1. Requirements

3.2.2 The project shall transform the allocated and derived requirements into a documented software architectural design.

1.1 Notes

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the properties of those components, and the relationships between them. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

1.2 Applicability Across Classes

Classes C through E, and Safety Critical are labeled with "P (Center)" and "SO." P (Center) means that an approved Center-defined process which meets a non-empty subset of the full requirement can be used to achieve this requirement. SO means that the requirement applies only for safety critical portions of the software.

Classes C and Not Safety Critical and D and Not Safety Critical are labeled with "P (Center)." This means that a Center-defined process which meets a non-empty subset of this full requirement can be used to meet the intent of this requirement.

Classes F and G are labeled with "X (not OTS)." This means that this requirement does not apply to off-the-shelf software for these classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>A_SC</th>
<th>A_NSC</th>
<th>B_SC</th>
<th>B_NSC</th>
<th>C_SC</th>
<th>C_NSC</th>
<th>D_SC</th>
<th>D_NSC</th>
<th>E_SC</th>
<th>E_NSC</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable?</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>(P)</td>
<td>✔️</td>
<td>✔️</td>
<td>(P)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>☑️</td>
</tr>
</tbody>
</table>

Key:  ✔️ - Applicable | ✗ - Not Applicable | ☑️ - Applicable with details, read above for more | (P) - P(Center), follow center requirements or procedures

2. Rationale

Experience confirms that the quality and longevity of a software-reliant system is largely determined by its architecture. (See lessons learned "NASA Study of Flight Software Complexity" in section 6 of this Handbook.) The software architecture underpins a system's software design and code; it represents the earliest design decisions, ones that are difficult and costly to change later. The transformation of the derived and allocated requirements into the software architecture results in the basis for all software development work.

A software architecture:

- Formalizes precise subsystem decompositions.
- Defines and formalizes the dependencies among software work products within the integrated system.
- Serves as the basis for evaluating the impacts of proposed changes.
- Maintains rules for use by subsequent software engineers that assure a consistent software system as the work products evolve.
- Provides a stable structure for use by future groups through the documenting of the architecture, its views and patterns, and its rules.

3. Guidance

Architectural design is defined as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." More specifically, architecture is defined as "the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." The architecture process, after defining the structural elements, then defines the interactions between these structural elements. It is these interactions that provide the desired system behavior. Design rules are necessary for the enforcement of the architectural patterns for current and future software development (i.e., for open architecture systems).
The software architecture is drafted during the early life cycle phases of a project and baselined during PDR (Preliminary Design Review) (see SWE-019 and Topic 7.8 - Maturity of Life Cycle Products at Milestone Reviews). The drafting begins when the top-level (systems) requirements are collected and organized. The project's operational concepts document is prepared based on these top-level requirements. From this point the project development team develops, decomposes, and sub-allocates these requirements to multiple and more narrowly focused activities. (Tarullo 345) describes a model for creating software architectures by using the de-facto standard software modeling tool, UML (Unified Modeling Language) (v2.0). His approach fosters decomposition, which is a major practice used to control complexity in large (and small) software systems.) The evaluation and sub-allocation of these requirements result in a hierarchical ordering of the complete set of requirements, which forms the basis and initial structuring of the software architecture. Often this activity is accomplished by performing a functional or physical decomposition of the systems components and performance functions. As these allocated requirements are further matured and organized, a new set of statements evolves in the form of derived requirements. These derived requirements are nominally logical extensions of the original specified requirements. See SWE-049, SWE-050, and SWE-051 for more discussion on derived requirements.

NASA/SP-2007-6105, NASA Systems Engineering Handbook, 273 and the Defense Acquisition University's Systems Engineering Fundamentals Guidebook 174 both provide more detailed discussions of requirements decomposition. The latter document includes several example templates for conducting the decomposition activities. Some key concepts from these two references include: "Logical decomposition is the process for creating the detailed functional requirements that enable NASA programs and projects to meet stakeholders' needs." 273 "The allocation process is accomplished by "arranging functions in logical sequences, decomposing higher-level functions into lower level functions, and allocating performance from higher to lower level functions." 174 "The process is "recursive" (repeated application of processes to design next lower layer system products or to realize next upper layer end products within the system structure) and "iterative"(application of a process to the same product or set of products to correct a discovered discrepancy or other variation from requirements) and continues until all desired levels of the system architecture have been analyzed, defined, and baselined. 273

As the software development team starts its effort, it organizes the activities based on these allocated and derived requirements. The key step is to transform these requirements into a logical and cohesive software architecture that supports the overall systems architecture for the NASA project. The team develops a software architecture to serve as guidance for the development of the components and systems level software work products through a process known as architectural design.

Software architecture is commonly organized using the concepts of "views" and "patterns." A view is a representation of a set of system components and the relationships among them. Views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers or project managers. 313 Views are analogous to the different types of blueprints that are produced to describe a commercial building's architecture. Patterns in architectural design refer to the use of common or standard designs. "A pattern system provides, on one level, a pool of proven solutions to many recurring design problems. On another (level)it shows how to combine individual patterns into heterogeneous structures and as such it can be used to facilitate a constructive development of software systems." 191

The resulting software architecture also allows for the following: The verification of the software components, the integration of work products into systems, and the integration of the software systems into the rest of the project's systems. 224

SWE-057 calls for the software architecture to be documented. The required content for the Software Design Description document (see SWE-111) includes the CSCI architectural design. The actual format for recording and describing the architectural concept is left to the software project team (all projects are different!). As a minimum, include the following:

- An assessment of architectural alternatives.
- A description of the chosen architecture.
- Adequate description of the subsystem decomposition.
- Definition of the dependencies between the decomposed subsystems.
- Methods to measure and verify architectural conformance.
- Characterization of risks inherent to the chosen architecture.
- Documented rationale for architectural changes(if made).
- Evaluation and impact of proposed changes.

See Topic 7.7 - Software Architecture Description for additional information on the recommended kinds of content that usually appear in software architecture descriptions and for examples from a number of sources of outlines for documenting software architecture descriptions.)

In situations where the software architecture does need to be changed, dependency models now offer the potential for maintaining the architecture over successive revisions during the software life cycle by specifying rules explicitly that define the acceptable and unacceptable dependencies between subsystems. The dependency structure model is an example of a compact representation that lists all constituent subsystems/activities and the corresponding information exchange and dependency patterns. 295

The Software Architecture Review Board, a software engineering sub-community of practice, is a good resource of software design information including sample documents, reference documents, and expert contacts.

Additional guidance related to the software architecture development and documentation may be found in the following related requirements in this handbook:
4. Small Projects

Software architecture is one of those non-coding activities that can improve the quality of the software. Small projects may want a less-formal, more-affordable method of development. In general, if software development involves a low-risk and highly precedented system, the project can skimp on architecture. If the development involves high-risk and novel systems, the project must pay more attention to it. Smaller, less risky projects may do just enough architecture by identifying their project’s most pressing risks and applying only architecture and design techniques that mitigate them. Regardless of size, the resulting software architecture still needs to be adequately documented.

5. Resources

- Dependency Models to Manage Software Architecture, Sangal, N., Waldman, F., Crosstalk Magazine, November, 2005

5.1 Tools

Tools relative to this SWE may be found in the table below. You may wish to reference the Tools Table in this handbook for an evolving list of these and other tools in use at NASA. Note that this table should not be considered all-inclusive, nor is it an endorsement of any particular tool. Check with your Center to see what tools are available to facilitate compliance with this requirement.
Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering. UML is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development.

“IBM® Rational® Rhapsody® family provides collaborative design and development for systems engineers and software developers creating real-time or embedded systems and software. Rational Rhapsody helps diverse teams collaborate to understand and elaborate requirements, abstract complexity visually using industry standard languages (UML, SysML, AUTOSAR, DoDAF, MODAF, UPDM), validate functionality early in development, and automate delivery of innovative, high quality products.” (NOTE: Several versions are listed on the website for architecture, system engineering requirements analysis, design and model management, simulations to validate requirements and analyze architecture, and code generation. Unsure which versions are used within NASA. Listed requirements are those related to these topics.)

### 6. Lessons Learned

A documented lesson from the NASA Lessons Learned database notes the following:

**NASA Study of Flight Software Complexity. Lesson number: 2050:** This March 2009 study identified numerous factors that led to the accelerating growth of flight software size and complexity that in turn lead to flight software development problems. In particular the lesson learned states “Good software architecture is the most important defense against incidental complexity in software designs...” 571