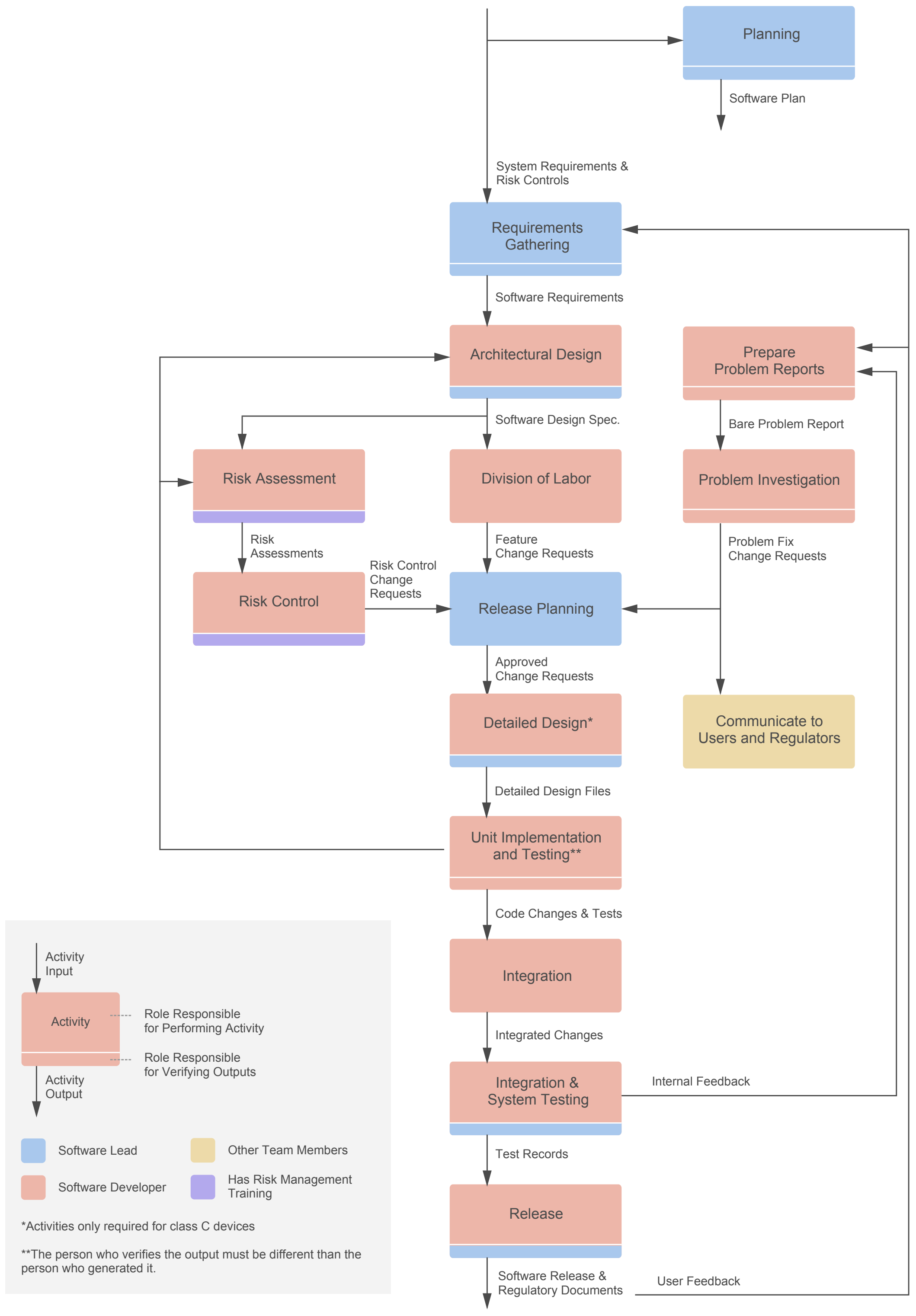
The Risk Assessment, Risk Control and other activities below are intended to meet ISO 14971 risk management standard [62304:4.2 14971:3.1 14971:3.2].

3 Activities

This section of the software plan describes the various activities involved with software development, maintenance, risk management, problem resolution, and version control (also known, in regulatory lingo as "configuration management"). The relationship between the inputs and outputs of these activities are displayed in the following diagram and are fully described in the sub-sections below. Since we are using an agile development life cycle, activities may be performed before their inputs have settled and thus inputs and outputs may not be consistent between releases. [Note that tasks are presented within each activity, although we do not explicitly demarcate them.]

[This software plan does not explicitly separate the software development process, software maintenance process, configuration management process, problem resolution process, and software-related risk management because we are using an agile software development life cycle and thus the processes overlap with one another significantly. The activities described here fulfill 62304:4.2 62304:5.1.1.a, 62304:5.1.1.b, 62304:5.1.6, 62304:5.1.7, and 62304:5.1.9.b as well as, software-related portions 14971:3.4.a, 14971:3.4.b, 14971:3.4.c, 14971:3.4.e, and 14971:3.5]

3.1 Activity Diagram



3.2 Planning

Input: User needs, system requirements, and risk controls

Setup a Git repository on GitHub. All software activity outputs will be stored in this Git repository, the associated GitHub issues, or the associated GitHub pull requests, unless explicitly noted otherwise [62304:5.1.1.b]. Problem reports and change requests are stored as GitHub issues. A GitHub issue tagged with the `bug` label is a problem report. If a problem report outlines a set of requested changes, then it can simultaneously act as a change request. GitHub issues tagged with the `obsolete` label are ignored.

The software developers working on the project are responsible for keeping all software activity outputs within version control at the times specified in the activity descriptions [62304:5.1.9.c, 62304:5.1.9.d, and 62304:5.1.9.e].

[Note that we do not explicitly use the term "software configuration management" since many developers will be unfamiliar with the term, and instead we use the term "version control." Git is a version control system that makes it simple to track and record the history of every file it contains in a precise and controller manner. The requirements listed in sections 62304:5.1.9.a, 62304:5.1.11, 62304:8.1.1, 62304:8.1.3, 62304:8.3, and 62304:9.5 are fulfilled by our use of Git and GitHub. Also note that this setup implies that all activity outputs that are stored in the Git repository, GitHub issues, or GitHub pull requests are configuration items. Furthermore, the version of every configuration item comprising the software system configuration is stored in the Git repository for the entire history of the project. Each activity describes the configuration items in more detail.]

In the Software Design Specification, record details about the project's build process, including tool versions, environment variables, etc. [62304:5.1.10 and 62304:5.8.5]. Also document how the software can be reliably delivered to the point of use without corruption or unauthorized change [62304:5.8.8].

Search through this document, and the other documents in this set of templates, for the text "TODO". Follow the instructions next to the "TODO" sections. Once you are done you may delete the instructions. Note that longer instructions may be demarcated with "ENDTODO".

Keep this planning document up to date as the project commences [62304:5.1.2].

In conjunction with the manufacturer's management, review and update as appropriate the:

- qualitative risk severity categories
- qualitative risk probability categories
- qualitative risk levels

contained within `risk.yml` [14971:3.4.d, 14971:D.3, 14971:D.4, 14971:D.8].

Output: The markdown version of this plan document and the Git repository hosted on GitHub.

Verification:

Review the document for typos our outdated information.

Ensure that activity in the software plan specifies:

- the activity inputs
- the activity outputs
- output verification steps, if any
- which role should perform and verify the activity [62304:5.1.6.a and 62304:5.1.6.b].

3.3 Requirements Analysis

Input: User needs, system requirements, and risk controls

Record system requirements in SYSTEM REQUIREMENTS SOFTWARE. Each system requirement must have a unique identifier so that we can trace software requirements back to the system requirements they fulfill [62304:5.1.3.a 62304:5.1.3.b].

Important software requirements should be enumerated at the start of the project [62304:5.2.1]. Software requirements must be tied to one or more originating system requirements via the system requirement's ids [62304:5.1.1.c]. If a software requirement can not be tied back to any system requirements, new system requirements should be added.

When software requirements are added or changed, re-evaluate the medical device risk analysis [62304:5.2.4] and ensure that existing software requirements, and system requirements, are re-evaluated and updated as appropriate [62304:5.2.5].

See the appendices for additional information.

Output: Software requirements, Git

Verification: Ensure software requirements:

- implement system requirements and are labeled with system requirement ids [62304:5.2.6.a 62304:5.2.6.f]
- implement risk controls
- don't contradict each other [62304:5.2.6.b]
- have unambiguous descriptions [62304:5.2.6.c]
- are stated in terms that permit establishment of test criteria and performance of tests to determine whether the test criteria have been met [62304:5.2.6.d].

3.4 Architectural Design

Input: Software requirements

Develop an initial software system architecture and document it in the SDS [62304:5.3.1]. The SDS should describe how the software system is divided into a hierarchy of software items [62304:5.4.1]. Software units are often thought of as being a single function or module, but this is not always appropriate.

Show the software and hardware interfaces between the software items and external software [62304:5.3.2]. Prefer block diagrams and flow charts to textual descriptions, and include these diagrams in the SDS. Indicate which software items are SOUP.

Identify any segregation between software items that is essential to risk control, and state how to ensure that the segregation is effective. For example, one may segregate software items by running them on different processors [62304:5.3.5].

The initial architecture does not need to be complete, since code construction can guide architectural decisions. However, it is worth spending a significant amount of time on the initial architecture. Once development commences (i.e., the unit implementation and testing activity), update the SDS as the architecture is refined.

Output: SDS

Verification: Ensure software architecture documented in the SDS:

- implements system and software requirements [62304:5.3.6.a].
- is able to support interfaces between software items and between software items and hardware [62304:5.3.6.b].
- is such that the medical device architecture supports proper operation of any SOUP items [62304:5.3.6.c].

3.5 Risk Assessment

[[:This activity is meant to fulfill sections 14971:4.1.a, 14971:4.1.b, 14971:4.1.c, 14971:4.2 14971:5, 14971:6.1, and 14971:6.2 of 14971 with respect to software related risks]]

Input: Software design specification

Begin by identifying known and forseeably hazards associated with the device [14971:4.3]. It is frequently necessary to consult with a clinical expert to understand and identify hazards in their clinical context.

Next, identify which software items could cause hazardous situations [62304:7.1.1 62304:7.3.3.a], and list them, along with the forseeably causes [62304:7.3.3.b]. Consider:

- whether requirements are appropriate and are meeting the needs of users
- incorrect or incomplete specifications of functionality [62304:7.1.2.a]
- software defects in the software item [62304:7.1.2.b]
- failure or unexpected results from SOUP [62304:7.1.2.c]
- how hardware failures or software defects in other items could result in unpredictable operation [62304:7.1.2.d]
- reasonably forseeably misuse by users [62304:7.1.2.e]
- the list of causes in Annex B of IEC80002-1:2009 [62304:5.1.12.a 62304:5.1.12.b]

Include the sequences of events that could result in the hazardous situation [62304:7.1.5]. If failure or unexpected results from SOUP is a potential cause contributing to a hazardous situation, review the list of anomalies for the SOUP (if any) for known anomalies relevant to this hazardous situation [62304:7.1.3].

Record the identified hazards, causes, hazardous situations, and harms in `risk.yml` as an individual risk [62304:7.1.4 and 62304:9.5].

Finally, estimate the severity and probability of each risk and record this as well [14971:4.4].

See the appendices for additional information.

Output: Risk assessment

Verification:

- Ensure that the hazard, hazardous situation, and harms recorded for new risks appropriately follow their ISO14971 definitions.
- Ensure that the risk probability is justifiable.
- Ensure that the new risks listed are appropriate, and are not unnecessarily detailed to the point of not contributing to improved safety.

3.6 Risk Control

Input: Risk assessment

Evaluate unmitigated risks listed in `risk.yml` [14971:5].

If any of the risks require reduction, then identify appropriate risk control measures. Consider risk control measure options, in the following order:

1. inherent safety by design (i.e., refactoring or architecting away the risks, or even removing requirements)
2. adding software or hardware
3. providing information to the user in the form of documentation or user interface elements---these should be avoided as much as possible.

Create a change request for the risk control measure [14971:6.3 62304:7.2.1 62304:7.3.3.c].

Output: Risk control related change requests

Verification:

- Ensure that risks controls don't introduce larger risks than they mitigate [14971:6.6.a, 14971:6.6.b and 62304:7.3.1, since risk control measures will be implemented as part of this activity]
- As appropriate, ensure that the inherent safety by design is preferred over adding software or hardware risk control measures.

3.7 Division of Labor

Input: Design files

All work on the software project should occur in response to approved change requests [62304:8.2.1]. There are many ways to divide new requirements work into change requests. Change requests associated with requirements which will be implemented soon may be split into smaller change requests, while requirements which may not be worked on for several months can be captured in a single large change request.

Output: Feature change requests

Verification: Not applicable to this activity

3.8 Release Planning

[This activity addresses 62304:6.3.1, since change requests resulting from maintenance and problem resolution are processed in the same manner in which risk control measures and feature change requests. Note that some releases are only meant for tracking development. The first commercial release is typically v1.0.]]]

Input: Feature and problem fix change requests

To organize and prioritize the development work, change requests are assigned to GitHub milestones. Change requests that have not yet been assigned to a GitHub milestone have not yet been approved, and should not be worked on [62304:8.2.1, 62304:6.2.4].

Once a change request is assigned to a milestone, it has been "approved" and may be worked on by a developer. The project lead will then assign developers to change requests to divide up the work. Software developers may also assign themselves to change requests, so long as it is not assigned to another developer and they don't have other outstanding tickets they can work on.

The project lead should coordinate with the business owner regarding which change requests to include in a release. When planning a release:

- Consider outstanding problem reports [62304:9.4].
- Look through historical problem reports and attempt to identify any adverse trends. For example, some software items may have many problem reports associated with them [62304:9.6 and 14971:9.a] or may have new or revised standards [14971:9.b].
- Review `risk.yml` for risk control measures that have not been implemented [62304:7.3.1 and 62304:7.2.2.c].
- Review `soup.yml`. Look for SOUP which has become obsolete, SOUP which should be upgraded, and for known anomalies in published anomalies lists as appropriate [62304:6.1.f]. See the appendices for additional details.

Create change requests as appropriate.

Output: The set of change requests which should be implemented for the next release

Verification: Not applicable to this activity

3.9 Detailed Design

Input: SDS

Begin a new Git branch, as discussed in the unit implementation and testing activity, but before implementing the change request, document a detailed design either within the SDS or as code comments, as appropriate, for each new software item [62304:5.4.2]. These detailed designs should be stored as closely as possible to their corresponding source files. As appropriate, write out function signatures for the essential procedures, functions, classes, and/or modules involved with the change request.

Detailed designs for interfaces between software items and external components (hardware or software) should be included as appropriate [62304:5.4.3].

Once you have completed the detailed design, open a pull request and assign the project lead to review the design.

Output: Software item designs

Verification: Ensure software requirements:

- is not more complicated than it needs to be to meet the requirements
- implements system and software requirements [62304:5.4.4.a]
- is free from contradiction with the SDS [62304:5.4.4.b].

3.10 Unit Implementation and Testing

[This activity addresses 62304:5.5.1]

Input: Detailed software item designs and software requirements

Create a new Git branch with a name that includes the change request number (e.g., `104-short-description`). Commit your code changes to this branch and push periodically [62304:5.1.1.d, 62304:6.1.e and 62304:8.2.2]. Commit messages should:

- explain why the current changes are being made, as appropriate

- reference the change request that prompted the changes (the `rdm hooks` command can streamline including these references).

During development, as appropriate:

- Analyze how this change request effects the entire software system, and consider whether any software items should be refactored or reused [62304:6.2.3].
- Consider whether any external systems that the software system interfaces with may be affected [62304:6.2.3].
- If software has been released, consider whether any existing data needs to be migrated.
- Write unit tests and new integration tests.
- If SOUP was added, removed, or changed, update the `soup.yaml` file (see the appendices for details).
- If the change request includes risk control measures, record the risk control measures in `risk.yml` along with the residual risk. Also add new software requirements for the risk control measure and record the software requirement id along with the risk [14971:6.2 and 62304:7.2.2.a].
- Perform the Risk Assessment [14971:6.6] and Risk Control Activities on any software items (including SOUP) which were are added or modified [62304:7.4.1.a], including new risk control measures[[62304:7.4.1.b, since they may have introduced new risks [62304:6.1.c, 62304:7.4 62304:7.3.1, 62304:7.4.3 since risk control measures will be implemented as part of this activity] or impact on existing risk control measures [62304:7.4.2].

When work on a change branch is nearing completion, a pull request should be created for this branch. A brief summary of the changes should be included in the pull request description. These comments will be included in the final release history. The description should also mention whether risk assessments were performed, or why not, and if tests were not required, why not.

Output: Code and documentation changes, stored in un-merged Git branches with corresponding approved pull requests

Verification: Code review by at least on other developer.

Code review should ensure the code changes made in the Git branch:

- implements the associated change request
- is consistent with the related detailed designs
- follows the project's software standards
- includes unit tests or justifies why they are not necessary
- any risk assessments are reasonable
- is covered by existing integration tests or includes a new integration test [62304:5.5.5 and 62304:8.2.3].

The developer performing the review should create a GitHub review and record their notes there. If any changes are requested, address them and re-submit the review once they have been addressed. The reviewer must approve the pull request from within the GitHub user interface [62304:8.2.4.c]. We suggest using the following format for your reviews:

```
- [x] Implements change request
- [x] Consistent with software system design
- [x] Documentation: Description of why fufilled, insufficient, or not needed.
- [ ] Unit Tests: Ditto
- [ ] Risk Assessment: Ditto
- [ ] Integration Tests: Ditto
```

This detailed checklist is not necessary for small changes or for changes early during the project.

Where the `x` indicates that the item was completed.

GitHub saved replies can help facilitate this process.

If, as is occasionally appropriate, someone outside of the core development team reviews a pull request, then mention who performed the review in the pull request body and tag the pull request with the `external-review` label.

3.11 Integration

Input: Unmerged, but approved, pull-request

Merge the approved Git branch into the `master` Git branch, correct any merge conflicts that occur. Once the branch has been merged successfully, delete the branch in GitHub [62304:5.1.5 and 62304:5.6.1].

Output: Merged pull request

Verification: Not applicable to this activity

3.12 Integration and System Testing

Input: Software system built using the changes from this release's change requests

The final integration prior to a release must formally record the test output in a test record. The test record must include:

- The list of tests that passed or failed [62304:5.6.7.a 62304:5.7.5.a]
- Verification that the results meet the pass/fail criteria listed in the Test Plan [62304:5.7.4.c 62304:5.6.7.a]
- The version of the software being tested (e.g., the Git commit hash) [62304:5.6.7.b 62304:5.7.5.b]
- The name of the person who ran the tests [62304:5.6.6, 62304:5.6.7, 62304:5.7.5.c, 62304:5.6.7.c and 62304:9.8].

Any test failures found during the formal release system testing shall be recorded as problem reports [62304:5.6.8 62304:5.7.4.d]. See the prepare problem report activity for details [62304:5.7.2]. If any change requests are implemented in response to these problem reports, the tests must be re-run [62304:5.7.3.a 62304:5.7.3.b]. If it is deemed unnecessary to re-run some of the tests, the justification as to why shall be included in the test record [62304:5.7.3.c note that the risk management activities for (c) will be handled as part of the unit implementation and testing activity].

Output: Test record and problem reports [62304:5.6.3, 62304:5.6.4, and 62304:7.3.3.d] **Verification:** Ensure code changes:

- the original problem is fixed and the problem report closed [62304:9.7.a]
- any adverse trends have been reversed [62304:9.7.b].

[Note that we combine our integration and system testing into one activity. We presume that if our integration tests and system tests are passing, no new problems were introduced, per 62304:9.7.d]

3.13 Release

[This activity addresses 62304:6.3.2, since development releases and maintenance releases are treated equivalently]

TODO: Write out how to archive the software system release. This will vary from project to project. Here are some exmples:

- If the output of the build process is a binary, then the binary should be saved somewhere.
- If the output is a set of Python scripts with out any SOUP, then the source code within the Git repository is already sufficient.
- If the output is a set of Python scripts with Python dependencies, then copies of the Python dependencies must be archived somewhere. Likewise, if there are other system dependencies, like postgres, then the debian package files (or perhaps a virtual box image) need to be archived.

ENDTODO

The purpose of the archive is to provide a means to re-test problems which may occur in an old version of the software.

Input: Implemented and verified change requests for the current milestone

When a new version of the software is released, the Git commit corresponding to the state of the code should be tagged with the version number.

Archived releases shall be kept until indefinitely.

[This section fulfills 62304:5.8.7.a and 62304:5.8.7.b; note that documentation and configuration items are archived automatically due to the fact that they are stored in Git]

Output: An archived software release

Verification: Ensure that

- all of the planned change requests have been implemented and integrated [62304:5.6.2 and 62304:9.7.c]
- the outputs of each activity are in a consistent state [62304:5.1.6.c, 62304:5.1.6.d, and 62304:5.8.6]
- the unit tests adequately verify the software units [62304:5.5.2]
- the integration tests adequately verify the software system [62304:5.6.5 and 62304:5.7.4]
- all software requirements have been tested or otherwise verified [62304:5.7.4.a and 62304:5.7.4.b]
- the software design specification is accurate and up-to-date
- integration and system testing activity has been completed after the last change request was integrated, and a test record has been written [62304:5.8.1]
- the Release History Document is up-to-date and none of the anomlies result in unacceptable risk [62304:5.8.2, 62304:5.8.3, and 62304:5.8.4]

Record these verification steps in a new software release record.

3.14 Problem Analysis

Feedback from users, internal testers, and software developers will be recorded in USER FEEDBACK SOFTWARE [62304:6.1.a, 62304:6.1.b and 62304:6.2.1.1; details about where direct customer feedback is recorded and tracked is not handled here. It is assumed that other processes (e.g., perhaps part of the broader quality management system) will handle this. We also do not go into detail here regarding what criteria should be used to determine whether feedback is considered a problem, as required by 62304:6.1.b].

3.15 Prepare Problem Report

[This activity addresses 62304:6.2.1.2]

Input: Feedback from users or other members of the development team

A problem report should be created whenever:

1. a user reports a problem while using a released version of the software system, or
2. when an internal user reports a new problem that has been found during software development or maintenance on the master Git branch [62304:5.1.1.e and 62304:5.1.9.f]. Note that small software bugs and test-failures, especially recently introduced bugs discovered by software developer working on the project, do not require a problem report. Problem reports provide a useful historical record of bugs, which can be used to identify software items which are especially risky.

When creating a new problem report, include in the description:

- The software item where the bug occurred (if known)
- If reported by a user, steps to recreate it
- If found in released software, the criticality of the problem
- Any relevant information that can be used to investigate the problem [62304:9.1].

Output: The problem report

Verification: Not applicable to this activity

3.16 Problem Investigation

[This activity addresses 62304:6.1.d and 62304:6.2.2]

Input: The problem report

1. Investigate the problem and if possible identify the cause and record it in the problem report [62304:9.2.a]
2. Evaluate the problem's relevance to safety using the software risk assessment activity [62304:6.2.1.3 62304:9.2.b]
3. Consider whether the software items implicated in the investigation have the proper safety class, and address as appropriate [62304:6.2.2 62304:7.2.2.b]
4. Summarize the conclusions from the investigation in the problem report [62304:9.2.c]
5. Create a change request for actions needed to correct the problem (also include an issue reference to the problem report [62304:8.2.4.a and 62304:8.2.4.b 62304:9.2.d], or document the rationale for taking no action [62304:9.2.d].

If the problem affects devices that have been released, make sure that quality control is aware of the situation and has enough information to decide whether and how to notify affected parties, including users and regulators. Record who you notified in the problem report [62304:9.3 62304:6.2.5.a and 62304:6.2.5.b; this document does not specify the process by which quality assurance will inform users, when they must inform users, etc. It is assumed these details are handled in another process, and that all that the software developers must do is pass along the appropriate details to quality assurance.].

Output: Details about the problem investigation documented in the problem report and either unapproved change requests or justification as to why change requests weren't necessary

Verification: Not applicable to this activity

4 Appendices

The subsections here provide guidance on following the software risk management, development, and maintenance activities.

4.1 Requirements Analysis

Writing software requirements is an art and a science; one must find balance between precision and usefulness.

The distinction between system requirements and software requirements can be challenging. System requirements describe the requirements of the entire system, including software and hardware. Software requirements must be traceable to all of the system requirements that they help fulfill. Software requirements are usually more detailed than the system requirements they refer to. Many system requirements will be fulfilled using both hardware and software.

The distinction between software requirements and the specifications is also typically challenging. Requirements should:

- not imply solution
- be verifiable
- be short, ideally one or two sentences long.

Specifications, on the other hand, should:

- be one of possibly many solutions
- be detailed.

Software requirements are often categorized as one of the following types [62304:5.2.2 and 62304:5.2.3]:

a. Functional and capability requirements [62304:5.2.2.a]

- performance (e.g., purpose of software, timing requirements),
- physical characteristics (e.g., code language, platform, operating system),
- computing environment (e.g., hardware, memory size, processing unit, time zone, network infrastructure) under which the software is to perform, and
- need for compatibility with upgrades or multiple SOUP or other device versions.

b. Software system inputs and outputs [62304:5.2.2.b]

- data characteristics (e.g., numerical, alpha-numeric, format) ranges,
- limits, and
- defaults.

c. Interfaces between the software system and other systems [62304:5.2.2.c]

d. Software-driven alarms, warnings, and operator messages [62304:5.2.2.d]

e. Security requirements [62304:5.2.2.e]

- those related to the compromise of sensitive information,
- authentication,
- authorization,
- audit trail, and
- communication integrity.

f. Usability engineering requirements that are sensitive to human errors and training [62304:5.2.2.f]

- support for manual operations,
- human-equipment interactions,
- constraints on personnel, and
- areas needing concentrated human attention.

g. Data definitions and database requirements [62304:5.2.2.g]

h. Installation and acceptance requirements of the delivered medical device software at the operation and maintenance site or sites [62304:5.2.2.h]

i. Requirements related to methods of operation and maintenance [62304:5.2.2.i]

j. Requirements related to IT-network aspects [62304:5.2.2.j@2015]

k. User maintenance requirements [62304:5.2.2.k]

l. Regulatory requirements [62304:5.2.2.l]

m. Risk control measures

Software requirements that implement risk controls should be tied to their originating risk control by tagging them with labels that match the risk control ids [62304:5.1.1.c].

4.2 Risk Management

This subsection provides a brief introduction to risk management in the context of software development.

Safety is freedom from unacceptable risk. Note that it does not mean that there is no risk, as that is impossible. The perception of risk can depend greatly on cultural, socio-economic and educational background, the actual and perceived state of health of the patient, as well as whether the hazard was avoidable.

Our obligation as medical device software developers is to develop safe devices, while balancing economic constraints---a device that is never made can not help patients, so there may be risk associated with not bringing a device to market if the device meets a clinical problem.

Risk is the combination of the probability of harm and the severity of that harm.

Harm is physical injury or damage of health of people (patients or users), or damage to property or the environment.

A **hazard** is a potential source of harm.

A **hazardous situation** is a circumstance in which people, property, or the environment is exposed to one or more hazard(s). Not every hazardous situation leads to harm.

Software is not itself a hazard because it can not directly cause harm, but software can contribute towards producing hazardous situations.

4.3 SOUP

Information in the `soup.yaml` file may duplicate information found in other files (e.g., `requirements.txt` or `package.json`).

Sometimes, especially when working on software items with low levels of concern, it can be appropriate to lump a few SOUP packages into a single item within the `soup.yml` file.

The `soup.yaml` should contain a sequence of mappings, each containing the keys in parenthesis below. Some keys are optional. All values must be strings.

The header of each sub-section contains the `title` of the SOUP [62304:8.1.2.a].

The `manufacturer` is the name of the company that developed the SOUP. If the manufacturer field is absent, then this SOUP was developed collaboratively by the free open-source software community, and does not have a manufacturer in the traditional sense [62304:8.1.2.b].

The `version` of each SOUP is a unique identifier, which specifies the version of the SOUP which is used in the software [62304:8.1.2.c]. The version may follow varying formats, such as `1.0.13`, `1.2r5`, or even `2021-05-05`, as appropriate. The `purpose` of each SOUP describes the functional and performance requirements that are necessary for its intended use [62304:5.3.3].

The `requirements` will be present if there are any noteworthy hardware and software requirements for the SOUP to function properly within the system [62304:5.3.4].

The known `anomalies` present in the SOUP which may affect the functioning of PROJECT should be recorded, as should the `anomaly_reference`, a location of the published anomalies list [62304:7.1.3].

When reviewing open anomalies:

- Follow a risk based approach; concentrate on high priority anomalies (assuming the SOUP manufacturer provides such a categorization).
- If the list of known anomalies is large (e.g., more than 100), without prioritization, then sample the list as appropriate for the risk associated with the SOUP.
- When possible, focus the review on anomalies which affect portions of SOUP which are used by PROJECT.