Carnegie Mellon University
Software Engineering Institute

Extensibility

Rick Kazman
Sebastián Echeverría
James Ivers

April 2022

TECHNICAL REPORT
CMU/SEI-2022-TR-002
DOI: 10.1184/R1/18863639

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 of extensibility. The report introduces extensibility and common forms of extensibility requirements for software architectures. It provides a set of definitions, core concepts, and a framework for reasoning about extensibility and satisfaction (or not) of extensibility 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 extensibility 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, to determine whether the architectural decisions made contain serious risks relative to extensibility 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 extensibility and common forms of extensibility requirements
- a description of a set of mechanisms, such as patterns and tactics, that are commonly used to satisfy extensibility requirements
- a means for an analyst to determine whether an architecture documentation package provides enough information to support analysis and, if so, to determine whether the architectural decisions made contain serious risks relative to extensibility requirements
- a means for an analyst to determine whether those extensibility 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 2020a, 2020b]. The purpose of this document, as with all 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 *extensibility*. 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 may 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 of 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, architectural decisions have implications for each of the other views.

In addition, other important decisions within a project will impact extensibility—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's being well executed.

As we will show, there is no single way to analyze for extensibility. One can (and should) analyze for extensibility 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, the analysis will be less detailed, and there will be less confidence in the results than if there were an existing implementation that 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.

2 On Extensibility

Extensibility is informally thought of as the ability of a software-intensive system economically and predictably to add functions, capabilities, and support for key quality attributes over a long period of time. By designing a system to be extensible, we acknowledge that the system will grow and change—along anticipated dimensions—and we consciously plan for this in the architecture. For this reason, it is an important quality attribute to design into a system from the start, assuming that the system is expected to have a long lifetime and to evolve over that lifetime. This is why we are interested in understanding extensibility and how it is supported by appropriate architectural decisions.

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

2.1 Extensibility–Definitions

Several definitions of extensibility can be found in the popular technical press and in academic writings. For example, this definition is from Techopedia:

> *Extensibility is a measurement of a piece of technology's capacity to append additional elements and features to its existing structure. A software program, for example, is considered extensible when its operations may be augmented with add-ons and plugins. Extensible programming languages have the ability to define new features and introduce new functionality within them. [Techopedia 2021]*

And this similar definition comes from Breivold and colleagues:

> *The capability of the software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to existing system [Breivold 2007]*

In much the same vein, a paper on the NASA SOFIA mission controls and communications system defines extensibility as follows:

> *Extensibility is the ability for the software to grow in performance and functionality, and to quickly and efficiently add new features, incrementally as they are needed (or funded), without the need to re-design the underlying infrastructure. [Papke 2000]*

Papke and colleagues then elaborate that modifiability and expandability are also needed for extensibility. They definite modifiability as the ability to add functionality and expandability as the ability to add new types of scientific instruments.

Our use of the term *extensibility* closely follows this last definition. The term extensibility, as used in this report, refers to the ability for (possibly independent) development teams to extend a system's functionality or its quality characteristics in known, planned ways, by exercising extension points that have been built into its architecture. We take the perspective of the owner of a system who wants to provide such extension mechanisms and the analyst who wants to ensure that extensibility characteristics designed into the architecture are adequate for achieving the desired outcomes with reasonable cost, schedule, and personnel implications and without compromising other important system qualities.

This final point is key: The above definitions characterize extensibility as the ability to *easily* add new kinds of functions, features, or capabilities. And they emphasize that this addition should be achieved by a design strategy in which the core of the existing system does not need to be modified to incorporate these extensions, but rather some design features—extension mechanisms or extension points—provide this as an organic capability that dramatically lessens the time, risk, and effort of extending the system. Some well-known examples are browser extensions and plug-ins in Chrome, Firefox, and Edge; music processing plug-ins in audio editing tools; and integrated development environment (IDE) plug-ins that understand the syntax of specific programming languages.

2.2 Extensibility as a Quality Attribute

The goals of extensibility overlap significantly with those of the quality attributes of integrability [Kazman 2020a], maintainability [Kazman 2020b], and modifiability [Bass 2021]. Hence it should be no surprise that the reasoning surrounding them, the mechanisms that an architect can use to achieve them, and the analytic approaches that an analyst can use to assess them all overlap substantially.

An architect has many "concerns" when designing for extensibility. Concerns are requirements that an architect must consider, although they will seldom if ever appear in a formal requirements specification [Cervantes 2016]. An architect needs to first understand where extension capabilities may be needed in a system. Then the architect must consider the following aspects of extensibility and provide architectural support for them:

- extension strategy: How are extensions packaged and integrated?
- testing strategy: How are extensions tested, both individually and in combinations?
- lifecycle management: When are extensions included in a system, and when are they activated? For example, can new extensions be added or removed at compile time, at deployment time, or at runtime? When do extensions get bound, invoked, checked, and so on.
- deployment strategy: How are extensions checked, approved, signed, and potentially impounded? These activities might occur outside of the system, via a separate governance process, but they must be coordinated with the system.
- dependency management: Is there support for extension orchestration, including ordering, composition, compatibility checking, and management of shared resources?

3 Evaluating the Extensibility of an Architecture

We cannot precisely evaluate the extensibility 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 extensibility of an architecture with respect to a set of anticipated extensions and extension 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 define a template for extensibility scenarios and provide examples of the kinds of extension tasks that a system might be subjected to. We need to understand the things that are involved in conducting extension activities to understand what it means to measure the extensibility of a system. To this end, in Section 5, we survey the techniques that the software engineering research literature has proposed for achieving extensibility. Finally, we use these scenarios in our architecture analysis playbook (in Section 7).

While we restrict our attention in this report to analyzing *architectural* information for extensibility, it is important to note that extensibility analyses have historically 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, maintained, and evolved a system over time. At that point, it can be very expensive and time consuming to repair such problems. Thus, our objective in analyzing an *architecture* for its extensibility characteristics is to find a sweet spot wherein we can gain insight into the potential extensibility characteristics of a system before much, if any, code has been developed.

3.1 Measuring Extensibility

Before discussing extensibility in detail, we need to first define several important terms:

- the *core* functionality of the system: This is the artifact that will be extended.
- *extension point*: This is an explicitly identified point in the system that supports addition of *extensions*. An extension point may employ multiple *extension mechanisms*. An extension may use multiple extension points. Each extension point should have a coherent purpose and scope.
- *extension mechanisms*: These are realizations of one or more tactics or patterns that enable an extension.
- the *extensions* to the system. These are new capabilities that build upon the core and employ the provided extension points.
- the *extended system*: This is the composition of the core system and one or more extensions. These may or may not be deployed together.

When we refer to and when we analyze specific extension tasks (defined as scenarios), we restrict ourselves to considering how well a system can be extended along predefined dimensions—that is, adding functions or capabilities in a rigorous, planned way. Given this context, performing an extension task on a large, complex software-intensive system typically involves taking advantage of *existing* extension points. An overarching goal of a system designed for extensibility is that the system can be extended without modifying the core system. Only in rare cases will new extension points or mechanisms be created and employed.

The creators and users of extension points are typically different groups of stakeholders (core architects and extension architects) and may even be in different organizations. The core architects are responsible for the overall architecture of the system, including designing, implementing, and testing one or more extension points. Typically, an architecture provides only a small number of such extension points. These extension points support vastly greater numbers of actual extensions.

These extension points should provide enough hooks into the architecture to extend the system in useful ways, but extensibility must be counter-balanced against compromising other qualities of the architecture, such as performance and security. Core architects also need to provide the appropriate abstractions in the extension points that they make available to ease the job of extension creators.

Given these extension points, extension creators can now use these points to extend the system. This process involves

- designing and implementing the extension, which requires understanding the extension points and their capabilities. For example, to develop extensions for the Chrome browser, the architect would need to understand the development process and, in particular, the extension API.[1] Furthermore, if some preexisting functionality is to be packaged as an extension and the extension point is not quite compatible with this functionality, the architect would need to do some additional work to ensure smooth integration [Kazman 2020a].
- determining whether any preexisting elements of the architecture—core or extensions—are affected by the insertion of extensions. If so, those other software elements and any supporting artifacts, such as tests or build scripts, must also be changed.
- debugging and testing the extension to validate that the extension and any secondary changes are correct, potentially including recertification work as needed.
- deploying[2] the new extensions and rolling back versions if defects are discovered in these newly deployed extensions.

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

- The *management of dependencies* is about ensuring that the architecture has been designed with a properly constrained set of extension points in mind. Well-designed extension points and mechanisms should ensure that
  - extensions have sufficient access to system resources and capabilities through extension points to implement their intended purpose
  - extensions can be created without requiring changes to the core system
  - extensions have localized impact and do not adversely affect the quality or behavior of the core system or other extensions

[1] https://developer.chrome.com/docs/extensions/reference/

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

    - extension points provide mechanisms at the right level of abstraction, simplifying extension creation, testing, and deployment
- 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 any extensions or the impact of new extensions on the rest of the system.
- The *management of deployments* is about ensuring that the architecture includes support for efficiently deploying a system and its extensions, potentially making provisions for detecting, labeling, and even impounding noncompliant extensions.

Explicit extension mechanisms built into an architecture should aid in achieving desired levels of the chosen characteristics. Consequently, when evaluating extensibility, 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 the desired characteristics.

Different scenarios will, of course, emphasize these concerns to different extents. For example, one scenario may simply explore how much work it is to add a new extension to the system, such as adding and deploying a new app on a phone or a new plug-in on a browser. Another scenario may probe the dependencies among extensions: they should be managed so that, for example, if the phone app requires a map service, it is sufficiently abstracted from the details of that service such that if the service upgrades or is replaced entirely, the phone app would be minimally affected. A third scenario might probe the extent to which a new extension could (or could not) affect the state of the entire system or of another extension. For example, could a phone book app change how calls are received and made on a phone?

We discuss the three categories of system characteristics in more detail in the following subsections.

Sidebar: Extensibility-Driven Design

There is a spectrum of kinds of systems that are extensible. Where a given system lives on this spectrum is a key design decision affecting important architectural characteristics. At one extreme of this spectrum, you can have "rich" systems that provide business or mission value without any extensions. Most systems created today fall into this category—they have no built-in extensibility, and all changes must be made by modifying the core.

At the other extreme, you can have systems that provide little or no business or mission value without the addition of extensions (think of platforms here). At this latter extreme, a system could provide just an extensible core capability, and all the functionality that users care about is achieved by extension points. Middleware platforms, such as node.js, and the Eclipse IDE are near this end of the spectrum.

In between are systems that provide an important and independently useful set of functions within the core, but which are almost always extended in a vast number of ways. Modern browsers fall into this portion of the spectrum.

Given that this is a spectrum, and that there is no "best" point on this spectrum, a key aspect of design and analysis should be to ensure that the level of extensibility provided by the architecture is properly aligned with the stakeholders' goals for the system. And these goals can be efficiently expressed with quality attribute scenarios that focus on anticipated extensions to the system and their desired characteristics.

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 extensions from the rest of the architecture, and how independent are they from each other?[3] This gives us insight into the probability that a new extension or a change to an existing extension will ripple to other extensions or to the core architecture. 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 2020a, 2020b].
- *highly cohesive*: Do extensions and extension points 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 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]. Typically, extension points and mechanisms are aimed at a small, cohesive scope of responsibilities.
- *understandable*: How easy is it to determine where a particular piece of functionality resides within an extension? This gives us insight into how completely the responsibilities within extension points, and extensions built upon them, are documented and understood, which, in turn, affects the average cost of a change. If responsibilities are well understood (because they are clear and well documented), then modifications will, on average, be more systematic and hence less costly [Glass 1992]. Here we seek to understand which extension points allow access to which functionality, information, or resources.

3.1.2 Management of System State

To assess the degree to which the architecture's extension points support the management of system state, we are primarily concerned with assessing how these extension points influence the amount of work it takes to set up and run test cases. We want to understand the degree to which the system, including its extensions, is "state controllable" and "state observable" [Binder 2000]. To comprehend the management of system state more fully, analysts may ask the following questions:

- *state controllable*: How well do the extension points and mechanisms support the control of system state? If the architecture allows the state of the system to be precisely controlled, then

[3] When considering quality attributes such as maintainability [Kazman 2020b] and integrability [Kazman 2020a], coupling can and should be assessed broadly. System-wide measures of coupling are appropriate for these qualities. For extensibility, however, system-wide measures of coupling are not appropriate. The focus must be narrower: on coupling among extensions and coupling between extensions and the core.

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, reducing the time required to test new or changed extensions.

- *state observable*: How much do the extension points and mechanisms 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 an algorithm that requires random values for some parameters or return different results based on time or its instantaneous load. In other cases, architecture decisions can introduce nondeterminism, for example, load-balancing strategies. 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 or allowing the time or load to be injected and hence precisely controlled.

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:[4]

- *granular:* Can updates to parts of the system be deployed separately? In particular, can updates to the core and its extensions be done separately (perhaps by different organizations)? 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. In an avionics system, *granular* could mean that we can update the cockpit display unit software now and update the mission computer software later. In a browser, we would like to be able to deploy new extensions or new versions of existing extensions with minimal effort and with no changes required to the core system. And we would like to be able to deploy a new version of the core system with little or no impact on the existing extensions. If deployments are all or nothing (in these examples, all instances of a service, all avionics software components, or all parts of a browser), 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.

[4] These characteristics are orthogonal to managerial decisions about deployments, such as who controls deployment, can base and extensions be deployed at different times by different parties, and does deployment of extensions depend on approvals?

- *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]? For example, does the extension mechanism provide a way to determine the compatibility of extensions with the extension points and quarantine or warn about non-compliant or untrusted extensions? An example is the Chrome browser, which will warn users who are trying to install untrusted extensions. Another example is operating systems that require drivers to be signed by known third parties before allowing them to be installed.
- *efficient:* How quickly can new extensions be deployed (and, if needed, rolled back) and with what level of effort? Can a new deployment be rolled out to many sites simultaneously? To do this efficiently, some automation of deployment and rollback is typically desired.

3.2 Reasoning About Extensibility Characteristics

For a large, complex software-intensive system to be extensible, its architecture will be designed to support anticipated extensions through the provision of extension points. When evaluating an architecture with respect to any quality attribute—such as extensibility—we need to assess the design decisions in an architecture that (purport to) lead to extensibility. That is, we need to understand how well the architecture has been designed to support the kinds of extensions that are anticipated.

Given the characteristics described in Section 3.1, we now consider how to operationalize this knowledge. Specifically, a designer planning an extension strategy, or an analyst examining the extension mechanisms already built into an architecture, should consider the following aspects of the design:

- The appropriateness of the extension points: Do they cover the majority of ways in which the system is expected to be extended and appropriately constrain the capabilities of extensions?
- The degree of encapsulation of extensions and the degree of coupling between extensions and the rest of the system:
  - How easy is it to plug or unplug extensions with predictable side effects?
  - To what degree do the extension points allow for extensions to be created and inserted with low effort? Is adaptation required so that the interface assumptions of the extensions match those of the extension mechanisms [Kazman 1997]?
  - Can extensions affect the state of the core system or of other extensions?
  - How can extensions be composed, and how can these compositions be reasoned about and analyzed?
- Cohesion of the extension mechanisms: Do the extension points and mechanisms require changes to be made to multiple points in the architecture?
- The lifecycle of an extension—including how it is deployed, invoked, tested, controlled, and terminated—and the implications of these lifecycle choices on qualities such as performance, testability, robustness, and security. Is it possible to monitor and control the resources that an extension uses? Can an extension be inserted (or replaced) at runtime or does this require a restart or rebuild of the system?

Unfortunately, there are few existing metrics that can be confidently brought to bear on these questions. While there are well-known coupling and cohesion metrics, their application to extensions and extension mechanisms is not straightforward. These metrics are defined at the level of a method or class [Chidamber 1994] or an entire system [Mo 2016], so it is unclear to how target them at just the subset of the architecture relevant to extensibility. As mentioned in the introduction to this section, extensibility characteristics can be measured more precisely when richer sets of information—derived from a project's code and its history—are available.

Sidebar: Leading and Trailing Indicators

When analyzing a complex system, an analyst may need to choose among a variety of analysis techniques. These techniques will vary along a number of dimensions: how easy or hard they are to apply, how much training is required to use them, where to apply them, and the kinds of data they require. Analysis techniques for architectures include metrics, simulations, prototypes, analytic models, checklists, questionnaires, scenario-based analysis, and instrumentation. Let us consider two key aspects of these techniques: where they may be applied and the kinds of data they require.

Metrics and measures based on data collected from a system as it is running or as it evolves can be highly accurate and informative. But this data can only be collected after something has been implemented, and thus metrics and measures are categorized as trailing indicators. Techniques such as scenario-based analysis can be applied on a system as it is being designed and used to obtain leading indicators.

The decision to use leading or trailing indicators exists in any domain where the artifact being evaluated is complex. For example, in healthcare there are many reliable leading indicators of a person's health outcomes such as physical activity, weight, tobacco use, substance abuse, and so forth. Mortality rate, on the other hand, is a trailing (sometimes called "lagging") measure. In a complex software product, trailing indicators are often used to measure qualities, such as maintainability, by analyzing a project's code base, issue tracker, and commit history. These trailing measures are likely to be highly accurate but can only be collected once the system has been built and used. Architectural analysis can offer insights—albeit likely less precise ones—at a fraction of the cost and risk.

All other things being equal, we prefer to use techniques that rely on leading indicators than trailing indicators. Leading indicators, if they are reliable, can dramatically reduce cost and risk. Architecture analysis, relying on leading indicators, gives you the opportunity to analyze before you build. There is less precision in this analysis, but it is still valuable because you can do this analysis cheaply and early in a project's life, thus reducing risk. Metrics, on the other hand, rely on trailing indicators. The system, or a major part of the system, must already be built before you can measure it.

Thus, a designer or an analyst considering a brand-new system must appeal to other analysis techniques—such as checklists, questionnaires, scenarios, and prototyping—to determine the appropriateness of a set of mechanisms. We will provide some of these in Section 6.

4 Extensibility 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 2021]. A quality attribute scenario is simply a brief description of how a system is required to respond to some stimulus. Quality attribute scenarios, different from use cases, 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 of a quality of a system. The use of scenarios to specify quality attribute requirements for software dates back at least to 1994 [Kazman 1994]. 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 [Bass 2021]. More recently we have seen characterizations of the qualities of scalability and consistency [Klein 2015], integrability [Kazman 2020a], and maintainability [Kazman 2020b].

A quality attribute scenario has six parts [Bass 2021]. 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, 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 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.

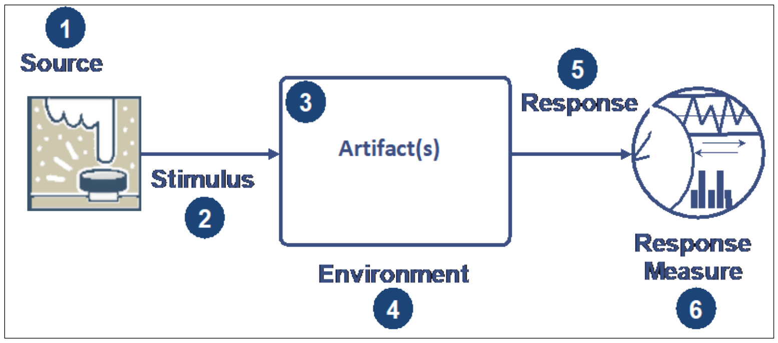

Figure 1: The Form of a General Scenario

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

4.1 General Scenario for Extensibility

There is no single scenario that specifies all the possible measurements that could characterize a quality attribute like extensibility. But we do see 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 the extensibility 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 extensibility:

Scenario Part	Possible Type for Each Value
Source	One of the following: • core developer • external developer
Stimulus	One of the following: • adds new extension • creates new version of existing extension
Artifact	One of the following: • specific set of extension mechanisms • extension point
Environment	One of the following: • development/maintenance time • integration time • deployment time • runtime

Scenario Part	Possible Type for Each Value
Response	One or more of the following: • extensions are {completed, integrated, tested} • new extensions are successful in correctly (syntactically and semantically) interacting with the core system • extensions in the new configuration do not violate any resource limits • extensions in the new configuration do not impact the correct behavior of other extensions
Response Measure	Cost, in terms of one or more of the following: • # components changed • % code • # lines of code changed • effort • money • calendar time • effects on quality attribute response measures • # new defects introduced

4.2 Example Scenarios for Extensibility

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, you would choose values that are as precise as possible, in the context of your system.

4.2.1 Scenario 1: New Track Information

This example scenario describes how an external developer can use the provided extension points to add a new type of moving track information to the system.

Scenario Part	Value
Source	External party
Stimulus	Add new type of moving track information to the system
Artifact	Fusion algorithm extension point
Environment	Development time
Response	Increase resolution of fused tracks Code-based modification made using existing extension points
Response Measure	No part of the core system needs to be recompiled Development can be done by an external party in less than 3 person-weeks

4.2.2 Scenario 2: New Phase Discriminator

This scenario describes how to extend the system by adding a new phase discriminator from an external party.

Scenario Part	Value
Source	External party
Stimulus	Add a phase discriminator to identify rotational motion
Artifact	Fusion algorithm extension point
Environment	Development time
Response	Increase resolution of fused tracks Code-based modification made
Response Measure	No part of the core system needs to be recompiled Development can be done in 1 calendar month by an external party

4.2.3 Scenario 3: New Input Validation Filter

This scenario describes how to create an extension to add a new input validation filter into the system.

Scenario Part	Value
Source	External party
Stimulus	Add an input validation filter extension to clean up input and avoid injection attacks
Artifact	User interface component extension point
Environment	Development by external party
Response	Filter can be added to the system using extension points and works as expected
Response Measure	No part of the core system needs to be recompiled Development can be done in 2 person-weeks by an external party

4.2.4 Scenario 4: New Location View

This scenario shows how an external party can extend the system by adding a new location view type using the provided extension points.

Scenario Part	Value
Source	External party
Stimulus	Create a new UI view extension to show location data on a map instead of as a list
Artifact	User interface component extension point
Environment	Development time
Response	New location view can be added to the system and displays the same location data
Response Measure	No part of the core system needs to be recompiled New view can be easily accessed, similarly to existing view Development can be done in 4 person-weeks by an external party

4.2.5 Scenario 5: New Format for Importing Data

This scenario shows how an external party can extend the system by adding support for importing data in a new format.

Scenario Part	Value
Source	External party
Stimulus	Import health data from a World Health Organization provider
Artifact	Data management component extension point
Environment	Development time
Response	Data in specific JSON format can be imported into the system and converted into internal structure for storage using existing extensions points
Response Measure	No part of the core system needs to be recompiled Development can be done using extension points in 2 person-weeks by an external party

5 Mechanisms for Achieving Extensibility

We have thus far focused most of our attention on analyzing an architecture for extensibility. But analysis and design are two sides of the same coin. Now we turn our attention to the architectural design task of *achieving* extensibility.

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. Here we generically refer to these design concepts as "mechanisms." These mechanisms are the architect's main tools to achieve a desired set of extensibility characteristics.

In practice, the terminology used for technical mechanisms is informal, and often the term 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 2021].
- Architectural Tactics. *Tactics* are smaller building blocks of design than architectural patterns, focused on a single element or interaction, in contrast to a pattern that defines a collection of elements [Bass 2021].

5.1 Tactics

Since tactics are simpler and more fundamental than patterns, we begin our discussion of mechanisms for extensibility with them. Tactics are the building blocks of design, the raw materials from which patterns, frameworks, and styles are constructed. Each set of tactics is grouped according to the quality attribute goal that it addresses. The goals for the extensibility tactics found in Figure 2 are to enable a system, in the face of a request to extend the system, to manage dependencies, adapt, and manage deployments so that the extension to the system can be made efficiently.

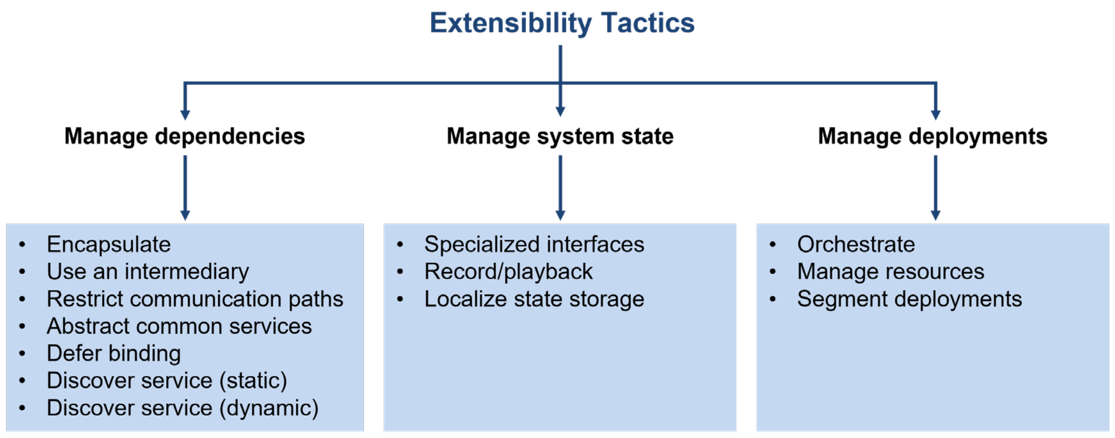

Figure 2: Extensibility Tactics

These tactics are known to influence the responses (and hence the costs) in the general scenario for extensibility (e.g., number of components affected, effort, calendar time, new defects introduced). The tactic descriptions presented below are derived from *Software Architecture in Practice* [Bass 2021] and "Toward Design Decisions to Enable Deployability" [Bellomo 2014]. Table 1 summarizes the tactics presented in this section and how each relates to the characteristics and measures presented in Section 3.1.

Table 1: Extensibility Tactics and Their Relationships to Extensibility Characteristics

Tactic	Extensibility Characteristic							
	Loosely Coupled	Highly Cohesive	Understand-able	State Controllable	State Observable	Granular Deployability	Controllable Deployability	Efficient Deployability
Encapsulate	+		+					
Use an intermediary	+							
Restrict communication paths	+							
Abstract common services		+	+					
Defer binding	+							
Discover service (static)	+		+					
Discover service (dynamic)	+		+					
Specialized interfaces				+	+			
Record/playback				+	+			
Localize state storage					+			
Orchestrate	+					+	+	
Manage resources	+						+	
Segment deployments							+	+

Note: A plus sign indicates that the tactic positively addresses an extensibility characteristic 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 tactic has no consistent effect on the measure.

Now we discuss each of the tactics presented in Figure 2 in more detail below. For each tactic that we discuss, we not only describe the tactic but also relate it to the characteristics described in Section 3.1 as a way of describing the intent and impact of the tactic.

5.1.1 Managing Dependencies

Encapsulate

Encapsulation is the foundation upon which all other extensibility tactics are built. It is, therefore, seldom seen on its own, but its use is implicit in the other tactics described here. Encapsulation, if done properly, will reduce coupling between modules and increase the cohesion within modules.

Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities. Encapsulation is also arguably the most common modifiability tactic because it reduces the probability that a change to one module propagates to other modules. Couplings that might have depended on the internals of the modules now go to the interface for the module. These strengths are, however, reduced because the interface limits the ways in which external responsibilities can interact with the module (perhaps through a wrapper). The external responsibilities can now only directly interact with the module through the exposed interface (indirect interactions, however, such as dependence on quality of service, will likely remain unchanged). 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.

Encapsulation may, for example, hide interfaces that are not relevant for extensibility and expose only those that will act as extension points. An example is a set of extension points that can be used to extend the filtering capabilities of a system, without giving access to any other part of the system, and where changes behind those extension points would not be noticeable to extensions.

Use an Intermediary

Intermediaries are used for breaking dependencies among a set of components. Intermediaries can be used to resolve different types of dependencies. For example, intermediaries like a publish-subscribe bus, shared data repository, or dynamic service discovery all reduce dependencies between 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.

Determining the specific benefits of a particular intermediary requires knowing what the intermediary actually does. An intermediary can be useful to allow future additions to a system without having to modify the core system. It can also help analysts or developers more easily discover new extension modules added to a system.

Restrict Communication Paths

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 is seen in service-oriented architectures (SOAs), 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.

This tactic can be very useful when defining extension points. By restricting interaction to only certain defined modules, the amount of knowledge needed to create an extension is decreased, and the possible ways to interact with the main system can also be tightly controlled.

Abstract Common Services

Where two modules provide services that are similar but not quite the same, it may be useful to hide both specific modules behind a common abstraction for a more general service. The more general service is more cohesive than either of the original services. This form of abstraction also serves to make an architecture more understandable. The abstraction might be realized as a common interface implemented by both or may involve an intermediary that translates requests for the abstract service to more specific requests for the modules hidden behind the abstraction. This is a form of encapsulation that hides the details of the modules from other components in the system. In terms of extensibility, this means that future extensions must deal with only a single abstraction (realized as an extension point) to access existing services, rather than needing knowledge of different modules and adding dependencies to them.

Abstracting common services allows for consistency when handling common infrastructure concerns (e.g., translations, security mechanisms, and logging) via the extension point. When these concerns change, or when new versions of the components implementing these concerns change, the changes can be made in a smaller number of places.

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 depend only 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.

Deferring binding is one of the most used tactics to achieve extensibility. Allowing binding to occur later lets developers easily add more functionality through defined methods without the need to recompile the core system. Deferred binding also allows developers to easily add different and customized extensions to a system without previous knowledge of those extensions.

Discover Service (Static)

This tactic allows one service to locate another by querying a directory service.[5] The service being searched for can be described by type of service, name, location, quality of service (e.g., timing performance or availability), or some other attribute.

This tactic is often implemented using a database or a wiki that describes the service, provides its location, and describes the service's API. An advantage of static service discovery is the relatively

5 We use the word "service" generically here to indicate some packaging of functionality that is discoverable at runtime. This does not imply a service-based architecture.

low cost to set one up in environments that do not change often. This approach does not work well in environments where the locations of services need to change often (e.g., containers or virtual machines where Internet Protocols are assigned dynamically). Dynamic environments require more up-front investment and generally are implemented by commercial off-the-shelf products that support the indirection needed for load balancing and other runtime needs.

Having a static way to discover a service allows a core system to easily find new functionality implemented as an extension. This can help make an architecture more understandable and more loosely coupled. Including a mechanism for static service discovery in an architecture can also be a governance mechanism that can assist extensibility by setting an expectation of minimal information that will be made available for all new extensions. The specific set of information that is required for the discovery mechanism varies, with inclusion of syntactic descriptions being common and inclusion of data semantics not being common.

Discover Service (Dynamic)

Dynamic discovery enables the discovery of service providers at runtime, allowing the binding between a service consumer and a concrete service to occur at runtime. This is similar to the discover service (static) tactic, but it defers the discovery and binding. And as with the static version of this tactic, it aids in making an architecture more loosely coupled and more understandable.

From an extensibility perspective, inclusion of a dynamic discovery capability sets the expectation that the core system will clearly advertise services available for integration with future components and the minimal information that will be available for each service. The specific information available will vary, but as a runtime mechanism, it is typically oriented to data that can be mechanically searched during discovery and runtime integration (e.g., identifying a specific version of an interface standard by string match).

5.1.2 Managing State

Specialized Interfaces

Having specialized testing interfaces allows you to control or capture variable values for a component either through a test harness or through normal execution. Examples of specialized test routines, some of which might otherwise not be available except for testing purposes, include these:

- a *set* and *get* method for important variables, modes, or attributes
- a *report* method that returns the full state of the object
- a *reset* method to set the internal state (for example, all the attributes of a class) to a specified internal state
- a method to turn on verbose output, various levels of event logging, performance instrumentation, or resource monitoring

Specialized testing interfaces and methods should be clearly identified or kept separate from the access methods and interfaces for required functionality so that they can be removed if needed. (However, in performance-critical and some safety-critical systems, it is problematic to field different code than that which was tested. If you remove the test code, how will you know the code you field has the same behavior, particularly the same timing behavior, as the code you tested? For other kinds of systems, however, this strategy is effective.)

Record/Playback

The state that caused a fault is often difficult to recreate. Recording the state when it crosses interfaces allows that state to be used to “play the system back” and to recreate the fault. Record refers to capturing information crossing interfaces, and playback refers to using the recorded state as input for further testing.

Localize State Storage

To start a system, subsystem, or component 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, this becomes difficult if not impossible. The state can be fine-grained—individual variables—or coarse-grained to represent broad abstractions or overall operational modes. The choice of granularity depends on how the states will be used in testing. A convenient way to “externalize” state storage (that is, to make it able to be manipulated through interface features) is to use a state machine (or state machine object) as the mechanism to track and report current state.

5.1.3 Managing Deployments

Orchestrate

Orchestrate is a tactic that uses a control mechanism to coordinate and manage the invocation of particular services so that they can be unaware of each other. Orchestration mechanisms encourage loose coupling and greater granularity of components (such as extensions) in an architecture. The creators are incentivized to make smaller, simpler extensions with fewer assumptions about how they will be used if there exists an orchestration mechanism in the architecture.

Workflow engines are a common implementation of the orchestrate tactic. Workflow management is fundamentally about the organization of work or activities. A workflow is a set of organized activities that coordinate software components to complete a business process. A workflow may consist of other workflows (each of which may themselves consist of aggregated services). The workflow model encourages reuse and agility, leading to more flexible business processes. Business processes can be managed under a philosophy of business process management (BPM) that views processes as a set of competitive assets to be managed. The processes that businesses seek to manage are typically composed within and executed by a SOA infrastructure. Complex orchestration can be specified in a language such as BPEL (Business Process Execution Language).

Orchestration helps with the integration of a set of loosely coupled reusable services to create a system that meets a new need. This tactic allows future extensions to focus on integration with the orchestration mechanism instead of point-to-point connections with multiple components.

Manage Resources

A resource manager is a specific form of intermediary that governs access to computing resources. It is similar to the restrict communication paths tactic. With this tactic, software components are not allowed to directly access some computing resources (e.g., threads or blocks of memory). Instead, they request those resources from a resource manager. Resource managers are typically responsible for allocating resource access across multiple components in a way that preserves some invariants (e.g., avoiding resource exhaustion or concurrent use), enforces some fair access policy, or both. Examples of resource managers include transaction mechanisms in databases, use of thread pools in enterprise systems, and use of the ARINC 653 specification for space and time partitioning.

A resource manager is an extensibility tactic that supports the addition of new extensions by clearly exposing resource requirements and managing their common use. It promotes lower coupling and can be particularly useful for avoiding unintended coupling that can occur when resources are allocated implicitly and idiosyncratically by the individual components of a system. A resource manager can be used to prevent extensions from overusing some resources and to avoid conflicts of resource usage between extensions.

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 and helps lower the potential issues of extending an existing system.

Segmented deployments can be useful when dealing with extensions. They allow different extensions to be deployed separately, ensuring that each extension works before adding another one. This makes deployments more controllable and efficient.

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. This approach simplifies and regularizes analysis, and it provides more confidence in the completeness of the analysis.

Table 2 summarizes the extensibility 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 extensibility, and finally the other quality attributes that are negatively impacted by these patterns (tradeoffs).

Note 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. For example, the use of an intermediary always negatively affects performance (specifically latency). This is inevitable; the interposition of an intermediary adds processing and communication steps. However, the resulting latency of the system may be acceptable. Perhaps the added latency is 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 extensibility and modifiability while "costing" only a small amount of latency.

It is also important to note that the tradeoffs described below are general. Other architectural mechanisms or decisions applied with the pattern may change the impacts. An example would be lightweight proprietary protocols and standards in place of web services to mitigate the performance challenges of using heavier weight standards. The performance challenge could be mitigated while other problems are introduced. These are the kinds of assessments that analysts need to make when assessing the appropriateness of the patterns selected and implemented.

This pattern list is not meant to be exhaustive. The purpose of this section is to illustrate common extensibility patterns—Microkernel, Layers, Publish-Subscribe, and Intercepting Filter—and to show how analysts can break patterns down into tactics that help them analyze quality attribute scenarios. Table 2 maps the patterns to the extensibility tactics described in Section 5.1. The patterns themselves are described in the following subsections.

Table 2: Extensibility Tactics Mapped to Common Patterns

Tactic	Microkernel/ Plug-in	Layers	Publish-Subscribe	Intercepting Filter
Encapsulate	x	x	x	
Use an intermediary	x		x	
Restrict communication paths	x	x	x	
Abstract common services		x		
Defer binding	x		x	
Discover service (static)	x			
Discover service (dynamic)	x			
Specialized Interfaces				x
Record/Playback				x
Localize State Storage				
Orchestrate			x	
Manage resources	x		x	
Segment deployments	x			

5.2.1 Microkernel/Plug-In

The Microkernel or Plug-in architectural pattern provides a structure to add additional features to a core application after it has been completed, without the need to recompile it. This pattern defines a set of extension points through which extensions called plug-ins can be added to the core without requiring changes to it. Plug-ins provide extensibility and flexibility and isolate functionality and features added to the system. Traditionally the core of such a system provides only minimal functionality, which is the case of microkernels implemented in operating systems. However, the amount of functionality included in the core system can be defined without directly affecting the possibility of using plug-ins for extensibility. How much functionality is included in the core defines how generic the overall system is and how malleable it is to be changed through the addition of plug-ins.

Plug-in modules are independent components that can add new functionality to a system and tend to be independent of each other. The core system has a method to find plug-ins currently installed, such as a registry, in order to load them along with the system. Plug-ins are connected to the core system through extension points, which can be of different types, to support different types of plug-ins. The exact method used to connect plug-ins to the core system is not defined by this pattern and can be static (e.g., runtime libraries) or dynamic (e.g., using messages or a broker).

Benefits for extensibility:

- The main goal of the Microkernel pattern is to allow extensibility in a clear and cost-effective way, as plug-ins can be developed at different times than the core system and have clearly defined interfaces to implement to connect to it.
- The Microkernel pattern can be used to define explicitly what types of functionality can be added, and to which parts of the system, through extension points, allowing as much or little flexibility as desired in extensions.
- Plug-ins can be easily added or removed from a system, which allows for highly customizable versions of the system and for an easily segmented deployment of new functionalities.

Tradeoffs:

- Even though tests of specific plug-ins can be easily performed, testing the many potential combinations of different plug-ins installed to a system can become very complex when too many plug-ins exist, as the combinatorial explosion of potential setups may not be possible to properly test out.
- Managing multiple versions of plug-ins can be hard, as well as conflicts between different plug-ins in terms of use of resources or assumptions from the environment they will run on.
- Scalability can be adversely affected if too many plug-ins are installed, as monitoring and controlling the resources needed by multiple plug-ins can be hard to accomplish.

5.2.2 Layers

Layering is arguably the most common of all architectural patterns [Bass 2021, 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 extensibility. 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 communication paths* 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 their 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 extensibility:

- Layered systems, if they contain only unidirectional dependencies, minimize the ripple effects of changes.
- Layers allow a clear separation of the core system and extension points where new planned functionalities can be added later.
- Layers are often deployed using tiers, which can be deployed and monitored independently to detect and isolate issues in new functionality.

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.

5.2.3 Publish-Subscribe

Publish-Subscribe is an architectural pattern in which components communicate primarily through asynchronous messages managed by a publish-subscribe mechanism (usually a bus of some kind). Publishers have no knowledge of subscribers' identity and vice versa; each is only aware of event types. A publish-subscribe mechanism can be implemented in different ways. In some cases, a distributed infrastructure is used to convey events between publishers and subscribers across a network; an enterprise service bus provides a publish-subscribe mechanism. In other cases, a local library can be used to register callbacks and implicitly invoke methods on subscribers when events arrive. Regardless, the result is extremely loose coupling between publishers and subscribers in terms of identity [Buschmann 2007]. Publish-Subscribe does not, however, prescribe the events that are permissible; that requires a separate decision process and agreement among participants.

The Publish-Subscribe pattern is often built on top of a broker. In such cases, publishers publish messages to a message broker (or event bus), and subscribers register their subscriptions with the broker. The broker is thus only responsible for message forwarding and associated functions such as filtering or prioritization.

Benefits for extensibility:

- New components do not need to understand the identity of interacting components and can limit their knowledge to the events being communicated (including all dimensions of distance).
- New components can be integrated to information flows simply by adding a subscription.
- Existing components are not affected by the need to send information to new components; this is managed by the publish-subscribe bus.

Tradeoffs:

- Routing communication through a publish-subscribe bus takes time, which can negatively affect performance scenarios that are sensitive to communication latency. This latency may not be deterministic, particularly when the number of subscribers varies during operations.

- Reliance on asynchronous communication results in less deterministic behavior, which can negatively affect testability. Race conditions are a bigger concern in asynchronous systems.
- Reliance on a publish-subscribe bus to mediate obscures the identity of communication peers, which can negatively affect security. Publishers do not know the identity of their subscribers and vice versa. This can lead to challenges with key management for digital signatures. The publish-subscribe bus would need to store keys for every publisher and subscriber or use a single key for all. Using a publish-subscribe bus also puts the onus of enforcing authorization and authentication on the bus and removes any possibility for local control by a publishing component.

5.2.4 Intercepting Filter

Intercepting Filter is an architectural pattern where filters can be easily added in a standard way to predefined parts of the system, to preprocess or postprocess the data being received by or sent from that component. These filters can be added or removed without changing existing code. This pattern originated from web application controllers, where filters were used to alter the request or response data handled by the controller.

The Intercepting Filter pattern can be useful when planning the extensibility of the system if there is a clearly defined location where data is going to be processed by an important part of the system. Even though setting up the mechanisms for the Intercepting Filters to work out can take some effort, once they are set up, they can act as an extension point where new functionality to pre- or post-process data passing through that component can be easily added in the future.

Benefits for extensibility:

- It is easy to add new filters for common pre- or post-processing functionality, without affecting the existing codebase, and they may be enabled by changing a configuration file.
- It is easy to add, remove, or combine filters in different ways, adding flexibility to further extensions to the core system.
- This pattern can help with testing a new extension, since this pattern isolates new filters through a clearly defined interface, simplifying the development and testing process for a filter extension.

Tradeoffs:

- The addition of filters before or after a process handles its data will influence performance. The magnitude of the effect will depend on the number of filters applied to pre- or post-process the data and their algorithmic complexity.
- These filters can have a positive impact on security, as they can be used to provide standardized data cleansing, removing the need for each component to do this on its own.

6 Analyzing for Extensibility

An analyst's job is to judge the appropriateness of the extension points and mechanisms built into the architecture of a system, in light of the extensibility stimuli that the system will need to withstand. And as stated above, "appropriateness" is really a function of the risks and costs of the anticipated extensions. Analysts can specify these potential or anticipated extensions using scenarios, as we exemplified above, and for consistency and repeatability they can guide stakeholders to derive those scenarios from the general scenario for extensibility.

Analyzing for extensibility at different points in the software development lifecycle will take different forms. The different analysis options are sketched in Table 3. If analysts only have a reference architecture or a functional architecture, for example, then they cannot make detailed predictions or claims about the level of difficulty associated with achieving a desired extensibility response measure. 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 extensibility.

On the other hand, if the analysts have received a defined and documented [Clements 2010] product architecture—perhaps including views such as functional, hardware, and software architecture—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 3, the analysts can also begin to think about employing analysis models.

The point is that there is no one-size-fits-all analysis methodology and 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 expected degree of confidence in the analysis results will vary according to the quality of the available artifacts.

Table 3: Lifecycle Phases and Possible Analyses for Extensibility

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 Model-based analyses
Implemented System	Set of elements—services, processes, threads, etc.—and their interaction mechanisms—calls, pub/sub, messages, etc.—along with the mapping of these elements to hardware and networks	Checklist Measurement-based analyses Model-based analyses

6.1 Tactics-Based Questionnaire

Architectural tactics have been presented thus far as design primitives, following the work of Bass, Cervantes, and their colleagues [Bass 2021, 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 extensibility.

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 extensibility tactics-inspired questions presented in Table 4. The analyst asks each question and records the answers in the table.

Table 4: Tactics-Based Questionnaire for Extensibility

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage Dependencies	Does the system *encapsulate* its functionality so that only the parts that are needed for extension are exposed by an interface?				
	Does the system *use intermediaries* for breaking dependencies between components and future extensions, for example, removing a data producer's knowledge of its consumers?				
	Does the system provide a means to *restrict communication paths* between the system and extensions?				
	Does the system *abstract common services*, providing a general, abstract interface for similar services?				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system support *deferring binding* of functionality so that it can be replaced later in the lifecycle? For example, does the system use plug-ins, extensions, or user scripting to extend the functionality of the system?				
	Does the system have a known *(static) discovery service* that describes components (services) and provides their locations and their APIs?				
	Does the system provide a *dynamic discovery service*, enabling the binding between a service consumer and a service at runtime?				
Manage State	Does the system or do the system components provide *specialized interfaces* to facilitate testing, monitoring, and extension?				
	Does the system provide mechanisms that allow information that crosses an interface to be recorded to use it later for testing purposes (*record/playback*)?				
	Is the state of the system, or a significant subsystem, stored in a single place to facilitate testing (*localized state storage*)?				
Manage Deployments	Does the system include an *orchestration mechanism* that coordinates and manages the invocation of extensions so they can be ignorant of each other?				
	Does the system provide a *resource management capability* that governs access to computing resources?				
	Does the system support *segmenting deployments*, rolling out new releases gradually (in contrast to releasing in an all-or-nothing fashion)?				

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 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 is the implementation (realization) 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 4 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 Extensibility

As presented in Bass [2021], one can view an architecture design as the result of applying a collection of design decisions. We view architecture design and analysis as two sides of the same coin [Cervantes 2016]: any design decision made by an architect should be analyzed. Design and analysis are not distinct activities—they are intimately related. Below we present a systematic categorization of these decisions so that an architect or analyst can focus attention on those 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. They are

1. allocation of responsibilities
2. coordination model
3. data model
4. resource management
5. mapping among architectural elements
6. binding time
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 alright if a particular decision exists in two different categories because the duty of the architect and the analyst is to assess whether 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 extensibility is presented below.

Category	Checklist
Allocation of responsibilities	Consider what types of extensions are likely to be needed through potential changes in technical, legal, and mission forces. Do the following for each potential extension: • Consider the responsibilities that would need to be added, modified, or deleted to make the extension work. • Consider what *other* responsibilities are impacted by this change. • Assess whether the allocation of responsibilities to modules 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	Consider which parts of the system—devices, protocols, and communication paths—used in extension points are likely to change. For those devices, protocols, and communication paths, assess the impact of changes (ideally limited to a small set of modules). For those elements for which future modifications are likely, assess whether the coordination model, such as publish-subscribe, reduces coupling; defers bindings such as enterprise service bus; or restricts dependencies such as layering.
Data model	Consider which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur as a result of an extension. Also consider 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, consider whether 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, assess whether the necessary attributes are visible to that user and whether the user has the correct privileges to modify the data, its operations, or its properties.

Category	Checklist
	Do the following for each potential change or category of change: • Consider which data abstractions would need to be added, modified, or deleted to make the change. • Consider whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions. • Consider which other data abstractions are impacted by the change. For these additional data abstractions, consider whether the impact would be on the operations, their properties, or their creation, initialization, persistence, manipulation, translation, or destruction. • Assess whether the data model was designed so that items allocated to each element of the data model are likely to change together.
Mapping among architectural elements	Consider whether it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time. Consider the extent of modifications necessary to accommodate an extension. This might involve a determination of the following, for example: • execution dependencies • assignment of data to databases • assignment of runtime elements to processes, threads, or processors Assess whether changes are performed with mechanisms that utilize deferred binding of mapping decisions.
Resource management	Consider how the integration, removal, or modification of a responsibility caused by an extension will affect resource usage. This involves the following example activities: • Consider what changes might introduce new resources, remove old ones, or affect existing resource usage. • Consider what resource limits will change and how. • Assess whether the resources are sufficient to meet the system requirements after deploying the extension.
Binding time	Do the following for each change or category of change: • Consider the latest time at which the extension will need to be deployed. • Assess whether the defer-binding mechanism will deliver the appropriate capability at the time chosen. • Consider the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. • Assess whether the design introduces so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of technology	Consider what extensions are made easier or harder by the technology choices. • Consider whether technology choices help to make, test, and deploy extensions. • Assess how easy it is to modify the choice of technologies (in case some of these technologies change or become obsolete). Assess whether the chosen technologies support the most likely anticipated extension types. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

7 Playbook for an Architecture Analysis of Extensibility

This playbook outlines an approach to combine the checklists and questionnaires presented in the previous sections with information about architectural mechanisms to analyze an architecture to validate the satisfaction of an extensibility 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 for satisfying the quality attribute requirement. The process ends with the Evaluation phase, when the analysts apply their understanding of the requirements and architecture solution approaches to make judgments about those approaches. The phases and steps are summarized in Table 5.

Table 5: 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 impact on other quality attribute requirements
	Step 7–Assess the cost/benefit of the architecture approach

The analysts 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 analysts must decide whether there is sufficient information 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–Assess Requirement Satisfaction and the degree of confidence in the results. Early in the development lifecycle, lower confidence may be acceptable, and analysts can work with lower quality artifacts and simpler analyses, as suggested in Table 3. Later in the lifecycle, analysts need higher confidence and therefore higher quality artifacts and more and deeper analyses.

7.1 Step 1–Collect Artifacts

In this step, analysts collect the artifacts that they will need to perform the analysis. These will typically include quality attribute requirements and architecture documentation but may also include other forms of information such as results from prototyping or modeling or historical system information (if the architecture being evaluated already exists).

The first artifact that analysts need is the extensibility requirements to validate. All such requirements must be stated so that they are measurable. One way of doing this is to write them as quality attribute scenarios, as discussed above. Let’s use a variant of example Scenario 5: New Format for Importing Data from Section 4.2, where we have specified the artifact as “Data management component extension point.” Let’s call this Scenario 6.

Scenario 6: Importing health data from an external provider

Scenario Part	Value
Source	External party
Stimulus	Import health data from a World Health Organization provider
Artifact	Data management component extension point
Environment	Development time
Response	Data in prespecified JSON format can be imported into the system and converted into internal structure for storage using existing extension points
Response Measure	No part of the core system needs to be recompiled Development can be done using extension points in 2 person-weeks by an external party

Next, the analysts need the other quality attribute requirements. As noted above, architecture designs embody tradeoffs, and decisions that improve extensibility may have a negative impact on the satisfaction of other quality attribute requirements. In Step 6–Assess Impact on Other Quality Attribute Requirements, the analysts 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 analysts need 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 or tactics. Naming of architecture elements may indicate the mechanism being used. The analysts may also look at the file structure and naming of source code repositories or packages to find mechanisms. The analysts may need to use all of these to identify the mechanisms that are being used to satisfy the extensibility requirement. Frequently, multiple mechanisms are needed to satisfy a requirement. If the analysts have 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 analysts can enumerate all the relevant mechanisms chosen (and not chosen).

For the example requirement above about importing data in a new format, let's say the project has already finished most of its development, and it is deployed in a production environment. The development team created documentation that describes the different extension points that are provided by the system, and the ways to use them. They also provided a rationale writeup explaining the reasons for what can and can't be done with each extension point. They indicated that they used the Microkernel/Plug-in pattern to simplify the extension of some specific areas, in particular for importing external data. They used the deferred binding tactic to allow plug-ins to be added to the system without requiring recompilation and the dynamic service discovery tactic to allow the system to automatically discover services provided by plug-ins after they are installed.

The analysts perform a quick check to decide if the referenced mechanisms are likely to contribute to satisfying the extensibility requirement. In this case, all mechanisms that are enumerated above are known to positively impact extensibility measures. They describe one pattern and two tactics for extensibility—so the check passes.

In contrast, if the documented rationale (or the architect) stated that the architecture used a *ping/echo* mechanism to achieve the above extensibility requirements, this would raise a red flag since *ping/echo* is usually associated with improving robustness, but not extensibility. The analysts might decide to pause the architecture analysis at this point and gather more information from the architect. The point of this quick check is not to analyze the mechanisms or decisions in detail but simply to assess whether the architecture analysis is on the right track, in terms of the available information and the mechanisms that have been chosen, before devoting more effort to a deeper analysis.

In some cases, the appropriateness of a mechanism is less clear. For example, the rationale for extensibility design choices might specify that an *encapsulation* mechanism is used. Encapsulation can support extensibility to some extent, but this mechanism by itself is usually insufficient since it only ensures that there is a separation of responsibilities; it may not provide any significant help for extending the system. In cases like this, the analysts should proceed carefully: the architect may have chosen an inappropriate or inadequate mechanism or used a mechanism in an improper way.

7.3 Step 3–Locate the Mechanisms in the Architecture

Following our example, the analysts need to use the architecture documentation, an interview with the architect(s), or reverse-engineering to locate the mechanisms realizing the extension points in the architecture. As seen in the tactics-based questionnaire, it is important to consider *how* a mechanism—tactic or pattern—is implemented.

Our scenario is concerned with being able to import data from an external provider in a new format. The analysts may be able to look at the documentation and find a structural diagram that defines the existing extension points and where they are located. With this information, the analysts can find whether one of these supports the creation of a plug-in to handle the importing of data. The analysts should also be able to find details on what functionality is and is not available to that type of plug-in and evaluate whether the type of data import that is required would be possible and reasonable using the extension point or points.

The analysts should next look for documentation related to the deferred binding tactic and dynamic service discovery tactic that complement the extension points. The deferred binding tactic may be found in the architectural diagrams and rationale, where it defines how plug-ins are connected to the system and loaded. Information about the dynamic service discovery tactic is likely to be documented separately, in instructions for how to offer new services provided by plug-ins, aimed at people extending the system. If this documentation is not available, the analysts may have to look at other existing plug-ins to understand their structure and configurability, or at the system itself.

Finally, the analysts must be able to conceptually integrate the mechanisms. The rationale for satisfying the requirement stated that binding of the plug-ins could be deferred to allow extensions to be loaded on an as-needed basis to import data. This raises a question: When are data management plug-ins loaded? This is an issue of *binding*, one of the categories of questions in the Architecture Analysis Checklist. One answer could be that all plug-ins are loaded only when the system is restarted. Another is that plug-ins are loaded dynamically, without requiring a restart, every time a new plug-in is added to the system. However, the analysts find that, in reality, the system uses a combination of both: plug-ins that use some extension points require a restart of the system to be available, while others, including the ones handling data management, are loaded after they are deployed, while the system is running.

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–Identify the Mechanisms Used to Satisfy the Requirement. That establishes only the presence of the mechanisms, not their suitability or adequacy for meeting the response goal of 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, if they find a dynamic service discovery mechanism, but only for services that provide various printing capabilities, then this use of the mechanism is not likely to directly improve the extensibility of the data management component when importing different types of data.

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 uses the Microkernel/Plug-in pattern along with the deferred binding tactic. As just mentioned, one set of decisions about using that mechanism is concerned with when the data management plug-ins are loaded (refer to the *binding time* category in the Architecture Analysis Checklist). In this case, alternatives include

- at build time (this would imply not using the deferred binding tactic)
- when the software system is restarted (hence, no new plug-ins can be loaded while the system is running)
- while the system is executing (so new plug-ins can be loaded without the need to restart)

If the architect decided to allow plug-ins to be loaded while the system is executing, then there is a set of subsequent derived decisions about how and when to do this. One option would be to load new plug-ins based on a timer, which triggers the discovery of new plug-ins. A second option would be to load plug-ins based on user input (such as pressing a button to refresh plug-ins). Another option would be for plug-ins to be loaded immediately once they are added to the system, which would require the system to constantly monitor the plug-in storage for new ones to appear.

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–Identify the Mechanisms Used to Satisfy the Requirement. The analysts must evaluate how the mechanism was instantiated, which usually involves tracing the decisions about the instantiation to the derived decisions and the selected alternatives that address them. And the analysts must endeavor to determine whether the mechanism, as envisioned and instantiated, is likely to actually meet the requirement.

As analysts identified the mechanisms in Step 3–Locate the Mechanisms in the Architecture, they also started to identify derived decisions. For example, in the options outlined above, the analysts identified that data management plug-ins do not require a system restart but require a user action to be loaded.

The analysts' next derived decision might be "Where does the data management component get the information to load a data import plug-in, and how and where are plug-ins deployed to the system?" This is an *allocation of responsibilities* decision in the Architecture Analysis Checklist. For the extensibility requirement that the analysts are validating, good answers to this question might include the following:

- All data management plug-ins are loaded in the same way into memory, from modules found in a specified folder.
- All data management plug-ins contain a configuration file with metadata about the plug-in and its functionality, located in the same folder as the plug-in.
- All data management plug-ins are accessed through a common interface that acts as the main extension point for data management functionality.

If these statements are all true, then the data management component has a clear extension point for loading and accessing data-related plug-ins, so our requirement to extend the system by adding a new plug-in to convert data from a specific external format can be addressed without changes to the core of the system. (On the other hand, if the driving quality attribute requirement was, for example, the performance of the process of importing new data, then the architect might have chosen to include in the system direct calls to some data management functionality that is built into the rest of the system. Adding functionality to import data in a different format would then require changing the data management component, which would make the system less extensible).

Another derived decision is related to how the plug-in information is represented in the configuration file. This is covered in the *data model* category in the Architecture Analysis Checklist. The analysts check that the information and schema required by the core when loading a plug-in are consistent with the information presented in the data management plug-in configuration file. For example, if information about the input and output formats of a data management plug-in is limited to only prespecified types of formats, then extending the system to import a new data format will require changes in the extension point itself, impacting the satisfaction of our extensibility requirement.

Finally, some mechanisms have special cases that warrant special attention. For example, modifications to the top or bottom layers in a layered mechanism introduce concerns about interfaces outside the mechanism. Changes to the functionality available in the core system may require additions to the extension points used by plug-ins if this new functionality is to be exposed to the extensions.

In our example, analysts should pay attention to the way a data management plug-in obtains and stores the data to be converted, as there may be limitations in the potential sources and targets of the converted data. Our requirement to import data in a new format will be more easily satisfied if we understand the source and target of that data.

7.5 Step 5–Assess Requirement Satisfaction

The analysts have now completed preparation and orientation and can begin 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, if the analysts are assessing a quality attribute requirement for portability to a different hardware platform, and the mechanism being used is the Layers pattern with a hardware abstraction layer as the lowest layer, then the analysis should include checks for layer skipping, which may introduce unwanted 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 6 is to import data from an external source in a new format. Our measures are that the development can be done "in 2 person-weeks by an external party" using the provided extension points, without needing to recompile the existing code. As discussed, the architecture mechanisms identified are Microkernel/Plug-in, deferred binding, and dynamic service discovery. In Step 4–Identify Derived Decisions and Special Cases, the analysts identified several derived decisions that need to be considered in the analysis:

- Where does the data management component get the information to load a data import plug-in, and how and where are plug-ins deployed to the system?
- How is the plug-in information represented in the configuration file?

The analysts might begin with the last question—a question about the data model—since they need to understand the structure of the configuration file to address the other questions. The configuration file schema is defined in the documentation about extension points, and it describes how it uses XML with a predefined schema to indicate the plug-in metadata. All elements that use the configuration file are semantically coupled—they must all represent and interpret the contents in the same way. The analysts begin recording analysis issues related to semantic cohesion:

- Issue 1: It is not clear how flexible the XML schema is. The documentation does not indicate whether all fields are mandatory or if some are optional. If the developer needs to figure this out by trial and error, it will increase the effort to add a new data import plug-in.

- Issue 2: It is not clear if all the data types necessary to configure the new plug-in are available in the schema. Modifying the schema would likely increase the effort to add the new plug-in.

This analysis thread is based on an observation that the configuration file uses XML syntax. For example, if the analysts found that the file had an YAML schema, which is not strongly typed, then they might record an issue about handling invalid values robustly. If invalid configuration values are not robustly detected, this could increase the effort to add a new plug-in.

Continuing our example, the analysts find that the plug-ins are dynamically linked to the core. This architecture decision reduces syntactic coupling between the core code and the plug-in code files, because clearly defined extension points implemented as interfaces are the main element that both parts of the code must understand in the same way. On the other hand, this decision potentially adds runtime errors such as those created by missing the dynamically linked plug-in or by the plug-in not properly implementing the extension point. The analysts record the following issue:

- Issue 3: The plug-in code is dynamically loaded by the core system. This introduces the possibility of runtime errors when loading the plug-ins. The system needs a robust error check method to capture these types of errors and handle them appropriately.

Next, in our simplified analysis example, the analysts investigate how the component that loads data management plug-ins provides access to core functionality that may be needed by the plug-in, such as logging and user notifications. The architecture documentation states that a data management plug-in has access to a singleton object called Logger, which provides unified logging functionality. This analysis triggers another issue to record:

- Issue 4: There is syntactic coupling between the Logger and plug-ins. If changes are made to the core logging interface, every plug-in that uses that functionality would be affected and would need to be recompiled.

In this simple example, the analysts rapidly identified four issues where architecture decisions might impact the ability to achieve the desired response measures in Scenario 6. Some of the issues, such as Issue 3 about dynamic linking, are unlikely to change as the details of the architecture are refined. Other issues, such as Issue 1 about the XML schema documentation, may be resolved as the details of the architecture are refined and are to be expected when analyzing an architecture early in the development cycle.

7.6 Step 6–Assess 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 analysts assessed in Step 5–Assess Requirement Satisfaction may be detrimental to the satisfaction of other quality attribute requirements.

Typical tradeoffs impact software performance (throughput or latency), testability, robustness, availability, maintainability, and usability. In Step 1–Collect Artifacts, the analysts collected other quality attribute requirements that were available at this point in the development lifecycle. Now, they 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:

- Does the architecture provide services for common concerns such as fault handling, communication, and logging for new plug-ins?
- Are there real-time latency requirements for this system that could be affected by importing new data? And is there a latency goal for how quickly the data is imported? Finally, the ability to add code via different extension points could introduce nondeterminism, which may affect deadlines.
- Does the architecture prescribe security mechanisms to prevent plug-ins from accessing secure or sensitive parts of the system? Does the architecture define a way to coordinate plug-ins that are trying to access the same resources? Are plug-ins sandboxed, or can they access most or all system resources?
- Are there performance or scalability requirements that could be negatively affected by the addition of large numbers of plug-ins to the system?

In this step, the analysts assess how the mechanisms and decisions that make it easy to add a new plug-in to import data impact the satisfaction of scenarios related to other quality attributes and concerns. For each requirement, the analysis may be fast (e.g., there are clearly defined services for logging and fault handling of plug-ins) or more involved (e.g., evaluating the impact in real-time requirements of deploying and using the new plug-in). In any case, the analysts should expect to find at least a couple of additional issues.

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

In carrying out the steps leading up to this point, the analysts should have developed a good understanding of the essential challenges in 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 expected frequency and variety of extensions to be added to the system is low and timing is clearly defined, a plug-in system may add unnecessary complexity. On the other hand, if different kinds of functionality will be added frequently and at different points in the project's lifetime, providing a flexible plug-in mechanism is almost certainly worth the effort.

Often the cost/benefit analysis is not clear and requires some judgment, 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 the quality attribute of extensibility—the ability of a software-intensive system economically and predictably to add functions, capabilities, and support for key quality attributes over a long period of time. We focused on analyzing the specific challenges of achieving extensible systems and analyzing for extensibility, based on an understanding of architectural mechanisms and their characteristics.

We provided a set of sample scenarios for extensibility and, from these and other examples, inferred a general scenario. This general scenario can be used as an elicitation device and helps with analysis as it delineates the response measures that stakeholders will care about when they consider this quality attribute. We have also described the architectural mechanisms—tactics and patterns—for extensibility. 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 the analysts can understand the design decisions made, or not made, their rationale, and their potential consequences.

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

9 Further Reading

The tactics on which much of this report are based were first described in *Software Architecture in Practice* [Bass 2021], in the technical reports on *Integrability* [Kazman 2020a] and *Maintainability* [Kazman 2020b], and in a research paper on tactics for deployability [Bellomo 2014].

Documentation of the Chrome manifest, extension mechanisms, and APIs for web browsers can be found at https://developer.chrome.com/docs/extensions/. Virtually all modern web browsers follow the Chrome APIs, so these provide an excellent case study on how to create extensions at scale.

References

[Bass 2021]
Bass, L.; Clements, P; & Kazman, R. *Software Architecture in Practice*, 4th ed. Addison-Wesley. 2021.

[Bellomo 2014]
Bellomo, S.; Ernst, N.; Nord, R.; & Kazman, R. Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. Pages 702–707. In *Proceedings of 44th IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society. 2014.

[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.

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

[Breivold 2007]
Breivold, H. P.; Crnkovic, I.; & Eriksson, P. Evaluating Software Evolvability. Pages 96–103. In *Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden (SERPS 2007)*. IT University of Göteborg. 2007.

[Buschmann 2007]
Buschmann, F. et al. *Pattern-Oriented Software Architecture*, Volumes 1–5. Wiley. 1996–2007.

[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.

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

[Kazman 1997]
Kazman, R.; Clements, P.; Bass, L.; & Abowd, G. Classifying Architectural Elements as a Foundation for Mechanism Matching. Pages 14–17. In *Proceedings of COMPSAC 1997*. IEEE Computer Society. August 1997.

[Kazman 2020a]
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

[Kazman 2020b]
Kazman, Rick; Bianco, Philip; Ivers, James; & Klein, John. *Maintainability*. CMU/SEI-2020-TR-006. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=650480

[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.

[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

[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.

[Padilla 2019]
Padilla, M. O.; Davis, J. B.; & Jacobs, W. Comprehensive Architecture Strategy (CAS). The Open Group. September 2019. https://www.opengroup.us/face/documents.php?action=show&dcat=87&gdid=21082

[Papke 2000]
Papke, B.; Brock, D.; & Graybeal, J. Extensible and Flexible Software Architecture for the SOFIA Mission Controls and Communications System. In *Proceedings of SPIE 4014, Airborne Telescope Systems*. Society of Photo-optical Instrumentation Engineers. June 2000.

[Techopedia 2021]
Extensible. Techopedia. 2021. https://www.techopedia.com/definition/7107/extensible

[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*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** April 2022		3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Extensibility			5. **FUNDING NUMBERS** FA8702-15-D-0002
6. **AUTHOR(S)** Rick Kazman, Sebastian Echeverria, James Ivers			
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2022-TR-002
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 of extensibility. The report introduces extensibility and common forms of extensibility requirements for software architectures. It provides a set of definitions, core concepts, and a framework for reasoning about extensibility and satisfaction (or not) of extensibility 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 extensibility 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, to determine whether the architectural decisions made contain serious risks relative to extensibility 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, extensibility, quality attributes, quality attribute requirements, software architecture			15. **NUMBER OF PAGES** 53
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