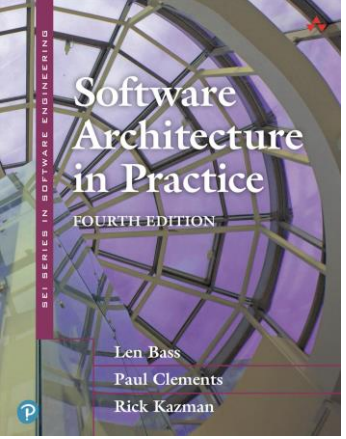
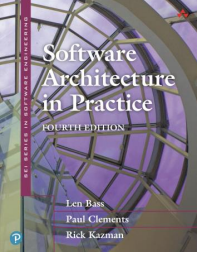


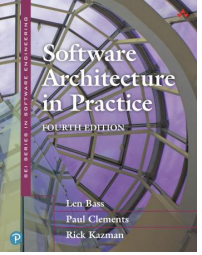
Chapter 7: Integrability

Integration is a basic law of life; when we resist it, disintegration is the natural result, both inside and outside of us. Thus we come to the concept of harmony through integration.
—Norman Cousins



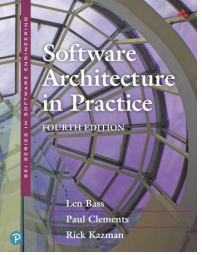
Chapter Outline

- What is Integrability?
- Integrability General Scenario
- Tactics for Integrability
- Tactics-Based Questionnaire for Integrability
- Patterns for Integrability
- Summary



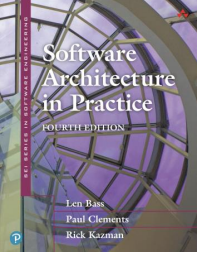
What is Integrability?

- Software architects need to be concerned about more than just making separately developed components cooperate; they are also concerned with the *costs* and *technical risks* of integration tasks.
- These risks may be related to schedule, performance, or technology.



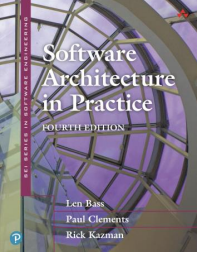
What is Integrability?

- Consider that a project needs to integrate a set of components *C*1, *C*2, ... *Cn*, into a system *S*.
- Our task is to design for, and analyze the costs and technical risks of, integrating {*Cn*+1, ... *Cm*}.
- We assume we have control over *S*, but the {*Ci*} may be outside our control.



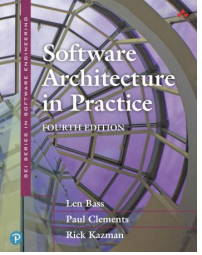
Integrability Concerns

- Integration difficulty—the costs and the technical risks—can be thought of as a function of the size of and the "distance" between the interfaces of {*Ci*} and *S*:
  - Size is the number of potential dependencies between {*Ci*} and *S*.
  - Distance is the difficulty of resolving differences at each of the dependencies.



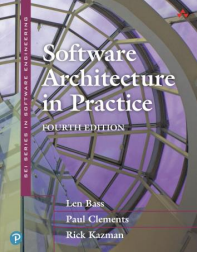
Integrability Concerns

- Distance may mean:
  - *Syntactic distance*
  - *Data semantic distance*
  - *Behavioral semantic distance*
  - *Temporal distance*
  - *Resource distance*



Integrability General Scenario

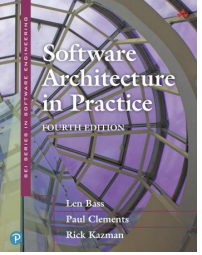
Portion of Scenario	Description	Possible Values
Source	Where does the stimulus come from?	One or more of the following: ▪ Mission/system stakeholder ▪ Component marketplace ▪ Component vendor
Stimulus	What is the stimulus? That is, what kind of integration is being described?	One of the following: ▪ Add new component ▪ Integrate new version of existing component ▪ Integrate existing components together in a new way
Artifact	What parts of the system are involved in the integration?	One of the following: ▪ Entire system ▪ Specific set of components ▪ Component metadata ▪ Component configuration
Environment	What state is the system in when the stimulus occurs?	One of the following: ▪ Development ▪ Integration ▪ Deployment ▪ Runtime



Integrability General Scenario

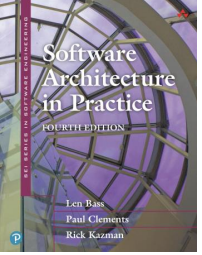
Response	How will an "integrable" system respond to the stimulus?	One or more of the following: ▪ Changes are {completed, integrated, tested, deployed} ▪ Components in the new configuration are successfully and correctly (syntactically and semantically) exchanging information ▪ Components in the new configuration are successfully collaborating ▪ Components in the new configuration do not violate any resource limits

Response measure	How is the response measured?	One or more of the following:
		▪ Cost, in terms of one or more of:
		  ▪ Number of components changed
		  ▪ Percentage of code changed
		  ▪ Lines of code changed
		  ▪ Effort
		  ▪ Money
		  ▪ Calendar time
		▪ Effects on other quality attribute response measures (to capture allowable tradeoffs)

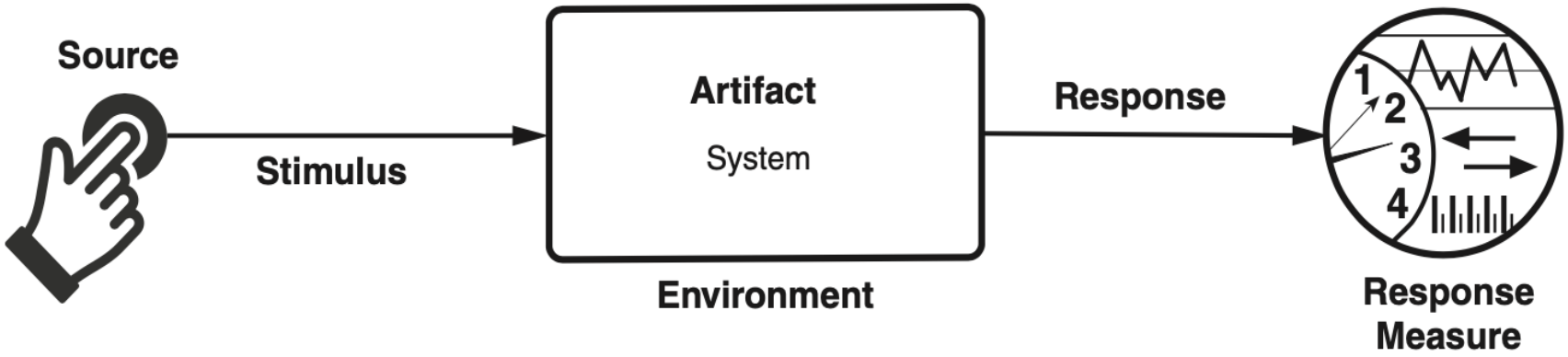


Sample Concrete Integrability Scenario

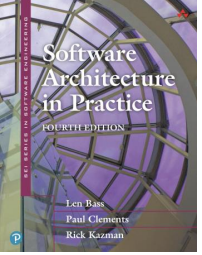
- *A new data filtering component has become available in the component marketplace. The new component is integrated into the system and deployed in 1 month, with no more than 1 person-month of effort.*



Sample Concrete Integrability Scenario

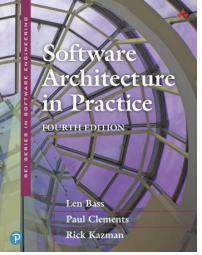


Source	Stimulus	Environment	Response	Response Measure
Component marketplace	New data filtering component becomes available	Development	The new component is integrated and deployed	1 month, with no more than 1 person-month of effort

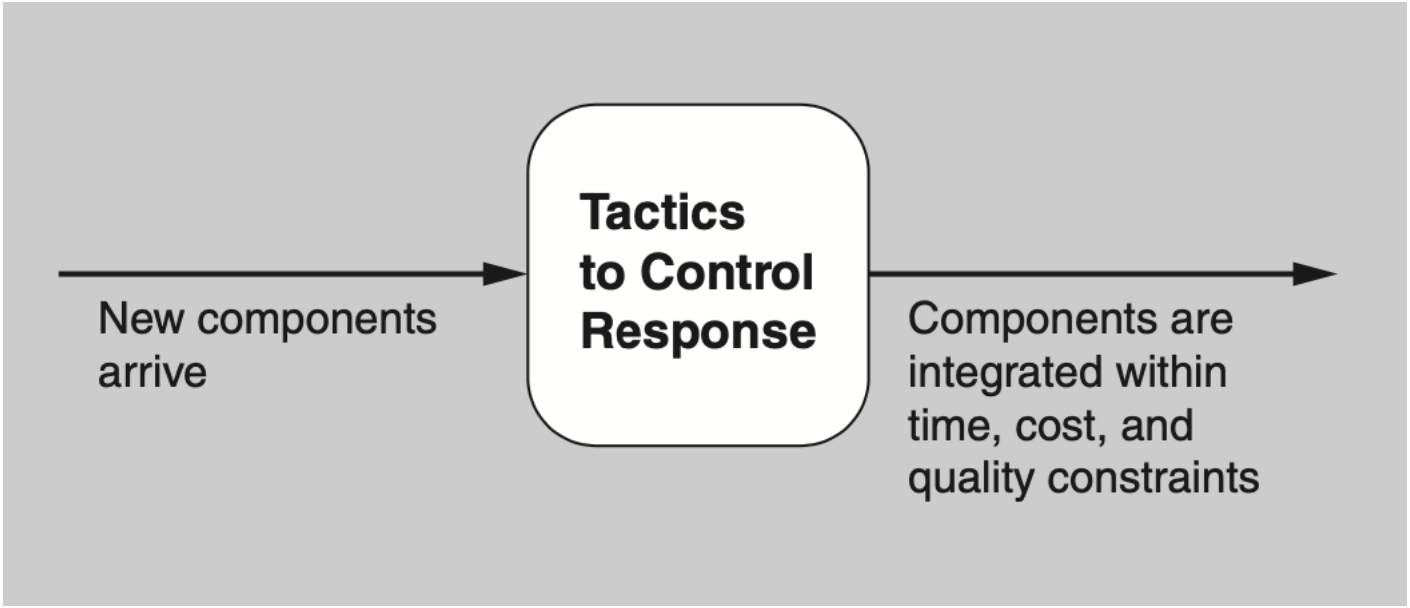


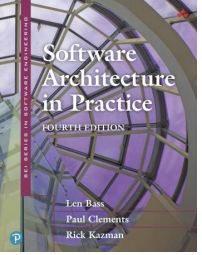
Goal of Integrability Tactics

- The goals for the integrability tactics are to reduce the costs and risks of adding new components, reintegrating changed components, and integrating sets of components together to fulfill evolutionary requirements.

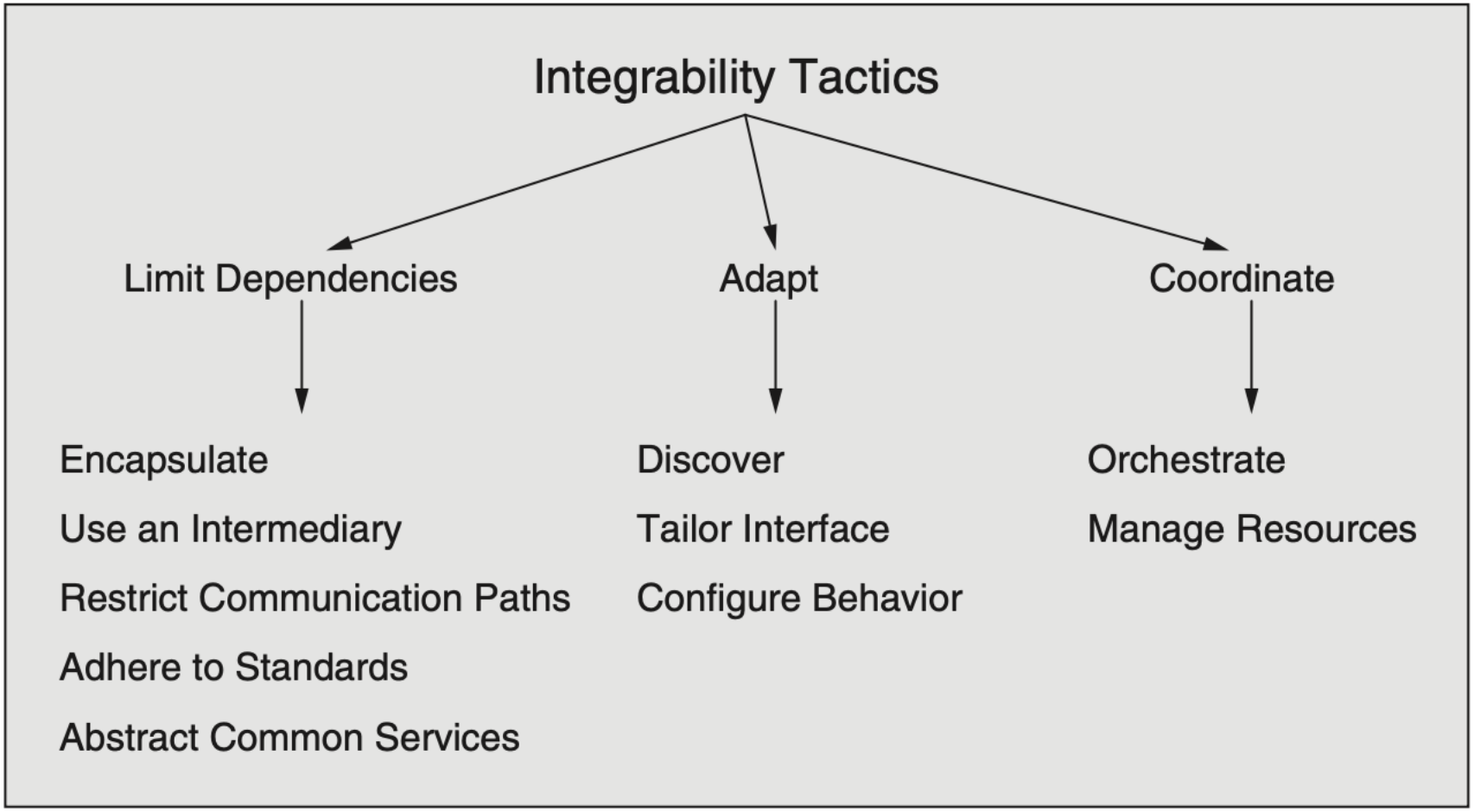


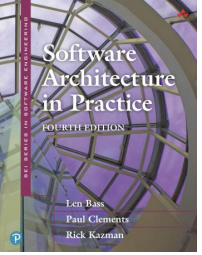
Goal of Integrability Tactics





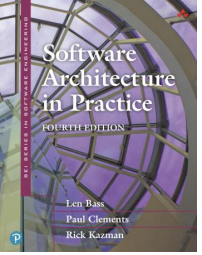
Integrability Tactics





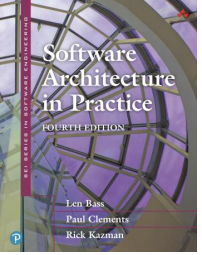
Limit Dependencies

- *Encapsulate*. Encapsulation introduces an explicit interface to an element and ensures that all access to the element passes through this interface. Dependencies on the element internals are eliminated, because all dependencies must flow through the interface.
- *Use an Intermediary*. Intermediaries are used for breaking dependencies between a set of components *Ci* or between *Ci* and the system *S*. Intermediaries can be used to resolve different types of dependencies (syntactic, behavior, data semantic, etc.).
- *Restrict Communication Paths*. This tactic restricts the set of elements with which a given element can communicate. In practice, this tactic is implemented by restricting an element's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (i.e., restricting access to only authorized elements).



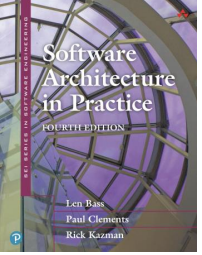
Limit Dependencies

- *Adhere to Standards*. Standardization in system implementations is a primary enabler of integrability and interoperability, across both platforms and vendors. Some standards focus on defining syntax and data semantics. Others include richer descriptions, such as those describing protocols that include behavioral and temporal semantics.
- *Abstract Common Services*. Where two elements provide services that are similar, it may be useful to hide both behind a common abstraction. This abstraction might be realized as a common interface implemented by both, or it might involve an intermediary that translates requests for the abstract service to more specific requests. The resulting encapsulation hides the details of the elements from other components in the system.



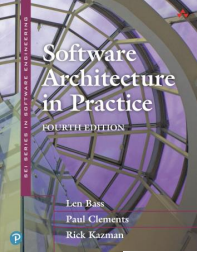
Adapt

- *Discover*. A discovery service is the mechanism by which applications and services locate each other. Entries in a discovery service are there because they were registered. This registration can happen statically, or it can happen dynamically when a service is instantiated.
- *Tailor Interface*. Tailoring an interface is a tactic that adds capabilities to, or hides capabilities in, an existing interface without changing the API or implementation. Capabilities such as translation, buffering, and data smoothing can be added to an interface without changing it.
- *Configure Behavior*. The behavior of a component can be configured during the build phase (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).



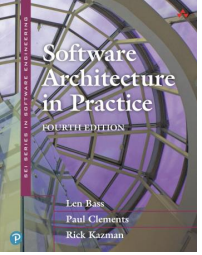
Coordinate

- *Orchestrate*. Orchestrate is a tactic that uses a control mechanism to coordinate and manage the invocation of services so that they can remain unaware of each other. Orchestration helps with the integration of a set of loosely coupled reusable services to create a system that meets a new need.
- *Manage Resources*. A resource manager is a 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), but instead request those resources from a resource manager.



Tactics-Based Questionnaire for Integrability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Limit Dependencies	Does the system **encapsulate functionality** of each element by introducing explicit interfaces and requiring that all access to the elements passes through these interfaces?				
	Does the system broadly **use 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?				
	Does the system **adhere to standards** in terms of how components interact and share information with each other?				



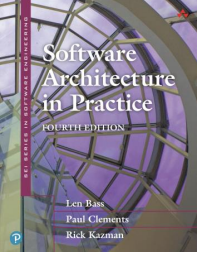
Tactics-Based Questionnaire for Integrability

Adapt

Does the system provide the ability to statically (i.e., at compile time) **tailor interfaces**—that is, the ability to add or hide capabilities of a component's interface without changing its API or implementation?

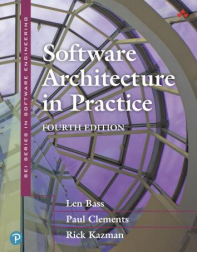
Does the system provide a **discovery service**, cataloguing and disseminating information about services?

Does the system provide a means to **configure the behavior** of components at build, initialization, or runtime?



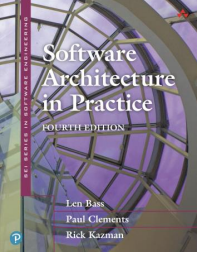
Tactics-Based Questionnaire for Integrability

Coordinate	Does the system include an **orchestration mechanism** that coordinates and manages the invocation of components so they can remain unaware of each other?
	Does the system provide a **resource manager** that governs access to computing resources?



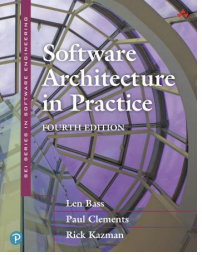
Tailor Interface Patterns for Integrability

- *Wrappers*. A wrapper is a form of encapsulation whereby some component is encased within an alternative abstraction. A wrapper is the only element allowed to use that component; every other piece of software uses the component's services by going through the wrapper. The wrapper transforms the data or control information for the component it wraps.
- *Bridges*. A bridge translates some "requires" assumptions of one arbitrary component to some "provides" assumptions of another component. The key difference between a bridge and a wrapper is that a bridge is independent of any particular component. Also, the bridge must be explicitly invoked by some external agent—possibly but not necessarily by one of the components the bridge spans.
- *Mediators*. Mediators exhibit properties of both bridges and wrappers. The major distinction between bridges and mediators, is that mediators incorporate a planning function that results in runtime determination of the translation, whereas bridges establish this translation at bridge construction time.



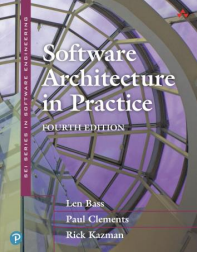
Tailor Interface Patterns Benefits

- Benefits:
  - All three patterns allow access to an element without forcing a change to the element or its interface.



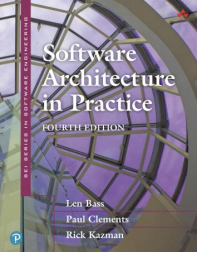
Tailor Interface Patterns Tradeoffs

- Tradeoffs:
  - Creating any of the patterns requires up-front development work.
  - All of the patterns will introduce some performance overhead while accessing the element, although typically this overhead is small.



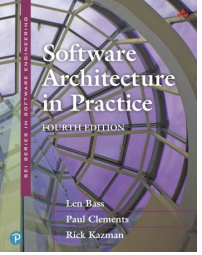
SOA Pattern for Integrability

- The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and consume services. Services are largely standalone entities: usually deployed independently, and often belong to different systems or even different organizations.
- Components have interfaces that describe the services they request and that they provide.
- A service's QAs can be specified and guaranteed with a service level agreement (SLA).
- Components perform their computations by requesting services from one another.



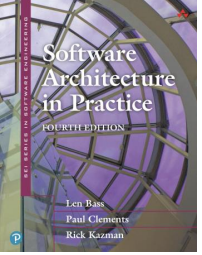
SOA Pattern Benefits

- Services are designed to be used by a variety of clients, leading them to be more generic.
- Services are independent. The only method for accessing a service is through its interface and through messages over a network.
- Services can be implemented heterogeneously, using whatever languages and technologies are most appropriate.



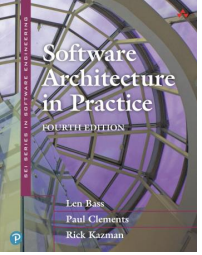
SOA Pattern Tradeoffs

- SOAs, because of their heterogeneity and distinct ownership, come with a great many interoperability features such as WSDL and SOAP. This adds complexity and overhead.



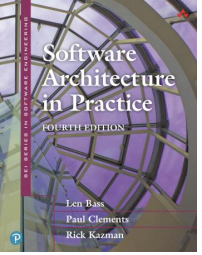
Dynamic Discovery Pattern for Integrability

- Dynamic discovery applies the *discover* tactic to enable the discovery of service providers at runtime.
- A dynamic discovery capability sets the expectation that the system will advertise the services available for integration and the information that will be available for each service.



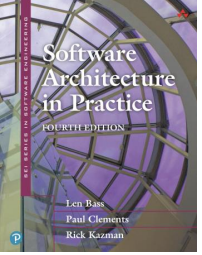
Dynamic Discovery Pattern Benefits

- This pattern allows for flexibility in binding services together into a cooperating whole. For example, services may be chosen at startup or runtime based on their pricing or availability.



Dynamic Discovery Pattern Tradeoffs

- Dynamic discovery registration and de-registration must be automated, and tools for this purpose must be acquired or generated.



Summary

- Integrability can be seen as a problem of bridging the size of and distance between the interfaces of a set of components $\{C_i\}$ and a system S .
- The tactics and patterns for integrability 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.