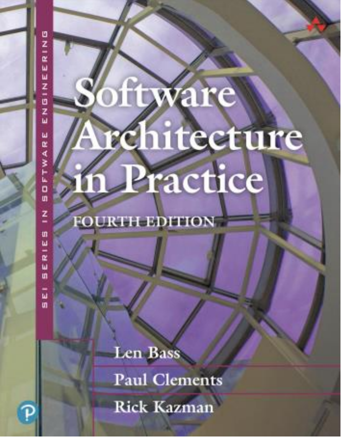
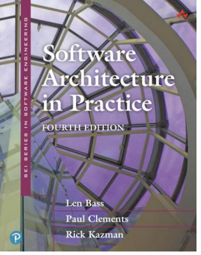

Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

Chapter 15: Software Interfaces

NASA lost its $125-million Mars Climate Orbiter because spacecraft engineers failed to convert from English to metric measurements when exchanging vital data before the craft was launched. . . .

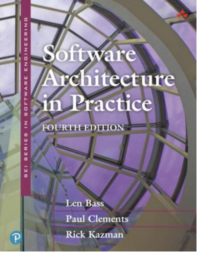
A navigation team at [NASA] used the metric system of millimeters and meters in its calculations, while [the company that] designed and built the spacecraft provided crucial acceleration data in the English system of inches, feet and pounds. . . .

In a sense, the spacecraft was lost in translation.
—Robert Lee Hotz



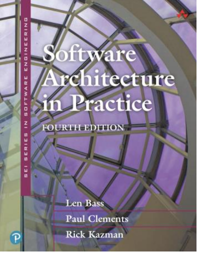
Chapter Outline

- Interface Concepts
- Designing an Interface
- Documenting the Interface
- Summary



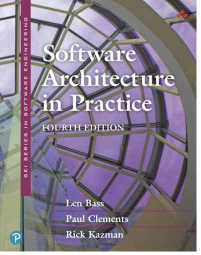
Interface Concepts

- An *interface* is a boundary across which elements interact, communicate, and coordinate.
- An element's *actors* are the other elements, users, or systems with which it interacts.
- The collection of actors with which an element interacts is called the *environment* of the element.
- The constructs that provide points of direct interaction with an element, are called *resources*.



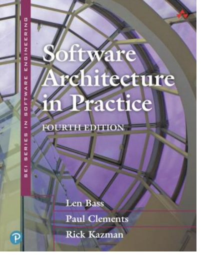
Interface Concepts

- **Multiple Interfaces**
  – It is possible to split a single interface into multiple interfaces.
  – Each of these has a related logical purpose, and serves a different class of actors.
  – Multiple interfaces provide a kind of separation of concerns. A specific class of actor might require only a subset of the functionality available.
  – Conversely, the provider of an element may want to grant actors different access rights, or to implement a security policy.



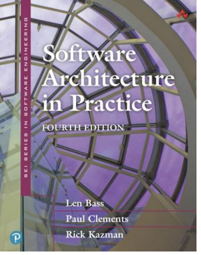
Interface Concepts

- **Resources**
  - *Resource syntax*. The syntax is the resource's signature. The signature includes the name of the resource, the names and data types of arguments, if any, etc.
  - *Resource semantics*. What is the result of invoking this resource?


Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

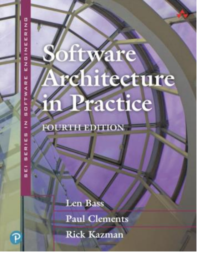
Interface Concepts

- **Operations, Events, and Properties**
  - The resources of provided interfaces consist of operations, events, and properties.
  - These resources are complemented by a description of the behavior caused or data exchanged.
  - *Operations* are invoked to transfer control and data to the element for processing.
  - *Events* may be described in interfaces. Incoming events can be the receipt of a message, or the arrival of a stream element to be consumed. Active elements produce outgoing events used to notify listeners.
  - An important aspect of interfaces is metadata, such as access rights, units of measure, or formatting assumptions. Another name for this interface metadata is *properties*.


Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

Interface Evolution

- An interface is a contract between an element and its actors. Three techniques to change an interface are:
  - *Deprecation*. Deprecation means removing an interface. Best practice when deprecating an interface is to give extensive notice to the actors of the element. In practice, many actors will not adjust in advance, but rather will discover the deprecation only when the interface is removed.
  - *Versioning*. Multiple interfaces support evolution by keeping the old interface and adding a new one. The old one can be deprecated when it is no longer needed or the decision has been made to no longer support it. This requires the actor to specify which version of an interface it is using.
  - *Extension*. Extending an interface means leaving the original interface unchanged and adding new resources to the interface that embody the desired changes.



Interface Evolution

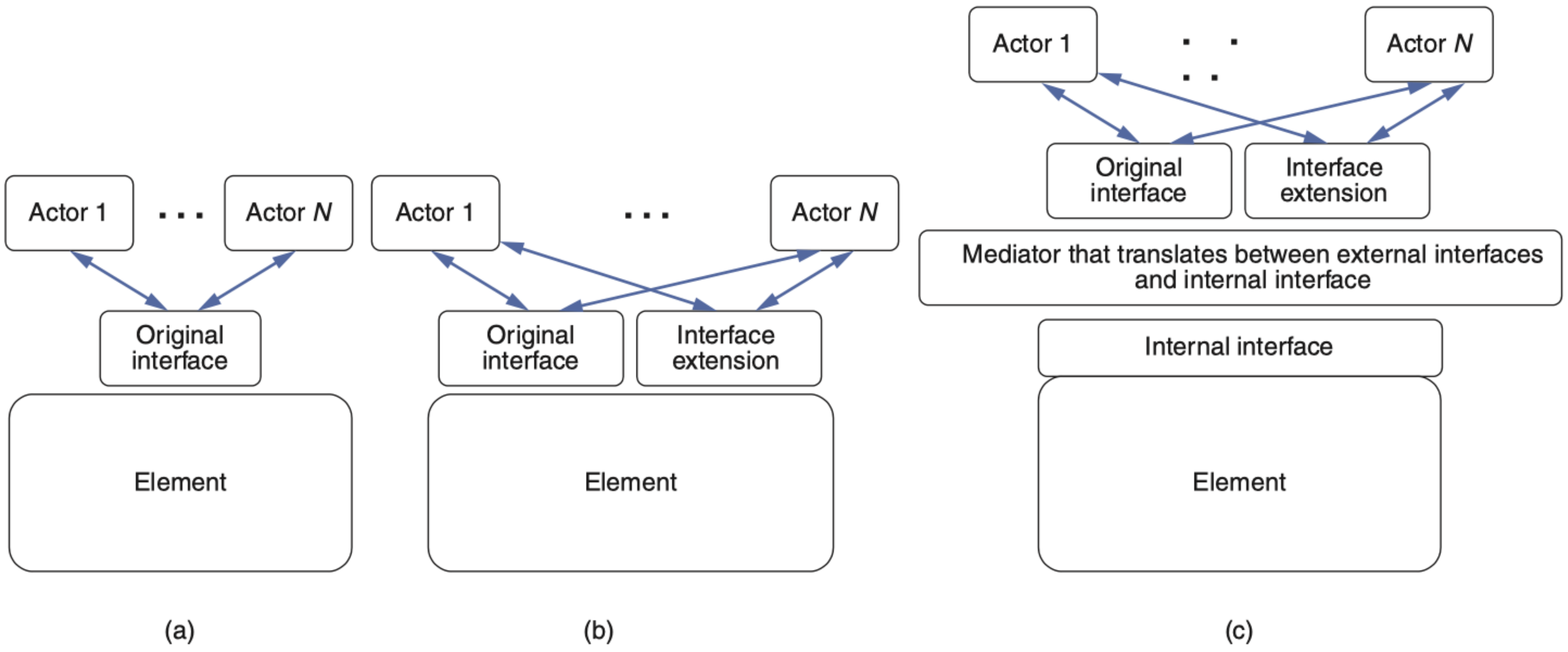
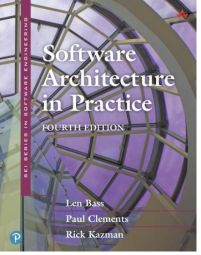
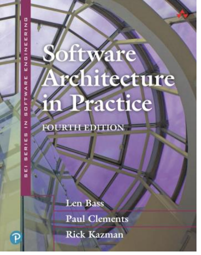


FIGURE 15.1 (a) The original interface. (b) Extending the interface. (c) Using an intermediary.



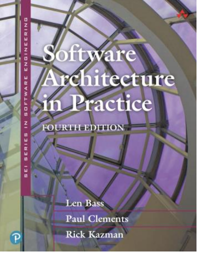
Designing an Interface

- Decisions about which resources should be externally visible should be driven by the needs of actors that use the resources. Adding resources to an interface implies a commitment to maintain those resources. Once actors start to depend on a resource, their elements will break if the resource is changed or removed.
- Some additional design principles for interfaces are:
  - *Principle of least surprise*. Interfaces should behave consistently with the actor's expectations.
  - *Small interfaces principle*. If two elements need to interact, have them exchange as little information as possible.
  - *Uniform access principle*. Avoid leaking implementation details through the interface. A resource should be accessible to its actors in the same way regardless of how they are implemented.
  - *Don't repeat yourself principle*. Interfaces should offer a set of composable primitives as opposed to many redundant ways to achieve the same goal.



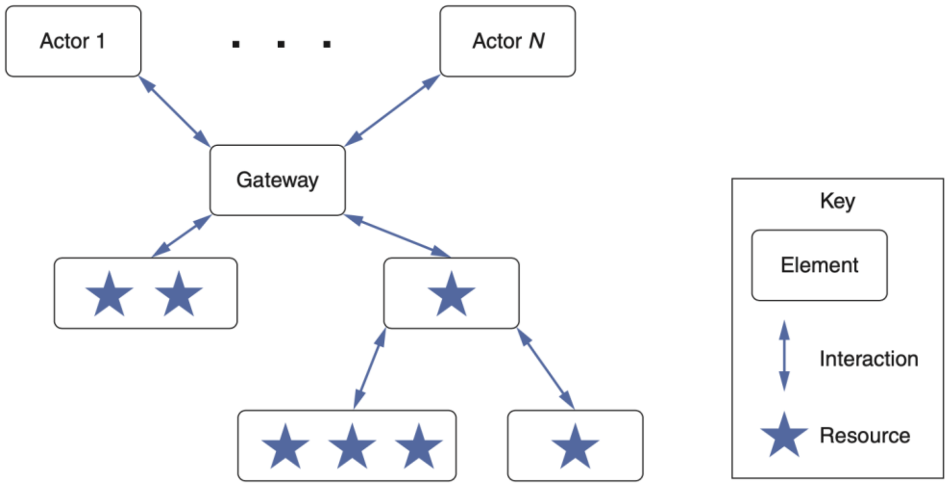
Designing an Interface

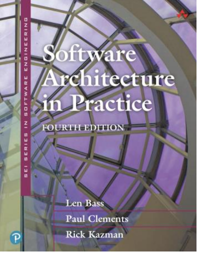
- A successful interaction with an interface requires agreement on the following aspects:
  - Interface scope
  - Interaction style
  - Representation and structure of the exchanged data
  - Error handling



Interface Scope

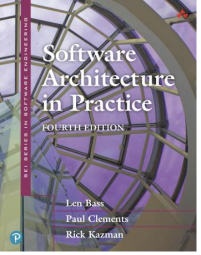
- The scope of an interface defines the resources available to the actors.
- A common pattern for constraining and mediating access to resources is to establish a *gateway* element. Gateways are useful for the following reasons:
  - A gateway can translate between elements and actors.
  - Actors may need access to, or be restricted to, specific subsets of the resources.
  - The specifics of the resources—number, protocol, type, location, properties—may change over time, and the gateway can provide a more stable interface.





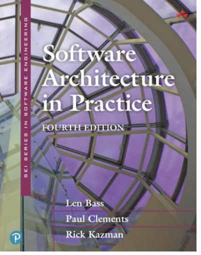
Interaction Styles

- Interfaces are meant to be connected together so that elements can communicate (transfer data) and coordinate (transfer control).
- There are many ways for such interactions to take place, depending on:
  - the mix between communication and coordination, and
  - whether the elements will be co-located or remotely deployed.
- The most widely used styles are: RPC and REST



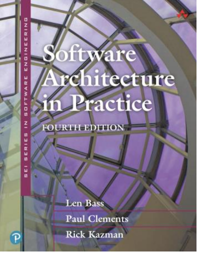
RPC

- *Remote Procedure Call (RPC)*.
  - RPC is modeled on procedure (method) calls in imperative languages, except that the called method is located elsewhere on a network.
  - The programmer codes the call as if a local method were being called (with some syntactic variation); the call is translated into a message sent to a remote element where the actual method is invoked.
  - The results are sent back as a message to the calling element.



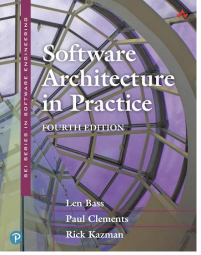
REST

- *Representational State Transfer (REST)*. REST is a protocol for web services. It imposes six constraints on the interactions between elements:
  - *Uniform interface*. All interactions use the same form (typically HTTP). Resources are specified via URIs.
  - *Client-server*. The actors are clients and the resource providers are servers using the client-server pattern.
  - *Stateless*. All client-server interactions are stateless.
  - *Cacheable*. Caching is applied to resources when applicable.
  - *Tiered system architecture*. The “server” can be broken into multiple elements, which may be deployed independently.
  - *Code on demand (optional)*. It is possible for the server to provide code to the client to be executed.



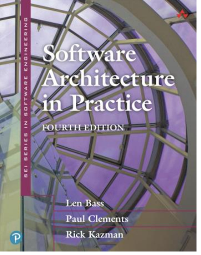
Representation and Structure of Exchanged Data

- Choosing how to represent interchanged data has the following dimensions:
  - *Expressiveness*. Can the representation serialize arbitrary data structures? Is it optimized for trees of objects? Does it need to carry text written in different languages?
  - *Interoperability*. Does the representation used by the interface match what its actors expect and know how to parse?
  - *Performance*. Does the chosen representation allow efficient usage of the available communication bandwidth?
  - *Implicit coupling*. What are the shared assumptions that could lead to errors and data loss when decoding messages?
    Transparency. Is it possible to intercept the exchanged messages and easily observe their content?



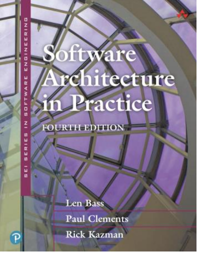
EXtensible Markup Language (XML)

- XML annotations to a textual document, called *tags*, are used to specify how to interpret the information in the document by breaking the information into fields and identifying the data type of each field. Tags can be annotated with attributes.
- XML is a meta-language: Out of the box, it does nothing except allow you to define a customized language to describe your data. Your customized language is defined by an *XML schema*, which specifies the tags you will use, the data type used to interpret fields, and the constraints on the document.



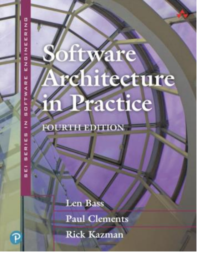
JavaScript Object Notation (JSON)

- JSON structures data as nested name/value pairs and array data types.
- Like XML, JSON is a textual representation featuring its own schema.
- JSON data types are derived from JavaScript, and resemble those of any modern programming language. This makes JSON serialization and deserialization much more efficient than XML.



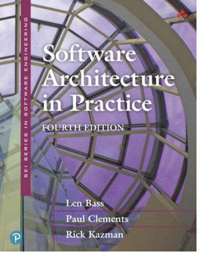
gRPC

- The most recent version of RPC, called gRPC, transfers parameters in binary, is asynchronous, and supports authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts.
- By default gRPC uses *protocol buffers*.



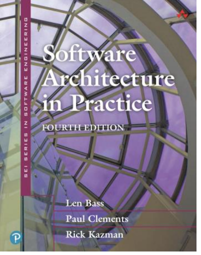
Protocol Buffers

- Like JSON, Protocol Buffers use data types that are close to programming-language data types, making serialization and deserialization efficient.
- As with XML, Protocol Buffer messages have a schema that defines a valid structure, and that schema can specify both required and optional elements and nested elements.
- However, unlike both XML and JSON, Protocol Buffers are a binary format, so they are extremely compact and efficient.



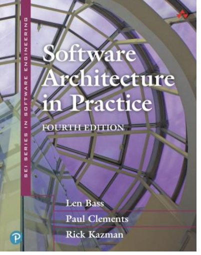
Protocol Buffers

- A protocol buffer specification is used to specify an interface. Kept in a .proto file
- Language specific compilers used for each side of an interface. This allows different languages to communicate across a message based interface.
- The collection of .proto files defines all of the interfaces and hence all of the microservices.



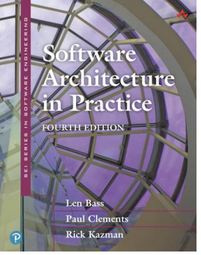
Error Handling

- When designing an interface, architects naturally concentrate on how it is supposed to be used in the nominal case.
- But a well-designed system must take appropriate action in the face of undesired circumstances.
- Strategies to do so include the following:
  - Failed operations may throw an exception.
  - Operations may return a status indicator with predefined codes, which need to be tested to detect erroneous outcomes.
  - Properties may be used to store data indicating whether the latest operation was successful or not, or whether stateful elements are in an erroneous state.
  - Error events such as a timeout may be triggered for failed asynchronous interactions.
  - The error log may be read by connecting to a specific output data stream.



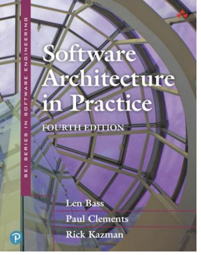
Documenting the Interface

- The interface documentation indicates what other developers need to know about an interface to use it.
- As you document an element's interface, keep the following stakeholder roles in mind:
  - *Developer of the element*
  - *Maintainer*
  - *Developer of an element using the interface*
  - *Systems integrator and tester*
  - *Analyst*
  - *Architect looking for assets to reuse in a new system*
- Describing an element's interface means making statements about the element that other elements can depend on.
- Documenting an interface means that you have to describe which services and properties are parts of the contract.



Summary

- Architectural elements have interfaces, which are boundaries over which elements interact. Interface design is an architectural duty.
- A primary use of an interface is to encapsulate an implementation, so that it may change without affecting other elements.
- Interfaces state which resources the element provides to its actors as well as what the element needs from its environment.
- Interfaces have operations, events, and properties; these are the parts of an interface that the architect can design. To do so, the architect must decide the element's
  - Interface scope
  - Interaction style
  - Representation, structure, and semantics of the exchanged data
  - Error handling



Summary

- Some of these issues can be addressed by standardized means. For example, data exchange can use mechanisms such as XML, JSON, or Protocol Buffers.
- All software evolves, including interfaces. Three techniques that can be used to change an interface are deprecation, versioning, and extension.
- The interface documentation indicates what other developers need to know about an interface to use it in combination with other elements.
- Documenting an interface involves deciding which element operations, events, and properties to expose to the element's actors, and detailing the interface's syntax and semantics.