EXECUTIVE UPDATE
Business & Enterprise Architecture, Vol. 24, No. 9

CUTTER CONSORTIUM
Access to the Experts

The Role of Business Architecture in Shaping Software Design

by William Ulrich, Fellow, Cutter Consortium

Many organizations have found it challenging to achieve the degree of stability, maintainability, and scalability originally promised by proponents of service-oriented architecture. Software service design and deployment have a direct bearing on an organization's ability to deliver customer value and execute its overall strategy. While Web, cloud, and related software service categories remain strategic targets, underlying design challenges remain. This *Executive Update* details how organizations can utilize business architecture to inform and shape software designs to achieve more stable, maintainable, and scalable software systems.

Software Service Design Considerations

IT professionals have sought clear, concise business perspectives upon which to base software design efforts for decades.

Software design is the practice of specifying a software artifact with the intent of accomplishing a specific objective, subject to technical and business constraints. A generally accepted software design principle is that high-quality software systems are comprised of highly cohesive software services that are not interdependent on other software services. The term “software service” generally refers to any type of business, data, integration, platform, Web, cloud, or related software service category. Achieving this high degree of cohesion and loose coupling requires designing software services that implement a single, clearly defined task, which increases reusability, streamlines future enhancements, and expedites rapid deployment of new systems in less time and at less cost.

IT professionals have sought clear, concise business perspectives upon which to base software design efforts for decades. The domain-driven design community has stated that “the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not dealt with in the design, it won’t matter that the infrastructural technology is well-conceived.” The fact that most software at most organizations is underperforming, error-prone, and increasingly resistant to business demands indicates that business domain complexity is not being addressed effectively.

A typical medium-to-large organization relies on hundreds, or even thousands, of software systems to manage payment allocations, communications, customers, agreements, workflow, and other functions across its business ecosystem. Surly software designers did not intentionally set out to create impossibly complex, highly

The *Executive Update* is a publication of Cutter Consortium’s *Business & Enterprise Architecture* practice. ©2021 by Cutter Consortium, an Arthur D. Little company. All rights reserved. Unauthorized reproduction in any form, including photocopying, downloading electronic copies, posting on the Internet, image scanning, and faxing, is against the law. Reprints make an excellent training tool. For information about reprints and/or back issues of Cutter Consortium publications, call +1 781 648 8700 or email service@cutter.com. ISSN: 2470-0894.

redundant software systems that collectively undermine business strategy, customer experience, and the ability to compete. To the contrary, this situation evolved over decades, driven by siloed business communities that offered business analysts and software designers little if any consistently defined, formal representation of their business ecosystem. The degree of software reliability, adaptability, stability, and interoperability can make the difference between building a strong customer base and excelling over the competition, or having an organization fail entirely.

Addressing the Root Cause of Ineffective Software Design

Ad hoc, silo-oriented business analysis efforts repeat themselves multiple times a day, resulting in ever-increasing levels of software incongruity and redundancy.

Can software designers and developers create and deploy high-quality, cohesive, and loosely coupled software in the absence of well-articulated business abstractions? The above example and other evidence strongly suggest that this is not possible. Addressing these issues requires getting to the root cause behind them. The primary reason behind these challenges involves the lack of consistency and clarity of the business perspectives being used as input to software design efforts.

Consider, for example, that business requirements, a key software design input, tend to be crafted by small, segregated teams of business analysts and software designers that rely on isolated snapshots captured across multiple business unit silos. Ad hoc, silo-oriented business analysis efforts repeat themselves multiple times a day, resulting in ever-increasing levels of software incongruity and redundancy. Seemingly simple concepts like customer, product, partner, asset, payments, financial accounts, and more are muddled, implemented in many different ways, invariably using conflicting terms and definitions. This assortment of incongruous business perspectives lacks the consistency and clarity required by software architects, business analysts, and software designers.

One errant assumption is that all business stakeholders share the same perspective, vocabulary, and mental model; a false assumption in practice. Consider an example where a team of business analysts at one organization assumed that everyone knew what people meant when they used the term “account.” Business stakeholders within and across departments were asked to define “account,” and the variations were striking, rendering the term meaningless. Depending on who was interviewed, an account was a customer, an agreement with a customer, an identifier used to identify a customer, or a place used to track monetary balances.

The bottom line is that basing software designs on siloed, inconsistent business perspectives produces more poorly defined software that cannot scale, interoperate, and ultimately support an organization’s strategic objectives.

These siloed business modeling teams were working “in the blind” with no shared perspective, which tends to be the rule, not the exception. Assuming that an effort is not cancelled, as four of the six projects experienced in the above example, the end result degrades the ability to enhance the customer experience, respond to changing market conditions, control or lower operating costs, manage supply chains, and increase business agility. The bottom line is that basing software designs on siloed, inconsistent business perspectives produces more poorly defined software that cannot scale, interoperate, or ultimately support an organization’s strategic objectives.

Formalizing Business Abstractions

If well-defined software services are to be cohesive, loosely coupled, readily reusable, and easily maintainable, then it logically follows that the business abstractions from which those software services are derived must meet the same criteria. But as previously indicated, this historically has not been the case, leaving most organizations with bloated portfolios of software that will continually degrade their ability to deliver customer value and compete. To reverse course, analysis, design, and development teams must leverage formal, holistic representations of their business ecosystem as input to software design efforts.

One overriding factor driving the need to address these issues is software system interoperability. When systems cannot interoperate, it leads to customer continuity issues, degradation of the customer experience, bad decisions due to data replication, an increase in manual work, and the proliferation of high-risk desktop solutions. Operating costs soar, productivity drops, customer issues are ignored, regulatory compliance fails, and business risks escalate.

Basing software design and development work on highly rationalized, ecosystem-wide business abstractions delivers software systems that seamlessly interoperate, scale effectively, and are more maintainable.

Reversing these trends requires creating and provisioning business abstractions that adhere to and enable the creation of software designs that meet quality standards. To accomplish this, business abstractions must meet three criteria:

1. Business ecosystem representations must be based on formal principles that provide rationalized, consistent, nonoverlapping, and unambiguous views of the business ecosystem.

2. The scope of business abstractions must span the scope of the business ecosystem being modeled.

3. Data and solution architectures, business requirements, and software designs must leverage these business ecosystem abstractions at scale.

An example of a business ecosystem would involve a multi-divisional financial services corporation with shared customers, partners, assets, and finances. Various divisions manage shared customers associated with common markets, orders, agreements, products, financial accounts, and related perspectives. Basing software design and development work on highly rationalized, ecosystem-wide business abstractions delivers software systems that seamlessly interoperate, scale effectively, and are more maintainable. This approach, if applied systematically, will reduce software bloat and correspondingly reduced the overall costs to maintain and enhance those systems. And the need for manual work and desktop solutions will dissipate.

The idea of crafting ecosystem-wide business abstractions can send people into a panic, eliciting cries of "boiling the ocean." This would be true if ecosystem abstractions targeted operating model perspectives, which tend to explode into countless, detailed business processes and technical models that are difficult to decipher, navigate, or leverage in practice. But these operating model views are not the basis for establishing rationalized, consistent, and readily interpreted business ecosystem abstractions. To the contrary, creating readily decipherable, widely accessible business ecosystem abstractions requires applying formal principles that ensure clarity, congruity, and absolute consistency. Meeting these criteria requires organizations to articulate and leverage a formally defined business architecture.

Business Architecture's Role in Representing a Business Ecosystem

Business architecture establishes a foundation for strategy execution, but the benefits of using it as input to software design should not be underestimated.

Business architecture provides an overarching, ecosystem-wide perspective for an organization, resting upon a formal, principled foundation of capabilities, information concepts, value streams, and organizational views. It establishes a foundation for strategy execution, but the benefits of using business architecture as input to software design should not be underestimated.

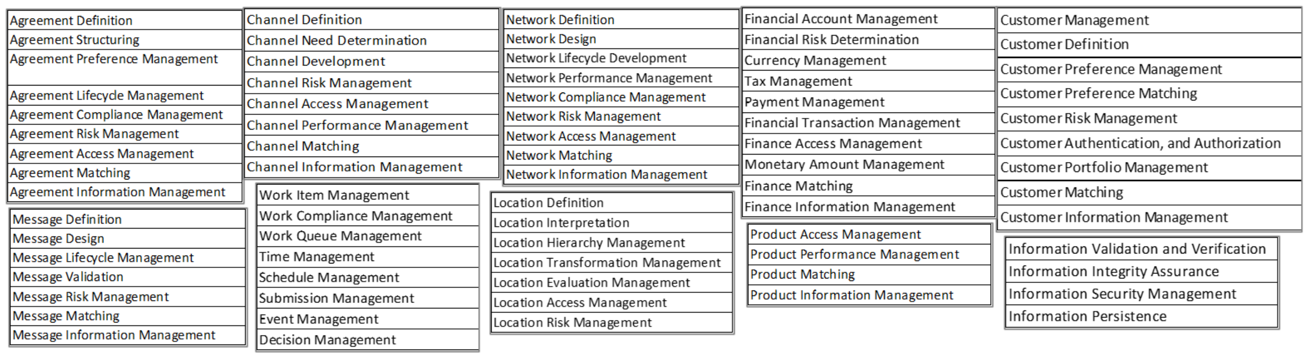
The first misconception about business architecture is that it is built over and over again or changes radically from project to project or strategy to strategy. In fact, just the opposite is true. Business architecture is built once, and while it can mature, the baseline remains largely the same as long as there is no radical departure from one's business model(s).

Business architecture serves as a long-term, cross-ecosystem abstraction that offers a way to envision an organization that is not obscured by siloed business unit constraints. At the highest level of maturity, every business strategy, plan, program, business design, innovation, requirement, data model, and software design would leverage a shared understanding of what the business does, the information it uses, the stakeholder value being delivered, and organizational structure.

Capabilities play a central role in software service design. When creating a capability map, an organization decomposes its business ecosystem into well-defined, nonoverlapping business objects and applies actions against objects. Objects are stateless, which, for example, allows an "agreement" business object to be established at the onset of its evolution, even though agreement is in a "pending" state. Similarly, a customer's state may be "prospective," eliminating the need for redundant business objects such as "opportunity" and "prospect." The concept of basing capabilities on stateless business objects ensures that an object is not replicated as it transitions through various states.

The second part of articulating and formulating capabilities involves identifying the actions that can be applied to those business objects. Figure 1 illustrates a sample set of capabilities, highlighting a subset of actions applied against selected business objects. Formal capability definitions, which are mandatory for every capability in a business ecosystem, are omitted for brevity.

The figure depicts a sample cross-section of capabilities based on clearly defined business objects that include agreement, customer, channel, message, financial account, payment, financial transaction, currency, tax, monetary amount, product, decision, and event. Actions coupled with selected business objects include performance



Agreement Definition
Agreement Structuring
Agreement Preference Management
Agreement Lifecycle Management
Agreement Compliance Management
Agreement Risk Management
Agreement Access Management
Agreement Matching
Agreement Information Management

Message Definition
Message Design
Message Lifecycle Management
Message Validation
Message Risk Management
Message Matching
Message Information Management

Channel Definition
Channel Need Determination
Channel Development
Channel Risk Management
Channel Access Management
Channel Performance Management
Channel Matching
Channel Information Management

Work Item Management
Work Compliance Management
Work Queue Management
Time Management
Schedule Management
Submission Management
Event Management
Decision Management

Network Definition
Network Design
Network Lifecycle Development
Network Performance Management
Network Compliance Management
Network Risk Management
Network Access Management
Network Matching
Network Information Management

Location Definition
Location Interpretation
Location Hierarchy Management
Location Transformation Management
Location Evaluation Management
Location Access Management
Location Risk Management

Financial Account Management
Financial Risk Determination
Currency Management
Tax Management
Payment Management
Financial Transaction Management
Finance Access Management
Monetary Amount Management
Finance Matching
Finance Information Management

Product Access Management
Product Performance Management
Product Matching
Product Information Management

Customer Management
Customer Definition
Customer Preference Management
Customer Preference Matching
Customer Risk Management
Customer Authentication, and Authorization
Customer Portfolio Management
Customer Matching
Customer Information Management

Information Validation and Verification
Information Integrity Assurance
Information Security Management
Information Persistence

Figure 1 — Selected capability examples.

management, risk management, access management, validation, compliance management, and more. The capabilities in Figure 1 primarily represent Level 1 and 2 capabilities, which, in a formal capability map, decompose to Levels 3, 4, 5, or even 6, representing increasingly detailed refinements to the actions being applied to a given business object.

While consistently defined across a business ecosystem, capabilities as realized in practice allow for unique, behavioral interpretations.

Each capability produces a unique outcome that never overlaps with another capability's outcome. For example, Customer Risk Threshold Setting would create a "risk threshold" for a customer. Outcomes contribute to stakeholder value when "cross-mapped" to value streams. Realization of a capability in a business unit or partner environment is called a capability "instance." While consistently defined across a business ecosystem, capabilities as realized in practice allow for unique, behavioral interpretations. For example, a Customer Preference Management capability may implicitly derive customer preference in one instance (e.g., customer likes cookware), while a second instance involves the explicit setting of a preference (e.g., do not share customer data).

The information map bases information concepts on the same business objects defined in the capability map. Each information concept has a definition, types, states, and cross-concept relationships. Capabilities rely on and modify information. Information concept relationships form the basis for data architecture and software solution designs because they unambiguously define a rationalized view of information across the business ecosystem (my *Update* from earlier this year goes into greater detail). Capabilities and information concepts are business unit and scenario agnostic, creating a complete representation of the business ecosystem. Business unit and scenario agnosticism mean that any business unit can rely on these business abstractions for strategic planning, impact analysis, enterprise design, program definition, data modeling, and software design.

The third essential business architecture domain is the value stream, which frames how an organization delivers customer, partner, and internal stakeholder value. Value streams are the main vehicle for framing capability-based investments, where investments may be associated with the capabilities that deliver stakeholder value. Business architecture's ability to represent a business ecosystem goes beyond capabilities, information concepts, and value streams to include stakeholder, organization, strategy, policy, product, and initiative domains. With a business architecture in place, data and solution architects, business analysts, and software designers have access to formally vetted business abstractions that serve as a basis for crafting target-state architectures, business requirements, and software designs.

Note that organizations can quickly articulate a business architecture baseline using industry reference models. Leveraging industry reference models allows a business architecture team to reduce the time it takes to establish a business architecture baseline from months to weeks, or less.

Business Architecture's Role in Software Design

> *Business architecture offers important insights into framing and defining software services and data used by those services.*

Business architecture offers important insights into framing and defining software services and data used by those services. According to a 2006 paper by Microsoft's Ulrich Homann, "Business capability mapping and service-orientation provide a new set of complementary tools that stretch the concepts of business beyond the physical corporate boundaries to include the entire value chain or ecosystem of business functions within the map." The logic behind positioning capabilities as a basis for software service design lies in the fact that they share common principles:

- **Reusable.** Capabilities are nonredundant, nonoverlapping concepts and may be reused across a business ecosystem, meaning all derived software services are nonredundant and reusable.
- **Stateless.** Business objects upon which capabilities are based are stateless, reducing replication. Software service statelessness is a design principle that enables reusability and scalability.

- **Composable.** Capabilities are organized into collections under higher-level capabilities and grouped under value streams. Software service composability facilitates service reuse in multiple solutions comprised of composed services.

- **Autonomous.** Capabilities do not rely on other capabilities, meaning they work independently. Software service autonomy reduces service dependence on execution environments and resources.

- **Defined outcome.** Every capability has a unique, nonoverlapping outcome. A software service is a logical representation of a repeatable activity that has a specified outcome.

- **Discoverable.** Capabilities are defined once for an ecosystem and available in a formally defined business architecture knowledgebase. Software services are discoverable so they may be found and reused as required.

To leverage capabilities in a software service design context, business architects must adhere strictly to formal business architecture mapping principles, specifically by creating capabilities based on stateless, nonoverlapping business objects and enforcing nonredundancy.

Figure 2 depicts the relationship between capability and software service. Each business architecture domain is identified by color in Figure 2, with business object shown in a neutral color because it is neither a business architecture domain nor a software domain.

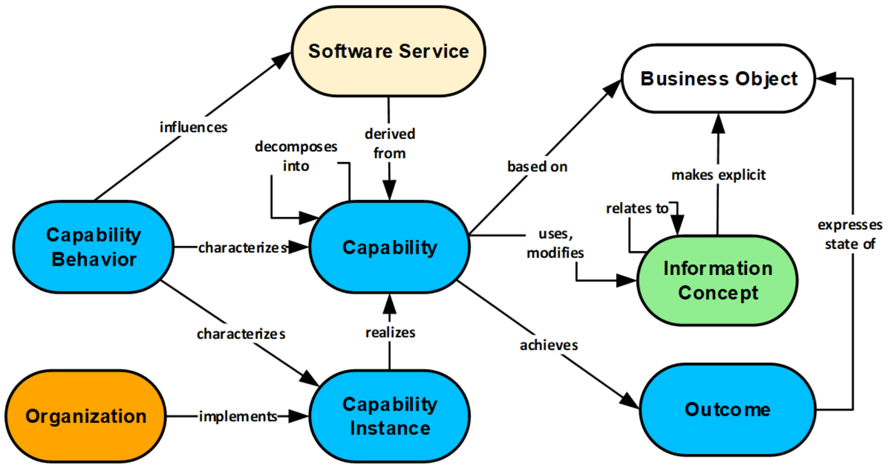


Figure 2 — Relationship between capability and software service.

Key relationships highlighted in Figure 2 are:

- **Capability** — based on a single business object, achieves a specific outcome, and decomposes into more capabilities, with each decomposition representing more granular actions against the business object
- **Capability instance** — realizes a capability in practice, occurs when implemented in a given part of that organization, and can have a unique capability behavior
- **Information concept** — makes a business object explicit including object types and states, relates to other information concepts, and is used and modified by capabilities
- **Software service** — derived from a capability and influenced by capability behavior, which may reflect unique behaviors associated with capabilities instances.

Using the previously identified Customer Preference Management capability, Figure 3 depicts an example of how the associations in Figure 2 are realized in practice. The main business object in this example is "customer," which is defined as "a legal entity that has, plans to have, or has had an agreement with the organization, or is a recipient or beneficiary of the organization's products or services."

The software service or services being designed in the Figure 3 example manage customer preferences. A sample capability definition for Customer Preference Management, a Level 2 capability under a Level 1 capability called Customer Management, is as

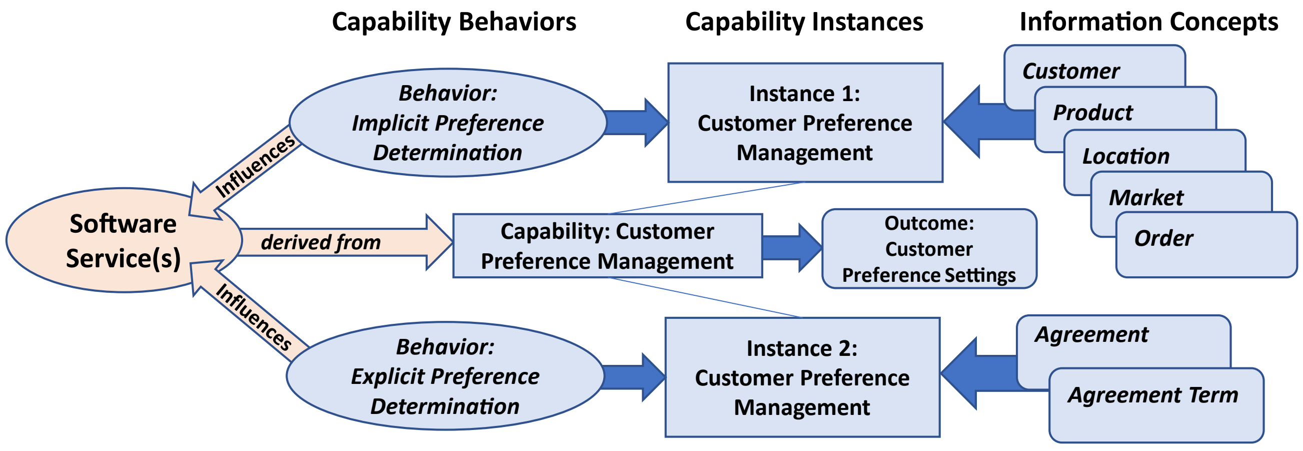

Capability Behaviors
Capability Instances
Information Concepts
Behavior: Implicit Preference Determination
Instance 1: Customer Preference Management
Customer
Product
Location
Market
Order
Influences
Software Service(s)
derived from
Capability: Customer Preference Management
Outcome: Customer Preference Settings
Influences
Behavior: Explicit Preference Determination
Instance 2: Customer Preference Management
Agreement
Agreement Term

Figure 3 — Software service derivation from capability, instances, behaviors, and outcome.

follows: “Ability to capture, represent, analyze, and act upon explicit or implicit wants, needs, inclinations, leanings, likings, predispositions, penchants, or options — expressed formally or informally — as determined or derived from a combination of sources and associated with a customer.”

The above capability definition provides insights into behavioral context, which is important in software service derivation. The overall scenario summarized in Figure 3 is detailed as follows:

- The Customer Preference Management capability is implemented in two business areas, creating two capability instances (business units were omitted to simplify the example).
- Each instance has a unique behavior where instance one derives preferences implicitly and instance two derives preferences explicitly.
- Implicit preference determination requires looking at customer, product, location, market, and order information to infer customer tastes, buying patterns, and related preferences.
- Explicit preference determination involves a customer turning certain preferences on or off.
- The capability itself uses an aggregate of the information concepts shown in Figure 3, where each instance may only require a subset.
- Software designers can determine whether they want to create a single software service with the ability to determine preferences implicitly and explicitly, or whether they want to create two software services, one for each unique behavior associated with each of the capability instances.

The Figure 3 example provides several insights for software design teams. First, a capability called Customer Preference Management uses a combination of information concepts to create a customer preference outcome. A software service automating that capability, therefore, requires data that corresponds to those information concepts to executive effectively. The capability is implemented in two business areas, creating two capability instances, each of which in this example behave uniquely to create the customer preference outcome.

Software designers using these business architecture–based business ecosystem abstractions can confidently rely on a common definition of customer and other information concepts, gain clarity of the scope of coverage for this service based on the capability instances, and design a service or services to address the unique behaviors associated with the capability instances.

The solution scales because the selected capability, information concepts, and organizational perspectives were drawn from a business architecture that maintains a principled, holistic, and formally derived abstraction of the business ecosystem as a whole. Every planning, project, design, and deployment team within this business ecosystem can confidently rely on a single, consistently defined set of business perspectives that may be used to design and deploy scalable, interoperable, and maintainable software systems.

Every planning, project, design, and deployment team within this business ecosystem can confidently rely on a single, consistently defined set of business perspectives that may be used to design and deploy scalable, interoperable, and maintainable software systems.

Software Service Derivation, Decomposition & Composability

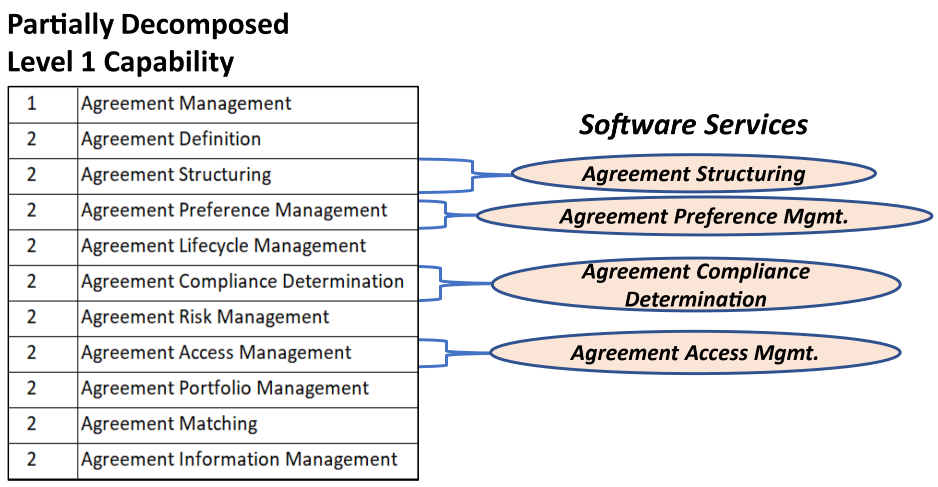
Researchers George Feuerlicht and Josip Lozina provide guidance on software service composability and decomposition:

> *During the decomposition stage complex business functions are progressively decomposed into elementary functions and then mapped to corresponding candidate service operations. This is consistent with maximizing cohesion as elementary business functions typically accomplish a single conceptual task and exhibit high levels of cohesion.*

Fortunately for software designers, capabilities already create detailed decompositions as a fundamental aspect of a principled capability map.

Formal capability decomposition in a business architecture means that what Feuerlicht and Lozina term the decomposition stage, “where complex business functions are progressively decomposed into elementary functions,” is dramatically streamlined for software design teams. Business architecture not only eliminates countless hours of business modeling work for business analysts and software designers, but also ensures that reuse, interoperability, and scalability are incorporated into software designs from planning through deployment.

With capability decomposition in place, software designers can make decisions on the degree of granularity desired in a given set of software services. Software service granularity is often driven by a multitude of factors that include governance, technical, and other considerations. Every design team will make its own decisions in consult with its enterprise architects. The important consideration is that a well-defined, principled set of capabilities, decomposed to the right levels, allows designers to design coarse-grain as well as finer-grain capabilities. For example, the Agreement Management capability frames a coarse-grain software service as shown in Figure 4. This capability embodies all actions that might apply to the agreement business object. It is high level, but perhaps appropriate for purposes of a given design team.



Partially Decomposed Level 1 Capability

1	Agreement Management
2	Agreement Definition
2	Agreement Structuring
2	Agreement Preference Management
2	Agreement Lifecycle Management
2	Agreement Compliance Determination
2	Agreement Risk Management
2	Agreement Access Management
2	Agreement Portfolio Management
2	Agreement Matching
2	Agreement Information Management

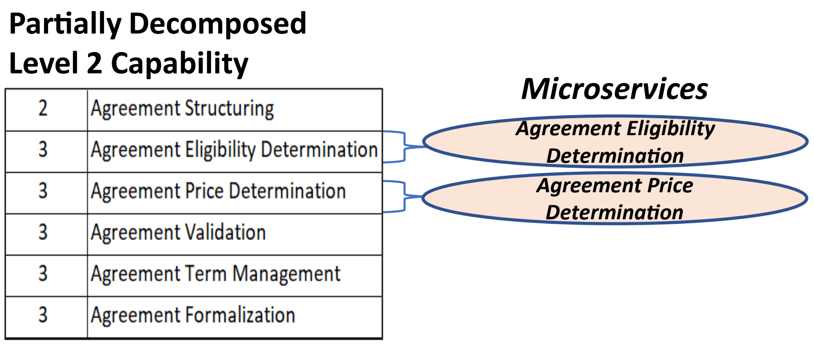
Software Services
Agreement Structuring
Agreement Preference Mgmt.
Agreement Compliance Determination
Agreement Access Mgmt.

Figure 4 — Capabilities as basis for software service design.

Figure 4 highlights how lower-level capabilities may be used to derive software services. Agreement Management capability decomposition identifies a set of Level 2 capabilities that serve as candidates for finer-grain software service definition. Figure 4 also depicts four candidate software services named after their corresponding capabilities: Agreement Structuring, Agreement Preference Management, Agreement Compliance Determination, and Agreement Access Management. Software designers are free to assign any naming convention as long as they maintain traceability to the capabilities.

Agreement Structuring as defined in this example formalizes all aspects of an agreement and brings it into executable form. Agreement Preference Management allows preferences to be implicitly inferred or explicitly set by any party with agreement access. Agreement Compliance Determination ensures that parties to the agreement adhere to agreement terms and Agreement Access Management allows or restricts access to the agreement by sentient objects, including people and intelligent assets.

Figure 5 highlights a second example, where a Level 2 capability, Agreement Structuring, is decomposed to five Level 3 capabilities, each of which forms the basis for microservice design. Formally defined, capability iterations, as shown in Figure 5, offer software designers a formal, rigorous business baseline upon which to make software service design decisions. In practice, Level 3 capabilities such as Agreement Eligibility Determination and Agreement Price Determination decompose into lower-level capabilities to provide software designers and developers deeper insights into exactly what these capabilities do.


Partially Decomposed Level 2 Capability
Microservices
Agreement Eligibility Determination
Agreement Price Determination

2	Agreement Structuring
3	Agreement Eligibility Determination
3	Agreement Price Determination
3	Agreement Validation
3	Agreement Term Management
3	Agreement Formalization

Figure 5 — Capabilities as basis for software microservice design.

When leveraging a capability map, it is important to recognize that it ideally defines what a business does in its entirety, which includes what some might people call “technical capabilities.” In reality, all capabilities are business capabilities. For example, data management is addressed through object-specific, profile, state, type, history, and matching relationship capabilities, along with generalized Information Management capabilities that address persistence, transformation, security, and other actions. A second example involves portal management, which IT often considers a technical capability. Industry reference models, however, leverage Submission Management capabilities to define all aspects of portal-related functionality. Other focal points often misconstrued as technical capabilities include Asset Management, Channel Management, Network Management, and Work Management.

Value streams play a role in software service design by providing a basis for composing capabilities into value-oriented groupings.

When an organization deploys a software service, it should ensure that an association between that software service and the capability upon which it is based is maintained by capturing it in the business architecture knowledgebase. This way, any architect, analyst, designer, or developer can check the knowledgebase first before designing or building a new software service. If the software service automating that capability already exists, developers can review it for reuse, enhance it, or consider other options. If this pattern is strictly applied, software reuse will scale, software portfolios will shrink, and software replication will recede.

Value-Oriented Software Service Design & Orchestration

Value streams play a role in software service design by providing a basis for composing capabilities into value-oriented groupings. Every organization has a fixed number of value streams in its business ecosystem that frame the entirety of external and internal stakeholder value delivery. Capability outcomes have a “value-enabling” relationship to value stream stages. Value stream/

capability cross-mapping provides a value-oriented framework for executing a wide variety of business scenarios. Value streams in practice may be implemented differently across different business units or situations, giving a good deal of latitude to service designers to determine how these compositions may be orchestrated based on the situation.

From a strategic perspective, value streams are an essential strategic planning tool for organizations and serve as the basis for capability-based planning and investing.

For example, individual finance capabilities, such as Financial Account, Financial Transaction, Payment, Monetary Amount, Tax, and Currency Management, may each be defined as a unique software service. These software services may then be composed into an aggregate service that addresses all aspects of requesting a payment through the allocation of monetary amounts into one or more financial accounts. This composite service may then be leveraged across every value stream implementation that involves billing and collection, establishing a highly scalable basis for reuse that will reduce the inconsistencies and redundancies found across organizations today.

From a strategic perspective, value streams are an essential strategic planning tool for organizations and serve as the basis for capability-based planning and investing. Strategic plans target stakeholder value delivery improvements, mapping impacts to specific value stream stages, which in turn expose value-enabling capabilities to be targeted for investment.

A Call to Action: Leveraging Business Architecture for Software Design

Organizations face many challenges in today's high-demand, complex business environments. Continuing to use siloed, incongruous, and redundant business abstractions as input to software designs will further destabilize already complex software systems and increase many of the challenges organizations already face today. Leveraging a formally defined business architecture based on rigorous principles, on the other hand, enables software design and development teams to deliver more stable, maintainable, interoperable, and scalable software systems, enabling business agility at a time when organizations need it the most. Organizations seeking to deliver more agile, robust software systems should establish a plan of action as follows:

It is time to reverse a multi-decade pattern of proliferating software systems that cannot communicate with other systems, scale poorly, cost too much, take too long to maintain, and undermine business strategy.

- Raise awareness of the business challenges associated with increasingly problematic software systems, along with the root cause of those issues.
- When raising awareness, draw from internal examples of software failures and the business issues that result.
- Position messaging from a business perspective, highlighting the business risks of degrading software systems versus the value delivered by stable, maintainable, and scalable systems.
- Communicate the need to establish an end-to-end strategy execution perspective built on a formal business architecture.
- Formally map the business ecosystem leveraging formal business architecture principles and industry reference models as a means of expediting use of the discipline.
- Work with solution architects, data architects, business analysts, and software designers to communicate and formalize an approach for leveraging business architecture across disciplines.

Using business architecture as input to software design efforts not only delivers more effective, business-aligned software systems, but eliminates much of the time development teams spend repeatedly abstracting and trying to interpret ad hoc, often inconsistent business perspectives across business unit siloes. It is time to reverse a multi-decade pattern of proliferating software systems that cannot communicate with other systems, scale poorly, cost too much, take too long to maintain, and undermine business strategy.

About the Author

William M. Ulrich is a Fellow of Cutter Consortium's Business & Enterprise Architecture practice, President of TSG, Inc., and cofounder of the Business Architecture Guild. He is a thought leader in the fields of strategy execution, business architecture, and business-driven IT transformation, and serves as senior advisor, mentor, and workshop leader to major corporations and governments. Mr. Ulrich contributed to the formation of the Business Architecture Guild's business architecture framework and de facto approach used worldwide. Active in the international standards community, he has contributed to a wide range of global business and IT standards. Mr. Ulrich is coauthor of *Business Architecture: The Art and Practice of Business Transformation*, *Information Systems Transformation*, and several other books and publications. He can be reached at experts@cutter.com.

About Cutter Consortium