Carnegie Mellon University
Software Engineering Institute

Integrability

Rick Kazman
Phil Bianco
James Ivers
John Klein

October 2019

http://www.sei.cmu.edu

Table of Contents

1 Goals of This Document

This document serves several purposes. It is

- an introduction to integrability and common forms of integrability requirements
- a description of a set of mechanisms, such as patterns and tactics, that are commonly used to satisfy integrability 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 integrability requirements
- a means for an analyst to determine whether those integrability requirements, represented as a set of scenarios, have been sufficiently well specified to support the needs of analysis

This document is part of 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. The purpose of this document, as with all of the documents in this series, is to provide a workable set of definitions, core concepts, and a framework for reasoning about quality attribute requirements and their satisfaction (or not) by an architecture and, eventually, a system. In this case the quality attribute under scrutiny is *integrability*. 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]. Among those widely found in practice are

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 a 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 analysis of software architectures because a software architecture is the major carrier and enabler of a system's driving quality attributes. And since software typically changes much more frequently than hardware, it is often the focus of integration effort. There will, however, be implications of architectural decisions made on each of the other views.

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

As a preview to the remainder of this document, we wish to stress that there is not one single way or one single time to analyze for integrability. One can (and should) analyze for integrability 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 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 Integrability

According to the Merriam-Webster dictionary, the adjective *integrable* means "capable of being integrated." Many definitions of integrability can be found in the software and system engineering literature. But they all follow similar wording and logic. For example, Henttonen [2007] defines it as follows: "Integrability means an ability to make separately developed components of a system to work correctly together."

For practical software systems, software architects need to be concerned about more than just "the ability to make separately developed components" cooperate; they are concerned with the *costs* and *technical risks* of anticipated and (to varying degrees) unanticipated future integration tasks. These risks may be related to schedule, performance, or technology. A general, abstract representation of the integration problem is that a project needs to integrate a unit of software[1] C, or a set of units C_1, C_2, … C_n, into a system S, as depicted in Figure 1. Note that S might be a platform, into which we integrate $\{C_i\}$, or it might be an existing system that already contains $\{C_1, C_2, \ldots, C_n\}$ and our task is to analyze the costs and technical risks of integrating $\{C_{n+1}, \ldots C_m\}$.

When we consider and analyze for integrability, we are attempting to predict the amount of effort, and the amount of risk, that would be involved in integrating $\{C_i\}$ with system S (specifically taking into account the integrability mechanisms designed into S). Thus, our objective in analyzing architectures is to estimate the costs and risks of a set of anticipated or postulated integration tasks as precisely as we can, given the available information. The identified costs and risks can then be used as a justification for adopting the architectural decisions as is, or for changing some of those decisions to mitigate the predicted risks.

We approach the topic of integrability from two perspectives: integrability as an architectural property and integration as a task. *Integrability* is about the degree to which an architect has anticipated and designed for integration tasks that the system may undergo. An integrability analysis of an architecture is predictive in nature. We are trying to predict the costs and risks of integrating some software unit at some future point in time. In an integrability analysis we assume the following:

- We know what S is, as we have some representation of its architecture. S is, after all, the primary focus of our analysis and is typically the artifact over which a system has the greatest engineering control. (Note also that while this representation will evolve over time, at any given moment we must choose a snapshot of S to analyze.)
- We have some idea of each C_i, but our level of understanding of each may C_i vary. The clearer our understanding of C_i, the more accurate the analysis will be. The level of knowledge about each C_i may vary due to our level of familiarity with each C_i, due to the

[1] Different organizations may use different terms for a unit of software or refer to a library, module, component, or CSCI (computer software configuration item). In practice many of these terms are used imprecisely, despite some of them (e.g., module and component) having clearly distinct meanings in the software architecture community.

fact that each C_i will evolve as time passes, or due to license reasons (for example, we might have the source code for one C_i but not for another).

We don't know what integration decisions will be made in the future—for example, what assumptions, protocols, or middleware S and C_i will employ—but we can analyze to predict the costs and risks associated with integrating $\{C_i\}$ into S as currently architected.

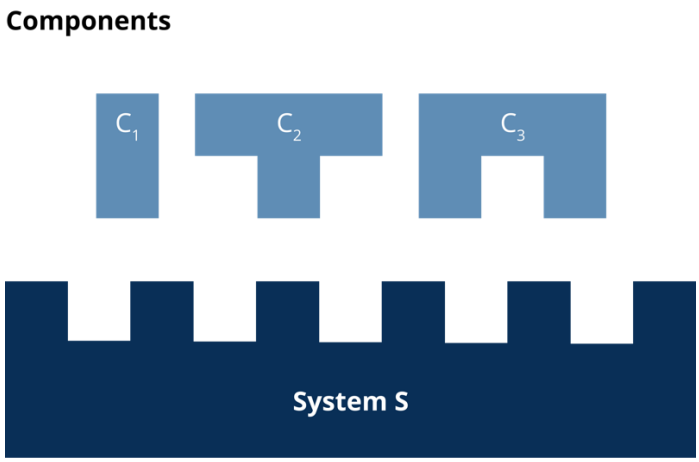

Figure 1: Abstract Representation of the Integration Problem

Integration, on the other hand, is a specific instance of a task in which two or more entities are being made to work together. The degree to which an integration task can be accomplished at acceptable levels of costs and risks depends, to a large extent, on the degree to which the architect of S appropriately designed integrability mechanisms into S's architecture. In contrast to integrability, integration is not predictive; it is done in the context of implementations and can rely on stronger assumptions:

- We know the specifics of both S and $\{C_i\}$. We have implementations of both, so we have a great deal more knowledge of the semantics of each.
- During integration, we may make changes to S and $\{C_i\}$. That is, we may change S to ease the integration of $\{C_i\}$, we may change $\{C_i\}$ to match the assumptions of S, we may change both, or we may insert something in between S and $\{C_i\}$ so that neither has to change.

Thus, an integrability analysis is an attempt to predict which changes to S or $\{C_i\}$ will have to be made during a future integration task, based on how far apart S and $\{C_i\}$ are. In practical terms, however, if changes are required to satisfy an integration scenario, then changes to S are typically more likely than changes to $\{C_i\}$. This is based on the assumption that the architect of S typically has full control over S but only limited or no control over $\{C_i\}$ as these may include third-party and commercial components.

One consequence of this process, however, is that after each change to S, we now have S', and all prior (and future) analyses of integrability may need to be revisited to determine if the costs and technical risks of integrating $\{C_i\}$ into S' are acceptable. Thus S is not static but will evolve, and this evolution may require reanalysis. Integrability (like other quality attributes such as maintaina-

bility) is challenging to analyze because it is about planning for a future with incomplete information. Simply put, some integrations will be simpler than others because they have been anticipated and accommodated in the architecture (or the architecture's build process) whereas others will be more complex because they have not been.

Consider a simple analogy: to plug a North American plug (an example of a C_i) into a North American socket (an interface provided by the electrical system S), the "integration" is trivial. However, integrating a North American plug into a British socket will require an adapter. And the device with the North American plug may only run on 110-volt power, requiring further adaptation before it will work in a British 220-volt socket. Furthermore, if the component was designed to run at 60 Hz and the system provides 50 Hz, the component may not operate as intended. From this example we can see that there is cost and risk related to the effort involved in performing an integration. The architectural decisions made by the creator of S—for example, to provide plug adapters or voltage adapters, or to make the component operate identically at different frequencies—will affect this cost and risk. We refer to these decisions in Section 5 as "mechanisms." That is, the architect chooses a set of mechanisms to ease the costs and risks of anticipated integration tasks.

Integration difficulty—the costs and the technical risks—can be thought of as a function of the size of and "distance" between the interfaces of $\{C_i\}$ and S:

- Size is the number of potential dependencies between $\{C_i\}$ and S.
- Distance is the difficulty of resolving differences at each of the dependencies.

"Interfaces" here are much more than simply application programming interfaces (APIs) [Clements 2010]. Interfaces between $\{C_i\}$ and S include the *semantics* of how they interact, the *interpretation* of the data that they exchange, and the *assumptions* or resources that they share. These unstated, implicit interfaces often add time and complexity to integration tasks (and modification and debugging tasks) as we will discuss in Section 7.

To employ another analogy from physical interfaces, consider the ubiquitous Universal Serial Bus (USB) standard. The USB standard was initially created to simplify the integration of external devices with personal computers. As such, it is much more than simply a cabling protocol. It also consists of a number of communication protocols, which standardize how external devices (components) can exchange information with a computer (systems). USB created a universal integrability mechanism for specific classes of devices—storage, modems, pointers, displays, etc.—by reducing the size and distance of the interfaces between $\{C_i\}$ and S.

While any integration task has a near-term cost, the accumulation of integration tasks will impose long-term costs on the system. The *trajectory* of these costs is one of the concerns that an analyst must consider when analyzing a system S with respect to integrability.[2] Thus we distinguish, in

[2] By "trajectory" we mean an anticipated series of future integrations. And, of course, some trajectories will pose a risk to system success.

this document, between “integration”—a specific task—and integrability—a quality of an architecture that can be designed in and analyzed in the context of specific integration scenarios.

3 Evaluating the Integrability of an Architecture

To evaluate the integrability of an architecture, we need to measure the potential dependencies between S and $\{C_i\}$. A dependency is a kind of coupling, which is defined as

- "a measure of the interdependence among modules in a computer program" [SWEBOK 2014]
- "1. manner and degree of interdependence between software modules 2. strength of the relationships between modules. 3. measure of how closely connected two routines or modules are" [ISO 2017]

While those definitions use the generic term "modules" to describe units of software, the concepts apply identically to the potential dependency between S and any $\{C_i\}$. When analyzing and evaluating an architecture's support for integrability, we would like to know that the potential dependencies between S and $\{C_i\}$ are low. We assume that we, as analysts, already know S—we have some appropriate architectural documentation describing S—and we enumerate the potential $\{C_i\}$, along with any desired response measures via scenarios. Our architectural analysis will then attempt to predict whether the dependency between S and $\{C_i\}$ will be appropriate, which is to say that it will allow us to meet our desired response measures.

For example, if S offers a mechanism that implements a standard that many organizations use, then any C_i that conforms to that standard will be more integrable because there are fewer integration decisions to make and validate. A standard helps achieve this because it predetermines some of the syntax and semantics of dependencies. Or, to take another example, if the architecture of S provides a publish-subscribe mechanism and the $\{C_i\}$ adhere to this protocol, publishing and subscribing to "topics," then integration is eased because there it reduces the dependencies between S and $\{C_i\}$. In each of these examples, the "distance" between S and $\{C_i\}$ is being reduced. We will discuss the notion of distance in more detail shortly.

Dependencies have traditionally been measured syntactically. For example, we say that module A is dependent on component B if A calls B, if A inherits from B, or if A uses B. But while syntactic dependency is important, and will continue to be important in the future, dependency can occur in forms that are not detectable by any syntactic relation. Two components might be coupled *temporally* or through *resources* because they share and compete for a finite resource at runtime (e.g., memory, bandwidth, CPU), share control of an external device, or have a timing dependency. Or they might be coupled *semantically* because they share knowledge of the same protocol, file format, unit of measure, metadata, or some other implementation detail. The reason that these distinctions are important is that dynamic and semantic dependencies are seldom well understood, explicitly acknowledged, or properly documented. And implicit knowledge is always a risk for a large, long-lived project as this knowledge only lives in the heads of a few key people. Implicit knowledge will inevitably increase the costs and risks of integration and integration testing.

Consider the trend toward services, and microservices, in computation today. This approach is fundamentally about *reducing* dependencies. Services only "know" each other via their published

interfaces and, if that interface is an appropriate abstraction, changes to one service have less chance to ripple to other services in the system. Services (and microservices and publish-subscribe) are examples of an industry-wide trend to decouple components in a system that has been going on for decades. Note, however, that service orientation, by itself, only addresses (that is, reduces) the syntactic aspects of dependency. It does not address the dynamic or semantic aspects. If a developer uses publish-subscribe or services but the supposedly decoupled components have detailed knowledge of each other and make assumptions about each other, then they are in fact tightly coupled and changing the details of their integration will be costly.

Thus, there is no silver bullet for integrability. Whether we are building traditional monolithic applications or service-based applications, we need ways to plan for the appropriate level of dependency—*all* forms of dependency—and to explicitly measure and monitor these dependencies. If we aim to exercise engineering control over the quality attribute (QA) of integrability over the lifetime of a system, then we must measure and manage dependency. This is now where we turn our attention.

3.1 Measuring Integrability

Integration difficulty—the cost and the risk—can be thought of as a function of the "size" of and "distance" between the interfaces of $\{C_i\}$ and S, where

- size is the number of potential dependencies between $\{C_i\}$ and S
- distance is the difficulty of resolving differences at each of the dependencies

Before we continue, we need to make two additional points clear:

1. Size is a way of inventorying the set of things that we need to be concerned about. It is an enumeration of the potential sources of costs and risks. It is not to be taken as a basis for analysis by itself. The heart of the analysis is in estimating the distances between the interfaces of $\{C_i\}$ and S.
2. "Interfaces" must be understood as much more than simply APIs, as we stated in Section 2. Interfaces between $\{C_i\}$ and S include the *semantics* of how they interact, the *interpretation* of the data that they exchange, and the *assumptions* or resources that they share. These include assumptions and constraints that do not typically get mentioned in a programming language interface description, such as "component A and B are both allocated to processor X so they need to share memory and CPU budget appropriately," "component A needs to run 50 ms before component B," "component A opens the device interface and component B expects it to be in that open state," or "component A and B both depend on some magic number that represents the max key length." These unstated, implicit interfaces often add time and complexity to integration tasks (and modification and debugging tasks).

We define the complexity of integrability with respect to pairs of cooperating components (x,y), where x and y are instances of integration points in S and $\{C_i\}$, respectively. The complexity of integrability is notionally defined by the following formula:

$$Integrability\_Complexity = \sum_{x,y} f(size(x,y) \times distance(x,y))$$

where size is the number of information elements shared between *x* and *y* (for example, parameters in an API) and distance is a measure of the differences in assumptions between *x* and *y*. This formula is not meant to be interpreted as a validated metric. Certainly there will be a positive correlation between integrability complexity—which we gauge using size and distance—and the cost or risk of actually completing required integration tasks. But we do not claim that, for example, doubling the value of this integrability complexity will double the integration cost or risk. The purpose of the formula is to illustrate the kinds of information that goes into assessing the costs and risks of integration.

As mentioned above, measuring distance is the greater challenge for the analyst. This is because distance may be measured along multiple dimensions. Let us now examine each of these dimensions in detail:

- **Syntactic distance**: The elements have different data types. For example, one element is an integer and the other is a floating point, or perhaps the bits within a data field are interpreted differently. Differences in data types are typically very easy to observe and predict. For example, such type mismatches could be caught by a compiler. Differences in bit masks, while similar in nature, are often harder to detect, and the analyst may need to rely on documentation or even scrutiny of the code.
- **Data semantic distance**: The elements have different data semantics; that is, even if they share the same data type, the values are interpreted differently. For example, one data value represents distance in meters and the other represents it in centimeters. This is typically difficult to observe and predict, although the analyst's life is improved somewhat if the elements involved employ metadata. Mismatches in data semantics may be discovered by comparing interface documentation or metadata descriptions, if available, or by checking the code, if available.
- **Behavioral semantic distance**: The elements behave differently, particularly with respect to different states and modes of the system. For example, a data element may be interpreted differently in system startup, shutdown, or recovery mode. Such states and modes may, in some cases, be explicitly captured in protocols. As another example, *x* and *y* may make different assumptions regarding control, such as each expecting the other to initiate interactions.
- **Temporal distance**: The elements may embody different assumptions about time. For example, they may operate at different rates (e.g., one element emits values at a rate of 10 Hz and the other expects values at 60 Hz) or different timing assumptions (e.g., one element expects event A to follow event B and the other element expects event A to follow event B with no more than 50 ms latency). While this might be considered to be a sub-case of behavioral semantics, it is so important (and often subtle) that we call it out explicitly.
- **Resource distance**: The elements may embody different assumptions with respect to shared resources such as devices (e.g., one element requires exclusive access to a device whereas the other expects shared access) or computational resources (e.g., one element needs 12 GB of memory to run optimally and the other needs 10 GB, but the target CPU has only 16 GB of physical memory, or three elements are simultaneously producing data at 2 Mbps each,

but the communication channel offers a peak capacity of just 5 Mbps). Again, this distance may be seen as related to behavioral distance, but it should be consciously analyzed.

In any of the above dimensions, there may be varying states of *knowledge* on the part of the analyst. The confidence in any analysis that an analyst performs will be contingent on having accurate knowledge.

Integrability is thus really about the distance between the elements of each potential dependency—the amount of work needed to resolve differences at each potential dependency. For example, one component expects to push data and the other one expects to request data when it needs it, or one component sends batches of data and another component expects data to arrive record by record or field by field. These two components will have a mismatch; they will not be (easily) compatible and some work will be needed to bridge their differing expectations. These kinds of incompatibilities are well known in object-relational implementations, where they are called "impedance mismatches." In the software architecture research literature, such problems have been referred to as "architectural mismatches" [Garlan 1995]. Let us now more carefully understand this concept.

3.2 Operationalizing the Analysis of Integrability

So what does it mean to measure the integrability of an architecture? This question, like all questions surrounding the quality attribute properties of an architecture, can only be assessed in a given context, and we specify that context using scenarios [Bass 2012]. That is, there is no global integrability number that matches any conceivable future integration. Instead, we focus each integrability analysis on the predicted costs and risks associated with a scenario that characterizes some likely future integration, and we ground this analysis in explicit assumptions and a clear scope. At the highest level, our process for analyzing for integrability (or for any other quality attribute, for that matter) involves three phases: preparation, orientation, and evaluation, as shown in Figure 2: Architecture Analysis Process. In the first phase, the analyst collects the artifacts to analyze—primarily scenarios and architectural documentation. In the second phase, the analyst identifies the architectural mechanisms relevant to the satisfaction of the scenarios and locates these mechanisms in the architecture. In the third phase, the scenarios are mapped onto the architectural description to determine whether they can be satisfied and to determine the costs, risks, and tradeoffs of this satisfaction.

Figure 2: Architecture Analysis Process

The process relies critically on the collection of scenarios in Phase 1. An integrability scenario specifies a set of components $\{C_i\}$ and a system S, along with some response measures—measures of cost, duration, effort, and risk. And these measures are derived from an analysis of

the size and distance of the predicted interfaces between $\{C_i\}$ and S. It follows naturally then, that for a given S, the integrability costs and risks to satisfy some scenarios will be greater than for other scenarios due to the characteristics of S—that is, the mechanisms that the architect built into S.

To return to our analogy, creating an adapter that allows a North American plug to be plugged into a British socket will cost something. Enhancing that adapter to convert from 220 to 110 volts will add cost to the adapter. Both British and American plugs have a positive terminal, a negative terminal, and a ground terminal. Thus their interfaces are the same "size" according to a simple syntactic measure. But the semantics of the electrical system introduces additional distance in the integration tasks. Furthermore, the cost and risk of a syntax adapter are much less than the cost and risk of a semantic adapter, and some semantics (e.g., 50/60 Hz or AC versus DC) may be cost-prohibitive to adapt. As analysts it is our job to assess these costs and their attendant risks.

We now turn our attention to crafting appropriate integrability scenarios, as these will be the architectural test cases that an analyst will need to consider.

4 Integrability Scenarios

As stated in *Software Architecture in Practice*, quality attribute names themselves are of little use, as they are vague and subject to interpretation. The antidote to this vagueness is to specify quality attribute requirements as scenarios [Bass 2012]. A quality attribute scenario is simply a brief description of how a system should respond to some stimulus. Quality attribute scenarios are not use cases—they are architectural test cases. Quality attribute scenarios are useful for more than capturing requirements, such expressing growth and exploratory scenarios, but they have proven to be an effective way to define quality attribute requirements more precisely than the usual free text seen in practice.

A quality attribute scenario has six parts [Bass 2012]. The two most important parts are a *stimulus* and a *response*. The stimulus is some event that arrives at the system, either during runtime execution (e.g., an invalid message arrives 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 context and 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 than invalid messages on internal interfaces.

During requirements elicitation, we may elicit and specify the parts of a scenario in any order. We often begin with stimulus and response, as these are typically the parts of a scenario that people focus on, although the environment, source, or artifact may in fact be the initial trigger for the requirement. In any case, once the scenario is specified, we usually arrange the parts to tell a small story, as depicted in Figure 3.

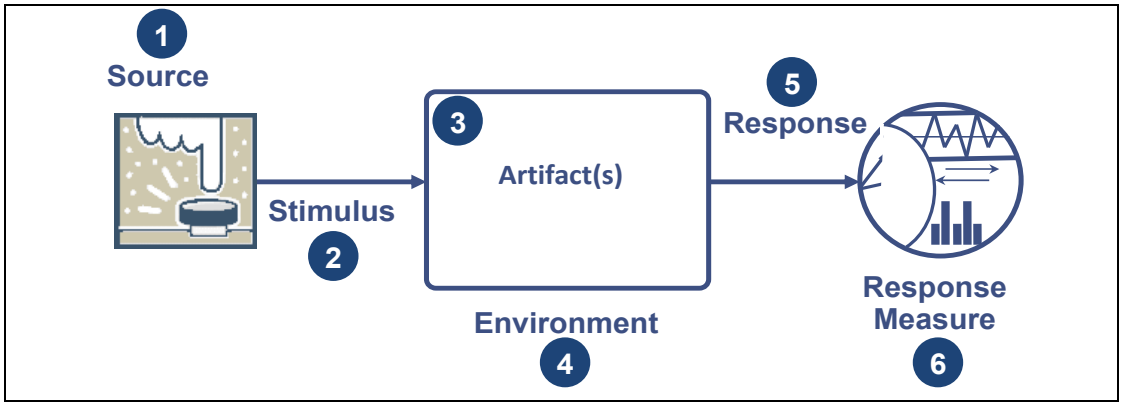

Figure 3: Six Parts of a Scenario

The degree to which an analyst can accurately analyze an architecture is directly correlated to the degree to which the scenarios are accurately specified. Below we provide a general scenario for integrability and then some example integrability scenarios derived from the general scenario. A general scenario is a system-independent scenario that allows stakeholders to communicate more effectively about quality attribute requirements and can assist stakeholders in developing concrete scenarios.

4.1 General Scenario for Integrability

There is no single scenario that specifies all of the possible measurements that could characterize a property like integrability. 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 integrability 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 integrability:

Scenario Part	Possible Type for Each Value
Source	One or more of the following: • mission/system stakeholder • component marketplace • component vendor
Stimulus	One of the following: • add new component • integrate new version of existing component • integrate existing components together in a new way

Scenario Part	Possible Type for Each Value
Artifact	One of the following: • entire system • specific set of components • component metadata • component configuration
Environment	One of the following: • development • integration • deployment • runtime
Response	One or more of the following: • changes are {completed, integrated, tested} • components in the new configuration are successful in correctly (syntactically and semantically) exchanging information • components in the new configuration do not violate any resource limits
Response Measure	Cost, in terms of one or more of the following: • # components changed • % code changed • LOC changed • effort • money Calendar time Effects on other quality attribute response measures (to capture allowable tradeoffs)

Note that the response measures are not specified in terms of the size and distance between *S* and $\{C_i\}$. As stated in Section 3.1, there will be a positive correlation between integrability complexity—which we gauge using size and distance—and the cost or risk of actually completing required integration tasks. Using cost and risk in scenarios keeps the focus on the end goals (e.g., business value) instead of technical details.

4.2 Example Scenarios for Integrability

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 use an easy-to-understand "typical" system. In practice, the analyst should choose values that are as precise as possible in the context of the system.

4.2.1 Example Scenario 1: New Software Component

This scenario describes the common situation in which an organization wants to be able to integrate new components that would be valuable additions to an already-deployed system *S* when they become available in the marketplace. Each future new component *C* would need to be integrated into *S*.

A new data filtering component will become available in the component marketplace. The new component will be integrated and deployed in 1 month, with no more than 1 person-month of effort. 100% of necessary messages will be correctly processed, and 100% of unnecessary messages will be correctly ignored.

From this description we can map the six parts of the scenario using the categories from the general scenario above:

Stimulus: add new data filtering component *C*

Source of stimulus: mission need

Artifact: <<specific set of components>>

Environment: system has been deployed

Response: component is fully integrated

Response measure:

- deployed in 1 month with no more than 1 person-month of effort
- 100% of necessary messages are correctly processed, and 100% of unnecessary messages are correctly ignored

Note that in this scenario—as in many scenarios—multiple response measures are specified. In this case the response measures specify elapsed time (deployed in 1 month), total effort (no more than 1 person-month), and correctness (100% of necessary messages are correctly processed, and 100% of unnecessary messages are correctly ignored) criteria. The choice of response measures is critical because these become tests that an analyst can apply when analyzing the architecture of *S*. An analyst would need to determine if the architecture decisions have provided sufficient support to make it likely that the desired response measures will actually be achieved.

It's also worth noting that the scoping of *C* to a data filtering component in the scenario, as opposed to a generic component, provides important information for the analyst. An analyst can make more reasonable assumptions about the number (size) of dependencies between *S* and a component that performs data filtering; likewise, the analyst can make more reasonable assumptions about the distance measures from Section 3.1 given the typical semantics of data filtering components. While these assumptions may not be perfect, they increase the confidence of an integrability analysis compared to a generic case in which few if any assumptions could be made.

4.2.2 Example Scenario 2: Modified Software Component

This scenario describes an even more common scenario than Example 1: the situation in which an organization wants to be able to integrate new releases of an externally maintained component that is already integrated in *S*. Because new releases commonly add new features and fix defects and vulnerabilities, re-integrating a component can be vital to the success of *S*.

Stimulus: updated version of an existing software component *C* needs to be integrated

Source of stimulus: vendor/component marketplace

Artifact: *S* and a specific *C*

Environment: system has been deployed, and all unit tests have been completed on *C*

Response: component integration test is completed, and the component is fully integrated

Response measure:

- within 1 person-day of effort
- all integration tests pass, and no resource limits are violated

In this case we just specify two response measures: one for effort and one for correctness. Note that the correctness response measure includes ensuring that no resource limits are violated by the newly integrated component. In practice, ensuring this response measure may require prototyping. Providing more specificity for *C* in a scenario, such as identifying which component in the system is expected to evolve independently, furnishes more knowledge for an analyst to work with. For example, a real-time operating system and an encryption package may differ significantly in terms of how tightly coupled they are with *S*.

4.2.3 Example Scenario 3: Using Existing Components to Meet New Needs

This scenario describes the case in which an organization wants to be able to rapidly integrate a set of existing components $\{C_i\}$ in a novel way in an established *S* to meet some new use case or mission need. *S* is thus the existing architecture, and the analyst's goal is to determine the degree to which *S* supports the achievement of the response goals. In this case, it is expected that such an integration would be relatively low cost and low risk, as indicated by the response measures.

Stimulus: new use case, mission need

Source of stimulus: stakeholder community

Artifact: *S* and $\{C_i\}$

Environment: assessment has been made that 100% of the new need can be satisfied with existing components

Response: new use case is fully operational by composing existing components

Response measure:

- within 2 person-days of elapsed time and less than 1 person-day of code changes to *S* or $\{C_i\}$
- no changes to existing system quality attribute response measures

The first response measure specifies that the cost of the change is expected to be low—just a person-day of effort. The second response measure specifies that this change will cause no collateral damage to the satisfaction of existing system quality attribute requirements.

4.2.4 Example Scenario 4: Integrating Version of Existing Component with New States/Modes

This scenario is similar in intent to the example in Scenario 2. In this case, however, the organization wants to support integration of a new release of *C* that has undergone significant semantic changes. Specifically, *S* should accommodate a *C′* that includes new states and/or modes.

Stimulus: component *C* is upgraded to *C′* (e.g., a new algorithm) with new states/modes

Source of stimulus: mission need

Artifact: *S* and *C*

Environment: system is deployed with a current version of the component, and the new component *C′* is fully unit tested

Response: new component version *C′* is integrated

Response measure: within 6 person-months of effort

In this scenario, the response measure would lead the analyst to believe that component *C* was not well decoupled from the rest of the system in terms of its behavioral semantics, hence the effort required to accommodate *C′*. And there is no additional response measure specifying that this change will have no ripple effects, and hence require no changes, to other system components. Scenarios such as this one help test the limits of what was anticipated by the architects of *S*.

From the analyst's perspective, however, the analysis is the same. The analyst must look at the size and distance between *S* and the anticipated *C′* to evaluate the architecture's integrability for this scenario; whether the result is expected to be large or small doesn't affect how the analysis is performed.

5 Mechanisms for Achieving Integrability

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."

But before we delve into a discussion of these mechanisms, we need to reiterate a point that we made in the introduction to this document: these architectural mechanisms are not, by themselves, guarantees of system success. They are important preconditions for success, of course—the technical foundations upon which the system is built—but without appropriate project governance and without a disciplined approach to all aspects of the software development lifecycle, these mechanisms will accomplish little. Having established that, let us now discuss and provide examples of two important kinds of architectural design mechanisms: *tactics* and *patterns*.

5.1 Tactics

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 integrability tactics found in Figure 4 are to reduce the costs and risks of adding new components, reintegrating changed components, and integrating sets of components together to fulfill evolutionary requirements. The tactics achieve these goals by reducing the potential dependencies between components or by reducing the expected distance between components.

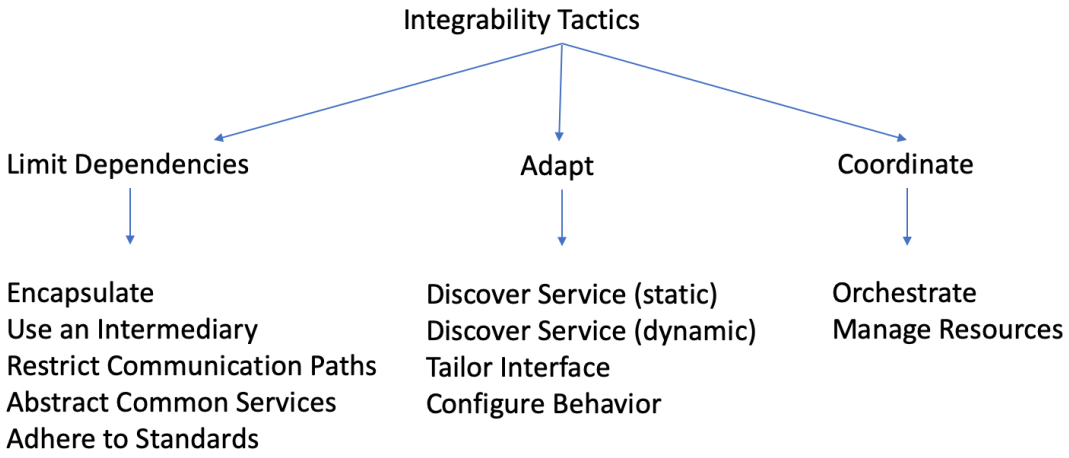

Figure 4: Integrability Tactics

These tactics are known to influence the responses (and hence the costs) in the general scenario for integrability (e.g., number of components changed, percent of code changed, effort, calendar time). The tactic descriptions presented below are inspired by and are derived in part from the third edition of *Software Architecture in Practice* [Bass 2012]. Table 1 summarizes the tactics

presented in this section, and how each relates to the principles of size and distance presented in Section 3.1.

Table 1: Summary of Integrability Tactics and How They Address the Principles of Size and Distance That Present Integrability Challenges

Tactic	Size	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Encapsulate	+	+	+	+		
Use an intermediary	*	*	*	*	*	
Restrict communication paths	+					
Abstract common services	+					
Adhere to standards	*	*	*	*	*	
Discover service (static)	+					
Discover service (dynamic)	+					
Tailor interface		+	+	*		
Configure behavior		*	*	*	*	
Orchestrate	+					
Manage resources						+

Note: A plus sign indicates that the tactic positively addresses challenges with the principle, while an asterisk indicates that the tactic might positively or negatively address the challenge, depending on the realization of the tactic.

Encapsulate

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

Encapsulation introduces an explicit interface to a module (a kind of component *C*). 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, be valuable in environments where systems use many sensors of the same type that are from many different manufacturers, each with their own device drivers.

These sensors may have different accuracy or timing properties needed for different missions. Architects also use wrappers or adapters to encapsulate the differences between these devices and provide the same interface so that the same components can be composed together for different missions.

Encapsulation may also hide interfaces that are not relevant for a particular integration task. An example is a library that used by a service can be completely hidden from all consumers and changed without these changes propagating to the consumers.

Encapsulation, then, can reduce the syntactic and data or behavior semantic distances between *C* and *S*, and may also effectively reduce the size of the interface between them.

Use an Intermediary

Intermediaries are used for breaking dependencies between a set of components C_i or between C_i and the system *S*. 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. Other intermediaries, like data transformers and protocol translators, resolve forms of syntactic and data semantic distance.

Determining the specific benefits of a particular intermediary requires knowledge of what the intermediary actually does. An analyst needs to determine whether the intermediary reduces the size of dependencies between a component and the system and which dimensions of distance, if any, it addresses.

Intermediaries are often introduced during integration to resolve specific dependencies, but they can also be included in an architecture to promote integrability with respect to anticipated scenarios. Inclusion of a communication intermediary like a publish-subscribe bus in an architecture and restricting communication paths to and from sensors to this bus is an example of using an intermediary with the goal of promoting integrability of sensors.

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 authorization (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 integrability tactic primarily reduces the size of potential dependencies between components C_i and the system *S*, essentially reducing the options to those provided by the allowed communication paths. A common example is to restrict all communication between C_i and *S* to a specific path, whose size is fixed. A restricted communication path is often paired with other tactics such as adherence to standards that also help with dimensions of distance.

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. This 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 integrability, this means that future components can be integrated with a single abstraction rather than separately integrated with each of the specific modules. When the tactic is combined with an intermediary, it can also normalize syntactic and semantic variations among the specific modules.

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

This integrability tactic primarily reduces the size of potential dependencies between components C_i and the system S, essentially restricting access to the specific modules to occur only through the abstract service. An abstract service is often paired with an intermediary that may perform processing to hide syntactic and data semantic differences among specific modules.

Adhere to Standards

Standardization in system implementations is a primary enabler of integrability and interoperability, both across platforms and vendors. Standards vary considerably in scope of what they prescribe. Some focus on defining syntax and data semantics. Others include richer descriptions, such as those describing protocols that include behavioral and temporal semantics.

Standards similarly vary in their scope of applicability or adoption. For example, standards published through widely recognized standards organizations like the Institute of Electrical and Electronics Engineers (IEEE), the International Organization for Standardization (ISO), and the Object Management Group (OMG) are more likely to be implemented consistently by different organizations. Conventions that are local to an organization, particularly if well documented and enforced, can provide similar benefits as "local standards," though with less expectation of benefits when integrating components from outside the local standard's sphere of adoption.

Adopting (or even defining) a standard can be an effective integrability tactic, though its effectiveness is limited to benefits in the dimensions of difference addressed in the standard and how likely it is that future component suppliers will conform to the standard. Restricting communication with a system S to require use of the standard often reduces the size of potential dependencies. Depending on what is defined in a standard, it may also address syntactic, data semantic, behavioral semantic, and temporal dimensions of distance.

Discover Service (Static)

Locate a service through searching a known directory service. The service can be located by type of service, name, location, quality-of-service description (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. The advantages of static service discovery is that there is a relatively 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 VMs where IPs 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.

Good descriptions can reduce integration problems when integrating a new component into an existing system because they expose a repository of facts about different components in the system or candidate components for integration and they speed information discovery. Including a mechanism for static service discovery in an architecture is a governance mechanism that can assist integrability by setting an expectation of minimal information that will be available for all components. 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 uncommon.

Using information that is not published, such as the identity of an interacting component or unpublished services, is not guaranteed to work if a different service with the same description matches in the future. Avoiding unpublished information can reduce the size of potential dependencies.

Note that the presence of this mechanism does not reduce the distance along any dimension, so it does not improve integrability as much as other mechanisms. Hence it should be used in conjunction with other integrability tactics. It does, however, allow engineers to make future decisions (e.g., which of three potential components is most compatible with a system) more quickly and with greater confidence. It also provides more information to help analysts reason about integrability scenarios.

Discover Service (Dynamic)

Dynamic discovery enables the discovery of service providers at runtime and 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.

From an integrability perspective, inclusion of a dynamic discovery capability sets the expectation that the system *S* 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).

Use of a dynamic discovery capability also sets an expectation that a future component should only rely on published service descriptions. Using information that is not published, such as identity of an interacting component or unpublished services, is not guaranteed to work if a different service with the same description matches in the future. Avoiding unpublished information can reduce the size of potential dependencies.

This tactic, by itself, does not reduce the distance along any dimension. However, it is commonly paired with tactics like adherence to standards that do allow future components to choose from a set option, potentially choosing an option with less distance along one or more of the dimensions of distance.

Tailor Interface

Tailor interface is a tactic that adds capabilities to, or removes them from, an interface without changing the API or implementation. Capabilities such as translation, buffering, and data smoothing can be added to an interface. An example of removing capabilities is hiding particular functions or parameters from untrusted users. A common dynamic example of this tactic is intercepting filters that add functionality such as validating data to help prevent attacks like SQL injection or translating between data formats. Another example is using techniques from aspect-oriented programming that weave in functionality for pre- and post-processing at compile time.

This tactic allows functionality that is needed by many services to be added or removed based on context and managed independently. It also allows services with syntactic differences to interoperate without modification to either service.

The tailor interface tactic is typically applied during integration; however, designing an architecture that facilitates interface tailoring can support integrability. Interface tailoring is commonly used to resolve syntactic and data semantic distance during integration. Interface tailoring can also be used to resolve some forms of behavioral semantic distance, though it can be more complex to do (e.g., maintaining complex state to accommodate protocol differences) and is perhaps more accurately categorized as introducing an intermediary.

Configure Behavior

This tactic is used by software components that are implemented to be configurable in prescribed ways that allow them to more easily interact with a range of different components. Behavior of a component can be reconfigured during build (recompile with a different flag), during system initialization (read a configuration file or fetch data from a database), or during runtime (specify a protocol version as part of your requests). A simple example is configuring a component to support different versions of a standard on its interfaces. Multiple options provide a greater chance for a match between the assumptions of *S* and a future *C*.

Plug-ins are a common form of configurable behavior. Plug-ins are usually bound during system initialization or during runtime. Plug-ins typically interact with the rest of the system through an intermediary, and each plug-in is typically expected to implement an abstract service interface in addition to any custom services that it provides. Plug-ins can help with integrability by providing

a system with a way to configure which plug-in interacts with a new component based on its compatibility.

Building configurable behavior into portions of *S* is an integrability tactic that allows *S* to support a wider range of potential *C*s. Determining the specific benefits of configurable behavior requires knowledge of what aspects of behavior can actually be configured. This tactic can potentially address syntactic, data semantic, behavioral semantic, and temporal dimensions of distance.

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.

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. Integrability cost measures are reduced when orchestration is included in an architecture in a way that supports the kinds of services that are likely to be integrated in the future. This tactic allows future integration activities to focus on integration with the orchestration mechanism instead of point-to-point integration with multiple components.

Orchestration primarily reduces the number (size) of dependencies with behavioral semantic and temporal distances between the system and new components by centralizing those dependencies at the orchestration mechanism. Syntactic and data semantic distance may also be reduced if the orchestration mechanism implements tactics like adherence to standards.

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 ARINC 653 for space and time partitioning.

A resource manager is an integrability tactic that primarily reduces the resource distance between a system *S* and a component *C* by clearly exposing resource requirements and managing their common use.

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 also provides more confidence in the completeness of the analysis.

Table 2 shows which integrability tactics are used to build each of the patterns described. Then we provide a brief description of each pattern, explain how the pattern promotes integrability scenarios, and identify other quality attributes whose scenarios are often 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 integrability 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 integrability patterns—SOA, Publish-Subscribe, Adapter, and Broker—and to show how analysts can break patterns down into tactics that help them analyze quality attribute scenarios. Table 2 maps the SOA, Publish-Subscribe, Adapter, and Broker patterns to the integrability tactics described in Section 5.1. The patterns themselves are described in the following sub-sections.

Table 2: Integrability Tactics Mapped to Four Common Patterns

Tactic	SOA	Publish-Subscribe	Adapter	Broker
Encapsulate	×	×	×	×
Use intermediary	×	×	×	×

Tactic	SOA	Publish-Subscribe	Adapter	Broker
Restrict communication paths	×	×	×	×
Abstract common services	×	×	×	×
Adhere to standards	×			
Discover service (static)	×			
Discover service (dynamic)	×	×		
Tailor interface	×		×	×
Configure behavior	×	×		
Orchestrate	x			
Manage resources	×			

5.2.1 Service-Oriented Architecture

SOAs are built as loosely coupled systems consisting of infrastructure components and applications that are orchestrated to provide defined capabilities. Typically, SOAs are designed using standard components and are tailored to specific mission needs. The key components of a standard SOA design are services, the enterprise service bus, the service registry and repository, service orchestration, event manager, and data management. Each of these components is tailored to support mission-specific requirements. For example, to meet its mission requirements a system may require unique data and service models or may use unique security services.

Solutions that use a service-oriented approach are intended to satisfy business or mission goals that include controlling lifetime system costs. Other goals might be easy and flexible integration with legacy systems (interoperability), streamlined business processes (maintainability), and agility to handle rapidly changing business processes (modifiability, integrability). Quality attributes such as interoperability, integrability, modifiability, and maintainability are the primary architectural drivers addressed by SOA adoption, and they are achieved by adhering to a set of design principles for service-oriented systems that are described below. However, there are other important quality attributes—such as availability, reliability, security, and performance—whose goals have to be addressed in a complete system design. In addition, an architectural decision that promotes scenarios in the first set of quality attributes may negatively impact scenarios in the second set of quality attributes. In this section, we discuss how SOA supports integrability scenarios and the tradeoffs that can negatively impact scenarios for other quality attributes [Bianco 2011, Josuttis 2007].

It should be noted that SOA has a few important differences from "microservices." The microservice pattern advocates for a bare minimum of centralized management compared to the strong governance approaches recommended for SOA. SOA services are focused on business reusability and are generally coarse grained while microservices are finer grained, emphasizing bounded contexts.

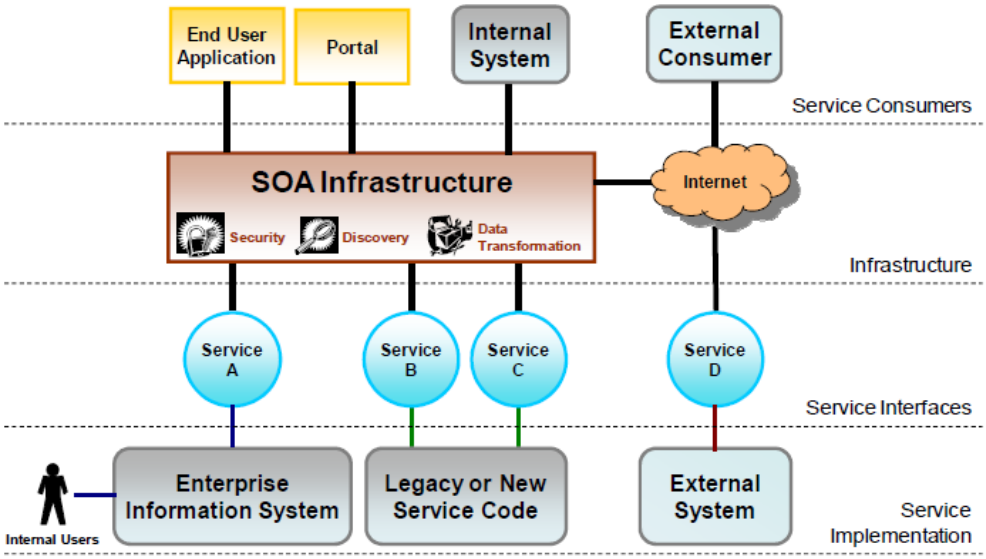

Figure 5: High-Level SOA Notional View [Bianco 2011]

The principles listed below apply to the full architecture of a service-oriented system:

- Standardization of data formats, protocols, interface conventions, policies, and constraints promotes interoperability.
- Loose coupling of service providers and consumers allows implementations to change independently.
- Creating services that are self-contained units of functionality allows use in multiple business processes (reusability).
- Composability allows business processes to change rapidly when the business environment changes.
- Discoverability provides an environment in which services are published in a known place that is accessible to service consumers.

Support for integrability:

- Common services that translate among data formats and protocols provide more options for integrating future components, increasing the chance of component compatibility with the system and resolving some forms of syntactic and data semantic differences at runtime.
- A service registry and enterprise service bus defer service binding and restrict communication to known message content, reducing the size of potential dependencies between a system and a new component.
- Service orchestration localizes some temporal and behavioral semantic issues, reducing the corresponding distances between a system and a new component.
- Decoupling service interfaces and service implementations (abstracting services) allows systems to configure their behavior to support multiple interfaces or present a simpler interface to future components. This decoupling also allows different product variants to be deployed, offering different features or quality of service in different deployments.

- Particularly when used with stateless services, an enterprise service bus can manage the number of service instances (resource management) to provide solutions that scale to meet the needs of future components.

Tradeoffs:

- SOA systems can be more complex, which can negatively affect modifiability and maintainability scenarios. The ability to dynamically determine message paths adds layers of processing logic and can introduce service compositions that are extremely complex and that require careful management and constraints in the business rules. Maintaining multiple versions of services and masking interface changes also increase complexity.
- SOA infrastructure provides a lot of services that can add latency to operations, which can negatively affect performance in terms of latency and operations / unit time. Dynamically determining message paths adds overhead. Using a complex set of rules that must be executed with common requests may produce significant variation in latency.
- SOA systems vary at runtime, resulting in a large number of configurations to understand. This can negatively affect testability and reliability scenarios. For example, dynamically determining message paths results in a large set of integrations for testing strategies. Defects can be injected when rules are written and deployed that affect configurations. An example would be a rule that routed a request to a service that wrapped a low-fidelity analysis engine when the requesting application needed a higher fidelity analysis.

5.2.2 Broker

Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services. The Broker pattern structures distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions. The Broker pattern promotes building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, and results in greater flexibility, maintainability, and composability.

The Broker architectural pattern comprises six types of participating components: clients, servers, brokers, bridges, client-side proxies, and server-side proxies [Bachmann 2007, Buschmann 2007].

Support for integrability:

- Components are able to access services provided by others through remote, location transparent service invocations, thus reducing point-to-point integrations. This reduces coupling by reducing the size and syntactic distance of the interactions.
- Designers can exchange, add, or remove components at runtime, potentially reducing the time and effort of integration.
- This pattern hides system- and implementation-specific details from the consumers of components and services. This reduces coupling—primarily syntactic and semantic distance—

and makes it less likely that consumers will need to be changed when the components and services they use are changed.

Tradeoffs:

- The Broker pattern negatively impacts performance (latency) by introducing additional calls and overhead.

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.

Support for integrability:

- 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 Adapters

An adapter creates an intermediary that wraps an underlying component and presents a new interface to the rest of the system. The adapter usually restricts communication with the component it wraps such that all interactions go through the adapter. When another component in the system interacts with the adapter, it transforms this interaction as needed and passes the request on to the wrapped component.

Adapters can be applied to solve different problems. One use is to create a common abstract service (e.g., a common sensor service) and then to use an adapter for each concrete service (e.g., a specific sensor) that implements the abstract service interface and passes appropriately transformed requests to the concrete service. Another use is to bridge differences between two specific components. The differences could be syntactic (e.g., calling for an adaptor to translate data), data semantic (e.g., translating based on different units of measure), or behavioral semantic (e.g., wrapping a component that expects to be polled with a thread that polls and broadcasts changes, making the component appear to implement a push control model).

Support for integrability:

- Adapters are custom things and can be written to resolve many different forms of differences along the dimensions described in Section 3.1.
- Adaptors for new components can be implemented during integration, allowing late binding of whatever specific behavior needs to be adapted.
- Adaptors for existing components can be included in an architecture, usually paired with a service abstraction, to reduce the number of variations (size) that future components may have to accommodate.

Tradeoffs:

- Adapters add functionality to perform their work, which can negatively affect performance (latency). The degree of the effect depends on how much work they do and how expensive that work is.
- Adapters negatively impact performance (latency) by introducing additional calls and overhead.

5.2.5 Analyzing Patterns

Note that publish-subscribe, broker, and most other patterns could be used to emulate other mechanisms. For example, if all publishers in a system “broadcast” messages that were specific to a single recipient, then all of the power and generality of publish-subscribe is lost. Further, if that recipient then responded back to the originator of the message (and to no other component), then the publish-subscribe mechanism would merely be replicating a call-return style of component interaction.

Thus, when analyzing the use of a pattern, we must distinguish between the intent of that pattern and all of its potential instantiations (which might in fact change or even undermine that intent, its pros and cons, and its tradeoffs). When we discuss patterns here, we focus on the intent of the pattern and not the myriad of ways that the pattern might be misapplied, undermined, or, at the very least, used in a way that is not consistent with its original intention. In such cases the analysis must take into consideration the specific instantiation of the pattern.

6 Analyzing for Integrability

An analyst's job is to judge the appropriateness of the mechanisms built into the architecture of a system *S*, in light of the anticipated set of components $\{C_i\}$ that will need to be integrated in the future. And as stated above, "appropriateness" is really a function of the risks and costs of the anticipated integrations. Analysts can specify these potential, or anticipated, integrations using scenarios, as we exemplified above, and for consistency and repeatability they can guide stakeholders to derive those scenarios from the Integrability General Scenario.

Analyzing for integrability 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 the integration of an arbitrary new component. 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 integrability.

On the other hand, if the analysts have received a defined and documented 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 metrics to measure the as-designed level of dependency in the system, in chosen subsets of the system, or between selected parts of the system.

The point is that there are 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 Integrability

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

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

6.1 Tactics-Based Questionnaires

Architectural tactics have been presented thus far as design primitives, following Bass [2012] and 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, they represent a taxonomy of the entire design space for integrability.

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.

For example, consider the list of integrability tactics-inspired questions presented in Table 4. The analyst asks each question and records the answers in the table.

Table 4: Example Tactics-Based Integrability Questions

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Limit Dependencies	Does the system provide a means to *encapsulate interfaces*, that is, introduce an explicit interface (an API and its associated responsibilities) that wraps and potentially adapts a component?				
	Does the system *make use of intermediaries* for breaking dependencies between components, for example, removing a data producer's knowledge of its consumers?				
	Does the system *abstract common services*, providing a general, abstract interface for similar services?				
	Does the system provide a means to *restrict communication paths* between components?				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system *adhere to standards* in terms of how components interact and share information with each other?				
Adapt	Does the system provide the ability to statically (i.e., at compile time) *tailor interfaces*, that is, the ability to add or remove capabilities of a component's interface without changing its API or implementation?				
	Does the system *provide a dynamic discovery* service, enabling the binding between a service consumer and a service at runtime?				
	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 means to *configure the behavior* of components at build, initialization, or runtime?				
Coordinate	Does the system provide a *resource manager* that governs access to computing resources?				
	Does the system include an *orchestration mechanism* that coordinates and manages the invocation of components so that can be ignorant of each other?				

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

- whether there are any obvious risks in the use (or non-use) 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).
- the specific design decisions made to realize the tactic and where in the codebase the implementation (realization) may be found. This is useful for auditing and architecture reconstruction purposes.

- any rationale or assumptions made in the realization of this tactic.

These questionnaires can be used by an analyst who poses each question, in turn, to the architect and records the responses, as a means of conducting an architecture analysis. To use these questionnaires, simply 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 italicized.
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/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 that 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 (including a decision to *not* use a tactic). Briefly explain the implications of this decision. For example, explain the rationale and implications of the decision in terms of the effort on cost, schedule, evolution, and so forth.

While this interview-based approach might sound simplistic, it can actually 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 shown in Table 4 forces the architect to do just that. And 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 Integrability

As presented in Bass [2012], 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 of the analyst is to assess whether every important decision has been considered.

Some of these design decisions might be trivial or highly constrained. For example, an architect may have no choice of technology decisions to make if he is required to implement the software on a prespecified platform over which he has little or no control. Or for some applications, the data model might be trivial. But for other categories of design decisions, the architect 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 integrability is presented below.

Category	**Checklist**
Allocation of Responsibilities	Consider which integrations are likely to occur through consideration of changes in technical, legal, and mission forces. Do the following for each potential integration: • Consider the responsibilities that would need to be integrated, modified, or deleted to make the change. • Consider what *other* responsibilities are impacted by the 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 whether the coordination models of S and $\{C_i\}$ are compatible and, if they are incompatible, how much work is required to bridge the differences. Consider which parts of S—devices, protocols, and communication paths—used for coordination 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 integration. 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 if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, assess whether the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. Do the following for each potential change or category of change:

- 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 if 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 integration. 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 the integration 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 after the integration are sufficient to meet the system requirements.
Binding Time	Do the following for each change or category of change: • Consider the latest time at which the integration will need to be made. • 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 does not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	Consider what integrations are made easier or harder by the technology choices. Will the technology choices help to make, test, and deploy integrations? How easy is it 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 integrations. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

6.3 Coupling Metrics

Coupling metrics can be used to analyze the strength of dependencies between two entities. While they have traditionally been used to measure the strengths of dependencies between modules of source code, the concept of coupling applies equally well to design concepts, hardware, people, and so forth. Essentially, coupling tells you how tightly bound together two things are, and this is correlated with how easy it is to change just one of those things and not have that change ripple to the other thing. This is important as higher coupling tends to correlate strongly with higher costs of integration, modification, and testing.

We can, therefore, measure the coupling among elements in a design. This is a good thing: it means that even before we have implemented the system we can begin to understand the characteristics of that system. This is why we create and analyze models. If we have a system that has already been implemented and for which we have access to the source code, executables, revision history, or adequate documentation, the analyst can consider several additional options for integrability analysis:

1. Analyze the system with respect to system-wide coupling metrics. Such a metric provides a single number that represents how tightly (or loosely) coupled the system is overall. While this number is useful—for example, to compare systems or to see how a system is evolving over time—it does not offer insight into the specific challenges of integrating a single component. On the positive side, a metric can be tracked over time, to give management insight into the overall level of coupling in a system, whether it is increasing or decreasing, and whether it crosses a threshold that represents a risk. To employ a medical analogy, this is similar to how a doctor monitors a patient's blood pressure or cholesterol level over time, and when it exceeds some threshold a medical intervention is indicated. Furthermore, this kind of application of a metric can be applied to any nontrivial subsystem, giving finer-grained insights.
2. Analyze a specific target component with respect to how tightly it is coupled to the rest of the system. This gives insight into the level of difficulty of changing this coupling (e.g., replacing this target component with a new one that performs approximately the same function). Assess a target component's level of coupling with the rest of the system in three ways: structural coupling, historical coupling, or dynamic coupling.
    a. To assess syntactic (structural) coupling, examine the software architecture artifacts (or reverse-engineer the system's source code if the architecture document does not exist or is not up to date) to count the number of connections from the target component to other system components. This gives a purely syntactic measure of coupling where, presumably, a higher number is worse. In addition, consider the strength of the coupling (for example, counting the number of parameters in an API or counting how many methods/messages or data types are passed).
    b. To assess historical coupling, review a project's revision history, tracking all co-commits (commits where two or more components are committed together). Each of those co-commits is a clue pointing toward some dependency between pairs of components. Low numbers of co-commits are typically uninteresting, but if two components change

together frequently, this indicates that they are somehow coupled. If such pairs of components also share some syntactic relation (for example, component A calls component B, or component A inherits from component B), then this co-commit behavior is unremarkable. However, if a pair of highly co-committed components does not share any syntactic relationship, then their coupling must be dynamic or semantic (data or behavioral).

c. To assess dynamic coupling, either run the system with a monitoring tool that reports calling behavior between system components or instrument the system to report such dynamic relationships. For example, if a pair of components are coupled together because one publishes an intent and the other subscribes to that intent, this dependency will not be detectable via a syntactic analysis such as that achieved from reverse-engineering the code. To perceive this dependency, the analyst will likely need to instrument the code.

Note that none of these three forms of coupling will provide complete, authoritative insight into (data or behavioral) semantic or temporal coupling, although historical and dynamic coupling should be strongly correlated with semantic and temporal coupling. Semantic and temporal coupling (and differences) between components are more diverse and often more subtle. These typically cannot be mechanically assessed. For example,

1. the meaning of data that is passed could differ, such as by using different units of measurement or different terms for the same concepts
2. the interfaces could assume different interaction protocols, such as one assuming that data will be pushed by a provider while the other assumes data will be pulled by a consumer
3. the interfaces could assume different temporal relations, such as different sampling rates

To overcome these information gaps, the analyst may need to consider if documentation exists for the involved components and if that documentation adequately addresses such questions. Or failing that, the components and their interfaces can be reverse engineered. Finally, to answer such questions, the analyst should contact the component-supplier engineering team to request information.

To assess a system's (or subsystem's) overall level of integrability, the analyst should employ a system-wide coupling measure. Examples of such measures are Propagation Cost (PC) [MacCormack 2006] and Decoupling Level (DL) [Mo 2016]. PC measures how tightly the elements of a system are coupled together: the more tightly coupled, the higher the score. DL measures how well components in a system are decoupled from each other: the more decoupled, the higher the score. Intuitively these metrics measure the same phenomena, but with one important difference: PC is sensitive to the size of the system being measured. The greater the number of files in the system, the smaller the PC. DL, on the other hand, is size-independent.

The advantage of metrics such as these is that they can be used to analyze architectural representations. The analyst need not wait for concrete implementation to begin analysis. However, these metrics, like most measures in software engineering, are "garbage in/garbage out." In other words, these metrics can only measure the coupling that a system (or system design) has explicitly

acknowledged and manifested. In the book *Documenting Software Architectures: Views and Beyond*, the notion of "interface" between components was explicitly made broad. In fact, that book spoke of "relations" between components, rather than interfaces. Relations were seen as much more than just APIs. If component A depends on or interacts with component B in any way, then there is a relationship between them. Consider the following examples:

- Two components might share the same processor or memory budget.
- They may collaboratively control a peripheral device; hence a change to one might affect the other.
- These components might have a timing dependency (A must complete before B can start, for example).
- They might share knowledge of a file format or key length, but this information is not manifested in any interface.

Such implicit dependencies between these files is a *relationship* but will not show up in any purely structural analysis of the design. These kinds of relationships—kinds of coupling—are typically not evident from a structural analysis.

In addition, coupling metrics only measure the *potential* for changes to ripple. If a set of components is very stable and seldom changes, or if the components have been designed so that the areas of change are completely encapsulated by their chosen abstractions and so the interfaces with other components never change, then little cost will accrue to this coupling.

So purely structural coupling metrics, while they reveal highly important information about the ability of a system to withstand changes in general, do miss some important types of dependencies, and they do not reflect the strengths of dependencies. However, there is some help. When analyzing a system for which the analyst already has some data, in the form of a revision history, she can conduct additional analyses that will shed light on integrability. If a revision history exists, the analyst can mine this history to derive a predicted co-change frequency for every pair of components. This is simply a ratio: the number of times that components A and B changed together over the number of times that they changed independently. The higher the number, the more highly coupled A and B are.

Revision history can also reveal implicit dependencies, often called "modularity violations" [Wong 2011]. Modularity violations are cases where groups of files change together, as considered from the project's revision history, but they have no structural relationships. This often highlights groups of files with implicit dependencies. Such dependencies deserve to be highlighted because they represent a form of coupling that project members typically do not understand or document. And these modularity violations are correlated with increased bug rates, change rates, and costs of change.

Fortunately, the definitions and implementations of PC and DL have been extended to include implicit coupling, so such forms of coupling can be captured and analyzed mechanically where revision history information exists.

We have now detailed three opportunities for extracting and analyzing coupling information between files:

1. at design time, where structural information (potentially annotated and augmented with other information about dependencies) can be analyzed
2. during maintenance, where the prior structural information can be augmented by information extracted from the project's revision history
3. at runtime, where the interaction behavior of components or their behavior with respect to usage of some shared resource can be tracked

7 Playbook for an Architecture Analysis on Integrability

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

The process has three phases and seven steps. The Preparation phase gathers the artifacts needed to perform the analysis and evaluation. The Orientation phase uses the information in the artifacts to understand the architecture approach to satisfying the quality attribute requirement. The process ends with the Evaluation phase, when the analyst applies his understanding of the requirement and architecture solution approach to make judgments about that approach. 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 analyst might identify missing artifacts during the Preparation phase and missing or incomplete information within those artifacts during the Orientation Phase. At the end of each step in the Preparation and Orientation phases, the analyst must decide whether 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 and the degree of confidence in the results. Early in the development lifecycle, lower confidence may be acceptable and the analyst can work with lower quality artifacts and simpler analyses, as suggested in Table 3. Later in the lifecycle, the analyst needs higher confidence and therefore higher quality artifacts and more and deeper analyses.

7.1 Step 1–Collect artifacts

In this step, the analyst collects the artifacts that she will need to perform the analysis. These include quality attribute requirements and architecture documentation.

The first artifact the analyst needs is the integrability requirement that she wants to validate. The requirement must be stated so that it is measurable, for example, as a quality attribute scenario as discussed above. Let's use one of the example scenarios from the earlier section, where we have specified the Artifact as "signal processing pipeline":

Scenario Part	Value
Source	mission need
Stimulus	add new data filtering component
Artifact	signal processing pipeline
Environment	system has been deployed
Response	component is fully integrated
Response measure	deployed in 1 month with no more than 1 person-month of effort 100% of necessary messages are correctly processed, and 100% of unnecessary messages are correctly ignored

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

Finally, the analyst needs architecture documentation. Early in the architecture development lifecycle, the documentation may be just a list of mechanisms mapped to quality attribute requirements, perhaps identifying tradeoffs. As the architecture is refined, partial models or structural diagrams become available, accompanied by information about key interfaces, behaviors and interactions, and rationale that provides a deeper link between the architecture decisions and quality attribute requirements. When the architecture development iteration is finished, then the documentation should include complete models or structural diagrams, along with 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 discussion of rationale, that can provide unambiguous identification of the mechanisms used to satisfy a quality attribute requirement. Other activities include looking at the structural and behavior diagrams or models and recognizing architecture patterns. Naming of architecture elements may indicate the mechanism being used. The analyst may need to use all of these to identify the mechanism or mechanisms that are being used to satisfy the integrability requirement. Frequently, two or more mechanisms are needed to satisfy a requirement. If the analyst has access to the architect(s), this is an excellent time to use the tactics-based questionnaires, as described in Section 6.1. In a short period of time, the analyst can enumerate all of the relevant mechanisms chosen (and not chosen).

For the example requirement above, "add a new data filtering component," let's say that the project is part way through the architecture development and has sketches of structural diagrams and an outline of the rationale discussion. The rationale states that a *pipe and filter* mechanism is being used to satisfy this integrability requirement, along with a *configuration file* mechanism to define the topology of the Signal Processing Pipeline.

The analyst performs a quick sanity check to decide if the referenced mechanisms are likely to contribute to satisfying the integrability requirement. In this case, both mechanisms are listed in the section above that discusses mechanisms for achieving integrability—one is a pattern and one is a tactic—so the check passes.

In contrast, if the documented rationale (or the architect) stated that the architecture uses ping-echo and exception detection to achieve this requirement, this would raise a red flag since those mechanisms are usually associated with improving availability. The analyst might decide to stop the architecture analysis at this point and gather more information from the architect. The point of this sanity check is not to analyze the mechanism or decision in detail but simply to assess whether the architecture analysis is on the right track before devoting more effort to it.

In some cases, appropriateness of a mechanism is less clear. For example, the rationale in this case might specify that a *shared repository* mechanism is used. A configuration file can have multiple readers, so it could be called a shared repository. In cases like this, the analyst should proceed carefully: the architect may have chosen an inappropriate mechanism, mislabeled the mechanism used, or used the mechanism in an atypical way that may or may not be appropriate.

7.3 Step 3–Locate the mechanisms in the architecture

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

Our scenario is concerned with the Signal Processing Pipeline. The analyst may be able to look at the documentation and find a structural diagram sketch that includes the Signal Processing Pipeline. With this diagram in hand, finding an instance of the pipe and filter mechanism should not be difficult because it is a major functional capability in the system. The analyst should also be able to locate the data filtering component in the pipeline.

The analyst should next turn to the configuration file mechanism and find an instance of a configuration file in one of the structural diagram sketches. This might be more difficult—it might be a single element in a larger diagram sketch, and the analyst will need to figure out which sketch that is. In this search, note the number of configuration files, identified as "Config1," "Config2," and "Config4."

Finally, the analyst must be able to conceptually put the mechanisms together. The rationale for satisfying the requirement said that the configuration file defines the topology of the Signal Processing Pipeline. This raises a question: When is the Signal Processing Pipeline instantiated? This is an issue of *Binding Time*, one of the categories of questions in the Architecture Analysis Checklist. One answer could be that the configuration file is read during the software build process, and

the pipeline configuration is fixed in the executable image. However, the analyst finds that, in reality, the configuration file is read during system initialization. He finds a component called Pipeline Configurator that reads the configuration file and instantiates the pipes and filters that comprise the Signal Processing Pipeline, and he sees that it reads from configuration file "PL.conf."

Before finishing this step, the analysts should check that the mechanisms are being used in parts of the architecture that relate to the requirement that they are analyzing. To assess how well a mechanism contributes to satisfying a quality attribute requirement, it is not sufficient to stop after the sanity check in Step 2. That establishes only the presence of the mechanisms, not their suitability or adequacy for the scenario being considered. The analysts must identify where and how the mechanism was instantiated in the architecture to assess whether it will have the desired effect. For example, if they find a pipe and filter mechanism, but it is used in the Display Overlay section of the architecture, then the use of that mechanism is not likely to improve the integrability of the Signal Processing Pipeline. Or if they did not find that one of the configuration files was read and used to configure the Signal Processing Pipeline topology, then the use of that mechanism is not likely to improve the integrability of the Signal Processing Pipeline.

7.4 Step 4–Identify derived decisions and special cases

Most architecture mechanisms are not simple, one-size-fits-all constructs. The instantiation of a mechanism requires making a number of decisions, with some of those decisions involving choosing and instantiating other mechanisms. For instance, our example employs a pipe and filter pattern (mechanism). One set of decisions about using that mechanism is concerned with when the pipeline topology is instantiated (refer to the *Binding Time* category in the Architecture Analysis Checklist). In this case, alternatives include

- at build time (topology is fixed)
- during software execution (topology is variable)
  - at software initialization (topology is fixed after initialization)
  - during software execution (topology is dynamic)
    - topology changes when system mode changes
    - topology changes based on time trigger
    - topology changes based on user input
    - …

If the architect decided to make the topology dynamic (the last major alternative above), then there is a set of subsequent derived decisions about whether and how to flush the data stream when the topology changes. Any of these alternatives leads to decisions about where the topology is defined; in our example, the architect has specified that the topology is defined using a configuration file mechanism.

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. The analyst must verify that the mechanism

meets the requirement and hence must evaluate how the mechanism was instantiated, which usually involves tracing the decisions about the mechanism instantiation to the derived decisions and the selected alternatives that address them.

As an analyst put the mechanisms together in Step 3, she started to identify derived decisions. For example, in the decision tree outlined above, the analyst identified that the pipeline topology is variable, but fixed after initialization.

The analyst's next derived decision might be "Where does the Pipeline Configurator component get the information needed to instantiate each pipe or filter in the topology, and how does it instantiate and initialize each pipe or filter?" This is an *Allocation of Responsibilities* decision in the Architecture Analysis Checklist. For the integrability requirement that the analyst is validating, a good answer to these questions is the following:

- All pipes are instantiated using a single, common interface, and all filters are instantiated using a single, common interface;
- all pipes are initialized using a single, common interface, and all filters are initialized using a single, common interface; and
- all information needed to instantiate and initialize a pipe or filter is specified in the configuration file.

If these are all true, then the Pipeline Configurator contains no information about any particular pipe or filter, so our requirement to integrate a new filter component will not necessitate any change to the Pipeline Configurator. (If the driving quality attribute requirement was, for example, reusability of *existing* independently developed filter components that have a variety of initialization interfaces, rather than integrability, then the architect might have chosen to make the Pipeline Configurator responsible for handling each of the reused filter components as a special case. Changing a filter component might then require changing the Pipeline Configurator, which would be less integrable.)

Another derived decision is how the Pipeline Configurator identifies the software to instantiate for a pipe or filter (see the *Binding Time* and *Allocation of Responsibilities* categories in the Architecture Analysis Checklist). The software for a pipe or filter could be statically linked, which necessitates rebuilding the software when a new pipe or filter is integrated, or the software for a pipe and filter could be dynamically linked and loaded at runtime, in which case the software might not need to be rebuilt. The second approach reduces the time and effort to integrate a new filter component. However, it also creates another derived decision: To change a filter component, a developer would need to change the information in the configuration file, so the analysts become concerned with how the configuration file is linked to the Pipeline Configurator. If a change to the configuration file triggers a rebuild of the software, then the benefit of dynamically linked pipes and filters is negated. Thus, the assertion that "the configuration file is dynamically linked, so all the developer needs to do is change the configuration file" is not correct if the pipes and filters are statically linked.

Let's look at one more derived decision: How is the information represented in the configuration file? This is covered in the *Data Model* category in the Architecture Analysis Checklist. The analyst checks that the schema for the configuration file data aligns with the interfaces to instantiate and initialize the pipe components and filter components. For example, if elements of the initialization interface of a new filter component cannot be specified using the configuration file schema, then integrating the new filter element might necessitate a more extensive change, impacting the satisfaction of our integrability requirement.

These are just a few examples of derived decisions. Other common derived decisions include

- error handling: Does the component that receives and records runtime errors need to change to integrate with the new filter (included in the Coordination Model category in the Architecture Analysis Checklist)?
- resource sharing: Does the new filter component consume more shared resources (e.g., processor cycles, memory, I/O)? If the architecture does not already address resource management, then other software modifications may be necessary (see the Resource Management category in the Architecture Analysis Checklist).

Finally, some mechanisms have special cases that warrant special attention. For example, adding a second writer to a shared repository introduces concerns about write conflicts. A distributed shared repository introduces concerns about replica consistency. Modifications to the top or bottom layers in a layered mechanism introduce concerns about interfaces outside the mechanism.

In our example, analysts should pay attention to changes to the first and last elements in a pipeline topology, as they introduce concerns about interfaces outside the pipe and filter mechanism. Our requirement to replace a filter component will be more broadly satisfied by instantiating a pipe and filter mechanism that does not mix data processing and data interface/interoperation in a single filter element.

7.5 Step 5–Assess requirement satisfaction

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

Recall that the requirement in our example is to add a new data filtering component, and our measures are the effort needed, that necessary messages are correctly processed, and that unnecessary messages are ignored. The architecture mechanisms are pipe and filter with the topology established at system initialization time based on the contents of a configuration file. In Step 4–

Identify derived decisions and special cases, the analyst identified several derived decisions that need to be considered in the analysis:

- Where does the Pipeline Configurator component get the information needed to instantiate each pipe or filter in the topology, and how does it instantiate and initialize each pipe or filter?
- How does the Pipeline Configurator identify the software to instantiate for a pipe or filter?
- How is the information represented in the configuration file?

The analyst might begin with the last question—a question about the data model—since he needs to understand the structure of the configuration file to address the other questions. The configuration file schema is not specified in the architecture documentation, but the artifacts include an excerpt of an example pipeline configuration, and the example looks like it might be using the YAML[3] key-value syntax. All elements that use the configuration file are semantically coupled—they must all interpret the contents in the same way. The analyst begins recording analysis issues related to semantic cohesion:

- Issue 1: The configuration file schema is not explicitly specified. If a developer needs to discover the schema, it increases the effort to add a new filter.
- Issue 2: YAML values are not typed. The configuration file reader (in our case, the Pipeline Configurator) must include robust error handling for invalid values. If invalid configuration values are not robustly detected, this could increase the effort to add a new filter.

This analysis thread is based on an observation that the configuration file uses YAML syntax. For example, if the analyst found that the file had an XML schema, which is strongly typed, then he might record an issue about whether all the data types necessary to configure the new filter are available in the schema, since modifying the schema would likely increase the effort to add a new filter. Note that compared to YAML, writers and readers of a schema-based XML configuration file have higher syntactic coupling to the file, but they have a lower semantic coupling to each other because the schema enforces some shared meaning of the data.

Continuing our example, the analyst finds that the pipeline elements are statically linked. This architecture decision simplifies some runtime error handling by precluding error conditions such as a missing dynamic library. On the other hand, it introduces constraints on the software build process—there is syntactic coupling between the linker and filter object code files. The analyst records an issue:

- Issue 3: The filter object code is syntactically coupled to the linker in the development toolchain. This constrains how filters are coded and built. This is type of coupling is hard to mitigate using integrability mechanisms.

Next, in our simplified analysis example, the analyst investigates how the Pipeline Configurator instantiates elements and how the pipe and filter mechanism connects elements together to create

[3] See https://yaml.org for more information about YAML Ain't Markup Language™.

the processing pipeline. The architecture documentation states that the Pipeline Configurator requires every pipe or filter element to provide an interface called NewInstance that both creates and initializes an instance of the element. The pipeline architecture constrains filters to provide a single DataIn and a single DataOut interface. This analysis triggers two more issues to record:

- Issue 4: There is syntactic coupling between the Pipeline Configurator and the filters. If the new data filter will reuse an existing implementation, then it will need to be wrapped with an adapter for the NewInstance, DataIn, and DataOut interfaces. The effort to develop this adapter will depend on the distance (as discussed in Section 3.1) between the pipelines required interfaces and the existing filter implementations provided interfaces.
- Issue 5: The data types for the DataIn and DataOut interfaces are not specified in the architecture documentation. This is early in the development lifecycle and the artifacts are not complete, but the architecture documentation and the structural diagrams to date indicate that the architect is making pipe elements in the pipe and filter pipeline responsible for data transformation to bridge between filter elements. This introduces semantic coupling between the pipe and filter. If this is the case, then the architect is not applying the mechanism correctly, since the Pipe and Filter pattern specifies that pipes do not transform data, so there is no semantic coupling between the pipe and filter.

In this simple example, the analyst rapidly identified five issues where architecture decisions impact the ability to achieve the desired response measures in the scenario. Some of the issues, such as Issue 3 about static linking, are unlikely to change as the details of the architecture are refined. Other issues, such as Issue 1 about the configuration file schema, 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 analyst found adequate to satisfy the requirement being evaluated in Step 5 may be detrimental to the satisfaction of other quality attribute requirements.

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

- (Performance) Is there a pipeline data processing latency requirement? Is there sufficient margin in the latency budget to permit a new filter to be added without other modifications to improve latency?
- (Performance) Is there a latency timeline for pipeline initialization? Is there sufficient margin in that latency budget to permit a new filter to be instantiated and configured without other changes?

- (Testability) Does the architecture prescribe the logging behavior and data collection interfaces for the new filter element?
- (Maintainability) How will any new wrappers or bridges needed to add the new filter affect maintainability?
- (Availability) Does the architecture prescribe the fault handling behavior for the new filter?

In this step, the analyst assesses how the mechanisms and decisions that make it easy to add a new filter impact the satisfaction of scenarios related to these other quality attributes and concerns. For each requirement, the analysis may be fast (e.g., seeing immediately that there is sufficient latency margin in the initialization timeline) or more involved (e.g., assessing the maintainability impacts or even building a prototype to measure latency). In any case, the analyst should expect to find at least a couple of additional issues.

7.7 Step 7–Assess the cost/benefit of the architecture approach

In carrying out the steps leading up to this point, the analyst has 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.

Any architecture approach adds new elements and interactions, and so makes the solution more complicated. Some approaches add new *types* of elements and interactions and may make the solution 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 introduced by this architecture approach is necessary. In some cases, the answer will be clear: in our example, if the Signal Processing Pipeline topology has only two filters connected by a single pipe, and this number is unlikely to grow significantly, then the complexity of the pipe and filter mechanism may not be warranted. If the pipeline topology is set at build time, then a separate configuration file may not be needed.

In other cases, the judgment is less clear; however, asking the question of necessity is still worthwhile.

8 Summary

In this report we defined integrability and focused on analyzing integration difficulty, the costs, and the technical risks of performing a set of desired integration tasks. This difficulty can be thought of as a function of the size of and distance between the interfaces of a set of components $\{C_i\}$ and a system S. We further explained that size is the number of potential dependencies between $\{C_i\}$ and S and distance is the difficulty of resolving each of the dependencies.

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

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

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

9 Further Reading

A general discussion of quality attributes, quality attribute scenarios, tactics, and patterns that provided the foundation for much of this report may be found in the book *Software Architecture in Practice* [Bass 2012]. That book, however, does not address integrability specifically. Another general discussion of quality attributes, particularly the vocabulary surrounding them, can be found in ISO/IEC/IEEE 24765 [ISO 2017].

A more in-depth discussion of the quality attribute of integrability specifically can be found in Hentonnen [2007].

MacCormack [2006] and Mo et al. [2016] define and provide empirical evidence for architecture-level coupling metrics.

Bibliography

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

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

[Bass 2012]
Bass, L.; Clements, P; & Kazman, R. *Software Architecture in Practice*, 3rd ed. Addison-Wesley. 2012.

[Berkeley 2004]
"Model-View-Controller: A Design Pattern for Software." 2004. https://pdfs.semanticscholar.org/b288/c7074ef1ae95174538147eae22b41fe3746e.pdf

[Bianco 2011]
Bianco, Philip; Lewis, Grace; Merson, Paulo; & Simanta, Soumya. *Architecting Service-Oriented Systems.* CMU/SEI-2011-TN-008. Software Engineering Institute, Carnegie Mellon University. 2011. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9829

[Brownsword 2004]
Brownsword, Lisa; Carney, David; Fisher, David; Lewis, Grace; Morris, Edwin; Place, Patrick; Smith, James; Wrage, Lutz; & Meyers, B.. *Current Perspectives on Interoperability*. CMU/SEI-2004-TR-009. Software Engineering Institute, Carnegie Mellon University. 2004. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7109

[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.

[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 ed. Addison-Wesley. 2010.

[Garlan 1995]
Garlan, D.; Allen, R.; & Ockerbloom, J. "Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts." Page 179. In *17th International Conference on Software Engineering*. Seattle, Washington. April 1995.

[Gartner 2019]
Gartner. "Service-Oriented Architecture (SOA)." IT Glossary. 2019. http://www.gartner.com/it-glossary/service-oriented-architecture-soa/

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

[ISO 2017]
ISO/IEC/IEEE 24765. "Systems and Software Engineering — Vocabulary," 2nd ed. 2017.

[Josuttis 2007]
Josuttis, N. *SOA in Practice: The Art of Distributed System Design*. O'Reilly. 2007.

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

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

[SWEBOK 2014]
Bourque, P. & Fairley, R. E., eds. *Guide to the Software Engineering Body of Knowledge*, Version 3.0. IEEE Computer Society. 2014.

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

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM19-1089