Carnegie Mellon University
Software Engineering Institute

Maintainability

Rick Kazman
Phil Bianco
James Ivers
John Klein

December 2020

TECHNICAL REPORT
CMU/SEI-2020-TR-006
DOI: 10.1184/R1/12954908

Software Solutions Division

http://www.sei.cmu.edu

Table of Contents

List of Figures

List of Tables

Abstract

This report summarizes how to systematically analyze a software architecture with respect to a quality attribute requirement for maintainability. The report introduces maintainability and common forms of maintainability requirements for software architectures. It provides a set of definitions, core concepts, and a framework for reasoning about maintainability and the satisfaction (or not) of maintainability requirements by an architecture and, eventually, a system. It describes a set of mechanisms, such as patterns and tactics, that are commonly used to satisfy maintainability requirements. It also provides a method by which an analyst can determine whether an architecture documentation package provides enough information to support analysis and, if so, determine whether the architectural decisions contain serious risks relative to maintainability requirements. An analyst can use this method to determine whether those requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis. The reasoning around this quality attribute should allow an analyst, armed with appropriate architectural documentation, to assess the risks inherent in today's architectural decisions in light of tomorrow's anticipated needs.

1 Goals of This Document

This document serves several purposes. It is

- an introduction to maintainability and common forms of maintainability requirements
- a description of a set of mechanisms, such as patterns and tactics, that are commonly used to satisfy maintainability requirements
- a means to aid an analyst in determining whether an architecture documentation package provides enough information to support analysis and, if so, whether the architectural decisions contain serious risks related to maintainability requirements
- a means to aid an analyst in determining whether those maintainability requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis

This document is one in a series of documents that collectively represent our best understanding of how to systematically analyze an architecture with respect to a set of well-specified quality attribute requirements [Kazman 2020]. The purpose of this document, as with all of the documents in this series, is to provide a workable set of definitions, core concepts, and a framework for reasoning about quality attribute requirements and their satisfaction (or not) by an architecture and, eventually, a system. In this case, the quality attribute under scrutiny is *maintainability*. The reasoning around this quality should allow an analyst, armed with appropriate architectural documentation, to assess the risks inherent in today's architectural decisions in light of tomorrow's anticipated tasks.

There are several commonly used and documented views of software and system architectures [Clements 2010]. The *Comprehensive Architecture Strategy*, for example, proposes four levels of architecture, each of which can be documented in terms of one or more views [Jacobs 2018]:

1. functional architecture: The Functional Architecture provides a method to document the functions or capabilities in a domain by what they do, the data they require or produce, and the behavior of the data needed to perform the function.
2. hardware architecture: A Hardware Architecture specification describes the interconnection, interaction, and relationship of computing hardware components to support specific business or technical objectives.
3. software architecture: A Software Architecture describes the relationship of software components and the way they interact to achieve specific business or technical objectives.
4. data architecture: A Data Architecture provides the language and tools necessary to create, edit, and verify Data Models. A Data Model captures the semantic content of the information exchanged.

The focus of this document is almost entirely on the *software* architecture because a software architecture is the major carrier and enabler of a system's driving quality attributes. And since software typically changes much more frequently than hardware, it is often the focus of maintenance effort. However, software decisions to improve maintainability may impact the other architecture views.

In addition, other important decisions within a project will impact maintainability—or any other quality attribute, for that matter. Even the best architecture will not ensure success if a project's governance is not well thought out and disciplined; if the developers are not properly trained; if quality assurance is not well executed; and if policies, procedures, and methods are not followed. Thus, we do not see architecture as a panacea but rather as a necessary precondition to success—and one that depends on many other aspects of a project being well executed.

As we will show, there is no single way to analyze for maintainability. We can (and should) analyze for maintainability at different points in the software development lifecycle, and at each stage in the lifecycle this analysis will take different forms and produce results accompanied by varying levels of confidence. For example, if there are documented architecture views but no implementation, then the analysis will be less detailed, and there will be less confidence in the results than if an existing implementation could be scrutinized, tested, and measured. We will return to this issue of types of analysis and confidence in their outputs several times in this document.

For these reasons, we do not advocate a purely checklist-based approach to analyzing for maintainability or any other quality attribute (although we do use checklists, as we will discuss in Sections 6.1 and 6.2). The approach that we advocate guides an analyst in terms of questions to ask and architectural characteristics to look for. In any analysis, context is crucial—the meaning of "maintainability" or any other quality attribute must be interpreted in the context of the project, its stakeholders, and their needs. We specify this context via scenarios, as we will show.

2 On Maintainability

We naturally think about the need to maintain physical equipment, but what does it mean to maintain software? Physical equipment has elements that need to be adjusted or replaced. For example, in an automobile, there are tires and belts that wear, lubricants and filters that get dirty, and washer fluid and fuel that are consumed. Elements within the physical system change over time, and we perform maintenance to remediate the effects of those changes. On the other hand, software does not change over time. Every time we read a binary value, the ones do not wear down until they turn into halves and eventually zeros. We don't need to clean the dirt off our zeros so they don't build up and turn into ones. Nonetheless, software typically becomes harder to maintain over time, despite our desires and best efforts to make our software maintainable.

Software in most organizations represents a long-term investment. And although the software itself does not wear or degrade, the software's environment changes. For example,

- the underlying hardware changes
- the software and ecosystem that we depend on can change
- software that we integrate may change
- an adversary's knowledge may change so that a weakness in our software becomes a cyber vulnerability

Each of these environmental changes, along with newly introduced (or discovered) defects, will trigger a need to maintain our software so that it can continue to operate.[1] We use the quality attribute term *maintainability* to refer to the property of our software that makes these changes possible within acceptable ranges of cost, schedule, and risk. In addition, our knowledge of the design of the system can degrade due to staff turnover, inadequate documentation, and the passage of time. While these concerns are not triggers, they do increase the difficulty of maintaining our software system.

This report begins with a survey of definitions for maintainability. We introduce a set of *quality attribute scenarios*, including a *general scenario*, to more precisely define maintainability requirements. We then discuss the mechanisms that can be employed in a software architecture to promote maintainability. We conclude with a discussion of the various ways that an analyst can analyze for maintainability, focusing on analysis checklists and analysis methods.

2.1 Existing Definitions

We create definitions for quality attributes, like maintainability, so that we can label and categorize quality requirements. These labels are then used by several groups during the development phase:

[1] We exclude considerations of new requirements, as these are catalysts to *upgrade*, rather than maintain, the software.

- **Stakeholders** and **requirements engineers** use these labels during requirements elicitation to create checklists, assess coverage and completeness, and collect similar requirements. This group is often concerned with *why* the software must be maintained.[2]
- **Architects** use quality attribute labels to identify the relevant parts of the design body of knowledge to help them choose and instantiate mechanisms that promote the desired quality and satisfy the requirement.
- **Analysts** and **evaluators** use the labels to choose methods to apply, validate, and verify that the requirement is achieved.

Architects, analysts, and evaluators are usually less concerned with the *why* of the requirement and more concerned with the scope and impact of what must be maintained and constraints on how the maintenance will be performed. Therefore, these groups need enough precision in the requirement definition that it is actionable and verifiable.[3]

How can we define maintainability? It is certainly an important quality of software systems. A recent literature survey found that maintainability was the most-studied design-time quality by academic researchers [Arvanitou 2017], and it appears in practitioner-focused quality attribute taxonomies such as SQuaRE [ISO/IEC 2011a] and dependable systems [Avizienis 2001]. However, the definitions used in these taxonomies are somewhat broad:

- The standard ISO 25010(E) defines maintainability as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [ISO/IEC 2011a].
- Another standard, IEEE 14764-2006, defines it as the capability of the software product to be modified, with the objective of software maintenance being to modify existing software while preserving its integrity [IEEE 2006].
- Avizienis and colleagues define it as the "ability to undergo repairs and modifications" [Avizienis 2001].

Other quality attribute taxonomies, such as those of NASA [Wilmot 2016] and the Carnegie Mellon University Software Engineering Institute [Bass 2012], include the quality attribute of modifiability but not maintainability. And a study of 15 years of architecture evaluation findings reported that modifiability was the quality attribute of most concern to stakeholders for both cyber-physical and IT systems [Bellomo 2015]. Why do we mention this? Because these terms are often confused. While modifiability is not the same as maintainability, as we will show, the *goals* of designing for modifiability overlap significantly with the goals for maintainability. A more modifiable system will almost certainly be a more maintainable system. This is because the mechanisms for achieving high levels of modifiability in an architecture overlap considerably with the mechanisms for achieving high levels of maintainability.

2 More formally, stakeholders and requirements engineers are concerned that a requirement is *necessary* and *appropriate* [BKCASE 2018, Table 3].

3 More formally, architects and analysts are concerned that the requirements are *unambiguous*, *complete*, and *verifiable* [BKCASE 2018, Table 3].

The Department of Defense (DoD) defines software maintenance in DoD Instruction 4151.20 as follows [USDAS 2018]:

Software Maintenance: Includes actions that change the software baseline (adaptive, corrective, perfective, and preventative) as well as modification or upgrade that add capability or functionality. Encompasses requirements development, architecture and design, coding, and integration and test activities. Software maintenance and software sustainment are considered synonymous.

A U.S. Government Accountability Office report titled *Weapon System Sustainment* defines the four types of changes as follows [GAO 2019]:

Corrective Sustainment	Perfective Sustainment	Adaptive Sustainment	Preventive Sustainment
Corrective sustainment activities diagnose and correct software errors after the software is released.	Perfective sustainment activities consist of upgrades to software to support new capabilities and functionality.	Adaptive sustainment activities modify software to interface with changing environments.	Preventive sustainment activities modify software to improve future maintainability or reliability.

The authors of that GAO report further note that the categorization of a change depends on intention:

For example, an Army command is modifying software to incorporate Windows 10. This action may be described as corrective in that it addresses errors in previous versions of Windows; perfective in that it upgrades the software to support new capabilities and functionality provided by Windows 10; adaptive in that it can accommodate changes to firmware and hardware environments; and preventive in that it improves reliability. [GAO 2019]

2.2 Maintainability as a Quality Attribute

Synthesizing these definitions, we can make some statements about maintainability as a quality attribute:

- Maintainability, as a system-wide property, is achieved by paying close attention to three categories of characteristics: the management of dependencies, the management of system state, and the management of deployments. Each of these categories is elaborated upon in Section 3.1.
- Maintainability is concerned with modifications after the software baseline is established.
- The goal of a maintenance activity is to correct defects, adapt to changing environments, or improve a system's future maintainability or other quality attributes.
- The description of a particular maintenance activity is in the eye of the beholder: A particular change (or type of change) can be labeled differently, depending on the maintainer's intention.

We measure maintainability as the amount of work required to modify, test, and maintain our software base in response to changes in environmental elements. This measure may depend on who is performing the maintenance task and that individual's level of skill or knowledge.

These statements are helpful and provide context and scope for a maintainability requirement. However, the definition does not have any criterion for satisfaction. For example, while the definition refers to *a measure*, it does not specify any measure: How can we say that one architecture is more maintainable than another? How can we say that an architecture is sufficiently maintainable? Further, the definition does not distinguish among modifications. For any software item, some modifications will be easy, while others will be more difficult. This definition allows us to talk about maintainability, but to specify requirements for maintainability we need a more precise definition.

Fortunately, we are not the first to encounter this challenge. Initially, philosophers and scientists in the 1920s [Suppe 1998], and engineers in the World War II era [Ackoff 1968], have had the challenge of bridging from abstract concepts and properties to a quantitative, empirical reality. The approach used is to create an *operational definition* of the property, which specifies the operations (i.e., steps and actions) needed to measure the phenomenon of interest. For example, there are at least three ways to operationally define the strength of a given material:

1. the force, in pounds, required to break it
2. its resistance to penetration
3. hours of use in a particular application before failure

A casual reader would agree that each of these definitions describes "strength"; however, each definition provides a different criterion with which to measure that strength. Operational definitions are not exclusive. In a particular system, the strength of a material might be judged using any (or several) of these definitions. Further, operational definitions of a concept need not be exhaustive. There may be other operational definitions of strength besides the three we mention above.

In addition to specifying the steps and actions needed to measure the phenomenon of interest, an operational definition must include the environmental conditions when the measurement is made. In the example above, the temperature of the material will likely affect the measurement. Furthermore, an operational definition should specify the details of the measurement actions. In Definition 1 above, are we pulling or crushing the material? In Definition 2, is the penetration an impulse (e.g., a bullet) or a sustained pressure? Operational definitions allow objective and repeatable specification and verification of a property.

We approach the topic of maintainability from two perspectives: maintainability as an architectural property and maintenance as a task. *Maintainability* is about the degree to which an architect has anticipated and designed for the actual maintenance tasks that the system may undergo. A maintainability analysis of an architecture is thus predictive in nature.

In the field of software quality attributes, we can use *quality attribute scenarios* to create operational definitions. In Section 4, we define quality attribute scenarios for the quality attribute of maintainability.

3 Evaluating the Maintainability of an Architecture

In engineering it is common to collect data on the "performance" of a component before applying it in practice. We use evidence drawn from a component's past in an attempt to predict its future—in particular we are interested in predicting the important properties of that component. This process is no different when analyzing a software architecture. While we can evaluate the maintainability of an implementation of an architecture by examining its past—its revision history—we do not have the benefit of this rich historical record when evaluating *new* designs. Thus, we must analyze and evaluate new architectural designs in terms of their discernable properties. We use concrete scenarios to guide this analysis.

We cannot precisely evaluate the maintainability of an architecture any more than we can evaluate its performance, availability, or integrability. All quality attribute names are categories, and categories are too imprecise to be used for evaluation. Thus, we are better served by measuring the maintainability of an architecture with respect to a set of anticipated maintenance tasks. We specify such tasks as scenarios. We use scenarios to probe and analyze system characteristics, as we explain in Section 4. In that section, we will define a template for maintainability scenarios and provide examples of the kinds of maintenance tasks that a system might be subjected to. We need to understand the things that are involved in conducting maintenance activities to understand what it means to measure the maintainability of a system. To this end, in Section 5, we survey the techniques that the software engineering research literature has proposed for achieving maintainability. Finally, we will use these scenarios in our architecture analysis playbook (in Section 7).

It is important to reiterate that while we restrict our attention in this report to analyzing *architectural* information for maintainability, historically maintainability analyses have focused on richer sets of information derived from a project's code and its history. The advantage of these richer sets of information is that we can potentially create more precise analyses. The disadvantage is that such rich information is available only after we have built, deployed, and actually maintained a system. At that point it can be very expensive and time consuming to repair such problems. Thus, our objective in analyzing an *architecture* for its maintainability properties is to find a sweet spot wherein we can gain insight into the potential maintainability characteristics of a system before much, if any, code has been developed.

3.1 Measuring Maintainability

When we refer to maintenance tasks, we are only considering actions taken to fix a defect, replace a portion of a system with a newer version, or add something to improve a characteristic of the system (but not change its function or capability). Given this context, performing a maintenance task on a large, complex software-intensive system typically includes some combination of the following activities:

- determining what software elements need to change
- diagnosing the precise nature of the changes to be implemented in each element and designing these changes

- changing the software elements and any supporting artifacts, such as tests or build scripts
- validating that the changes were correct, potentially including recertification work as needed
- deploying[4] the new software and rolling back versions if defects are discovered in the newly deployed versions

These activities are supported by the following categories of system characteristics:

- The *management of dependencies* is about ensuring that the architecture has been designed to anticipate a set of probable changes so that those changes are localized and do not "ripple" throughout the system. In this way, the anticipated changes may be made efficiently and with confidence.
- The *management of system state* aids in controlling the cost and complexity of identifying and diagnosing problems (bugs) and aids in confirming the correctness of modifications.
- The *management of deployments* is about ensuring that the architecture includes support for efficiently deploying a system or making modifications to a system in its operational environments.

Consequently, when evaluating maintainability, we need to assess the architecture in terms of these characteristics (which we elaborate in the subsections below). Specifically, the analyst's job is to gauge the degree to which the architecture supports each of these characteristics.

Different scenarios will, of course, emphasize these concerns to different extents. For example, when replacing commercial off-the-shelf (COTS) or open source software with a new release of the same product, the concern and the effort typically center on validating that the new release did not break anything that was previously working. This maintenance activity crucially relies on being able to manage the system state—putting the system into states that fully test the replaced software. The ability to precisely manage the system state keeps the costs and time of testing the software to an acceptable level.

Each of these concerns will now be discussed in more detail.

3.1.1 Management of Dependencies

To assess how well the architecture supports the management of dependencies, we primarily focus on the degree to which they are

- *loosely coupled*: How interdependent are the architectural elements (modules, packages, threads/processes, deployable units, etc.)? This gives us insight into the probability that a change to one element will ripple to other elements. In general, lower levels of coupling will reduce the average cost of a change [Yourdon 1979]. But coupling may exist along multiple dimensions—syntactic, semantic, temporal, resource based, and state based—and each of these can be managed in slightly different ways [Kazman 2020].
- *highly cohesive*: Do elements have a small number of related responsibilities, or do they have many unrelated responsibilities? The level of cohesion of an architecture (or any element of an

[4] This deployment process consists of some combination of installing, updating, or replacing existing software; activating the software; and in some cases, deactivating, uninstalling, or rolling back software versions.

architecture) gives us insight into the likelihood that a given change will affect multiple elements. Higher levels of cohesion will reduce the average cost of change [Yourdon 1979].

- *understandable*: How easy is it to determine where a particular piece of functionality resides within the architecture? This gives us insight into how completely the responsibilities within the system are documented and understood, which, in turn, affects the average cost of a change. If responsibilities are well understood (because they are well documented), then modifications will, on average, be more systematic and hence less costly [Glass 1992].[5]

Sidebar: The Role of Information in Reasoning About Maintainability

Providing maintainers with sufficient information to efficiently perform their tasks (e.g., determining what changes are needed) improves the efficiency and quality of future maintenance activities. While the meaning of "sufficient information" depends on a number of factors, projects should strive to achieve several general goals.

1. The right information should be available to maintainers. What constitutes the right information for a system depends on a number of factors, including the specific maintenance activities being performed on a system, that system's quality attribute requirements, and the architectural decisions realized in the system.

Generally, the information required to understand quality attribute requirements is an excellent starting point. This information is used to analyze whether particular decisions will meet key requirements, guide development teams in realizing the architecture, and help maintainers avoid accidentally undermining the satisfaction of those same requirements.

For example, if a goal is to make a sensor management package easy to maintain in the future, then information that will help future maintainers includes the set of affected code modules, their responsibilities, the relationships among those modules, and the commit history of those modules. Architecture documentation can capture all of this information—except for the commit history—clearly and succinctly with good models and accompanying descriptions.

Best practices for structuring architecture documentation recommend using different views to address different concerns (e.g., as specified in publications by Clements or ISO/IEC [Clements 2010, ISO/IEC 2011b]). The selection of views, and consequently what information should be included in each, depends on the quality attribute requirements that a system must satisfy. Clements and colleagues recommend documenting at least three types of views, as each supports reasoning about different quality attributes.

- A module view, showing code elements and their relationships, is used to reason about the difficulty of making code changes.

[5] Creating and maintaining quality documentation is a governance issue. While these activities are very important to project success, particularly for long-lived projects, this report will not further discuss these aspects of managing dependencies as they are not about making and analyzing architectural design decisions.

- A component-and-connector view, showing runtime components (processes, threads, services, etc.) and their interactions, is used to reason about use of protocols and runtime coordination.
- An allocation view, showing how elements from a module or component-and-connector view are allocated to the computing infrastructure, is used to reason about system properties such as latency and mean time to failure.

2. *Information should be easy for maintainers to find.* Any creator of documentation should keep in mind the audience of that documentation—who they are writing the document for—as well as the purpose of the documentation; typically one or more purposes include analysis, construction, or education. In practice, architectural documentation is often spread across multiple media. For example, models may be the primary representation of views, accompanying documents may contain additional details like rationale, and some decisions about how tactics and patterns are realized may be found only in the code.

Best practices make use of searchable media, establish common terminology among teams, and clearly link information (e.g., how elements of different views relate to one another). These practices make it easier to find critical information. As projects progress through their lifecycles, it's also important to document how code elements map to architectural elements, which speeds understanding where tactics and patterns are used in code.

3. Information should be accurate. In practice, accuracy becomes more challenging once developers start writing code and even more so as a system ages and evolves. If teams are not diligent about keeping documentation and models up to date with code changes, then documentation can quickly become an untrusted source.

Best practices in areas such as architecture knowledge management [Weinreich 2016], architecture training, and architecture recovery can dramatically improve the ability of maintainers to understand the architecture that they are working with. Typically, the larger and more complex the architecture, the greater the need for such practices.

Finally, some architecture decisions make architectures themselves easier to understand and maintain. Code modules of an architecture should exhibit low coupling and high cohesion to aid in understandability. In fact, for complex systems, these characteristics are structural prerequisites to understandability. If large numbers of code modules are highly coupled, it will be more difficult to understand any one of them in isolation. And if the important code modules do not exhibit high cohesion, then even one of these modules will be challenging to comprehend. These challenges can persist and drag down a project, irrespective of the quality of the documentation.

3.1.2 Management of System State

To assess the degree to which the architecture supports the management of system state, we are primarily concerned with assessing how the architecture influences the amount of work it takes to set up and run test cases so that they are state controllable and state observable [Binder 2000]:

- *state controllable*: How well does the architecture support the control of system state? If the architecture allows the state of the system to be precisely controlled, then testing effort is dramatically reduced. Addressing this concern architecturally can minimize the effort to put the system into a state that needs to be tested.
- *state observable*: How much does the architecture support the precise observation of system state so that results of test cases can be verified? If the state of the system can be completely known, then the testing effort is dramatically reduced.

The degree to which a system state is controllable is affected by the system's degree of determinism, which is the property that a set of inputs will always produce the same output. Some systems have inherent nondeterminism. For example, a system may use a processing algorithm that requires random values for certain parameters. In other cases, architecture decisions can introduce nondeterminism, for example, certain load-balancing strategies. In both cases, the architecture can reduce or eliminate the nondeterminism by controlling parameters during testing, for example, by controlling the seed of a pseudorandom number generator so that the generated sequence is repeatable.

The degree to which a system state is observable is also affected by the system's degree of determinism. In a deterministic system, observing its state only at the system boundary may be sufficient to decide that test execution is correct. On the other hand, in a nondeterministic system, observing its internal state (and often also performing analyses of the observed state) may be needed to determine that test execution is correct.

3.1.3 Management of Deployments

To assess and measure the degree to which an architecture supports the management of deployments, we are primarily concerned with the degree to which they are granular, controllable, and efficient:

- *granular:* Can updates to parts of the system be deployed separately? Granularity manifests differently, depending on the type of system. In a service-oriented architecture with multiple redundant instances of a service executing at the same time, *granular* means that we can replace the instances one at a time while the other instances continue executing. For example, in an avionics system, *granular* could mean that we can update the cockpit display unit software now and update the mission computer software later. If deployments are all or nothing (in these examples, all instances of a service or all avionics software components), there is less opportunity for control and hence there is greater technical risk [Lenhard 2013, Lewis 2014]. An architecture that provides options to deploy small units can reduce risk. When dependencies are managed effectively, it is easier to manage deployments.
- *controllable*: How precisely can deployments be controlled and monitored? Does the architecture provide the capability to deploy at varying levels of granularity, monitor the operation of the deployed units, and roll back unsuccessful deployments [Lewis 2014]?

- *efficient:* How quickly can new units be deployed (and, if needed, rolled back) and with what level of effort?

3.1.4 Properties and Measures

These properties, and potential measures for each of them, are summarized in Table 1. The metrics referenced in Table 1 are further described in Section 6.3.

Table 1: Maintainability Properties and Example Measures

Concern	Measurable Property	Example Measure
Management of dependencies	How tightly coupled are components (e.g., modules, packages, threads/processes, deployable units)?	• Decoupling Level (DL) [Mo 2016], propagation cost (PC) [MacCormack 2006], Quality Model for Object Oriented Design (QMOOD), metrics to measure static coupling [Goyal 2014] • DL and PC metrics to measure dynamic coupling and historical coupling • average number of modules that need to be tested per new deployment and standard deviation from the mean
	How cohesive are component responsibilities?	• average number of modules affected by a change and standard deviation from the mean • Chidamber & Kemerer (CK) Lack of Cohesion of Methods (LCOM) measure [Chidamber 1994]
	How completely are the responsibilities of each component understood and documented?	• % of modules documented • “completeness” of documentation (e.g., range of information that is supplied for each element) • % of documentation that is updated whenever components change
Management of system state	How much system state can be observed?	• % of system elements that allow state to be observed • % of stateful behavior that is observable for each element • period of time over which data is available (e.g., logged data is available for the last *n* minutes)
	How much system state can be controlled?	• % of system elements whose state/behavior can be controlled • % of state/behavior of each element that can be controlled
	How efficient is testing?	• number of human interactions required per test • amount of time and effort to determine root causes of bugs • time elapsed between code commit and testing complete
	How deterministic is execution?	• % of bugs that can be reliably replicated
Management of deployments	How fine-grained are deployments?	• number of deployed services • ratio of deployable artifacts to code artifacts
	How controllable is deployment?	• % of successful deployments • % of deployments requiring human intervention • level of human effort required per deployment
	How efficient is deployment?	• number of human interactions required per deployed component • time between code commit and component deployment

Analyzing for maintainability, then, is about predicting the mix of activities required for a set of maintainability scenarios and predicting how difficult—in terms of measures of cost and risk—those activities will be for a specific (existing or planned) architecture. These risks may be schedule related, performance related, or technical.

We cannot give any specific guidance on target values for the measures presented in Table 1. The analyst needs to consider the values in the context of requirements and anticipated risks. For example, a "best" number of deployed services cannot be determined; more services mean more overhead and a more complex deployment, but they can provide benefits for performance and availability.

Many of the measures of maintainability in Table 1 cannot be estimated from architectural artifacts alone. Many of these measures can only be measured from a built and deployed system. However, this does not mean that we cannot perform a useful analysis of an architecture with respect to maintainability. As we will show in our "playbook" for architecture analysis in Section 7, even where a measure does not exist or its value cannot be reliably obtained, an architecture can still be examined for its *fitness for purpose* with respect to the properties described in Table 1.

3.2 Architectural Measures of Maintainability

Much of the prior research on assessing the maintainability of an architecture has focused on the following four properties of the components of a design: size, complexity, coupling, and cohesion [e.g., Bogner 2017, Seref 2016]. The size of a software system—and, in particular, the size of its components—will affect how easy it is to modify. In addition, the complexity of each component will affect the level of effort required. But the system's components do not live in isolation, so the coupling between components and the cohesion of components will also affect the maintenance effort: high coupling and low cohesion increase the likelihood that a maintenance-based modification will affect multiple components, thus increasing the overall effort—the time to determine what to change, the time to make the change, and the time to test and deploy the changed code. We elaborate on measures of coupling and cohesion in Section 6.3.

What do we look for in an architecture to support these activities? And how do we measure (and hence predict) the level of support for these activities from an architectural specification? We begin addressing this question by enumerating the major concepts that architects (and others, such as maintainers) tend to use to perform, support, and manage maintenance activities. The techniques, architectural decisions, and activities needed to make a system maintainable have much in common with other quality attributes, and many of these concepts are themselves widely considered to be first-order quality attributes. For example, to make an architecture more maintainable, it is important to make it easy to isolate and replicate faults and to deploy new versions of updated components. It is also important to ensure that a bug fix affects only a localized part of the architecture and does not "ripple" throughout the system.

In addition, it is important to ensure that architects and developers make decisions (and apply tactics and patterns) at the appropriate level of granularity, as these can affect the achievement of specific capabilities. We gave an example of this with respect to the management of deployments in Section 3.1.3, but granularity is important for other measures as well. For example, two services might be loosely coupled, but if we look inside each service, its code modules might be tightly coupled. If we want to replace one service without affecting the other, this level of granularity of coupling is appropriate. But if we want to modify the tightly coupled services, then the high coupling is problematic. This situation could result as a consequence of applying a tactic that reduces coupling at the granularity of services (e.g., using an intermediary) but does not affect coupling within services. This example also reinforces the importance of applying measures with respect to specific scenarios rather than at the level of an entire architecture.

3.3 Operationalizing the Measurement of Maintainability

When analyzing an architecture for maintainability, we have some analysis tradeoffs to make. We can analyze with respect to a particular set of scenarios and obtain a reasonably precise understanding of the architecture's accommodation of those scenarios. But that understanding is necessarily narrow—limited to just those scenarios that we have considered. Alternatively, we can analyze with respect to metrics and get a broad understanding of the architecture's overall level of predicted maintainability, as measured by the metrics, but this gives us no insight into the specific risks involved in responding to specific scenarios. Furthermore, scenario-based analyses and design-level structural metrics (like DL and PC) can be used to gain insight into a design. This insight therefore can be achieved *before* committing to an implementation, that is, by analyzing a design specification. But precisely because these metrics measure early artifacts, like design specifications, they may not accurately reflect the eventual state of the system. Measures of an implementation—accounting for its code, its commit history, its runtime behavior, its development costs, and other considerations—will be more precise, but these measures can only be made later in the system's lifetime.

For this reason, we recommend doing both: evaluating with respect to scenarios to get a deep understanding of some *anticipated* forms of maintenance and, later in the lifecycle, adding evaluations using measures from code, history, and project metrics to obtain a more precise understanding of the qualities that the architecture (or any major subsystem within the architecture) helps to realize.

4 Maintainability Scenarios

As stated in the book *Software Architecture in Practice*, quality attribute names themselves are of little use, as they are vague and subject to interpretation. The antidote to this vagueness is to specify quality attribute requirements as scenarios [Bass 2012]. A quality attribute scenario is simply a brief description of how a system is required to respond to some stimulus. Quality attribute scenarios are not use cases—they are architectural test cases. That is, they provide insights into the qualities that the architecture supports and any risks associated with the fulfillment of these scenarios.

A quality attribute scenario provides an operational definition (as introduced in Section 2.1) of a quality property of a system. We use scenarios to analyze and probe the system's architectural characteristics, as realized by the set of design mechanisms that have been chosen. Scenarios probe the "characteristic space" of a system in much the same way that code tests cover the state space of a system. Our goal, in traditional software testing, is to cover as much of the system's state space as possible with our test suite. However, for any nontrivial system, we cannot completely cover the state space. Similarly, we cannot completely cover the characteristic space of a system, which is why we need to prioritize scenarios. Our goal, in eliciting and prioritizing scenarios, is to cover enough of the highest priority scenarios that the most important maintainability risks are considered and minimized.

The use of scenarios to specify quality attribute requirements for software has a long history, dating back at least to the 1990s [Kazman 1994, 1996]. Published examples include scenarios to specify requirements for seven of the most commonly occurring quality attributes [Bellomo 2015]—availability, interoperability, modifiability, performance, security, usability, and testability, as described in the work of Bass and colleagues [Bass 2012]—and Klein and Gorton's use of scenarios to specify quality attribute requirements for big data systems, including the qualities of scalability and consistency [Klein 2015]. And quality attribute scenarios have had a long history of being used to "drive" architecture analyses [Bengtsson 1999, Kazman 2002].

A quality attribute scenario has six parts [Bass 2012]. The two most important parts are a *stimulus* and a *response*. The stimulus is some event that arrives at the system, either during runtime execution (e.g., an invalid message arriving on a particular interface) or during development (e.g., a development iteration completes). The response defines how the system should behave when the stimulus occurs. For example, consider these two simple scenarios:

- In response to an invalid message arriving, the system should log the event and send an error response message.
- In response to a development iteration completing, the unit and integration tests should be run and the test results reported.

The stimulus and response form the core of our operational definition by specifying the operation that we will measure. The third part of a scenario, the *response measure*, defines how we will measure the response and the satisfaction criteria. The response measure includes a metric and a threshold.

The other three parts of the scenario provide more details. We specify the *source* of the stimulus, to provide context for the scenario. We also specify the *environment*, which is the conditions under which the stimulus occurs and the response is measured. Finally, we specify the *artifact*, which is the portion of the system to which the requirement applies. Often, the artifact is the entire system, but in the example above, we might need to treat invalid messages on external interfaces differently from invalid messages on internal interfaces.

During requirements elicitation, we may specify the parts of a scenario in any order. We often begin with stimulus and response, although environment, source, or artifact may be the initial trigger for the requirement. In any case, once the scenario is specified, we usually arrange the parts to tell a story, as shown in Figure 1.

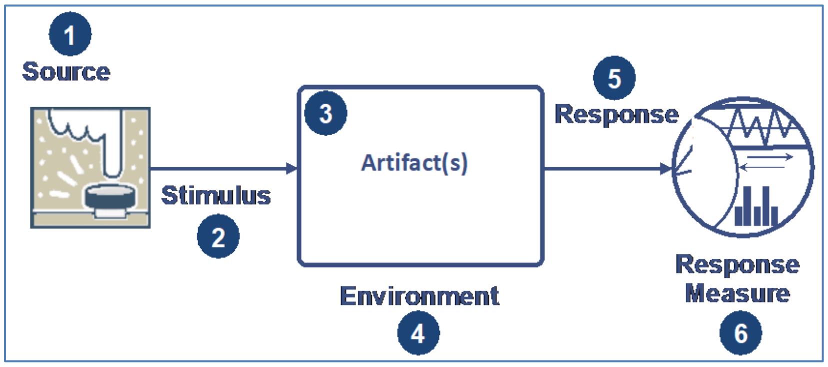

Figure 1: The Form of a General Scenario

Sidebar: Scenarios as Architectural Test Cases

In architecture analysis, scenarios are “architectural test cases.” We use them to determine whether the architecture—as envisioned or as created—is consistent with its specification. Before the system is built, we use scenarios to assess the quality of the architectural decisions. Once the system exists, we can continue to use scenarios to assess the quality of the architecture as it evolves.

For runtime quality attributes, scenarios may become much more than simply guides for analysts. They can be used as acceptance tests and made part of the regression test suite. Or they can even be manifested as system health measures that are logged or monitored continuously at runtime. If the checks are at runtime, checking can be built into a system monitor; if the checks are run at build time, checking can be built into a continuous integration pipeline. In either case, checking requires appropriate visibility into system response measures (e.g., the ability to track latency, resource usage, and mean time to failure). For non-runtime quality attributes (assuming that source code is available), we can monitor the quality or degradation of the architecture’s modular structure via architecture analysis tools, or we can monitor project management measures of the effort required to make changes (e.g., expected maintenance activities).

In this way, the quality of the architecture, including measures that reflect on its maintainability, can be continuously tracked and assessed. And if changes are made that undermine some architectural property, the test case fails and appropriate remedial action can be taken.

4.1 General Scenario for Maintainability

As we noted in the previous section, operational definitions are not exclusive. No single scenario specifies all of the possible measurements that could characterize a property like maintainability. However, if we look at the definitions of maintainability, we find some common themes. A *general scenario* maps those common themes into the parts of a quality attribute scenario, providing a template that we can use to create *concrete scenarios* for a particular system. The general scenario defines the *type* of the values for each part of the scenario, and a concrete scenario for maintainability of a system is created by specifying one or more system-specific values of the selected type for each part of the scenario. (We say "values" because, for example, a scenario might have more than one response measure.)

Here is the general scenario for maintainability[6]:

Scenario Part	Possible Type for Each Value	Discussion
Source	Program Manager	In the DoD software maintenance context, all changes to the software baseline are governed by the authority of the program manager, who collects information from a broad set of stakeholders.
Stimulus	Request to correct error (corrective change) Request to modify the quality of a function, e.g., more efficient, faster calculation, fewer resources required (perfective change) Request to operate in a changed environment (adaptive change) Request to replace a software element with a new version, e.g., apply a security patch (preventive change)	Although it is not necessary to categorize the request into one of the four categories of change, such categorization can provide context and motivation, and by considering all four categories, we ensure completeness. *Changed environment* can include changes to the platform or infrastructure, or changes to systems that our system interoperates with.
Artifact	Single software element Multiple software elements Entire software system	This part of the scenario defines the scope of the modification, if known. Possible artifacts may extend beyond system elements, e.g., tests and DevSecOps automation scripts.

[6] This general scenario is adapted from Bass and colleague's general scenario for modifiability [Bass 2012, Section 7.1].

Scenario Part	Possible Type for Each Value	Discussion
Environment	Original development organization Contractor (not original developer) Organic (not original developer)	In the DoD context, maintenance begins only after a baseline has been established. This is relevant to the scenario because we can assume that code and associated development processes have been established. The environment value may include refinements that enumerate the relevant capabilities of the maintenance organization, such as skills or access to networks or tools.
Response	One or more of the following: • Isolate the affected components and other artifacts. • Modify the code, tests, or DevSecOps artifacts. • Test and integrate the modification. • Validate, verify, or certify the modification. • Deliver or deploy the modification.	Deployment may be in scope for some types of systems and out of scope for other types of systems.
Response Measure	Cost, in terms of one or more of the following: • Number, size, or complexity of affected artifacts • Effort • Calendar time • Money (direct outlay or opportunity cost) • Extent to which this modification affects other functions or quality attributes • Introduction of new defects	Any modification will take time and cost money. The first four types of cost are directly associated with the modification, while the last two consider future costs.

4.2 Example Scenarios for Maintainability

Each of the following example scenarios is constructed by selecting one or more of the types of values from each of the six parts of the general scenario and specifying a system-specific value. For each example, we will use an easy-to-understand “typical” system. In practice, an analyst would choose values that are as precise as possible, in the context of the system.

In each example, notes in square brackets are added to trace back to general scenario types in cases where the traceability is not obvious.

4.2.1 Scenario 1: Apply Operating System Security Patches

This example scenario describes how critical security patches are applied to a networked avionics system such as a navigation system.

Scenario Part	Value
Source	Navigation System Program Manager
Stimulus	Direction to apply critical security patches to the Red Hat Enterprise Linux (RHEL) operating system (The vendor has asserted that these patches will not break any application functionality.)
Artifact	RHEL operating system, navigation system
Environment	CECOM Software Engineering Center (SEC) – This will be its first maintenance activity on this system. [organic, not original developer]

Scenario Part	Value
Response	Apply patches. Update the build and install automation scripts. Test the navigation system using existing automated test scripts. Deliver the navigation system to integration testing.
Response Measure	No changes are made to system application-level code. [affected artifacts] The SEC delivers the navigation system to integration testing within 14 days. [calendar time]

In this scenario—as in many scenarios—multiple response measures are specified. Furthermore, in some cases, multiple scenarios are needed to completely specify the quality attribute requirement. For example, the environment in the example scenario above was the first maintenance activity that the SEC performed, and one of the response measures was to deliver within 14 days. Another scenario, with identical stimulus and response values, might specify an environment for subsequent activities by the SEC, with a response measure of 5 days to deliver the modification.

4.2.2 Scenario 2: Software Error Isolation and Correction

This example scenario describes maintenance to the Flight Management System to correct a software error. The system clock cannot be set correctly on leap days (February 29).

Scenario Part	Value
Source	Navigation System Program Manager
Stimulus	Direction to correct the system clock leap-day software error
Artifact	System Initialization package
Environment	CECOM Software Engineering Center (SEC) – It has been maintaining this system for 3 years. [organic, not original developer]
Response	Isolate the error. Modify the code and add automated tests. Test the Flight Management System using automated tests. Deliver the Flight Management System software to integration testing.
Response Measure	The error is isolated and affected the artifacts identified within 14 days. [calendar time] The affected code modules are all located in the System Initialization package. [affected artifacts] The SEC delivers the navigation system to integration testing within 30 days. [calendar time]

4.2.3 Scenario 3: Planned Replacement of Radar Altimeter

This example scenario describes maintenance to the Flight Management System software to adapt to a planned replacement of the radar altimeter sensor. A lower cost, lower power sensor has become available, as anticipated in the original architecture design, so this modification to the Flight Management System is limited to replacing the Radar Altimeter Message Formatter module.

Scenario Part	Value
Source	Flight Management System Program Manager
Stimulus	Direction to perform planned replacement to integrate with the new radar altimeter

Scenario Part	Value
Artifact	Flight Management System–Radar Altimeter Message Formatter
Environment	NewCon, Inc. [contractor, not original developer]
Response	Replace with the new Message Formatter module, automated system tests, and automated build and install scripts. Test the Flight Management System using automated test scripts. Perform an integration lab test with the new radar altimeter using automated install scripts. Deliver the Flight Management System software to integration testing.
Response Measure	No "dead code" is left in the Flight Management System from old radar altimeter interface. [affected artifacts] Only the Message Formatter module is changed. [affected artifacts] The Modified Flight Management System is delivered within 60 days at cost of less than $250,000. [calendar time and direct outlay]

4.2.4 Scenario 4: Error During Update and Rollback to Previous State

This example scenario describes a failure occurring during an update with the system rolling back to a previous error-free version.

Scenario Part	Value
Source	Flight Management System Program Manager
Stimulus	Errors are detected during sensor manager update.
Artifact	Flight Management System–Sensor Manager
Environment	Maintenance phase; during update
Response	The system is rolled back to a previous working state, and diagnostic logs are captured.
Response Measure	Within "X" minutes

5 Mechanisms for Achieving Maintainability

An architect must choose a set of design concepts to construct a solution for any quality attribute requirement [Cervantes 2016], and the architecture that the analyst is given to examine will include design decisions about such concepts. We generically refer to these design concepts as *mechanisms*. We will discuss and provide examples of two important kinds of architectural design mechanisms: *tactics* and *patterns*.

A mechanism is an architectural approach that an architect may choose to achieve—and, ideally, control—a quality attribute response. Many discussions of mechanisms—for example, Bass and colleagues [Bass 2012]—focus on *technical mechanisms*, such as architectural patterns and tactics. Technical mechanisms are sufficient to satisfy requirements for quality attributes such as availability or consistency in a big data system. For other quality attributes, such as security and maintainability, technical mechanisms are necessary but not sufficient to satisfy some system-level requirements, and the technical mechanisms must be accompanied by *governance mechanisms*. For example, security defense-in-depth might begin with physical security, which requires governance to enforce access procedures. For maintainability, any modification to the software will be extremely difficult without governance, such as acquisition practices that ensure that appropriate architecture, design, and code documentation are produced; code reviews are performed; test suites are maintained; and employees are appropriately trained so that they do not undermine the integrity of the architecture with the changes they implement.

Governance mechanisms related to maintainability in the DoD context appear in discussions of the Modular Open System Approach (MOSA) [ODASD 2017] and DoD software acquisition practices [DIB 2019]. The rest of this section focuses on technical mechanisms for maintainability. Architecture approaches are commonly employed to satisfy the types of scenarios that we outlined in the previous section.

In practice, the terminology used for technical mechanisms is informal, and often the term *technical mechanism* itself is used to refer to any decision made during the architecture design process or to any fragment of the architecture that is intended to address some particular functional or quality attribute-related concern. In this report, we will consider two specific types of mechanisms:

- architectural patterns: Design patterns are conceptual solutions to recurring design problems that exist in a defined context. A pattern is architectural when its use directly and substantially influences the satisfaction of an architecture driver such as a quality attribute scenario [Cervantes 2016]. An architectural pattern defines a set of element types and interactions, the topological layout of the elements, and constraints on topology, element behavior, and interactions [Bass 2012].
- architectural tactics: Tactics are smaller building blocks of design than architectural patterns and focus on a single element or interaction, in contrast to a pattern that defines a collection of elements [Bass 2012].

Since tactics are simpler and more fundamental than patterns, we begin our discussion of mechanisms for maintainability with them.

5.1 Tactics

Tactics are the building blocks of design, the raw materials from which patterns are constructed. Each set of tactics is grouped according to the quality attribute goal that it addresses. The goals for the maintainability tactics shown in Figure 2 are to reduce the costs and risks of adding new components, modifying existing components, testing, and integrating sets of components to fulfill evolutionary requirements. As discussed in Section 3.1, the maintainability tactics achieve this by reducing the amount of coupling between components, reducing the distance between components, and making the testing and deployment of components easier, less costly, and more disciplined.

These tactics are known to influence the responses (and hence the costs) in the general scenario for maintainability (e.g., number of components changed, percent of code changed, effort, calendar time). The tactic descriptions presented below are derived, in part, from the third edition of *Software Architecture in Practice* [Bass 2012].

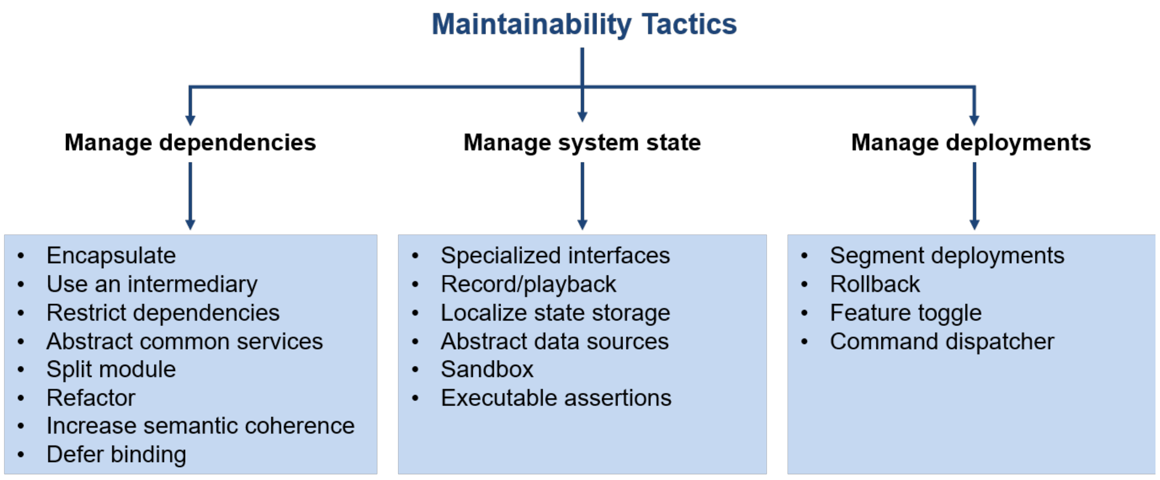

Figure 2: Maintainability Tactics

We discuss each of the tactics presented in Figure 2 in more detail below. For each tactic that we discuss, we describe the tactic and relate it to the measures defined in Section 3.1 as a way of characterizing the intent and impact of the tactic. The tactic descriptions are inspired by and derived, in part, from the third edition of *Software Architecture in Practice* [Bass 2012]. Table 2 summarizes the tactics presented in this section and how each relates to the measures presented in Sections 3.1 and 3.2.

Table 2: Maintainability Tactics and Their Relationships to Measures of Interest

Tactic	Coupling	Cohesion	Observ. of State	Control of State	Test Efficiency	Controllability	Granularity	Efficiency
Encapsulate	+	+	*	*	+			
Use an intermediary	+		*	*				
Restrict dependencies	+				+			
Abstract common services	+		*	*	*			

Tactic	Coupling	Cohesion	Observ. of State	Control of State	Test Efficiency	Controlla-bility	Granular-ity	Efficiency
Split module	*	+			+			
Refactor	+	+	*	*	*			
Increase semantic coherence		+						
Defer binding	+				–			
Specialized interfaces			+	+	+			
Record/playback				+	+			
Localize state storage			+	+	+			
Abstract data sources				+	+			
Sandbox					+			
Executable assertions			+					
Segment deployments						+	+	
Rollback						+		+
Feature toggle						+		+
Command dispatcher								+

Note: A plus sign indicates that the tactic positively addresses maintainability properties and hence measures, a minus sign indicates that the tactic has a negative effect, and an asterisk indicates that the tactic might positively or negatively address the measure, depending on its realization. A blank cell means that the property has no consistent effect on the measure.

It is worth discussing some of the asterisk entries in Table 2. For example, the encapsulate tactic may have positive or negative effects on the observation of state and control of state. Encapsulation may positively affect these properties if the state variables are appropriately encapsulated, but it may negatively affect those properties if the encapsulation hides the state variables. Similarly, refactoring may have a positive or negative effect on the observation of state, control of state, and test efficiency, depending on whether those were goals of the refactoring.

In addition, in Table 3, we show the anticipated effects of maintainability tactics on the various forms of coupling introduced in the work of Kazman and colleagues [Kazman 2020]. These forms of coupling are syntactic, data semantics, behavioral semantics, temporal distance, and resource distance.

Table 3: Selected Maintainability Tactics and Their Impacts on Various Aspects of Coupling

Tactic	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Encapsulate	+	+	+		
Use an intermediary					
Restrict dependencies					
Abstract common services	+	+	+	+	
Split module					

Tactic	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Increase semantic coherence					
Defer binding	+	+	+	+	+

Note: A plus sign indicates that the tactic positively addresses maintainability properties and hence measures. A blank cell means that the property has no consistent effect on the measure.

5.1.1 Managing Dependencies

One category of maintainability tactics deals with *managing dependencies* to limit the complexity of an anticipated change. The first five tactics in this category deal with structural decisions and constraints that inherently increase the maintainability of a software architecture.

Encapsulate: Encapsulation is the foundation upon which all other maintainability tactics are built. It is, therefore, seldom seen on its own, but its use is implicit in the other tactics described here.

Encapsulation introduces an explicit interface to a module. This interface includes an application programming interface (API) and its associated responsibilities. Encapsulation is also arguably the most common modifiability tactic because encapsulation reduces the probability that a change to one module propagates to other modules. This benefit is obtained because external dependencies are on the interface and not the implementation. Encapsulation reduces coupling, particularly syntactic coupling. Coupling that might have depended on the internals of the modules now depends only on the interface for the module. The external responsibilities can now directly interact with the module only through the exposed interface. Indirect couplings, however, such as temporal dependencies or dependence on quality of service will likely remain unaffected. Interfaces designed to increase modifiability should be abstract with respect to the details of the module that are likely to change—that is, they should hide those details.

Use an intermediary: Intermediaries are used for breaking dependencies among a set of components. Intermediaries can be used to intervene among (different types of) dependencies. For example, intermediaries like a publish-subscribe bus, shared data repository, or dynamic service discovery service all reduce dependencies among data producers and consumers by removing any need for either to know the identity of the other party. Intermediaries decouple components from one another, and thus may reduce coupling of all types.

Restrict dependencies: This tactic restricts the set of modules that a given module can interact with. In practice, this tactic is achieved by restricting a module's visibility (when developers cannot see an interface, they cannot employ it) and by restricting authorization to access it (restricting access to only authorized modules). This tactic, among others, is seen in service-oriented architectures, in which point-to-point requests are discouraged in favor of forcing all requests to be routed through an enterprise service bus so that routing and preprocessing can be done consistently. This tactic is typically used to limit syntactic coupling, although it could, in principle, restrict other forms of coupling as well.

Abstract common services: Where two modules provide services that are similar but not quite the same, it may be cost effective to implement the services just once in a more general (abstract) form.

Any modification to the (common) service would then need to occur just in one place, reducing modification costs. A typical way to introduce an abstraction is by parameterizing the description (and implementation) of a module's activities. The parameters can be as simple as values for key variables or as complex as statements in a specialized language that are subsequently interpreted. This tactic increases cohesion (the services that are abstracted must have a high degree of cohesion) and reduces all forms of coupling between the services and their clients (since a client cannot depend on the particulars of any given service).

Defer binding: When we bind the values of some parameters at a different phase in the lifecycle than the one in which we declared the parameters, we are applying the defer binding tactic. Deferring binding reduces all forms of coupling, since a client of some functionality can only depend on the explicit interface for the functionality that is exposed and not on, for example, its runtime behavior, memory usage, or any other property or side effect.

The final three tactics in this category are not specific structural decisions or constraints but rather architectural design techniques and approaches that, if done wisely, lead to greater maintainability.

Split module: If the module being modified includes a great deal of capability, the modification costs will likely be high because such modules have high complexity. Refining the module into several smaller modules should reduce the complexity and hence the average cost of future changes. This tactic is typically employed to increase the cohesion of the parts that remain after splitting.

Increase semantic coherence: If responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or moving a responsibility to an existing module. One method for identifying responsibilities to be moved is to hypothesize likely changes that affect a module. If some responsibilities are not affected by these changes, then those responsibilities should probably be removed. As its name suggests, the effect of this tactic is to increase coherence.

Sidebar: Tactics for Managing Dependencies and Their Relationship to Deployment Characteristics

Challenges in managing deployments are similar to those of managing a code base. The tactics that have been created over the years to aid in managing dependencies have found new life in the theory, practice, and tools for managing deployments. Tactics for managing dependencies primarily focus on reducing coupling and increasing cohesion so that a change made in one code element, such as a source file, is simple to understand and simple to do. When deployments become complex—for example, in large microservice architectures or in systems of systems—many of these same tactics have been found useful but are applied to units of deployment (e.g., containers, VMs, and services) rather than units of development (source code files).

Tactics such as encapsulation, restrict dependencies, abstract common services, and so forth all help achieve this goal. This should not be surprising for two reasons:

1. Tactics are abstract, so they are scale-free.
2. Microservices, being "micro," are just another way of packaging and interconnecting functionality, so it should be no surprise that the tactics that make sense for code packaged in files are largely the same as for code packaged in microservices.

Thus "encapsulation" has been rebranded as "containerization" or "componentization" [Lewis 2014]. The *defer binding* tactic is achieved in microservice architectures with a discovery service, and the *abstract common services* tactic has been realized as service proxies [Jamshidi 2018]. All of the tactics to manage the ever-growing complexity of architectures are found in microservice architectures, such as *refactor*, *split module*, and *increase semantic coherence*.

It must be stressed that these mechanisms, although they go by slightly different names, have been created by the microservices community for the very reasons that the tactics existed in the first place: to reduce coupling and increase cohesion among units of development and deployment.

5.1.2 Controlling and Observing System State

A second category of maintainability metrics deals with *controlling and observing system state* so that it is easy to test a system after some change has been made.

Specialized interfaces: Specialized testing interfaces allow a system to control or capture variable state values for a component either through a test harness or through normal execution. This tactic can improve the speed of testing and can improve the effectiveness of the regression suite by allowing modules to be more completely tested.

Record/playback: The state that caused a fault is often difficult to recreate. Recording the state when it crosses an interface allows that state to be used to replay the data stream and to recreate the fault. Record/playback refers to both capturing information crossing an interface and using it as input for further testing. This tactic also allows system state to be used to analyze and isolate bugs.

Localize state storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place. By contrast, if the state is buried or distributed, initialization of the system to a specific starting point is much more difficult. Furthermore, having a single storage location for all system states makes exporting a consistent state snapshot easier, reducing the time and effort to analyze and isolate failures.

Abstract data sources: Similar to controlling a program's state, easily controlling its input data makes it easier to test. Abstracting the interfaces makes it much easier to substitute test data. This can help increase the speed at which defects are revealed.

Sandbox: "Sandboxing" refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment. Testing is helped by the ability to operate the system in a way that has no permanent consequences or that can be rolled back without consequences. This can help increase the speed at which defects are revealed.

Executable assertions: Using this tactic, assertions are inserted at desired locations to indicate when and where a program is in a faulty state. These assertions are often designed to check that data values satisfy specified constraints or that the system is in an expected state. Such assertions can help increase the speed and effectiveness of testing. They do this by identifying faults closer to the point where they occur, improving isolation and hence remediation.

5.1.3 Deploying Software

Finally, there is a category of tactics that aids in *deploying software* and in managing those deployments. When a release of software is available, it can be deployed to one or more target environments. As previously mentioned, this deployment process consists of some combination of installing, updating, or replacing existing software; activating the software; and, in some cases, deactivating, uninstalling, or rolling back software versions.

Segment deployments: Rather than deploying in an all-or-nothing fashion, deployments are made gradually, often with no explicit notification to users of the system. This tactic has been realized in techniques such as phased rollout, incremental rollout, canary release, and blue/green deployment. This rollout may be for instances of a system or for the component parts of the system, such as services, that are gradually released. By gradually releasing, the effects of new deployments can be monitored, measured, and, if necessary, rolled back. This tactic minimizes the potential negative impact of an incorrect deployment by shrinking the granularity of deployments.

Rollback: If a deployment has defects or does not meet user expectations, then it can be rolled back to its prior state. Since deployments can involve multiple coordinated updates of multiple services, applications, or components, the rollback mechanism must be able to keep track of all of these deployments and must be able to reverse the consequences of any update made by a deployment. This tactic increases the degree to which the deployment can be controlled.

Feature toggle: Even when code is fully tested, issues may arise after new features are deployed. For that reason, it is convenient to be able to integrate a kill switch (or feature toggle) for new features. The kill switch automatically disables a feature in the system at runtime, without having to make a new deployment. If the new feature has any problem and needs to be removed, a software-enabled trigger for the kill switch is sufficient, and there is no need to roll back the software. This provides fine control over the granularity of deployments and increases the efficiency of deployments (particularly when features must be rolled back).

Command dispatcher: Deployments are often complex and require many steps to be carried out and orchestrated precisely. For this reason, deployment is often scripted, and deployment scripts should be treated like code. A command dispatcher executes the deployment script so that human error is minimized. This increases the degree of control and efficiency of deployments.

5.2 Patterns

As stated above, architectural tactics are the fundamental building blocks of design. Hence, they are the building blocks of architectural patterns. By way of analogy, we say that tactics are “atoms” and patterns are “molecules.” During analysis, it is often useful for analysts to break down complex patterns into their component tactics so that they can better understand the specific set of quality attribute concerns that patterns address and how they do so. This approach simplifies and regularizes analysis, and it provides more confidence in the completeness of the analysis.

In the three tables below, we show the maintainability tactics that are used to build each of the patterns described. Then we provide a brief description of each pattern, a discussion of how the pattern promotes maintainability, and finally the other quality attributes that are negatively impacted by these patterns (tradeoffs). It is important to understand the constituent tactics used in a pattern because no pattern is applied in a pure form. Analysts need to recognize how the instantiation of a pattern may compromise one or more constituent tactics, resulting in the pattern *not* actually achieving the desired quality attribute response.

Note also that just because a pattern negatively impacts some other quality attribute, this does not mean that the levels of that quality attribute will be unacceptable. Perhaps the added latency is negligible—only a small fraction of end-to-end latency—on the most important use cases. In such cases, the tradeoff is a good one, providing benefits for maintainability and modifiability while costing only a tiny amount of latency. For example, the use of an intermediary almost always negatively affects performance (specifically latency). This is inevitable; the interposition of an intermediary adds processing and communication steps. This is not to say, however, that the resulting latency of the system will be unacceptable. The added latency may be “down in the noise,” and the benefit from having the intermediary may be enormous. This is why we advocate consciously analyzing and, if possible, modeling and measuring quality attribute responses to consider the merits and tradeoffs of each in context.

It is also important to note that the tradeoffs described below are generalizations. Other architectural mechanisms or decisions applied with the pattern may change the impacts. For example, layering might be affected by “layer bridging,” where an element in Layer n directly accesses an element in Layer $n - 2$, without going through the abstraction provided by Layer $n - 1$. But if this layer bridging is isolated to a small and well-understood set of dependencies, and if this bridging is done to achieve a performance goal, then the tradeoff might be perfectly justified. These are the kinds of judgments that analysts need to make when assessing the appropriateness of the patterns selected and implemented.

Finally, this pattern list is not exhaustive. The purpose of this section is to illustrate the most common maintainability patterns—such as Model-View-Controller (MVC), Publish-Subscribe, Adapter, and Blue-Green Deployment—and to show how analysts may, by identifying the tactics employed in patterns, better understand a pattern’s quality attribute properties, strengths, weaknesses, and tradeoffs.

The tables below map the patterns to the maintainability tactics described above. The patterns themselves are described in the following subsections.

Table 4: Maintainability Tactics Mapped to Common Patterns – I

Tactic	Layers	Pipes and Filters	Publish-Subscribe	Adapter	MVC
Encapsulate	×	×	×	×	×
Use an intermediary			×	×	×
Restrict dependencies	×		×	×	×
Abstract common services	×		×		×
Split module	×				
Increase semantic coherence	×				×
Defer binding		×	×		
Specialized interfaces					
Record/playback					
Localize state storage					
Abstract data sources	×				
Sandbox					
Executable assertions					
Segment deployments					
Rollback					
Feature toggle			×		
Command dispatcher					

Table 5: Maintainability Tactics Mapped to Common Patterns – II

Tactic	Memento	Façade	Strategy	Intercepting Filter
Encapsulate	×	×	×	×
Use an intermediary		×		
Restrict dependencies	×	×	×	×
Abstract common services			×	×
Split module				×
Increase semantic coherence		×		
Defer binding			×	×
Specialized interfaces			×	
Record/playback				
Localize state storage	×			
Abstract data sources				
Sandbox				
Executable assertions				×
Segment deployments				
Rollback	×			

Tactic	Memento	Façade	Strategy	Intercepting Filter
Feature toggle				
Command dispatcher				

Table 6: Maintainability Tactics Mapped to Common Patterns – III

Tactic	Blue-Green Deployment	Canary Deployment	Rolling Deployment	Circuit Breaker
Encapsulate			×	
Use an intermediary				x
Restrict dependencies				x
Abstract common services				
Split module				
Increase semantic coherence				
Defer binding	×	×	×	
Specialized interfaces				
Record/playback				
Localize state storage				
Abstract data sources				
Sandbox		×		
Executable assertions				
Segment deployments	×	×	×	
Rollback	×	×	×	
Feature toggle	×	×	×	
Command dispatcher	×	×	×	

5.2.1 Layers

Layering is arguably the most common of all architectural patterns [Bass 2012, Buschmann 2007]. All complex systems experience the need to develop and evolve portions of the system independently. The developers of the system need a clear and well-documented separation of concerns so that modules of the system may be independently developed and maintained. Hence the software must be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse. To achieve this separation of concerns, the Layers pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage is, ideally, unidirectional. Layers completely partition a defined set of software, and each partition is exposed through a public interface.

Layers *restrict dependencies* by placing a constraint that each layer is allowed to use only the lower layer next to itself, which reduces the likelihood that changes to one layer will propagate past the next higher layer. Layers are also often designed to *abstract a set of common services* into its own layer to promote consistency and avoid the duplication of code. It is also common to see a data abstraction layer in data-intensive systems to reduce coupling with the underlying data store.

Benefits for maintainability

- Layered systems, if they contain only unidirectional dependencies, minimize the ripple effects of changes.
- Layers promote consistency and reduce instances of replicated code.
- Layers are often deployed using tiers, which can be deployed and monitored independently to detect and isolate issues.

Tradeoffs

- Layering can initially make a system more complex to build.
- Performance is typically negatively impacted because many invocations need to traverse additional layers. But in many cases, the actual impact on latency and throughput may be very small, which is a good tradeoff.

5.2.2 Pipe-and-Filter

Many systems are required to transform streams of discrete data items from input to output. Many types of transformations occur repeatedly in practice, so it is desirable to develop these as independent, reusable parts. The architectures of such systems benefit if they can be divided into reusable, loosely coupled components (filters) that share a common set of interaction mechanisms (pipes). In this way, they can be flexibly combined with one another. The components are easily reused and can be separately maintained and evolved.

The interaction in the Pipe-and-Filter pattern is characterized by a series of transformations of streams of data. Data arrives at a filter's input ports, is transformed, and then passes via output ports through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

The Pipe-and-Filter pattern uses the *encapsulation* tactic. The internals of filter processing are largely hidden. In this type of architecture, a developer needs to understand only the inputs and outputs to connect and use a filter. Another tactic that Pipe-and-Filter-based architectures usually use is *defer binding*. It is common that pipes and filters are often specified in configuration files and connected using that information when the system is started.

Benefits for maintainability

- Filter implementations can be changed without affecting other filters or configurations.
- Pipe-and-Filter architectures support rolling deployments, which can allow fine-grained deployment. This deployment can allow single filters to be independently deployed and monitored.

Tradeoffs

- This pattern is typically not a good choice for an interactive system as it disallows cycles, which are important for user feedback.
- Large numbers of independent filters can add substantial amounts of computational overhead since each filter runs as its own thread or process.

- Pipe-and-Filter-based architectures may not be appropriate for long-running computations without the addition of some form of checkpoint/restore functionality, as the failure of any filter (or pipe) can cause the entire pipeline to fail.

5.2.3 Publish-Subscribe

Publish-Subscribe is an architectural pattern where components communicate primarily through asynchronous messages [Buschmann 2007], sometimes referred to as "events" or "topics." The publishers have no knowledge of the subscribers, and subscribers are only aware of message types. Systems using the Publish-Subscribe pattern rely on implicit invocation; that is, the component publishing a message does not directly invoke any other component. Components publish messages on one or more events or topics, and other components register an interest in the publication. At runtime, when a message is published, the publish-subscribe (or event) "bus" notifies all of the elements that registered an interest in the event or topic. In this way, the message publication causes an implicit invocation of (methods in) other components. The result is a loose coupling between the publishers and the subscribers.

The Publish-Subscribe pattern has three types of elements:

- publisher component: sends (publishes) messages
- subscriber component: subscribes to and then receives messages
- event bus: manages subscriptions and message dispatch as part of the runtime infrastructure (also known as message-broker, message-oriented middleware, and enterprise service bus)

The Publish-Subscribe pattern incorporates many tactics. The publishers have no knowledge of the subscribers and vice versa. They rely only on the publish-subscribe interface, which is a realization of the *encapsulation* tactic. Pre- and post-processing of events allow architects to *abstract common services* such as transformation and data validation. The event distribution mechanism is an *intermediary* and *restricts dependencies*, which results in loose coupling among components. The publishing of topics and the subscribing to topics can *defer binding* to runtime.

Benefits for maintainability

- Publishers and subscribers are independent and hence loosely coupled. Adding or changing subscribers requires only registering for an event and causes no changes to the publisher.
- System behavior can be easily changed by changing the event or topic of a message being published, consequently changing which subscribers might receive and act on the message. This seemingly small change can have large consequences, as features may be turned on or off by adding or suppressing messages. This capability supports the feature toggle tactic.
- Events can be logged easily to allow for record and playback to reproduce error conditions that can be challenging to recreate manually.

Tradeoffs

- Some implementations of Publish-Subscribe can negatively impact performance (latency). For example, in some cases, a component cannot be sure how long it will take to receive a published message. In general, systemic performance and resource management are more difficult to reason about in Publish-Subscribe systems.

- Publish-Subscribe can negatively impact the determinism produced by synchronous systems. The order in which methods are invoked, as a result of an event, can vary in some implementations.
- Publish-Subscribe can negatively impact testability. Seemingly small changes in the event bus—such as a change in which components are associated with which events—can have a wide impact on system behavior and quality of service.
- Some Publish-Subscribe implementations limit the mechanisms available to improve security (integrity). Since publishers do not know the identity of their subscribers and vice versa, end-to-end encryption is limited. Messages from a publisher to the event bus can be uniquely encrypted, and messages from the event bus to a subscriber can be uniquely encrypted; however, end-to-end encryption requires all publishers and subscribers to share the same key.

5.2.4 Adapters

An Adapter is a pattern that adds an abstraction—a component with its own interface—between a concrete service and clients of that service [Gamma 1994]. The Adapter pattern is commonly used when clients should not change their behavior even as the services they use (and the interfaces of those services) change. Clients can invoke the Adapter component, via its interface, which redirects them into calls to existing components that are being reused. The Adapter functions as a wrapper on top of an existing component, allowing two incompatible interfaces to interact and removing all knowledge of the concrete service from the client. The client needs to depend only on the Adapter interface.

This pattern enables sets of components that provide similar functionality but expose different interfaces (for example, components that manage sensors from different manufacturers) to work together, serving as an intermediary. The Adapter allows components with incompatible interfaces to work together by wrapping its own interface around an already defined component interface, promoting encapsulation. This helps isolate changes to underlying components by restricting dependencies that require communication to go through the adapter. Developers need only to write a new adapter rather than changing many components. This pattern is most useful for making software that was designed previously to work in new environments.

Benefits for maintainability

- Adapters facilitate communication and interaction between two or more incompatible components. This allows new sensors' or components' details to be hidden from the rest of the components to localize changes to an underlying adapter implementation.

Tradeoffs

- Adapters may negatively impact performance (latency) by introducing additional calls and overhead.

5.2.5 Façade

The Façade pattern is quite similar in intent to the Adapter pattern. A Façade is an interface with an explicit goal of hiding the complexity of the underlying modules [Gamma 1994]. A Façade is employed to hide complexity—to provide a simpler interface to a client than what the service exposes. In contrast, the Adapter pattern provides a more generic interface—one that the client expects—so that the client does not need to adapt, but this interface may not be inherently simpler than what the services expose.

A Façade is composed of several important tactics. This interface uses *encapsulation* by hiding the interaction details of the underlying components. The façade *restricts dependencies* by serving as an intermediary (e.g., performing additional functionality before or after calls) between a set of components and the rest of the system. A façade also isolates the changes to the rest of the system and prevents ripple effects when systems are refactored to achieve other quality goals.

Benefits for maintainability

- The number of dependencies on subsystems are greatly reduced, preventing ripple effects from changes such as COTS upgrades.
- The simple interface ensures that clients use the set of interfaces that are encapsulated in a consistent manner, reducing potential defects.

Tradeoffs

- The indirection of going through a façade introduces overhead, as compared to direct calls to the underlying functionality.
- The constraints imposed by the façade, which is typically simpler than the functionality it is hiding, may not be appropriate for all use cases and could result in duplication of underlying components or a loss of some functionality.

5.2.6 Model-View-Controller (MVC)

User interface software is typically the most frequently modified portion of an interactive application. For this reason, it is important to keep modifications to the user interface software separate from the rest of the system. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. Both representations should reflect the current state of the data.

The MVC pattern separates application functionality into three kinds of components [Krasner 1988]:

- a *model*, which contains the application's state data and responds to state queries
- a *view*, which displays some portion of the underlying model and interacts with the user
- a *controller*, which mediates between the *model* and the *view* (or views), manages the notifications of state changes, and allows the user to switch among views

The MVC components are connected to each other via some form of notification, such as events or callbacks. These notifications contain state updates. A change in the model needs to be communicated to the views so that they can be updated. An external event, such as a user input, needs to be communicated to the controller, which may, in turn, update the view, the model, or both. Notifications may be either push or pull.

The relationships between the components of MVC are shown in Figure 3.

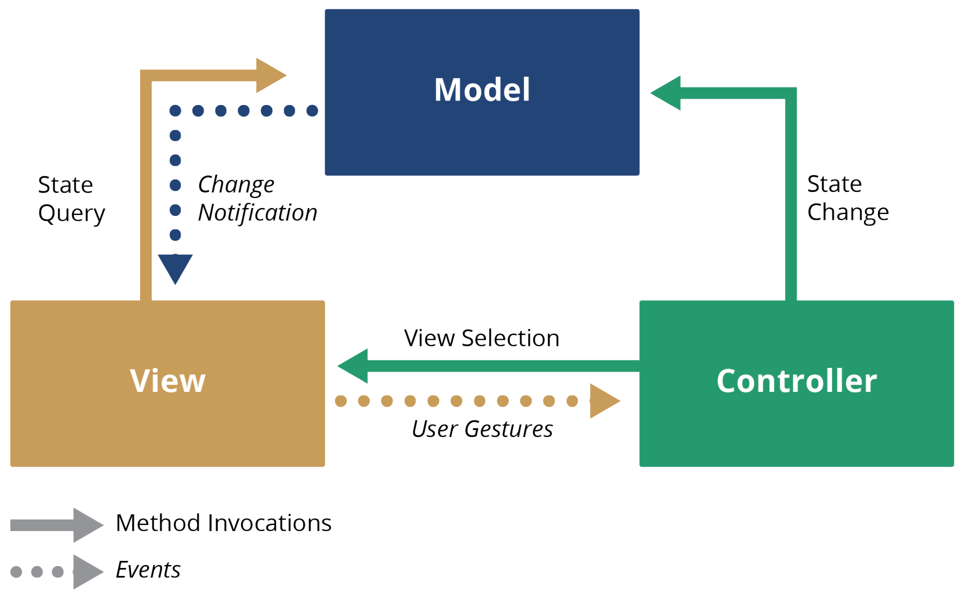

Figure 3: Model-View-Controller Pattern

The model encapsulates the application state, and the controller defines application behavior and serves as an intermediary. The restrict dependencies tactic is used since all interactions are defined by the pattern. Semantic coherence is increased by having clearly defined responsibilities for each of the major components.

Benefits for maintainability

- Because the MVC components are loosely coupled, it is easy to develop and test them separately.
- Changes to one of the MVC components have minimal impact on the others.
- Multiple views are relatively simple to implement and maintain separately.

Tradeoffs

- The design and implementation of three distinct kinds of components, along with their various forms of interaction, can be costly, and this cost may not make sense for relatively simple user interfaces.
- The match between the abstractions of MVC and commercial user interface tool kits is not perfect. The view and the controller split apart input and output, but these functions are often combined into individual widgets. This splitting can result in a conceptual mismatch between the architecture and the user interface tool kit.

5.2.7 Memento

The Memento pattern was originally created to allow the state of an object to be saved externally so that it can be restored later if necessary. This can be useful at different levels of abstraction for tactics such as *rollback* when problems are found after deployment changes [Gamma 1994].

The Memento pattern is implemented with three elements:

- An *originator* is an object that has an internal state that needs to be saved so that changes can be undone.
- A *caretaker* can change the state of the originator but also needs to be able to undo those changes.
- A *memento* is a copy of the originator's state.

The caretaker applies some operation (or operations) that cause the state of the originator to change. Before doing so, the caretaker requests a memento object from the originator. To roll back to the state prior to these operations, the caretaker can return the memento to the originator, thus restoring it to its previously saved state. Since only the originator is allowed to access the memento encapsulation, the originator's encapsulation is preserved.

The memento itself should never be modified by the caretaker. When using this pattern, care should be taken because the originator can affect other objects or resources ("ripple" effects). The Memento pattern is meant to operate on a single object, so those ripple effects will not be undone.

Benefits for maintainability

- Encapsulation of the originator is maintained since there is a constraint on access to the memento. This is an example of the restrict dependencies tactic.

Tradeoffs:

- If very large data sets need to be saved by the originator, this may have a negative impact on system resource utilization, including memory and processor utilization.
- Large data sets in originators can also impact system restart time, thus impacting availability.

5.2.8 Blue-Green Deployment

The Blue-Green Deployment pattern promotes the use of two production environments and allows for segmented deployment. These two environments should be identical whenever possible. Any time one of the production environments is live, we call this the *green environment*. The *blue environment* is used to finalize testing as developers prepare a new release. After the software passes all critical tests, the developers switch to the blue environment with the upgrade and blue becomes live. If developers have confidence that the upgrade is working in production in the blue environment, they start deploying upgrades to the idle green environment. On the other hand, if users experience any issues, they can safely roll back to the green environment that has not been updated [Fowler 2010].

Benefits for maintainability

- When faults or errors are detected after deployment, rollback can be done quickly by switching back to the environment that hasn't been upgraded.
- This pattern allows for segmented deployment with a smaller scope that can be tested and monitored more easily than a large deployment.
- Being able to switch easily minimizes the impacts on users when deployments fail and even when they are successful. Deployment can be done with zero downtime.

Tradeoffs

- Two identical production environments are not practical for some systems and can be prohibitively costly.
- Some systems suffer from a lack of hardware for testing, and this pattern compounds that problem by siphoning resources away from a test environment.
- Maintaining fidelity between the environments requires careful management.

5.2.9 Canary Deployment

Canary deployment builds on the Blue-Green pattern. It is very similar but adds a further refinement to the segment deployment tactic. In a canary release, a *segmented deployment* is often rolled out to a subset of the deployed instances (such as servers). Then some subset of user requests is routed to the new instances, while others continue to run on the previous (stable) version. This routing can be based on organizations (for example, roll out internally before externally), a small percentage of instances, or a more sophisticated classification of user profiles. Incrementally rolling out the upgrades minimizes risk and allows for testing a new deployment and gathering detailed metrics about how the production environment is affected as the load is gradually increased [Sato 2014].

Benefits for maintainability

- The Canary pattern has all the benefits of Blue-Green Deployment.
- It enables finer grained metrics collection as the number of users increases.
- It carries lower risk than simply switching all traffic to a new deployment (as is done with the Blue-Green pattern).

Tradeoffs

- All of the tradeoffs from Blue-Green Deployment apply here.
- The Canary pattern adds complexity to managing routing rules or configurations. And small changes in configurations may produce different results, which can lead to invalid or misinterpreted metrics.

5.2.10 Circuit Breaker

Since deployments may involve large numbers of moving parts and large numbers of dependencies, it is often desired to decouple the effects of a failure of some part of the system from other parts. With a Circuit Breaker, a service is wrapped and the wrapper monitors the state of the service [Nygard 2017]. If it is determined that the service is not operating consistently within its specification, the circuit breaker is tripped, and all subsequent calls to the service return an error immediately. This ensures that such dependencies do not slow down other parts of the system due to, for example, repeated timeouts, which increase the degree of control over the deployment.

Benefits for maintainability

- The Circuit Breaker pattern makes it possible to deploy new (instances of) services with the knowledge that they will not seriously undermine existing latency and availability properties.

Tradeoffs

- The Circuit Breaker pattern adds some up-front complexity in terms of the need to map the service being deployed.

5.2.11 Rolling Deployment

The Rolling Deployment pattern is used for systems that can be broken down into several nodes that can be deployed independently. These nodes can be an entire server running a monolithic application or a small container running a single component or a few components.

Whenever an update is ready for deployment, developers deploy a new node running the updated or new functionality. The previous version runs at the same time as the new version while developers monitor the behavior of the new node until they are confident that the new version is stable.

Rolling deployment allows for rollback since the previous version is still running until the new version is monitored sufficiently. Feature toggle is also supported since the node can be easily brought offline by a command or configuration change.

Benefits for maintainability

- This pattern minimizes the impact on users since developers can roll out deployments quickly with almost no downtime.
- Finer grained deployments simplify the root cause analysis of faults or changes in the required quality of the service.

Tradeoffs

- Since developers roll out changes in a production environment server by server, there will be multiple (at a minimum two) versions of the software running at the same time. This can lead to unpredictable results, which can make it more difficult to determine the root cause of newly emerging issues.
- Some servers will need to be offline while they are being upgraded, thus reducing processing capacity.

5.2.12 Strategy

The Strategy pattern enables specific functionality (such as an algorithm) to be chosen at runtime, rather than being hard-wired in to the behavior of the system. This pattern is an instantiation of the *defer binding* tactic. It allows for certain functionality (e.g., data validation) to vary independently from the clients that use them. The pattern defines context components, strategy components, and concrete strategy components. Some or all of the context components' behavior is implemented by a concrete strategy component. The strategy component will choose the concrete strategy components based on parameters, configurations, or runtime conditions. This pattern provides value when interfaces are stable, but implementations change frequently [Gamma 1994]. The Strategy pattern is the basis for dependency injection.

The Strategy pattern encapsulates underlying implementations and allows for changes based on the desired quality of the service. This is also an instance of providing a specialized interface for testing where test components can be substituted easily for operational components to inject error conditions.

Benefits for maintainability

- The Strategy pattern supports maintainability by allowing test components to be chosen when the real components have yet to be implemented.
- Decoupling the implementation from the interface allows the client and component to be changed independently.
- This pattern allows functionality to be added without changing the calling context in which the pattern is invoked.

Tradeoffs

- The Strategy pattern can reduce code readability since additional files need to be examined to understand the complete runtime behavior.
- The Strategy pattern removes some complexity from components but introduces a dependency on any framework used to implement the pattern (for example, the Spring Framework for dependency injection in Java).
- The added indirection will increase latency, but as we have noted in other contexts, this increase may be small and thus the tradeoff may be a good one.

5.2.13 Intercepting Filters

An intercepting filter allows for a set of services to be implemented as a set of pre- or post-processing filters without requiring changes to the core component that they serve. The binding of the processing can be deferred until compile, load, or runtime. These filters may be woven in using technologies such as aspects from aspect-oriented programming, in which functionality is woven in during compilation or by language frameworks that weave aspects at runtime.

Intercepting filters have similar benefits to the Strategy pattern but have a different strategic use case. The intercepting filter is more applicable for some standard functionality that needs to be invoked in many places and in many different contexts, often requiring a chain of filters. The Strategy pattern is more applicable for a stable interface with several useful implementations' variants.

The intercepting filter is an example of splitting modules to simplify them by removing explicit pre- and post-processing from a functional module. The pre- and post-processing are very similar to executable assertions where input value ranges can be checked before or after execution.

Benefits for maintainability

- The Intercepting Filter pattern supports maintainability by allowing faults to be injected into components via preprocessing so that conditions can easily be met for unit testing.
- Filters can be easily turned on when they are needed and off when they are not needed.
- This pattern allows functionality that will be used broadly to be added in many places without changing the underlying components.

Tradeoffs

- The Intercepting Filter pattern can reduce code readability since additional (often nonlocal) files need to be examined to understand the complete behavior in cases where important functionality is required (e.g., security token checks, data validation).
- Intercepting filters remove the complexity of components but may introduce a dependency on a framework that implements the pattern. The Spring framework is an example of a common dependency injection framework.
- Long chains of filters can increase latency considerably.

6 Analyzing for Maintainability

An analyst's job is to judge the appropriateness of the mechanisms built into the architecture of a system in light of the set of maintenance tasks anticipated. The degree of appropriateness is a function of the risks and costs of the anticipated maintenance tasks. Analysts can specify these potential or anticipated tasks using scenarios, as we exemplified above. For consistency and repeatability, analysts can guide stakeholders to derive those scenarios from the general scenario for maintainability.

Analyzing for maintainability at different points in the software development lifecycle will take different forms. The different analysis options are sketched in Table 7. If analysts are analyzing a system in its early design stages, they may only have a reference architecture or a functional architecture, for example. In this case, they cannot make detailed predictions or claims about the level of difficulty associated with a specific anticipated maintenance task. What the analyst can employ, at that early stage, is a checklist or tactics-based questionnaire. These analysis techniques will reveal the designer's intentions with respect to maintainability.

On the other hand, if the analysts have received a defined and documented product architecture—perhaps including multiple hardware and software architecture views—but little or no coding has been done, they can still employ checklists and tactics-based questionnaires to understand the design intent. But, as shown in Table 7, the analysts can also begin to think about employing metrics to measure the as-designed level of coupling in the system, in chosen subsets of the system, or between selected parts of the system.

There is no one-size-fits-all analysis methodology or tools that we can recommend. The analysis team needs to respond appropriately to whatever artifacts have been made available for analysis. And the analysis team and the product owner need to understand that the accuracy of the analysis and the expected degree of confidence in the analysis results will vary according to the quality of the available artifacts.

Table 7: Lifecycle Phases and Possible Analyses for Maintainability

Lifecycle Phase	Typical Available Artifacts	Possible Analyses
Early Design	Set of selected mechanisms, tactics, and patterns	Checklist Tactics-based questionnaire
Software Architecture Defined	Set of containers for functionality (e.g., modules, services, microservices) and their interfaces	Checklist Tactics-based questionnaire Coupling metrics • structural • semantic

Lifecycle Phase	Typical Available Artifacts	Possible Analyses
Implemented System	Set of containers for functionality (e.g., modules, services, microservices) and their interfaces Commit history Issue-tracking history Runtime profiles/traces	Checklist Coupling metrics • structural • semantic • history based • dynamic

6.1 Tactics-Based Questionnaires

Architectural tactics have been presented thus far as design primitives, following the work of Bass, Cervantes, and their colleagues [Bass 2012, Cervantes 2016]. However, since tactics are meant to cover the entire space of architectural design possibilities for a quality attribute, we can use them in analysis as well. Each tactic is a design option for the architect at design time. But used in hindsight, tactics represent a taxonomy of the entire abstract space of design possibilities for maintainability.

Specifically, we have found these tactics to be very useful guides for interviews with the architecture team. These interviews help analysts gain rapid insight into the design approaches taken, or not taken, by the architect and the risks therein. These risks might be one or more of the following:

- *risks of omission, such as* the architect did not use this tactic and should have
- *risks of commission, such as* this tactic is not really required, which increases costs with little or no commensurate benefit
- *risks on how a tactic was implemented, such as* team members implemented a tactic themselves when a better, more mature alternative already existed
- *managerial risks, such as* the tactic has not been properly communicated to the team

Although the information could be derived from other sources such as document review or reverse engineering, interviews with the architect are typically quite efficient and can be very revealing. For example, consider the list of maintainability tactics-inspired questions presented in Table 8. The analyst asks each question and records the answers in the table.

Table 8: Tactics-Based Questionnaire for Maintainability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage Dependencies	Does the system consistently **encapsulate the** functionality? This typically involves isolating the functionality under scrutiny and introducing an explicit interface to it.				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system consistently **use an intermediary** to keep modules from being too tightly coupled? For example, if A calls concrete functionality C, the system might use an abstraction B that mediates between A and C.				
	Does the system **restrict dependencies** between modules in a systematic way? Or is any module free to interact with any other module?				
	When two or more unrelated modules regularly change together—that is, they are consistently affected by the same changes—do you regularly **refactor** to isolate the shared functionality as common code in a distinct module?				
	Does the system **abstract common services** in cases where it provides several similar services? For example, this technique is often used when the system must be portable across operating systems, hardware, or other environment variations.				
	Do you regularly make modules simpler by **splitting the module**? For example, if the system has evolved from a large, complex module, would you normally split it into two (or more) smaller, simpler modules?				
	Does the system consistently support **increasing semantic coherence**? For example, if responsibilities in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or moving a responsibility to an existing module.				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system regularly **defer binding** of an important functionality so that it can be replaced later in the lifecycle, perhaps even by end users? For example, does the system use plug-ins, add-ons, or user scripting to extend the functionality of the system?				
Manage System State	Does the system or do the system components provide **specialized interfaces** to facilitate testing and monitoring?				
	Does the system provide mechanisms that allow information that crosses an interface to be recorded to use it later on for testing purposes (**record/playback**)?				
	Is the state of the system, subsystem, or modules stored in a single place to facilitate testing (**localized state storage**)?				
	Does the system make it easy to **abstract data sources**, for example, by abstracting interfaces? Abstracting the interfaces makes inserting test data simpler.				
	Can the system be executed in isolation (a **sandbox**) to experiment with or test it without worrying about having to undo the consequences of an experiment?				
	Are **executable assertions** used in the system code to indicate when and where a program is in a faulty state?				
Manage Deployments	Do you **segment deployments**, rolling out new releases gradually (in contrast to releasing in an all-or-nothing fashion)?				
	Can you automatically **roll back** deployed components if you determine that they are not operating in a satisfactory fashion?				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system employ **feature toggles** to automatically disable a newly released feature (rather than rolling back the newly deployed component) if the feature is determined to be problematic?				
	Does the system use a **command dispatcher** to script complex sequences of deployment instructions?				

These questionnaires can be used by an analyst, who poses each question to the architect and records the responses, as a means of conducting an architecture analysis. To use these questionnaires, follow these four steps:

1. For each tactics question, fill the "Supported" column with Y if the tactic is supported in the architecture and with N otherwise. The tactic name in the "Tactics Question" column is bolded.
2. If the answer in the Supported column is Y, then in the "Design Decisions and Location" column, describe the specific design decisions made to support the tactic and enumerate where these decisions are manifested (located) in the architecture. For example, indicate which code modules, frameworks, or packages implement this tactic.
3. In the "Risk" column, indicate the anticipated or experienced difficulty or risk of implementing the tactic using a (H = High, M = Medium, L = Low) scale. For example, a tactic that was of medium difficulty or risk to implement (or which is anticipated to be of medium difficulty, if it has not yet been implemented) would be labeled M.
4. In the "Rationale" column, describe the rationale for the design decisions made, including a decision *not* to use this tactic. Briefly explain the implications of this decision. For example, explain the rationale and implications of the decision in terms of its effect on cost, schedule, evolution, and so forth.

Thus, when using this set of questions in an interview, the analyst records whether or not each tactic is supported by the system's architecture, according to the opinions of the architect. When analyzing an existing system, the analyst can additionally investigate the following:

- Are there are obvious risks in the use (or nonuse) of this tactic? If the tactic has been used, record how it is realized in the system (e.g., via custom code, generic frameworks, or externally produced components).
- What are the specific design decisions made to realize the tactic, and where in the code base the implementation (realization) is it found? This is useful for auditing and architecture reconstruction purposes.
- What rationale or assumptions are made in the realization of this tactic?

While this interview-based approach might seem simplistic, it can be very powerful and insightful. In architects' daily activities, they likely do not take the time to step back and consider the bigger picture.

A set of interview questions such as those in Table 8 forces the architect to do just that. This process can be quite efficient; a typical interview for a single quality attribute takes between 30 and 90 minutes.

6.2 Architecture Analysis Checklist for Maintainability

As presented in the work of Bass and colleagues, one can view an architecture design as the result of applying a collection of design decisions [Bass 2012]. We view architecture design and analysis as two sides of the same coin [Cervantes 2016]. Design and analysis are not distinct activities—they are intimately related. Any design decision made by an architect should be analyzed. What we present next is a systematic categorization of these decisions so that an architect or analyst can focus attention on the design dimensions likely to be most troublesome.

An architect faces seven major categories of design decisions. These decisions will affect both software and, to a lesser extent, hardware architectures. These categories are

1. allocation of responsibilities
2. coordination model
3. data model
4. management of resources
5. mapping among architectural elements
6. binding time decisions
7. choice of technology

These categories are not the only way to classify architectural design decisions, but they do provide a rational (and exhaustive) division of concerns. The concerns addressed in these categories might overlap, but it's all right if a particular decision exists in two different categories because the duty of the architect and the analyst is to ensure that every important decision has been considered.

Some of these design decisions might be trivial. For example, architects may have no choice of technology decisions to make if they are required to implement the software on a prespecified platform over which they have little or no control. Or for some applications, the data model might be trivial. But for other categories of design decisions, architects might have considerable latitude.

For each quality attribute, we enumerate a set of questions—a checklist—that will lead an analyst to question the decisions made (or not made) by the architect and for some of these decisions to refine the questions into a deeper analysis. The checklist for maintainability is presented below.

Table 9: Checklist for Maintainability

Category	Checklist
Allocation of Responsibilities	Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. Do the following for each potential change or category of changes: • Determine the responsibilities that would need to be added, modified, or deleted to make the change. • Determine what other responsibilities are impacted by the change. • Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module and places responsibilities that will be changed at different times in separate modules.
Coordination Model	Determine which functionality or quality attribute can vary at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number of modules. Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules. For those elements for which maintainability is a concern, use a coordination model that reduces coupling, such as publish-subscribe, defers bindings, such as enterprise service bus, or restricts dependencies, such as broadcast, façade, or layering.
Data Model	Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction. For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. Do the following for each potential change or category of change: • Determine which data abstractions would need to be added, modified, or deleted to make the change. • Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions. • Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, or their creation, initialization, persistence, manipulation, translation, or destruction. • Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes. Design the data model so that items allocated to each element of the data model are likely to change together.

Mapping Among Architectural Elements	Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors, virtual machines) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve determining, for example, • execution dependencies • assignment of data to databases • assignment of runtime elements to processes, threads, or processors Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.
Resource Management	Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example, • determining what changes might introduce new resources, remove old ones, or affect existing resource usage or contention • determining what resource limits will change and how Ensure that the resources after the modification are sufficient to meet the system requirements. Encapsulate all resource managers, and ensure that the policies implemented by those resource managers are themselves encapsulated and bindings are deferred to the extent possible.
Binding Time	Do the following for each change or category of change: • Determine the latest time at which the change will need to be made. • Choose a defer binding mechanism that delivers the appropriate capability at the time chosen. • Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. • Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	Determine what maintenance tasks are made easier or harder by technology choices. • Will technology choices help to make, test, and deploy modifications? • How easy is it to modify the choice of technologies (in case some of these technologies change or become obsolete)? Choose technologies to support the most likely maintenance tasks. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

6.3 Coupling and Cohesion Metrics

There are well-validated measures of some architecture properties that tend to lead to maintainability. These measures may give us insight into the tendency of an architecture to be maintainable. For example, there are several metrics for object-oriented designs, such as the Quality Model for Object-Oriented Design (QMOOD) quality attributes [Goyal 2014] and the CK metrics suite [Chidamber 1994], both of which include metrics for measuring coupling and cohesion.

Additionally, there are architecture-level coupling measures, such propagation cost (PC) [MacCormack 2006] and Decoupling Level (DL) [Mo 2016]. PC measures how tightly the elements of a system are coupled; the more tightly they are coupled, the higher the score. DL measures how well components in a system are decoupled from one another; the more decoupled they are, the higher the score.

In addition, the DL metric has been used to assess nonstructural coupling between components by measuring “history coupling” [Xiao 2014, Cai 2019]. For example, Gall and colleagues showed that logical relations among files can often be reflected by how they were changed together as recorded in the revision history [Gall 1998]. History coupling is calculated based on a representation of a system where two components are said to be “historically coupled” if they are changed together in a commit. Thus, if a project has a recorded commit history, we can calculate this coupling measure. This measure is of interest because historical co-commit relations can reveal nonstructural forms of coupling, such as control, data, timing, and resource-based coupling.

Each of these metrics has been extensively empirically validated, so they can be used with a reasonable degree of confidence. Furthermore, the DL, PC, and QMOOD metrics can be applied when an architecture description has been created but little or no coding has been done. These metrics can give early insight into the properties of the design with respect to some important aspects of maintainability.

7 Playbook for an Architecture Analysis of Maintainability

This playbook outlines an approach to combine the checklists and questionnaires presented in the previous sections with information about mechanisms to analyze an architecture to validate the satisfaction of a maintainability requirement. The playbook provides a process, illustrated with a running example, that will guide experts to perform architecture analysis in a more repeatable way.

The process has three phases and seven steps. The Preparation phase gathers the artifacts needed to perform the analysis and evaluation. The Orientation phase uses the information in the artifacts to understand the architecture approach to satisfying the quality attribute requirement. The process ends with the Evaluation phase, when the analysts apply their understanding of the requirement and architecture solution approach to make judgments about that approach. The phases and steps are summarized in Table 10.

Table 10: Phases and Steps to Analyze an Architecture

Phase	Step
Preparation	Step 1–Collect artifacts.
Orientation	Step 2–Identify the mechanisms used to satisfy the requirement.
	Step 3–Locate the mechanisms in the architecture.
	Step 4–Identify derived decisions and special cases.
Evaluation	Step 5–Assess requirement satisfaction.
	Step 6–Assess the impact on other quality attribute requirements.
	Step 7–Assess the costs/benefits of the architecture approach.

The analyst might identify missing artifacts during the Preparation phase and missing or incomplete information within those artifacts during the Orientation phase. At the end of each step in the Preparation and Orientation phases, the analyst must decide whether sufficient information is available to proceed with the process.

This process can be applied at almost any point in the development lifecycle. The quality of the architecture artifacts—breadth, depth, and completeness—will inform the type of analysis and evaluation performed in Step 5 and the degree of confidence in the results. Early in the development lifecycle, lower confidence may be acceptable, and the analyst can work with lower quality artifacts and simpler analyses, as suggested in Table 7. Later in the lifecycle, the analyst needs higher confidence and therefore higher quality artifacts and more and deeper analyses.

7.1 Step 1–Collect Artifacts

In this step, the analysts collect the artifacts that they will need to perform the analysis. These include quality attribute requirements and architecture documentation.

The first artifact the analysts need is the maintainability requirement that they want to validate. The requirement must be stated so that it is measurable, for example, as a quality attribute scenario (as discussed above). Let’s use a variant of the example in Scenario 4 from Section 4.2, where we have specified the artifact as “Flight Management System–Sensor Manager,” and we will add a new scenario in which an error occurs, requiring rollback. Let’s call these Scenarios 5 and 6, respectively.

Scenario 5: Sensor Replacement

Scenario Part	Value
Source	Flight Management System Program Manager
Stimulus	A sensor needs to be replaced.
Artifact	Flight Management System–Sensor Manager
Environment	Maintenance phase; new contractor, not the original developer
Response	Code is modified and tests are completed. The system is ready for integration testing and able to support rollback if issues are discovered.
Response measure	Within 60 calendar days and no code changes required outside the Sensor Message Formatter

Scenario 6: Errors Requiring Rollback

Scenario Part	Value
Source	Flight Management System Program Manager
Stimulus	Errors are detected during sensor manager update.
Artifact	Flight Management System–Sensor Manager
Environment	Maintenance phase; during update
Response	The system is rolled back to a previous working state.
Response measure	Within “X” minutes

Next, the analyst needs the other quality attribute requirements. As noted above, architecture designs embody tradeoffs, and decisions that improve maintainability may have a negative impact on the satisfaction of other quality attribute requirements. In Step 6, the analyst will check that the architecture decisions made to satisfy this requirement do not adversely affect other quality attribute requirements, and more information about the complete set of quality attribute requirements means greater confidence in the results of that step.

Finally, the analyst needs architecture documentation. Early in the architecture development lifecycle, the documentation may be just a list of mechanisms mapped to quality attribute requirements, perhaps identifying tradeoffs. As the architecture is refined, partial models or structural diagrams become available, accompanied by information about key interfaces, behaviors and interactions, and rationale that provides a deeper link between the architecture decisions and quality attribute requirements. When the architecture development iteration is finished, the documentation should include complete models or structural diagrams, along with the specification of interfaces, behaviors and interactions, and rationale.

7.2 Step 2–Identify the Mechanisms Used to Satisfy the Requirement

To begin the Orientation phase, there are several places to look to identify mechanisms used in the architecture. If the architecture documentation includes a discussion of rationale, that can provide unambiguous identification of the mechanisms used to satisfy a quality attribute requirement. Other activities include looking at the structural and behavior diagrams or models and recognizing architecture patterns. Naming of architecture elements may indicate the mechanism being used. The analyst may also look at the file structure and naming of source code repositories, if they exist, to find mechanisms. The analyst may need to use all of these to identify the mechanism or mechanisms that are being used to satisfy the maintainability requirement. Frequently, two or more mechanisms are needed to satisfy a requirement. If the analyst has access to the architect(s), this is an excellent time to use the tactics-based questionnaires, as described in Section 6.1. In a short period of time, the analyst can enumerate all of the relevant mechanisms chosen (and not chosen).

For the example requirement above about replacing a sensor, let's say that the project is already partially through the development, and parts are already deployed into a development environment. The original development team created documentation that illustrates a desire to segment deployment and a fairly robust rationale writeup justifying other decisions relating to maintainability. The rationale states that the team used a Façade pattern to abstract the complexity of managing sensors from the rest of the system to satisfy this maintainability requirement, along with a mechanism for rollback using the Memento pattern. Since the development is underway and partially deployed in a development environment, a quick walkthrough with developers showed a set of deployment scripts that they created to reduce human error during deployment. This mechanism is the command dispatcher tactic. Another mechanism that has yet to be implemented is to use executable assertions to help determine when and where a program is in a faulty state.

The analyst performs a quick check to decide if the referenced mechanisms are likely to contribute to satisfying the maintainability requirement. In this case, all mechanisms that are enumerated above are known to positively impact maintainability measures. They describe two patterns and three tactics for maintainability—so the check passes.

In contrast, if the documented rationale (or the architect) stated that the architecture used a broker for protocol translation and data format transformations to achieve this requirement, these choices would raise a red flag since those mechanisms are usually associated with improving syntactic interoperability, but not necessarily maintainability. The analyst might decide to stop the architecture analysis at this point and gather more information from the architect. The point of this quick check is not to analyze the mechanism or decision in detail but simply to assess whether the architecture analysis is on the right track before devoting more effort to it.

In some cases, the appropriateness of a mechanism is less clear. For example, the rationale in this case might specify that a health monitoring mechanism is used. Simply checking if something is alive can support maintainability in a crude sense but often is insufficient since it only uncovers failures and not a faulty state. In cases like this, the analyst should proceed carefully; the architect may have chosen an inappropriate mechanism, mislabeled the mechanism used, or used the mechanism in an atypical way that may or may not be appropriate.

7.3 Step 3–Locate the Mechanisms in the Architecture

Following our example, the analyst needs to use the architecture documentation, or an interview with the architect(s), to find where these mechanisms are used in the architecture. As seen in the tactics-based questionnaires, it is important to consider *how* a tactic or pattern is implemented.

Our scenario is concerned with the replacement of a sensor in the flight management system. The analyst may be able to look at the documentation and find a structural diagram that includes where the sensor inputs and outputs are handled. With this diagram in hand, finding instances of the façade mechanism for managing sets of sensors should not be difficult because it is a major abstraction in the system. The analyst should also be able to locate a diagram that provides insight into the state capture necessary for using the Memento pattern to allow maintainers to roll back to a previous build of the whole sensor manager that was known to work properly.

The analyst should next look for documentation relating to the *segmented deployment* tactic and the *command dispatcher* tactic that describes the partial and automated builds. This documentation is often separate from software architecture documentation. These mechanisms are often documented using flow charts and accompanying descriptions of each step of a release pipeline. Sometimes automated build scripts are not specifically described in the architecture but found while walking through the code repository. In this case, the developers recognized that the build process was complex and error prone, which led them to solve the problems they encountered by using scripts.

Finally, the analyst must be able to conceptually integrate the mechanisms. The rationale for satisfying the requirement (e.g., no code changes outside the sensor manager) said that the Façade pattern was used to abstract sensor management and data formats from the rest of the system. This raises a question: where in the system would you find responsibilities to determine if sensors' values were in expected ranges? This is an issue of *managing dependencies*, one of the categories of questions in the Tactics-Based Questionnaire. One answer could be that the sensor manager has the responsibility to implement a circuit breaker to return errors whenever sensor data out of range is requested or pushed to other components. However, the analyst finds that, in reality, faulty values are transmitted throughout the system and a set of executable assertions are used in multiple components that check to see if the sensor values are in expected ranges.

Before finishing this step, the analysts should check that the mechanisms are being used in parts of the architecture that relate to the requirement that they are analyzing. To assess how well a mechanism contributes to satisfying a quality attribute requirement, it is not sufficient to stop after the sanity check in Step 2. That establishes only the presence of the mechanisms, not their suitability or adequacy for the scenario being considered. The analysts must identify where and how the mechanism was instantiated in the architecture to assess whether it will have the desired effect. For example, they find a façade mechanism, but it is used to encapsulate a relatively straightforward group of components and provides little to no simplification. This use of that mechanism is not likely to improve the maintainability of the flight management system.

7.4 Step 4–Identify Derived Decisions and Special Cases

Most architecture mechanisms are not simple, one-size-fits-all constructs. The instantiation of a mechanism requires making a number of decisions, with some of those decisions involving choosing and instantiating other mechanisms. For instance, our example employs a Façade pattern and Memento pattern (employing the rollback tactic). One set of decisions about using that mechanism is concerned with the scope or granularity. (Refer to the *manage dependencies* and *manage system state* categories in the Tactics-Based Questionnaire).

This case includes several alternatives:

- Which modules are encapsulated by the façade? Should the system restrict dependencies by forcing all other modules to go through the façade? If so, how does it enforce this? Will there be exceptions to support other qualities such as performance?
- Should the architect "split" the façade into finer grained pieces to reduce its complexity?
- How does the architect determine the components for where and how its state will be copied to facilitate rollback (using the Memento pattern)?
- Should developers refactor modules to remove any responsibilities that serve different purposes (e.g., they are not semantically coherent)?
- Will the system and/or façade provide specialized interfaces to support testing?

If the architect decided to support specialized interfaces (the last major alternative above), then there is a set of subsequent derived decisions about how the specialized interfaces are realized. They can be hard coded to always serve as checks at runtime, or they can be disabled through configuration or parameters to execute only during testing. Another option is to use a form of dependency injection to insert checks during testing or diagnostic modes.

To assess how well a mechanism contributes to satisfying a quality attribute requirement, it is not sufficient to stop after the quick check in Step 2. The analyst must verify that the mechanism meets the requirement and hence must evaluate how the mechanism was instantiated, which usually involves tracing the decisions about the mechanism instantiation to the derived decisions and the selected alternatives that address them.

As the analysts identified the mechanisms in Step 3 above, they also started to identify derived decisions. For example, using the questions outlined above, the analysts identified that the façade's granularity may need to be refactored as the system evolves and hence they should pay attention to the dependencies on the façade to achieve the maintainability objective of restricting code changes to the sensor manager.

The analyst's next derived decision might be "How did the architect determine the components where the state will be copied to facilitate rollback (e.g., Memento pattern)?" This is a *manage deployments* decision in the Tactics-Based Questionnaire. For the maintainability requirement that the analyst is validating, a good answer to these questions includes the following:

- The architecture supports rolling out new releases gradually.
- The architecture supports rollback, and the system has a well-defined release pipeline.
- The system has a well-defined, flexible command dispatcher that supports automated builds when release triggers occur.

If these are all true, then the analyst should be able to identify all the artifacts that need to be copied when a release trigger occurs without huge manual effort. (On the other hand, if the driving quality attribute requirement was, for example, reducing the cost of field service technicians rather than maintainability, then the architect might have chosen to make the release all or nothing, creating a line-replaceable unit [LRU]. Changing anything in the LRU would require a completely tested unit to be sent into the field, and incremental updates would not be possible.)

Another derived decision could be to simplify modules by separating responsibilities (see the *manage dependencies* categories in the Tactics-Based Questionnaire). The software might have sensor management and sensor fusion responsibilities intermingled and managed under the façade. This introduces a large, complex set of responsibilities in modules managed under the façade that are costly to change. This complexity also makes it, on average, more challenging to isolate where problems are occurring. Separating sensor management and sensor fusion simplifies the modules and can reduce the underlying implementation and root cause analysis complexity. However, it also creates another derived decision: when splitting these responsibilities, should the architects continue to restrict the communication paths using the façade to retrieve sensor values, or should they allow direct access to the new modules, in effect removing responsibilities from the original façade?

Finally, some mechanisms have special cases that warrant special attention. For example, using a mechanism to flexibly inject tests can lead to complexity. Problems can occur when different developers inject code into the same places. The issue is that developers often do not control which tests execute first. This can be especially difficult when one test is purposefully inserting a fault while the other is testing normal functionality (happy cases). The functional test could fail and may result in wasted effort to debug a false positive.

In this example, analysts should pay attention to changes to the release pipeline and the command dispatcher build scripts. These will help ensure that the state needed to roll back and the specialized testing interfaces are managed appropriately. The façade, while important to isolating changes for the “no code changes to other components” measures, is generally well understood. Deployment technologies, on the other hand, are less well understood and changing rapidly.

7.5 Step 5–Assess Requirement Satisfaction

The analyst has completed preparation and orientation and begins the Evaluation phase. The analysis performed to assess whether the architecture satisfies the quality attribute requirement will depend on the nature of the requirement and the mechanism(s) being applied. For example, the analyst assesses a quality attribute requirement for portability to a different hardware platform, and the mechanism used is the Layers pattern with a hardware abstraction layer as the lowest layer. The analysis should include checks for layer skipping, which introduces syntactic dependencies. The analysis should also include examining the interface that the hardware abstraction layer provides to other layers and checking that all those interface services could likely be constructed on other hardware platforms.

Recall that the requirement in Scenario 5 is to add a new sensor. Our measures are “within 60 calendar days” and the code changes are isolated to the sensor manager. The architecture mechanisms are Façade, Memento, segment deployment, command dispatcher, and executable assertions. In Step 4, the analyst identified several derived decisions that need to be considered in the analysis:

- Which modules are encapsulated by the façade? Should the system restrict dependencies by forcing all other modules to go through the façade? Or will there be exceptions to support other qualities such as performance?
- Should the architect “split the façade” into finer grained pieces to reduce complexity?
- How does the architect determine the components for where the state will be copied to facilitate rollback (using the Memento pattern)?
- Should developers refactor modules to remove any responsibilities that serve different purposes (e.g., they are not semantically coherent)?
- Will the system and/or façade provide specialized interfaces to support testing?

The analysts might begin with state management for rollback—a question of how to determine which artifacts need to be preserved—since they want to understand the scope of the state that needs to be preserved. The segment deployments strategy and command dispatcher build files are not defined in the architecture. In this example, the analyst begins recording analysis issues related to managing deployment:

- Issue 1: The build files are built in an ad hoc manner. The developers use different tools. Some groups are using cmake and Jenkins for continuous integration while others are creating their own make files and running them on the command line.
- Issue 2: The release pipeline is not defined. This is related to Issue 1. The developers should determine which artifact sources the pipeline will accept (e.g., Jenkins versus Git repositories) and what will trigger a release (e.g., continuous deployment, scheduled, pulled release).

This analysis thread is based on an observation that the teams use different tools and different release triggers. For example, if the analyst found that a staged deployment had two teams using separate release pipelines, then considerable effort could be required to create custom approvals and gates for each change or change type. Our maintainability measures for effort, calendar time, and number of human interactions would be higher every time this condition occurred.

Continuing our example, the analyst finds that the groups are permitted to use different versions of shared libraries to maintain backward compatibility. This architecture decision allows the teams to develop independently but increases complexity while incurring technical debt. The analyst records an issue:

- Issue 3: Maintaining multiple versions of libraries for backward compatibility increases the complexity of each update.

Next, in our simplified analysis example, the analyst investigates how the specialized testing interfaces are realized. The architecture documentation states that the test interfaces are realized by weaving in aspects to execute code at defined join points.[7] This practice can create challenges with determinism when the developers write aspects for the same join points. The analyst identifies two additional issues to record:

[7] A *join point* is a point in program execution that is exposed by the language definition as an event.

- Issue 4: You do not control which tests execute in what order since some join points may depend on runtime conditions that cause problems even if you set the precedence you need. This issue can occur when one test is purposefully inserting a fault while the other is testing functionality. The functional test could fail and may result in wasted effort to debug a false positive.
- Issue 5: Aspects can impact the readability of code since the reader of the code may be oblivious to advice[8] that acts on a join point.

In this simple example, the analyst rapidly identified five issues where architecture decisions impact the ability to achieve the desired response measures in Scenario 5. Some of the issues, such as Issue 3 about multiple versions of libraries, should be carefully managed to reduce future maintenance costs. Other issues—such as Issue 1, where builds are managed in an ad hoc manner—can be fixed by defining a release pipeline and a set of tools to be used across groups.

7.6 Step 6–Assess the Impact on Other Quality Attribute Requirements

Architecture decisions rarely affect just one quality attribute requirement. The tradeoffs inherent in design decisions mean that the mechanisms and decisions that the analyst found adequate to satisfy the requirement being evaluated in Step 5 may be detrimental to the satisfaction of other quality attribute requirements.

Typical tradeoffs impact software performance (throughput or latency), testability, maintainability, availability, and usability. In Step 1–Collect Artifacts, the analyst collected other quality attribute requirements that were available at this point in the development lifecycle. Now, the analyst will scan those and select the ones that might be impacted by the architecture mechanisms and decisions analyzed in Step 5–Assess Requirement Satisfaction. For example, there may be quality attribute requirements that cover concerns such as the following:

- Is there a restart timing requirement for when a development team must roll back changes? Can this restart requirement be met if there is a large amount of state that needs to be restored?
- Are there real-time latency requirements for sensor management and fusion? The ability to add code at different join points introduces nondeterminism, which may affect deadlines.
- Does the architecture provide services for common concerns such as fault handling, communication, and logging for new sensors?
- Does the architecture prescribe security mechanisms for detecting and adding new connections to other resources that serve logically as a new sensor (e.g., UAV video feeds)?

In this step, the analyst assesses how the mechanisms and decisions that make it easy to add a new sensor impact the satisfaction of scenarios related to other quality attributes and concerns. For each requirement, the analysis may be fast (e.g., securely adding a UAV was largely ignored by designers) or more involved (e.g., assessing the restart time when rolling back during significant updates). In any case, the analyst should expect to find at least a couple of additional issues.

[8] *Advice* is a method meant to run when program execution encounters selected join points.

7.7 Step 7–Assess the Costs/Benefits of the Architecture Approach

In carrying out the steps leading up to this point, the analyst should have developed a good understanding of the essential challenges of satisfying the quality attribute requirement, the approaches taken by the architect (choice of mechanisms, instantiation of the mechanisms, and how derived concerns are addressed), and the tradeoffs embodied in the approaches taken.

Any architecture approach adds new elements, interactions, or responsibilities and makes the solution more complicated. Some approaches add new *types* of elements and interactions and, in doing so, may make the solution substantially more complex. There is a level of complexity needed to solve real-world problems; this is unavoidable. The final step is to judge whether the complexity (and hence additional cost) introduced by this architecture approach is necessary and appropriate. This is a cost/benefit analysis. In some cases, the answer will be clear. In our example, if the sensors are simple and homogenous, then a façade introduces unnecessary complexity. If the sensors are diverse and fusion involves many steps, then providing a simplified, coarse-grained interface through a façade is worth the effort.

Often the cost/benefit analysis is not clear, but probing the design space and the design decisions taken with this analytical perspective in mind is still worthwhile as it will catalyze important analysis questions.

8 Summary

In this report, we defined maintainability and focused on analyzing maintenance difficulty and the cost and risk of performing a set of desired maintenance tasks.

We provided a set of sample scenarios for maintainability and, from these and other examples, inferred a general scenario. This general scenario can be used as an elicitation device and help with analysis as it delineates the response measures that stakeholders will care about when they consider this quality attribute. We also described the architectural mechanisms—tactics and patterns—for maintainability. These mechanisms are useful in both design—to give a software architect a vocabulary of design primitives from which to choose—and in analysis, so that an analyst can understand the design decisions made (or not made), their rationale, and their potential consequences.

To address the needs of analysts, we described a set of analytical tools, discussed the artifacts upon which each of these analyses depends, and identified the stage of the software development lifecycle in which each of these analyses could be employed. And we delved into three specific architecture analysis techniques for maintainability: tactics-based questionnaires, an Architecture Analysis Checklist, and coupling metrics.

Finally, we provided a playbook for applying an architecture analysis for maintainability. This playbook combines the checklists and questionnaires with information about architectural mechanisms to analyze an architecture to validate the satisfaction of a maintainability requirement.

9 Further Reading

A general discussion of quality attributes, quality attribute scenarios, tactics, and patterns that provided the foundation for much of this report can be found in the book *Software Architecture in Practice* [Bass 2012]. That book does not address maintainability specifically, but it does discuss modifiability, which is closely related. Another general discussion of quality attributes, particularly the vocabulary surrounding them, can be found in ISO/IEC/IEEE Standard 24765 [ISO/IEC 2011a].

A more in-depth discussion of the quality attribute of maintainability specifically can be found in the work of Henttonen and colleagues [Henttonen 2007]. The notion of a "maintainability index"—a single number that can characterize the level of maintainability for an entire code base—has existed for several decades. However, these indices, which are a combination of code complexity and code size measures, have not been widely adopted as they are not actionable, have not been empirically supported, and do not account for design complexity.

MacCormack, Mo, and their colleagues define and provide strong empirical evidence for architecture-level coupling metrics that account for design complexity, measuring the complexity of an entire software architecture based on its code components [MacCormack 2006, Mo 2016]. Similarly, the QMOOD metrics, described by Goyal and Joshi, and the Chidamber and Kemerer metrics attempt to measure the complexity of object-oriented designs, analyzing classes and their relationships [Goyal 2014, Chidamber 1994]. These, too, have had much more thorough empirical evaluations.

Bibliography

URLs are valid as of the publication date of this document.

[Ackoff 1968]
Ackoff, R. L. & Sasieni, M. W. *Fundamentals of Operations Research*. John Wiley & Sons. 1968.

[Arvanitou 2017]
Arvanitou, E. M.; Ampatzoglou, A.; Chatzigeorgiou, A.; Galster, M.; & Avgeriou, P. A Mapping Study on Design-Time Quality Attributes and Metrics. *Journal of Systems and Software*. Volume 127. 2017. Pages 52–77. doi: https://doi.org/10.1016/j.jss.2017.01.026

[Avizienis 2001]
Avizienis, A.; Laprie, J.-C.; & Randall, B. *Fundamental Concepts of Computer System Dependability*. Presented at IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments. May 2001.

[Bachmann 2007]
Bachmann, F.; Bass, L.; & Nord, R. *Modifiability Tactics.* CMU/SEI-2007-TR-002. Software Engineering Institute, Carnegie Mellon University. 2007. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8299

[Bass 2012]
Bass, L.; Clements, P; & Kazman, R. *Software Architecture in Practice*. Third edition. Addison-Wesley. 2012.

[Bellomo 2015]
Bellomo, S.; Gorton, I.; & Kazman, R. Insights from 15 Years of ATAM Data: Towards Agile Architecture. *IEEE Software*. Volume 32. Number 5. 2015. Pages 38–45.

[Bengtsson 1999]
Bengtsson, P. & Bosch, J. Architecture-Level Prediction of Software Maintenance. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press. 1999. Pages 139–147.

[Binder 2000]
Binder, R. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley. 2000.

[BKCASE 2018]
Body of Knowledge and Curriculum to Advance Systems Engineering. System Requirements. In *Guide to the Systems Engineering Body of Knowledge (SEBoK)*. 2018. https://www.sebokwiki.org/wiki/System_Requirements

[Bogner 2017]
Bogner, J.; Wagner, S.; & Zimmermann, A. Automatically Measuring the Maintainability of Service- and Microservice-Based Systems—a Literature Review. In *Proceedings of IWSM/Mensura '17*. ACM. 2017. Pages 107–115.

[Buschmann 2007]
Buschmann, F.; Schmidt, D. C.; Kircher, M.; et al. *Pattern-Oriented Software Architecture*. Volumes 1–5. Wiley. 1996–2007.

[Cai 2019]
Cai, Y.; Xiao, L.; Kazman, R.; Mo, R.; & Feng, Q. Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture. *IEEE Transactions on Software Engineering*. Volume 45. Number 7. July 2019. Pages 657–682.

[Cervantes 2016]
Cervantes, H. & Kazman, R. *Designing Software Architectures: A Practical Approach*. Addison-Wesley. 2016.

[Chidamber 1994]
Chidamber, S. & Kemerer, C. Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*. Volume 20. Number 6. 1994. Pages 476–493.

[Clements 2010]
Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*. 2nd edition. Addison-Wesley. 2010.

[DIB 2019]
Defense Innovation Board. *Software Is Never Done: Refactoring the Acquisition Code for Competitive Advantage*. DoD. 2019. https://media.defense.gov/2019/Apr/30/2002124828/-1/-1/0/SOFTWAREISNEVERDONE_REFACTORINGTHEACQUISITIONCODEFORCOMPETITIVEADVANTAGE_FINAL.SWAP.REPORT.PDF

[Fowler 2010]
Fowler, M. Blue Green Deployment [blog post]. *martinFowler.com*. 2010. https://martinfowler.com/bliki/BlueGreenDeployment.html

[GAO 2019]
U.S. Government Accountability Office. *Weapon System Sustainment: DoD Needs to Better Capture and Report Software Sustainment Costs*. GAO-19-173. GAO. February 2019.

[Gall 1998]
Gall, H.; Hajek, K.; & Jazayeri, M. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the 14th IEEE International Conference on Software Maintenance*. November 1998. Pages 190–197.

[Gamma 1994]
Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. 1994.

[Glass 1992]
Glass, R. *Building Quality Software*. Prentice-Hall. 1992.

[Goyal 2014]
Goyal, P. & Joshi, G. QMOOD Metric Sets to Assess Quality of Java Program. In *Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. 2014. Pages 520–533.

[Henttonen 2007]
Henttonen, K.; Matinlassi, M.; Niemelä, E.; & Kanstrén, T. Integrability and Extensibility Evaluation from Software Architectural Models—A Case Study. *The Open Software Engineering Journal*. Volume 1. 2007. Pages 1−20.

[IEEE 2006]
IEEE. *ISO/IEC/IEEE International Standard for Software Engineering – Software Life Cycle Processes – Maintenance*. Standard IEEE 14764-2006. IEEE Computer Society. 2006.

[ISO/IEC 2011a]
ISO/IEC. *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*. International Standard ISO/IEC 25010:2011(E). 2011.

[ISO/IEC 2011b]
ISO/IEC/IEEE. *Systems and Software Engineering – Architecture Description*. International Standard ISO/IEC/IEEE 42010:2011. 2011.

[ISO 2017]
ISO/IEC/IEEE. *Systems and Software Engineering – Vocabulary*. 2nd edition. ISO/IEC/IEEE 24765. 2017.

[Jacobs 2018]
Jacobs, W.; Wigginton, S.; & Padilla, M. *Comprehensive Architecture Strategy (CAS)*. Defense Technical Information Center. 2018. https://apps.dtic.mil/sti/citations/AD1103295

[Jamshidi 2018]
Jamshidi, P.; Pahl, C.; Mendonça, N. C.; Lewis, J.; & Tilkov, S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*. Volume 36. Number 3. May/June 2018. Pages 24–35.

[Kazman 1994]
Kazman, R.; Abowd, G.; Bass, L.; & Webb, M. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*. Sorrento, Italy. ACM. 1994. Pages 81–90.

[Kazman 2002]
Kazman, R. & Bass, L. Making Architecture Reviews Work in the Real World. *IEEE Software*. Volume 19. Number 1. Jan./Feb. 2002. Pages 67–73.

[Kazman 2020]
Kazman, R.; Bianco, P.; Ivers, J.; & Klein, J. *Integrability.* CMU/SEI-2020-TR-001. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=637375

[Klein 2015]
Klein, J. & Gorton, I. Design Assistant for NoSQL Technology Selection. In *Proceedings of the First International Workshop on Future of Software Architecture Design Assistants*. ACM. 2015. Pages 7–12.

[Krasner 1988]
Krasner, G. & Pope, S. A Cookbook for Using the Model–View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Technology*. Volume 1. Number 3. Aug./Sep. 1988. Pages 26–49.

[Lenhard 2013]
Lenhard, J.; Harrer, S.; & Wirtz, G. Measuring the Installability of Service Orchestrations Using the SQuaRE Method. In *Proceedings of the Sixth International Conference on Service-Oriented Computing and Applications*. IEEE. 2013. Pages 118–125.

[Lewis 2014]
Lewis, J. & Fowler, M. Microservices [blog post]. *martinFowler.com*. 2014. https://martinfowler.com/articles/microservices.html

[MacCormack 2006]
MacCormack, A.; Rusnak, J.; & Baldwin, C. Y. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*. Volume 52. Issue 7. July 2006. Pages 1015–1030.

[Mo 2016]
Mo, R.; Cai, Y.; Kazman, R.; Xiao, L.; & Feng, Q. Decoupling Level: A New Metric for Architectural Maintenance Complexity. In *Proceedings of the International Conference on Software Engineering*. Austin, TX. May 2016. Pages 499−510.

[Nygard 2017]
Nygard, Michael. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Programmers. 2017.

[ODASD 2017]
Office of the Deputy Assistant Secretary of Defense. *Initiatives: Modular Open Systems Approach*. 2017. https://www.dsp.dla.mil/Programs/MOSA/

[Oman 1992]
Oman, P. & Hagemeister, J. Metrics for Assessing a Software System's Maintainability. In *Proceedings of the Conference on Software Maintenance*. IEEE. 1992. Pages 337–344.

[Sato 2014]
Sato, D. Canary Deployment [blog post]. *martinFowler.com*. 2014. https://martinfowler.com/bliki/CanaryRelease.html

[Seref 2016]
Seref, B. & Tanriover, O. Software Code Maintainability: A Literature Review. *International Journal of Software Engineering & Applications*. Volume 7. Number 3. May 2016. Pages 69–87.

[Suppe 1998]
Suppe, F. Operationalism. In *Routledge Encyclopedia of Philosophy*. Edited by T. Crane. Taylor & Francis. 1998. https://www.rep.routledge.com/articles/thematic/operationalism/v-1

[USDAS 2018]
Office of the Under Secretary of Defense for Acquisition and Sustainment. *Depot Maintenance Core Capabilities Determination Process*. DoD Instruction 4151.20. 2018. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/415120p.pdf

[Weinreich 2016]
Weinreich, R. & Groher, I. Software Architecture Knowledge Management Approaches and Their Support for Knowledge Management Activities: A Systematic Literature Review. *Information and Software Technology*. Volume 80. December 2016. Pages 265–286.

[Wilmot 2016]
Wilmot, J.; Fesq, L.; & Dvorak, D. Quality Attributes for Mission Flight Software: A Reference for Architects. In *37th IEEE Aerospace Conference*. Big Sky, Montana. March 2016. https://ieeexplore.ieee.org/document/7500850

[Wong 2011]
Wong, S.; Cai, Y.; Kim, M.; & Dalton, M. Detecting Software Modularity Violations. In *Proceedings of the International Conference on Software Engineering*. Honolulu, HI. May 2011. Pages 411–420.

[Xiao 2014]
Xiao, L.; Cai, Y.; & Kazman, R. Design Rule Spaces: A New Form of Architecture Insight. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM. June 2014. Pages 967–977.

[Yourdon 1979]
Yourdon, E. & Constantine, L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. 1979.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** December 2020	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Maintainability		5. **FUNDING NUMBERS** FA8702-15-D-0002
6. **AUTHOR(S)** Rick Kazman, Phil Bianco, James Ivers, and John Klein		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2020-TR-006
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** SEI Administrative Agent, AFLCMC/AZS, 5 Eglin Street, Hanscom AFB, MA 01731-2100		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** n/a
11. **SUPPLEMENTARY NOTES**		
12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

This report summarizes how to systematically analyze a software architecture with respect to a quality attribute requirement for maintainability. The report introduces maintainability and common forms of maintainability requirements for software architectures. It provides a set of definitions, core concepts, and a framework for reasoning about maintainability and the satisfaction (or not) of maintainability requirements by an architecture and, eventually, a system. It describes a set of mechanisms, such as pat-terns and tactics, that are commonly used to satisfy maintainability requirements. It also provides a method by which an analyst can determine whether an architecture documentation package provides enough information to support analysis and, if so, determine whether the architectural decisions contain serious risks relative to maintainability requirements. An analyst can use this method to determine whether those requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis. The reasoning around this quality attribute should allow an analyst, armed with appropriate architectural documentation, to assess the risks inherent in today's architectural decisions in light of tomorrow's anticipated needs.

14. **SUBJECT TERMS** architecture analysis, maintainability, quality attributes, quality attribute requirements, software architecture	15. **NUMBER OF PAGES** 74
16. **PRICE CODE**	

17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102