PART II

SUPPORTING THE IT VALUE CHAIN

> The customer and partner community is communicating quite clearly… that what they're looking for is an *integrated family of applications* that minimize their cost structures going forward.
>
> —Tom Siebel
> CEO, Siebel Systems[131]

Having completed an examination of the enterprise IT value chain, let's turn to the question of how to support it in an integrated and comprehensive manner. Currently, IT as a value chain is supported by fragmented data and processes, contained within functional silos, suffering from much redundancy and lack of integrity.

As Roger Burlton notes,

> It's one thing to have enterprise information available whenever you want it. It's something else for that information to have integrity. *Integrity* for data management purposes means that, if the information is redundant, it must be consistent…. [This is] key if you want to avoid different people making decisions based on information that varies but that should be the same.[132]

Because of this requirement for data consistency, the architectural analysis then moves to a data model. Enterprise IT can be represented by a comprehensible set of information concepts, which I will detail and then cross-reference to the process model.

The cross-referencing among process, data, and system will be further explored in terms of pattern analysis. If you don't know what a pattern is, read on!

Discussing the large-scale technical enablement of IT immediately brings up the question of other large-scale applications, such as those used by human resources, finance, customer relationship management, and supply chain organizations. The overall term for such applications is "enterprise resource planning," a concept now relevant to IT itself.

Enterprise Resource Planning?

> Companies that attempted to install ERP encountered grave difficulties, for they were unprepared for the shifts in jobs and power that focusing on end-to-end processes entailed. Companies that managed their installation in terms of process change rather than software were far more successful.
>
> —Michael Hammer[133]

There are differing representations of what the major enterprise "resources" are, but one reasonable version is the following:

- Liquid capital
- Fixed (productive) capital
- Stock of goods
- People

However, as Peter Drucker noted, a distinguishing feature of the 20th century was the emergence of knowledge (information) as a resource to be exploited in and of itself, with emergent properties at scale requiring significant experience, specialization, and infrastructure for support.[134] Hence the emergence of information as a first-class enterprise resource.[135] Treating information in this respect parallels the evolution of other enterprise resources; a small company may not need a dedicated human resources or supply chain system, but as the company grows, the pressures for increasing professionalization of their management grow.

When the $100 million company becomes a $30 billion enterprise with hundreds of distinct systems and thousands of servers, informal understanding becomes impossible.

The same is true of information, albeit at a larger scale: a $100 million company might have an IT organization of a couple dozen staff members, who understand the system architectures through experience. However, when that $100 million company becomes a $500 million or $30 billion firm with hundreds of distinct systems and thousands of servers, informal understanding becomes impossible and the IT capability requires its own management infrastructure, with capabilities analogous to the ERP systems found for other major value chains and functional areas.

The original manufacturing resource planning (MRP) systems focused on core manufacturing functions such as materials management and work planning. Their comprehensive approach expanded and evolved through MRP II and the first-generation ERP systems; ERP, circa 2006, generally means large-scale systems that support multiple large, complex business processes for entire enterprises, processes such as human resource management, manufacturing logistics and supply chain, and financial management, as well as next-generation resource management for customer relationships, intellectual property, and information and its technological infrastructure.

Enterprise Resource Planning for IT?

As I was starting to make sense of what metadata management might mean for an integration competency center, an enterprise architect approached me and asked, "What's the difference between an ITIL configuration management database and a metadata repository"? I pondered this for some time, and then saw (General Motors CIO) Ralph Szygenda's call for "ERP for IT." That gave me the answer—there is no essential difference. They are two attempts at answering the same problems of enterprise IT.

IT as a general organization capability, just like its counterparts in Finance or Human Resources or Manufacturing, has processes and data elements it needs to manage. It manages the definition of process, data, and system architectures; the creation and operation of physical data and software artifacts implementing them; hardware computing platforms supporting those artifacts; and process concepts: change and incident tickets, work orders, services and systems as cooperatively defined with the client, and more. It also manages the human and financial resources necessary to support the IT capability.

If this seems misguided, consider some questions.

The definition of an "entity" would generally be accepted as "metadata." Is the name of the analyst who defined that entity metadata? Many tools can and do store it. What about the project in which the model was created? What about the financials underlying that project? The software quality practices? Were inspections carried out on the data model?

A logical evolutionary step for metadata repositories was to extend their data dictionaries to include the programs that accessed the various data elements. But are the change tickets that put those programs into production metadata? Are incident tickets related to those programs metadata? Is the headcount and budget required to maintain the system in production ongoing?

(continued)

Hence my argument that it's all really just "ERP for IT" (a.k.a. IT resource planning, IT business management, or integrated IT management). The acronym ITRP, for IT resource planning, will be used.

The history of ERP systems is not distinguished.

ERP is used evocatively and provocatively in this book. Reality check: the reputation of real-world ERP systems is not distinguished. There have been widely publicized failures of implementation and acceptance.

Secondarily, ERP systems have a poor technical reputation; earlier versions traced their lineage back to mainframe, flat file–based systems with intricate, proprietary, and obscure architectures. Their monolithic architectures have proved inflexible and costly to upgrade. Such a platform would have serious challenges in supporting internal IT business processes, which depend on complex data structures requiring state-of-the-art infrastructure and are quite varied in their interactions.

However, it clearly has been an advance to have one system covering accounts payable, accounts receivable, payroll, and general ledger, where previously those systems might have been separate and joined by inefficient interfaces.

The major IT process areas produce and consume common data and suffer greatly when no system of record exists.

IT governance presents similar challenges: even thought the track record of ERP systems has not been stellar, there seems little alternative. As will be detailed in the following analysis, the major IT process areas produce and consume common data and suffer greatly when no system of record exists. Unification is the challenge of the day.

Component-based architectures (most recently SOA) are an alternative paradigm, holding the promise of loose coupling and easier interoperability among systems; however, this paradigm is still emerging. Standards-based integration of loosely coupled ITRP systems would be ideal but may be difficult to achieve given the momentum of more traditional enterprise application software approaches and the immaturity of SOA.

Tail Chasing

Chris: I was just talking to one of your consultants and he was recommending we put in place a portfolio management tool. It seems to me that such a tool is an application that will manage applications.

Kelly: Right. It's kind of like metadata, which is data about your data.

Chris: My head is hurting.

Kelly: And one of the services in your service catalog is "service creation and management." It's the service of managing services!

Chris: I guess I shouldn't get too bent out of shape; after all, the Human Resources people have to manage their own staff, and Finance gets a budget for its own operations. But it still seems like a hall of mirrors.

Kelly: Welcome to IT. We're going to talk about the requirements of requirements management, by the way. And don't forget your new Six Sigma initiative. You realize, of course, that the essence of Six Sigma is a process that manages processes?

chapter 3

A Supporting Data Architecture

I would not give a fig for the simplicity this side of complexity, but I would give my life for the simplicity on the other side of complexity.

—Oliver Wendell Holmes[136]

3.1 Metrics: Gateway from Process to Data

The metrics-based management control of processes requires carefully and clearly structured data.

AS YOU HAVE SEEN, PROCESS-CENTRIC THINKING is a hallmark of modern business practices.

COBIT, CMM, and ITIL, at their base, are process frameworks. They focus on overall functional capabilities and the sequences of activities (business processes) that *add value for the customer* (internal or external).

Business processes require optimization, and to optimize them they must be measured. The concept of metrics management is essential to process improvement frameworks such as Six Sigma. Processes are controlled by metrics.

But what is a metric? A metric is a measurement. It is information, not activity—information that *drives* activity.

What is information? Information is actionable, context-relevant data. So, metrics at their base are data.

This brings us nicely to the next major architectural view: data. When architecting systems (defined as combinations of people, process, and technology) the concept of "data" is critical. The frameworks *imply* shared data, but they do not go far in discussing its implications, which are significant.

Processes are controlled by metrics, and metrics are based on data.

In reading the major frameworks as requirements specifications, the need for a consistent data architecture emerges; however, to date the major frameworks have been circumspect about this reality, consigning it to some unspecified other forum (which defaults to the intellectual property of consulting firms or vendor application products).

Trends are always carried to the extreme, and BPM is no exception. One of the unfortunate extremes of BPM thinking (an extreme not represented by its careful thought leaders but evident among some practitioners) is the idea that process is everything and data is nothing, or is some mere technicality whose consideration can be deferred to the developers.

Data has been either absent or, at best, a second-class citizen in much of the IT governance literature and frameworks.

The consequences of this are clear from an enterprise architecture perspective: processes can't be fully optimized, because the "things" that the processes are managing are still unclear to the process stakeholders. In many cases redundancy is the result: two processes may be managing the same thing but calling it by two different names. Or—for example, with the broad ITIL concepts of Configuration Item and Change—different things may be lumped inappropriately together in a given process context.

Without a product-independent data perspective, ITRP and its implementers will be hostage to product vendors.

This is compounded by the current vendor landscape, in which many vendors are selling overlapping products that refer to the same logical concepts with different terminology—sometimes, this appears to be a deliberate strategy to create the illusion of product differentiation where none exists. Without a sound, product-independent data perspective, ITRP and its implementers will be hostage to product vendors.

Of the views this book takes—process and function, data, and system—data is the most precise. Even at the verbal, conceptual level, it provides the basis for system interoperability, business rules, and application design. The data necessary to the domain of IT management is a fascinating topic. It's not impossible, as some seem to feel—the data structures needed to run IT, although tricky in some ways, are comparable in number and complexity to other functional areas.

IT Metrics

Business processes require metrics, and at the most general and abstract the use of metrics to assess and guide process is called "performance management" or "business performance management" (sometimes abbreviated BPM and confused with business process management).

There is no established suite of IT metrics comparable to standard financial measures of corporate performance.

A hierarchical metrics structure is characteristic of performance management and the business intelligence methods supporting it. The hierarchy of metrics may progress from simple operational reporting to complex, derived leading indicators. Such approaches have become well established in many types of business activities, and attention is now turning to measuring IT similarly. Unfortunately, there is no established suite of IT metrics comparable to standard financial measures of corporate performance. This is an area of activity for a number of standards bodies, academics, and other players.[137]

Business intelligence software at its most sophisticated provides robust support for building expressions based on metrics. Before investing in a limited-function service-level management tool, it may be worthwhile considering this as a special case of a business intelligence problem and handle it through standard business intelligence or data warehousing techniques.

Both ITIL and COBIT have extensive coverage of metrics, which this book will not replicate. However, consider a couple of examples from those frameworks.

Example 1: Change Management. ITIL, in the "Change Management" section, calls for tracking the number of Incidents traced to Changes. This implies the existence of separate Incident and Change data entities, which can be joined together and summarized to derive the counts. This is a clear statement of data requirements.

Example 2: Technology Obsolescence. COBIT, in the Acquire and Maintain Technology Infrastructure Control Objective, calls for the metric "# of critical business processes supported by obsolete (or soon to be) infrastructure."[138] This implies the existence of business process and technology entities. The technology entity would require a life cycle state or obsolescence attribute of some kind (implying, in turn, a process for maintaining this information). A good data architect will also question whether business processes should be tied directly to technology platforms or tied first to IT services, which are then dependent on technologies.

This book's data model was derived through just such systematic consideration of ITIL, COBIT, and other industry literature serving as requirements statements.

IT performance measurement is a nontrivial and evolving field; refer to the references and footnotes for discussion of specific IT metrics. The discussion here is focused on the architectural requirements of metrics management generally, including their basis on clean, normalized, well-architected data (an aspect overlooked in most discussions).

Rollups and Dimensions

Most reporting is characterized by a requirement to summarize detailed data along standard hierarchies. In data warehousing, such standard hierarchies are known as "dimensions." In the ITSM and IT governance space, these common dimensions include the following:

- Organizational hierarchy
- Organizational and IT strategic goals
- Application portfolio (rolling up into both the organizational hierarchy and the service-level agreements, or SLAs)
- Service (in the ITSM sense, if separate from application)
- Program or project portfolio
- Data subject areas (hierarchical, not relational)
- SLAs
- Enterprise calendar
- Enterprise operational locations and hierarchy (e.g., District and Region)

Intractable process and political difficulties may emerge in the search for standardized reference data.

Some of these will be well understood by the enterprise's data warehousing group (e.g., calendar and location); others will be new ground. A well-known challenge in data warehousing is "nonconformed dimensions," that is, dimensions that are not in synch across different systems—for example, different calendars in use by different lines of business. Intractable process and political difficulties may emerge in the search for conformance. Generally, the dimensions should be managed by the IT portfolio and architecture processes (preferably with Data Management guidance), and the facts should be managed by the operational processes.

The application portfolio is perhaps the most important and difficult. Organizations may have 10 or more different lists of applications, causing no end of confusion around what IT is doing and who owns it.

The project portfolio can also be a source of pain. As with systems, people tend to refer to projects by myriad imprecise names. What is the system of record for projects? Does every system that references a project do so using an unambiguous project identifier or picklist derived from the system of record? Or is just the project name casually typed in with no check for accuracy?

Another problem is the challenging topic of "slowly changing dimensions." Suppose incidents are rolling up by application portfolio and organizational hierarchy, with trending reports over the years. Then reorganization happens. How should

this be handled? There are three approaches, all with pros and cons that need to be understood in depth.[139]

A related matter is the establishment of common IT reference data (generally termed "master data management," or MDM in the industry). Dimension conformance is an issue of MDM, but MDM is somewhat broader in implication (e.g., Server might not be a true dimension, but ensuring that there is an accurate single list is an MDM problem). See Figure 4.26 and the related discussion.

The metrics-based management control of processes requires carefully and clearly structured data.

Generally, the metrics-based management control of processes requires carefully and clearly structured data. This chapter thus turns to a detailed examination of the conceptual data structures involved in IT management.

3.2 A Conceptual Data Model

> Data modeling is arguably the most widely used technique in modern systems analysis and design, but it isn't always used well. Too often, technically oriented "modelers" jump straight into excruciating detail, dense jargon, and complex graphics, incomprehensible to process-oriented participants and other mere mortals.
>
> The root problem is a misconception—data modeling has been equated with database design. That's like equating architecture with the drafting of construction blueprints. Of course, the architect's work will eventually lead to precise, detailed blueprints, just as the data modeler's work will eventually lead to precise, detailed database designs, but…it can't start there, or the subject-matter experts will soon mentally "check out." Without their participation, the data model won't be a useful and accurate description of their business. And that's exactly how a data model should be regarded—*not as a database design, but as a description of a business.*
>
> —Alec Sharp[140]

Data Model Discussion

Chris: One thing that has us all puzzled is exactly how the ITRP concepts fit together and work with other non-ITSM concepts. There's a lot of terminology, and it seems like things overlap sometimes. For example, what is the relationship

(continued)

between a Configuration Item and an Asset? Also, some of what ITIL calls for is not exactly how we do business. Do *all* Configuration Items go through our data center change control process? What is the relationship between a Service Request and an Incident? Is a Service a Configuration Item? Is a Service Offering? Are Applications Services?

Kelly: That's why we're going to turn to one of the most important aspects of enterprise architecture: the creation of a conceptual data model.

Chris: A conceptual data model? What good is that? We're probably not going to build anything—we're going to purchase products. Sounds pretty technical.

Kelly: That's why I call it a *conceptual* data model, and yes, it's relevant even if you are purchasing products. There are a lot of vendors out there selling various flavors of IT enablement and IT governance tools, and they have a lot of overlap between their products, often with slightly different terminology.

A conceptual data model is not technical—it's about *clarifying the language* describing our problem domain so that we understand exactly what we mean by a Configuration Item and how it might relate to a Service. And this is something you need to put together independent of the products—because it's going to be your road map that helps you determine *what* products you need.

Chris: Will it help me translate the vendor-speak?

Kelly: Absolutely. One vendor may have a "service catalog entry" and an "order," and another vendor may call the same two things a "template" and a "service instance" In the conceptual data model (also called a "reference model"), they are Service Offering and Service. It doesn't matter what the vendors call them, but you need to understand that any service request management solution should have both concepts. Doing the data model helps us understand our requirements better and communicate them to the vendor.

How do we gain more precision around hard-to-define concepts like Change or Configuration Item.

How do we gain more precision around hard-to-define concepts like Change or Configuration Item? One technique used for many years is an "entity relationship model." Other (not necessarily synonymous) terms used in this general area are "conceptual model," "logical model," "domain model," "ontology," and "class model."

An entity relationship model helps clarify language by relating concepts together in certain ways:

- A Configuration Item may have many Changes applied to it, and a Change may be applied to multiple Configuration Items (many to many).

- A Machine may be related to one and only one Asset, and an Asset may be related to one and only one Machine (one to one).
- A Configuration Item may be a Service, Process, or Application (subtyping).

These relationships are visually represented as shown in Figure 3.1.[141]

Some data modeling methodologists emphasize naming the relationships (typically with a verb phrase such as "is a part of"), but others do not see this as critical, and this book does not systematically do this.[142]

Using these tools, we can start to carefully structure the relationships between the various loosely used terms of IT governance (Figure 3.2).

Vive La Difference

Your Organization's concepts and terminology will be different. Count on it. This does not make either your Organization or this model right or wrong. The point is to start asking the questions: Why does the model call for two concepts when we use one? One concept where we use two? Do we have any ability to relate concept A to B as the model calls for? Do we need it? Why do we relate X to Y when the model doesn't?

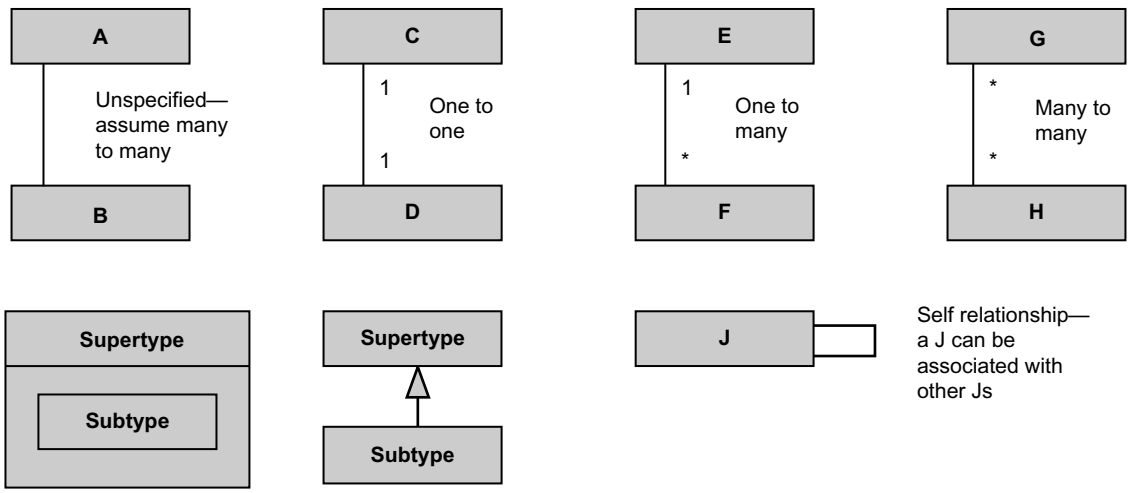


Figure 3.1 Data modeling key.

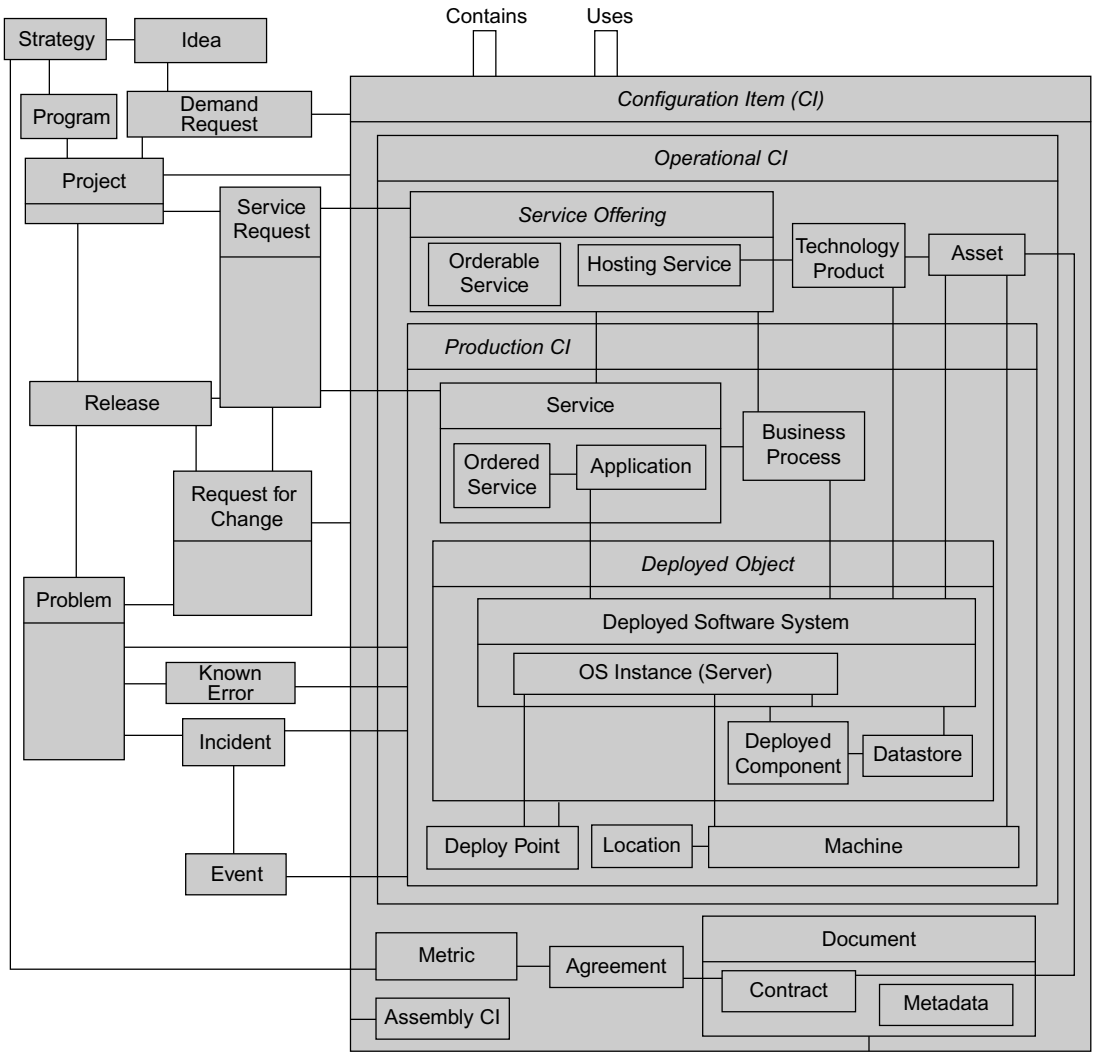


Figure 3.2 IT enablement conceptual model.

Overwhelmed?

If this model looks overwhelming to you, you might want to review the section later in this chapter titled "An Iterative and Incremental Approach to Configuration Data Maturation."

Pictures such as this only tell part of the story. They require a detailed discussion of each box (or entity), what it means, and how to interpret the lines (relationships) to the other boxes.

A conceptual data model refines language and concepts. It's not technical.

Figure 3.2 is a conceptual data model. It is primarily about refining language and concepts. The goal of this model is not technical precision but rather resonance with common industry usages, which overlap and are not well delineated. It's an attempt to push common usage toward more rigor and admittedly encounters a number of problems in this effort.

It also deliberately omits a number of details that would be necessary to realize a solution. Attributes obviously are not included (e.g., Serial Number on Machine or Date Signed on Contract). Omitted entities are generally intersection entities and dependent entities that elaborate on the core concepts. Some notes on possible approaches for elaborating this into a full logical data model are covered in the data definitions.

Building a model such as this for an industry that is not yet mature in its process best practices and terminology is challenging. The relationships among entities such as Release, RFC, Service Request, Project, and Configuration Item might have many permutations. This is a reference model, presented as a starting point for your own analysis. Reasonable professionals may come to different conclusions about which entities should be related to which.

This picture, technically speaking, is not the model. *It is only one view on the model.* One characteristic of a good conceptual data model is that its central concepts can be represented with a one-page view; there are always more details to add. Thus, in the subsequent sections other entities will appear, along with relationships not drawn in Figure 3.2.

It's All about the Language

Chris: Wow. What a picture. I'm getting a little glassy eyed.

Kelly: That's OK. Just take it a couple boxes at a time, and here are some useful reminders:

First, it's all about the language. This picture is a long way from anything we're going to build; it's here to help us understand how our project, incident, change, monitoring, configuration, and service management systems relate.

Second, there's a trick to reading the lines. Where you see an arrow or a box inside a box you should read it as "is a." For example, an Application *is a* Deployable Object. Where you see a number or star on either end, then you can read it as "has" or "is associated with." For example, an Application *has* Components, or an Asset is *associated* with a Machine.

Chris: That makes it easier. It's still pretty complicated though!

Kelly: Well, let's go through it in some detail.

3.3 IT Process Entities

This section is concerned with the IT entities that are *not* configuration items. Generally, all conceptual entities that are not configuration items can be thought of as "IT process entities."

We start with the first subgrouping, Strategy, and related entities.

Strategy

A Strategy is a top-level organizational direction or guidance toward the overall mission. The term Strategy is used generically here and might include concepts such as mission, goal, and objective detailed into a more concrete framework.

Strategies have two avenues into lower-level IT data: they drive Programs and Projects to implement new functionality, and they require the support of Business Processes to achieve ongoing success. (Notice that for graphical simplicity the Strategy–Business Process and Release–Configuration Item links were not drawn in the main data model in Figure 3.2 and appear as thinner lines. There will be other cases of such omissions.)

Strategies are related to other Strategies (this is the meaning of the "U"-shaped line on the left side of the Strategy entity).

Strategies should be measurable using Metrics; this relationship is critical to the establishment of digital dashboards.

Program

A Program is an ongoing, large-scale organizational commitment and corresponding investment toward meeting a major goal or objective of the enterprise. A Program typically consists of one or more Projects.

Idea

An Idea is an initial, typically business-generated, opportunity for IT services. It is minimally qualified.

Demand Request

An Idea becomes a Demand Request after going through some form of IT assessment for sizing or capacity impacts and preliminary feasibility. A Demand Request is a

fully qualified request for an IT service change, awaiting full funding authorization to become a Project.

Project

A Project is a defined set of manageable activities to achieve a well-specified mission (e.g., Demand Request fulfillment), usually represented by some set of deliverables or enumerated changes, with explicitly allocated resources (time, money, staff), executed and measured within the scope of those resources. A Project has one or more Releases (see the "Release" section). Projects in many cases are constrained to a fiscal year. A Project should always be associated to a Demand Request.

Projects may be non-IT (e.g., construction projects), but that usage is out of scope for this book.

A Project before it is approved may be considered a Demand Request.

Projects relate to Configuration Items either directly or (more rigorously) through defined, named Releases. This ambiguity can be seen in Figure 3.3.

Projects may be grouped into larger Programs (not represented in the model). A Program is an ongoing, large-scale organizational commitment and corresponding investment toward meeting a major goal or objective of the enterprise. A Program typically consists of one or more Projects.

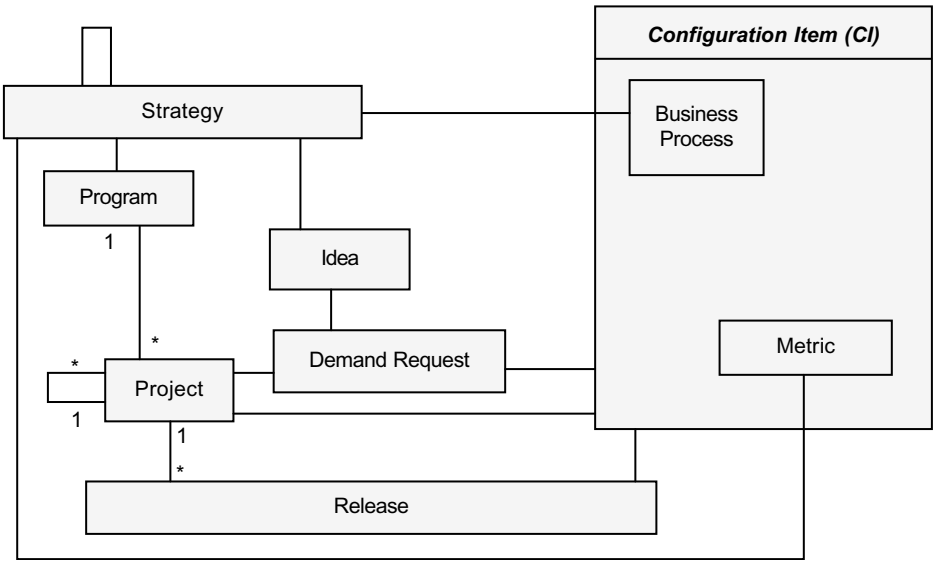


Figure 3.3 Strategy and related entities.

Strategy–Program–Project versus Idea–Demand

The model graphically depicts two competing paradigms: one from the top down, the other from the bottom up. A traditional top-down IT planning model would state that Strategy drives Program drives Project. However (especially when executed using an annual time frame), this is not an agile method for responsive IT. An event-driven, business-responsive demand process is also necessary. Aligning these two paradigms will be a different exercise for every organization; commonly, Demand Requests are evaluated against the annual strategy baseline.

(Request for) Change

A Change is a work order or authorization to alter the state of some Configuration Item.

A Change is an authorization to alter the state of some Configuration Item. ITIL defines Change as follows:

> The addition, modification or removal of approved, supported or baselined hardware, network, software, application, environment, system, desktop build or associated documentation.

It defines request for change (RFC) as follows:

> Form, or screen, used to record details of a request for a Change to any CI within an infrastructure or to procedures and items associated with the infrastructure.[143]

There is much additional discussion of Change in ITIL. However, the scope of Change in this framework is somewhat more limited; business-driven RFCs are Demand Requests.

This model does not distinguish between Changes and RFCs. However, an operational configuration management tool may detect unapproved Changes for which there are no RFCs; these can be considered Events and potentially Incidents.

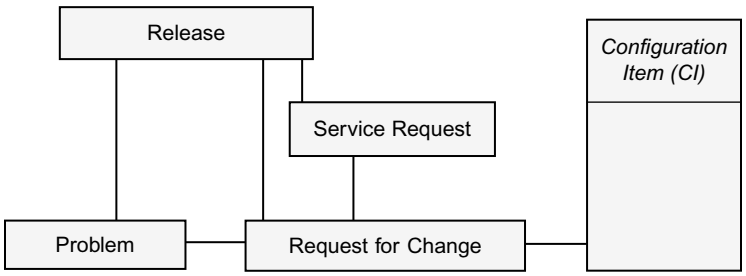


Figure 3.4 Change and Release context.

Change as Transaction

On a more architectural level, Changes can be analyzed using principles of transaction processing as a useful metaphor.[144] Changes, like transactional logical units of work, should have the following characteristics:

- Atomic
- Consistent
- Isolated
- Durable

In the context of enterprise IT, an *atomic* Change is "all or nothing"; either the Change is successfully applied or it is rolled back completely. If a Change has some part that would be rolled back and another part would stay, it should be framed as two Changes. ITIL does allow "partial rollback" but clearly indicates this is not preferred.[145]

A *consistent* Change means that the change, when deployed, leaves the item in a stable state. Characteristics no longer needed by the new version of the item should be removed as part of the change. New functionality should integrate seamlessly with the previous functionality without an undesired or unexpected effect. Any temporary states during the Change that deviate from normal practice are removed (e.g., temporary copies or parallel execution).

An *isolated* Change means (in theory) that it can go in without affecting other changes or item functionality, and is not affected by other concurrent changes. This would be hard to achieve in all cases but is nevertheless something to strive for. Achieving logical isolation of Changes is a goal for an integrated Release and Change Management process.

A *durable* Change is one that, once executed, is stable and permanent. For example, all instances of the new software in all deployment locations persist, and older software is not inadvertently reinstalled (e.g., during a system restoration process). This example requires attention to the Definitive Software Library.

Change–Configuration Item

This is perhaps the most important relationship in all of ITSM. Simply, a Change by definition affects configuration items (CIs), and CIs are objects under change control. This is far simpler to state and to model than to execute in the real world. A naïve approach to implementing this concept will result in unmanageable data. Clearly, it is not optimal for a Change record to have to be related to 1500 individual

CIs, yet this is what a simplistic approach will arrive at (e.g., in putting in an initial release of a software package with many separate binary assets).

Whether or not to inventory all binary software components in the CMDB is an important decision.

There are various techniques for mitigating and simplifying this, mostly involving encapsulation and abstraction. If a logical Application CI is defined, for example, it can be presumed to include all lower-level physical binary Components. Whether or not to inventory those binaries in the CMDB is one of the most critical decisions the ITSM implementer faces. For high security organizations this may be done, but it is questionable whether lower-criticality information systems organizations truly require it, especially in a world of purchased software where the physical architecture of a software product is less and less of a concern for the package vendor's customers.

Alternatively, the concept of assembly CI (which is also a CI) can be used. An Application plus its Datastores and Deploy Points might be a logical assembly CI. This is where the issue of Logical versus Physical CI comes in, pointing up the importance of having a defined process for maintaining logical Applications and related assembly CIs. It is *not recommended* to allow individuals the ability to create high-visibility logical CIs; this results in a chaotic environment. Everyone must *agree* that there is one Application (e.g., Quadrex), composed of, for example, these 50 Components.

Change–Service Request

Changes may require a Service Request to implement, for example, if database administration services are part of the service catalog and the addition of a new table is handled as a Service Request. This will depend on the maturity of the IT organization.

Change–Release

Changes are tied to Releases. In this framework, a Release is typically associated with a Project and results in one or more RFCs to add or alter CIs for a given IT service.

Production Change and the Software Development Life Cycle

RFCs in this architecture, and the concept of Change generally, are not applied to project deliverables. This is in keeping with the ITIL philosophy that "changes to any components that are under the control of an applications development project—for example, applications software, documentation or procedures—do not come under Change Management but would be subject to project Change Management procedures…. [The] Change Management process manages Changes to the day-to-day operation of the business. It is no substitute for the organisation-wide use of methods…to manage and control projects."[146] While the project

change management concepts are similar, they are managed in a project context that is quite different from production operations and out of scope for this book because they are extensively covered in the project management literature.

Release

Release is the gateway from the software development life cycle into the ITSM world.

In this framework, a Release is the gateway from the software development life cycle into the ITSM world. It is one of the most important concepts for which to develop an enterprise approach. A Release is (if narrowly defined) a distinct package of new or changed functionality deployed to production, usually enabling new capabilities and/or addressing known Problems.

ITIL says "a Release should be under Change Management and may consist of any combination of hardware, software, firmware and document CIs…. The term 'Release' is used to describe a collection of authorised Changes to an IT service."[147]

Releases, like Changes, should be transactional, although their larger grain makes this more challenging.

The concept of assembly CI may be helpful in supporting a Release's various elements. However, some consider a Release to primarily be a dependent entity of an Application.

Note that release management as an overall capability includes planning and harmonizing all Releases in the environment, not just managing Releases for an individual Project or Program (the enterprise release managers should interface with the program or project release managers).

Project–Release

The relationship between Project and Release can work two ways: a Project may have several (smaller-grained) Releases, and a large-grained enterprise Release may coordinate across multiple Projects. This flexibility of interpretation, coupled with narrower and broader scopes for Release, make it a particularly difficult concept from a conceptual modeling perspective.

Change–Release

A Release may have a number of Changes associated with it, but a Change should be "owned by" only one Release. That is to say, two different Releases should not be cited as justification for one Change. (See the "Justify Change" pattern in Chapter 5.)

A Release usually affects multiple CIs; however, CIs can be grouped, as with the assembly CI.

Project, Release, and Change

The ITIL conception of the relationship between Project, Release, and Change is presented in Figure 3.5.

In ITIL terms, an RFC precedes the establishment of a Project.

Note that in ITIL terms, an RFC precedes the establishment of a Project, in theory.[148] The Release might also result in smaller-grained RFCs for change control (e.g., actual physical deployments); thus, there is a conceptual difficulty in distinguishing Change granularity, which ITIL calls out as a risk[149] but does not present a systematic framework for resolving.

This may be problematic in terms of language and culture for organizations with a strong tradition of change control, possibly including a function named Change Management. They will not want their process (and system) "contaminated" with RFCs more focused on Project initiation; a forward schedule of change is as far as they may wish to go.

An alternate view is presented in Figure 3.6.

The controversy is primarily linguistic. The ITIL intent behind front-loading the RFC is presumably so that it is suitably assessed by all stakeholders. This is also the objective of the demand and portfolio management processes (as well as the function of enterprise architecture), and there is arguably more maturity in their conceptions of how to do this.[150]

Whether you subscribe to the ITIL view or this book's framework, these issues should be clarified in any large IT organization.[151]

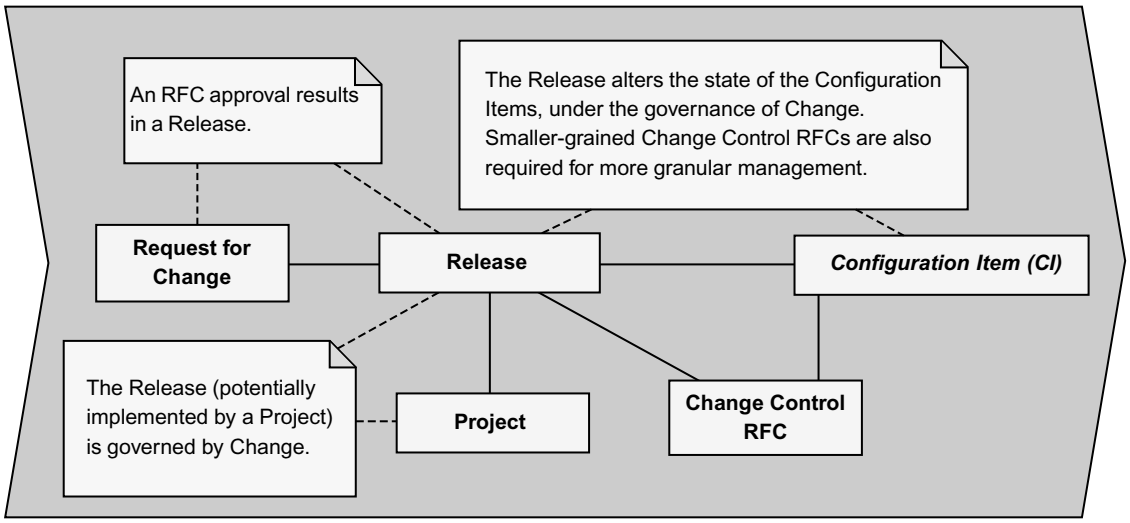


Figure 3.5 ITIL representation of RFC, Project, and Release.

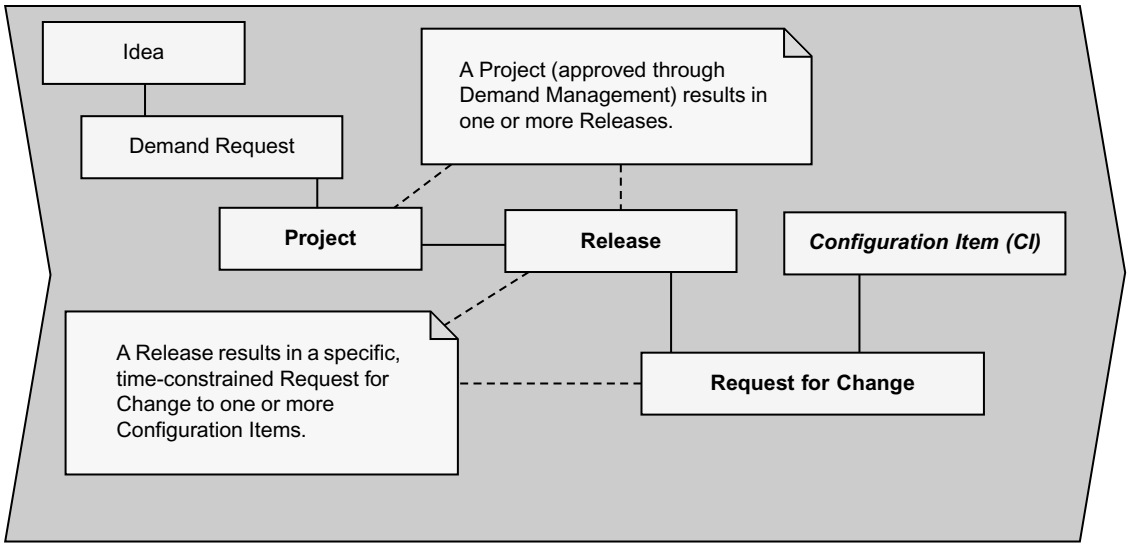


Figure 3.6 Alternate representation of RFC, Project, and Release.

Event

Events are the raw material of Metrics, which in turn drive Agreements and Contracts.

An Event is raw material. It is any operational signal emitted by any Production CI. Only a small fraction of Events are meaningful to ITSM, and an even smaller fraction result in Incidents. Events are one basis for Metrics, which in turn drive Agreements and Contracts.

One important type of Event is emitted by change control and detection systems, and that is the identification of physical change. This Event specifically indicates that for a given CI a state change has occurred that is of management interest. Change Events may be generated automatically by the CI in question or detected by active probing (e.g., tools such as Tripwire that compare the current state with a known baseline). The most sophisticated IT operations reconcile such change detection Events with the RFC process.

ITIL implies that an Event is equivalent to an automatically detected Incident.[152] Anyone who has experienced an autogenerated "ticket storm" will know that this definition is not suitable—most Events are not Incidents; extensive and well-architected correlation and filtering are required.

Of course, in the broadest sense, an Event can apply to any entity undergoing a state change of any kind. In this sense, a Contract might "raise" a logical Event when it expires. However, this is so broad that it's not a focus of this model.[153]

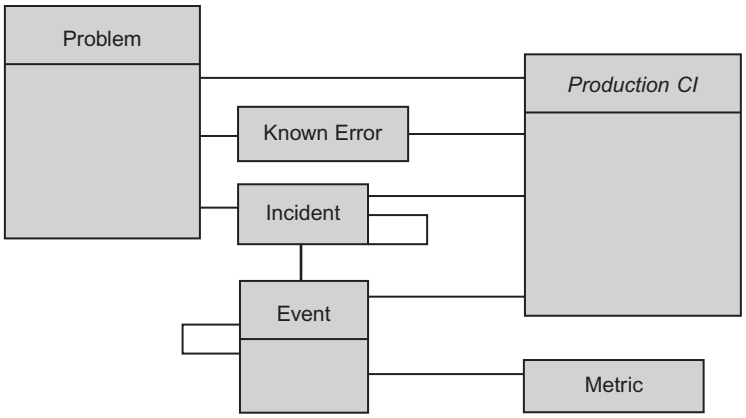


Figure 3.7 Event, Incident, Problem, and Known Error context.

By definition, an Event must have had a CI that emitted it.

Note that Events can be related to both discrete physical CIs, such as Servers and Datastores, and to logical Services. This is characteristic of monitoring correlation architectures, business service management (BSM) and end-to-end transaction monitoring. Rather than monitoring an individual, granular CI, the major Event of interest is an aggregation or derivation of multiple internal Events within the Service (e.g., expressed as overall transaction response time or customer-visible service failure).

Events are also indicators of capacity consumption and support measurements for that purpose: hardware utilization, memory, transactions, and so forth. Financial chargeback may depend on event management.

Advanced IT providers and infrastructure systems are starting to work with statistical analysis of Events, for example, to determine whether a certain repeated Problem has an identifiable Event signature that may help resolve it. This gets into cutting-edge research into pattern detection across large data sets, related to data mining.

A best practice for all operational Events is the embedding of an appropriate CI identifier. By definition, an Event must have had a CI that emitted it—it cannot arise out of the ether. This reinforces the case for managing unique and *terse* CI naming conventions, because many Event data structures will not be able to support long identifiers. See the "Application ID and Alias" pattern in Chapter 5.

The change Event is discussed further in the "Configuration Management" section in Chapter 4.

Incident

ITIL defines Incident as "any event which is not part of the standard operation of a service and which causes, or may cause, an interruption to, or a reduction in,

the quality of that service."[154] ITIL also states that a Service Request is a type of Incident, which seems perverse. (A Service Request is not an interruption unless you are trying to build a culture of hostile customer service!) This line of thinking is not supported here.

Incidents are independent of their mode of detection.

Service requests may be tied to Incidents through the CI against which the Incident is reported. In this interpretation, Incidents are independent of their mode of detection; this is necessary to support Incidents that may be reported or derived through enterprise monitoring without ever being reported through the centralized service desk.

An Incident has to be experienced. It is an occurrence. This distinguishes it from the Known Error concept used for knowledge management for the help/service desk (an error being a known condition in the abstract).

A Service Request may occur in response to an Incident. Incidents (especially when generated from monitoring tools) often require correlation and root cause analysis, which are supported through the relationship of Incidents and Events to each other.

Change–Incident

A Change may be in response to an Incident, without going through the more formal and heavyweight Release process. Alternatively, an Incident might be the result of a poorly executed Change. This means that the relationship between Change and Incident should probably have a type attribute so that it is clear which caused which (see the section on intersection entities later in this chapter).

Problem and Known Error

In ITIL, a Problem is "the *unknown* underlying cause of one or more Incidents," and a Known Error is "a Problem that is successfully diagnosed and for which a Work-around is known."[155]

However, this leaves a hole for Problems with known underlying causes that nevertheless have no workaround, so the ITIL specification won't do as a data definition. The definition here is that a Problem is generally a (known or unknown) root cause of many Incidents, although in the current model it is possible for an Incident to be caused by several Problems.

ITIL further states, "A Problem can result in multiple Incidents, and it is possible that the Problem will not be diagnosed until several Incidents have occurred, over a period of time. Handling Problems is quite different from handling Incidents and is therefore covered by the Problem Management process."[156]

A Known Error is a knowledge management hook—it is an entity that can house the known resolution techniques for a given Problem.

Problem–Release and Problem–RFC

Problems may be addressed by Releases, which might solve multiple Problems. An individual Problem might also be addressed by one or several RFCs. One possible approach is to say that Problems are generally handled by Releases (using demand management), and Incidents are handled directly by RFCs (when called for). Ideally, an RFC should be able to reference both Incidents (tactical) and Problems (longer term). This will depend on the capabilities of incident management and its degree of integration with Problem and Change.[157]

Service Request

Problems may be addressed by Releases, which might solve multiple Problems.

A Service Request is a logged interaction between an individual and the service desk that requires follow-up. Service requests may have various types, such as the following:

- Hardware or software request
- Incident report (i.e., the request is "resolve this incident")
- Configuration change request (the Service Request is the actual work request, not the authorization request)
- Security request

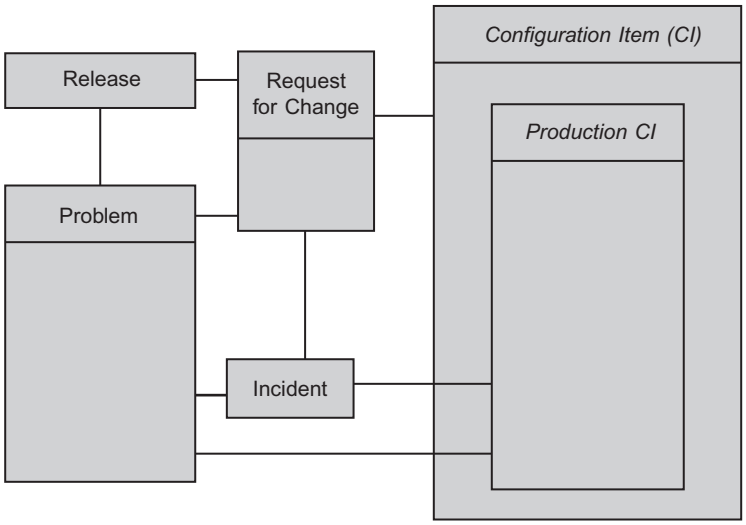


Figure 3.8 Problem, Release, and RFC context.

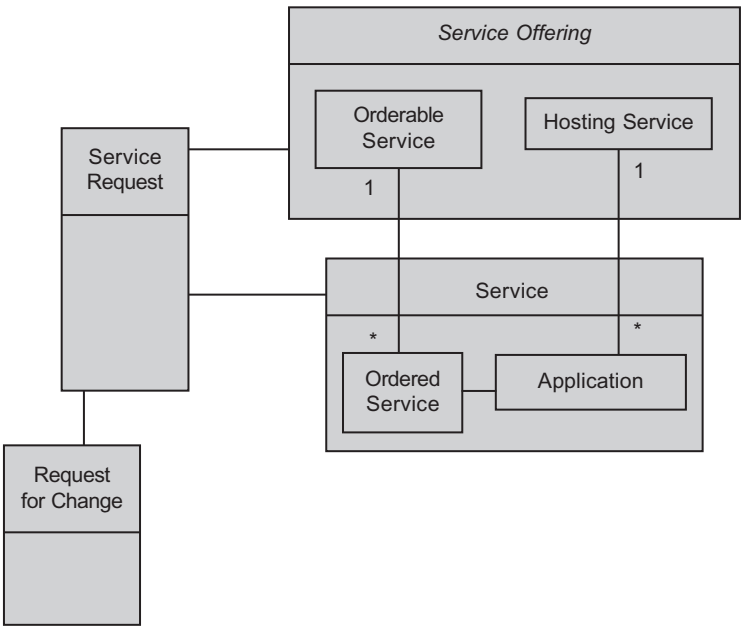
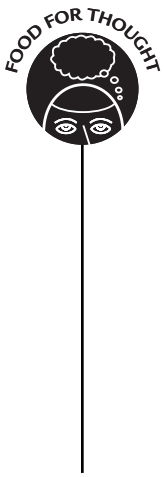


Figure 3.9 Service Request context.



Incoherent ITIL

The ITIL definition of Service Request is "every Incident not being a failure in the IT infrastructure."

The definition of Incident is "any event that is not part of the standard operation of a service and that causes, or may cause, an interruption to, or a reduction in, the quality of that service."[158]

Translation: *A Service Request is, or may be, an interruption.*

This is incoherent at best and perverse at worst. Service requests are part of normal operations. They are not interruptions.

RFCs might be seen as more closely related to Incidents, because these do pose a risk. However, changing systems is in a larger sense part of standard value chain activities, as opposed to true Incidents, which are usually understood to be unforeseen.

A critical distinction is that between Service Request and Project initiation. The service management architects will need to pay close attention to the differences among Service Offerings that may be straightforward products, Service Offerings that are more open ended (analogous to professional services or consulting), and work requests that should not be framed as Service Requests but should be routed to

demand management. Alternatively, the architects might view a Demand Request as a type of Service Request and drive to a more generalized approach (the "single pane of glass" philosophy).

See the "Clarify Service Entry Points" pattern in Chapter 5.

A Service Request is not a CI. It has a defined life cycle and typically figures in only one Business Process—its own fulfillment.

Service Request–Service Offering

A common relationship pattern is that Service Requests turn Service Offerings into Services.

Service Request–Service

A Service Request may occur with respect to an already-delivered Service. See the discussion later in this chapter.

Risk

> When a resource becomes essential to competition but inconsequential to strategy, the risks it creates become more important than the advantages it provides.
>
> —Nicholas Carr[159]

A Risk is a known possibility of adverse events, usually described by 1) likelihood of happening and 2) cost of occurrence. Risks are best seen as directly applying to CIs; a deficiency of modern risk management software is that it is often designed in a vacuum, with the risk management team entering their own representations of CIs,

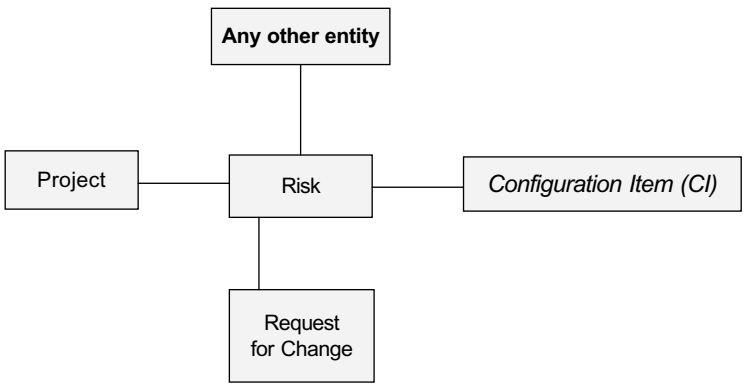


Figure 3.10 Risk context.

such as Application and Process, and not looking to a common system of record for this reference data. See the CMDB-based risk management pattern in Chapter 5.

Risk Relationships

Risks may theoretically be associated with virtually any entity in the model, but the primary targets should be CIs, Projects, and Change requests.

Account and Cost

An Account is a financial construct. According to Wikipedia, it is "a record of an amount of money owned or owed by or to a particular person or entity, or allocated to a particular purpose."[160] Other terms are "cost center" and "charge code."

The relationships of Account were not included in the main data model because of graphical complexity issues. Account is typically tied to a number of different entities, depending on the financial management approach being used (Figure 3.11).

Account might also be tied to any arbitrary CI, but this can imply considerable complexity.

Cost is an attribute, not an entity, and therefore does not appear in the conceptual model. Cost might be an attribute on any of the entities surrounding Account in Figure 3.11 and others (e.g., lower-level entities supporting Service, such as Application or database). A CMDB technically might allow any entity (not

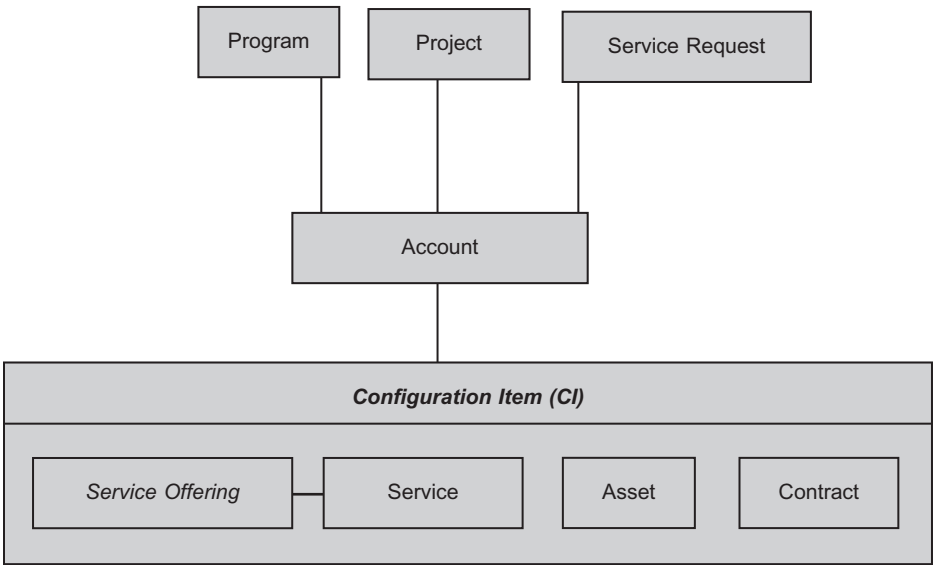


Figure 3.11 IT accounting relationships.

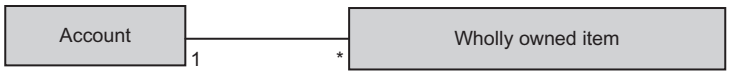


Figure 3.12 Account and wholly owned item.

just CIs as defined in this book) to have an associated cost, and determining which CIs might appropriately have a cost would be an important implementation task.

One common issue is allocation. If a given entity instance is related to one and only one Account, it "rolls up" and financial management is simpler—the account holders know that they bought the whole item. This is represented as a one-to-many relationship (Figure 3.12).

However, if the costs for a given IT item are to be split across multiple accounts, it turns the relationship into many to many, requiring resolution with a specific allocation percentage (Figure 3.13).

Attributes

Percentage, the first attribute, has appeared. This book does not go into much detail about attributes.

For example, if a network Service is shared across several accounts, a percentage allocation must be established for each Account (Figure 3.14).

Direct versus allocated (or indirect) costs are a substantial management challenge in IT. The desire for financial visibility runs into the issue of "dollars chasing

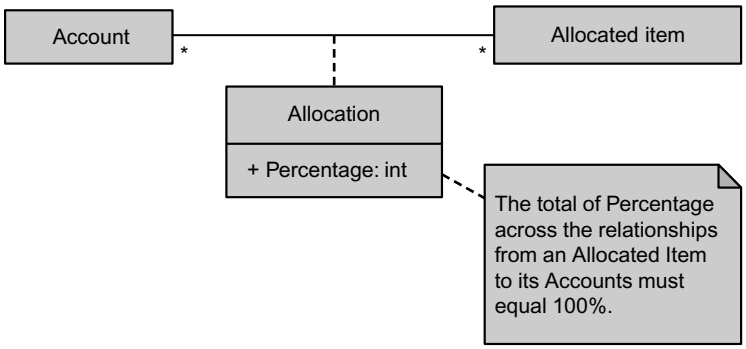


Figure 3.13 Model for allocating across accounts.

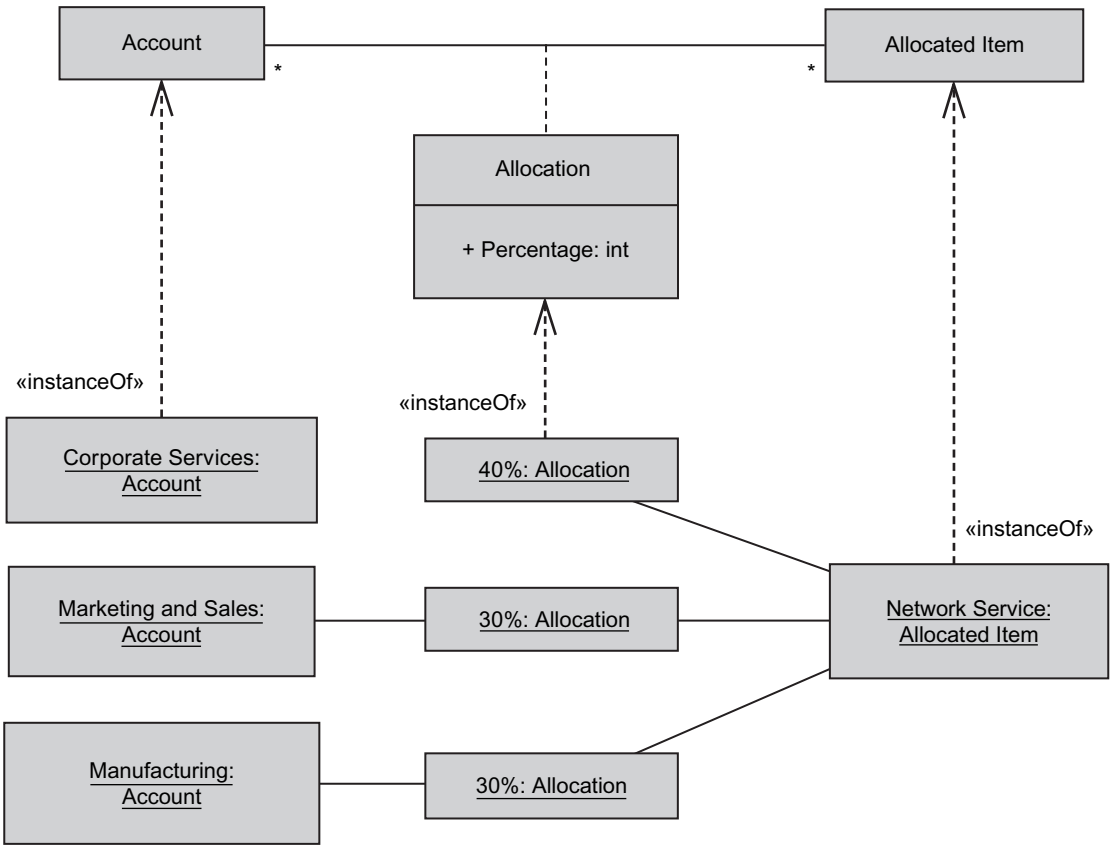


Figure 3.14 Example of allocated service.

dimes": the costs of managing the direct allocations outweigh the benefits in having granular visibility. In ITIL's words, the risk is that "the IT Accounting and Charging processes are so elaborate that the cost of the system exceeds the value of the information produced."[161] This book takes no position on what is an appropriate level of complexity but rather seeks to describe the general case capabilities needed to support a variety of approaches—one thing architects can be sure of is that requirements will change.

As Jeff Kaplan notes,

> Each IT service component (development, integration, help desk, network management, data center operations, maintenance, etc.) has a unit cost. Unit cost is the cost of providing one unit of service at predetermined service levels. Examples include cost per call, cost per connection, and so on. The specific units used are less important than is measuring each

service's variance from the standard cost. Using cost accounting, organizations should set a standard cost per unit for each service and project, based on the expected cost of providing an incremental unit of service.[162]

This passage, although informative, requires some thought to interpret as a requirements specification. First, the distinction between orderable and nonorderable services becomes important. A nonorderable service by definition has a large fixed cost that can be allocated arbitrarily against a user base, but doing so might not be advisable. For example, consider an investment in a high-capacity customer-facing online order system. This system must be kept running regardless of workload, and the marginal cost for heavy use as opposed to no use may be negligible. In naïve chargeback models, cost to the customer will vary *inversely* with usage, and this does not help IT credibility. (Even worse is when a unit's cost goes up—with stable consumption—because another unit has *decreased* its consumption.)

The concept of activity-based costing is a significant departure from older costing approaches. This book's interpretation of activity-based costing requirements applied to IT is that a concept of the business transaction is needed (this is the true "activity").

Role Management

The human organization will be more fluid than the core ITSM and metadata concepts.

The core data model has no Roles or people in it. *This is deliberate.* Organizational approaches to managing the processes and their data will vary, titles will change, and in general the human organization will be more fluid than the core ITSM and metadata concepts. Therefore, the Role structure is generalized; Parties (people or groups composed of other parties) have Roles with respect to any entity in the model.

Party, Person, and Group

A Party is either a group or a person, people are members of groups, and groups can contain other groups. The following are all Parties:

- Oracle Incorporated
- Bill Smith
- Support group APPL-2-CNS
- IT Service Management Forum

Party is a controversial concept in data modeling, because business users do not understand it. They understand concepts like "administrator" or "steward." However, these are *Roles.* (These are well-understood issues in data modeling.)

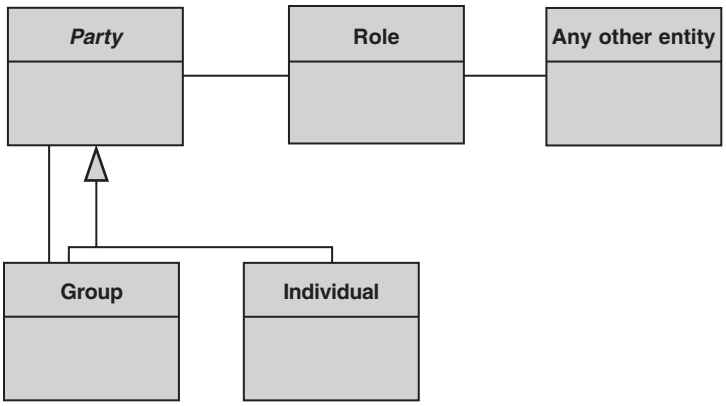


Figure 3.15 Role model.

Roles

Here are some example Role types and the entities they might interact with. Note that ITIL and other industry sources, such as the Enterprise Computing Institute, go into some depth about this, so this section doesn't include an exhaustive survey.

Role	Entity	Notes
Requester	Service request (as related to Service Offering or Service)	A requester can request a new instance of a Service Offering (which becomes a new service) or can request a Change to an existing Service.
Support group	Usually Application	A support group would usually be a group associated with one or more Applications. Sometimes, a support group might be associated with a Technology Product (e.g., a Windows Engineering group).
Developer	Project (preferably related to Release and Application)	A developer carries known expertise on a given system. For any Application, a complete record of all developers (especially at the senior level) who worked on it is recommended. To provide value, this list might be sorted by hours worked on the system; those who spent the most time on the system would be of highest interest. Other software development roles (e.g., architect, tester, and analysis) could be handled analogously.

(continued)

Role	Entity	Notes
Release manager	Project, Release, Change	A release manager is responsible for coordinating the output of a project into releases to be accepted into production.
Change coordinator	Change	A change coordinator is responsible for the successful execution of one or more Changes. They may be part of a specific capability team or part of an enterprise change team.
Operational change approval group	Operational CIs	An operational change approval group is often seen as a dynamic entity, composed of representatives from the support groups associated with the CIs in question, as well as overall change coordination from a central enterprise group. Often, the change approval group may have standing representation from major technology product areas (e.g., Unix engineering or network engineering) or other operational capabilities (e.g., security).
Here is a common Role type that may be problematic:		
Change Advisory Board	Any CI	ITIL calls for a unitary Change Advisory Board, admitting that the composition of that group may vary even within a single meeting.[163] However, different CIs may have radically different stakeholders. For example, if a Contract is a CI, it should be under change control, but the change approvers would be the senior IT executives, the contract office, and legal—your engineers would not be involved. The concept of a Change Advisory Board becomes so general that its usefulness is questionable. The better understood use of change approver is with respect to Production CIs. See the "Clarify Service Entry Points" pattern in Chapter 5 and related discussions throughout.

Support roles for a Service (e.g., an Application) may be ordered, which requires an escalation path (Figure 3.16).

Escalation paths may be of several types, typically functional and hierarchical; a functional escalation path is, for example, from level 1 to level 2 to level 3 support, and a hierarchical escalation path might walk the organization chart from application manager to director to vice president. Specialized escalation paths to technical subject matter experts (e.g., database administrators and senior software engineers) may also exist; alternately, the escalation path may become a tree with decision points and not just a linear progression.

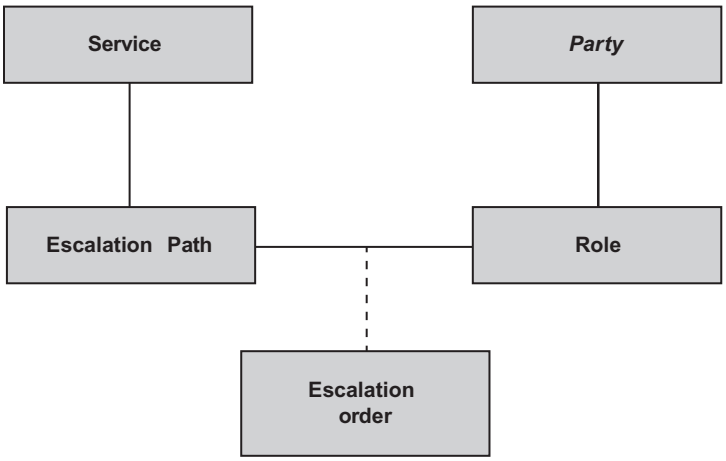


Figure 3.16 Escalation.

Figure 3.17 Classification taxonomy.

Classification

Taxonomies are used extensively in IT information management, for the same reasons they are used in science and other fields requiring knowledge management. A hierarchical tree structure is an intuitive and effective way to manage complexity. Typical taxonomies encountered in internal IT systems are functional decompositions, data subject hierarchies, application and technology categorizations, and so forth. There are commercial providers of taxonomies.

There is overlap between this entity and other treelike structures. The differentiation is that a classification taxonomy is merely a lightweight conceptual structure. Each node is of the same basic type. One does not typically establish dependencies between the taxonomy nodes or assign extensive attributes to them.

A valuable use of the taxonomy concept is to identify overlap or redundancy, for example, in an application portfolio. See the "Taxonomy-Based Rationalization" pattern in Chapter 5.

3.4 The Configuration Item and Its Subtypes

The Base Technology Stack

Before discussing the particulars of the CI and its subtypes, some discussion of the general IT stack is called for.

The concept of a "stack" has a long history in information technology, perhaps originating with the OSI networking model. In ITSM, an extended stack is often depicted something like the one shown in Figure 3.18.

Figure 3.18 shows a stylized representation of concepts present in much ITSM literature, advertising, and so on. One thing that all of these concepts have in common is that they may be seen as CIs.

A CI is a managed, specific object or element in the IT environment. It is one of the most problematic concepts in IT governance.

CI is one of the most necessary yet problematic concepts in IT governance. It is highly abstract: any managed "thing" in the environment, from an individual computer chip to an entire mainframe, can be a CI. This high level of generality makes the concept difficult to manage from the perspectives of process, data, and Application.

The ITIL definition of CI is as follows:

> [A CI is a] Component of an infrastructure—or an item, such as a Request for Change, associated with an infrastructure—that is (or is to be) under the control of configuration management. CIs may vary widely in complexity, size, and type, from an entire system (including all hardware, software, and documentation) to a single module or a minor hardware component.[164]

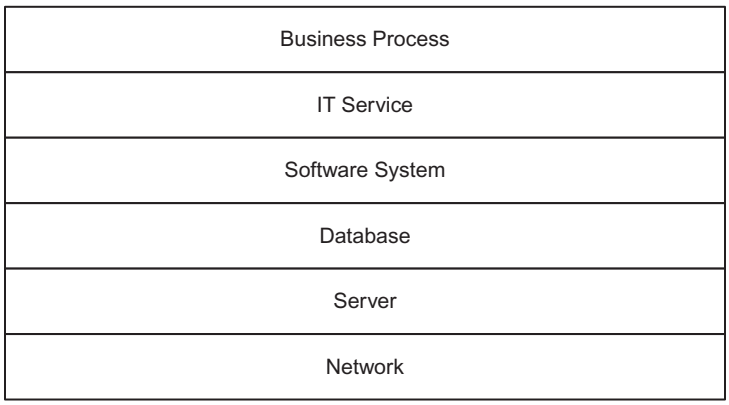


Figure 3.18 The generic IT stack.

The preceding sentences are imprecise from a data management point of view. Essentially, a CI as it is viewed by ITIL could be construed as *any piece of data representing any IT concept.* The phrase "item, such as a Request for Change, associated with…" extends the CI concept unmanageably—every data element in the IT problem domain becomes a CI. There is then a paradox: if an RFC is a CI, and a CI by definition is under change management, that means the RFC requires an RFC requires an RFC, and so forth.

Every data element in the IT problem domain becomes a CI.

Here is the ITIL specification as it describes the interrelationships of CIs:

> Configuration structures should describe the relationship and position of CIs in each structure…. CIs should be selected by applying a decomposition process to the top-level item using guidance criteria for the selection of CIs. A CI can exist as part of any number of different CIs or CI sets at the same time…. The CI level chosen depends on the business and service requirements.
>
> Although a "child" CI should be "owned" by one "parent" CI, it can be "used by" any number of other CIs….
>
> Components should be classified into CI types…. Typical CI types are: software products, business systems, system software…. The life-cycle states for each CI type should also be defined; e.g., an application Release may be registered, accepted, installed, or withdrawn….
>
> The relationships between CIs should be stored so as to provide dependency information. For example,…a CI is a part of another CI…a CI is connected to another CI…a CI uses another CI….[165]

This is again highly general. One issue in the industry is that some vendors have interpreted this specification to allow their customers too much freedom in defining CIs and their relationships. In some tools, a Server might be "a part of" a random access memory (RAM) chip; a printer might be "connected to" an extensible markup language (XML) schema—connections that obviously do not make logical sense.

More rigor is necessary. This analysis refines the ITIL representation and makes it more specific by applying data modeling (metamodeling) principles.

A CI typically has an indeterminate life cycle, unlike a Project, Service Request or Incident; these are defined and tracked partly in terms of their closure.

- For this book, a CI is a managed, specific object or element in the IT environment.
- A CI by definition is under change control *of some form.*
- Typically, a CI also has an indeterminate life cycle, unlike a Project, Service Request, or Incident; these are *events* and defined and tracked partly in terms of their closure.

- CIs are not instances of activities, although an activity definition may be a CI. They are real, not abstract.
- CIs typically also participate in multiple IT processes. If something is relevant only to one IT process, it is probably not a CI.

Applying the preceding principles means that certain things are not CIs, such as the following:

- Strategies, Programs, Ideas, Demand Requests, and Projects (Projects may have multiple CIs within them, but they themselves are not CIs)
- Events
- Incidents and Problems
- Requests for Change
- Service *Requests* (but a Service *Offering* is a CI)
- Data records in databases and files generally; they are under the "change control" of the accessing Application
- CI *records* (the representation is not the object); however, see the discussion of the Metadata CI type

This architecture proposes three major categories of CIs: base, Operational, and Production.

CIs should always be specific. "Oracle Financials," if present in the environment, would be a logical CI, containing and using many physical CIs (e.g., software Components and Datastores). A Generic "Human Resource Management Application" as a reference category would not be a CI.

CIs have subtypes, and those subtypes in turn can have subtypes. Figure 3.19 shows one representation.

Servers and Applications can have Incidents and Known Errors—but can a Contract?

The major types of CIs are as follows:

- (Base) CI
- Operational CI
- Production CI

They are "nested" (Figure 3.20).

This means that an Operational CI is also a base CI and a Production CI is also an Operational CI and a base CI.

Subtyping is often overapplied. An important reason to subtype (in conceptual modeling) is if a subtype can have a relationship that the parent does not participate in. Figure 3.21 shows this clearly: a Change can apply to any CI or subtype, a measurement can apply to an Operational CI or a Production CI, and an Event can only be associated with a Production CI.

Again, can a Contract have an Incident?

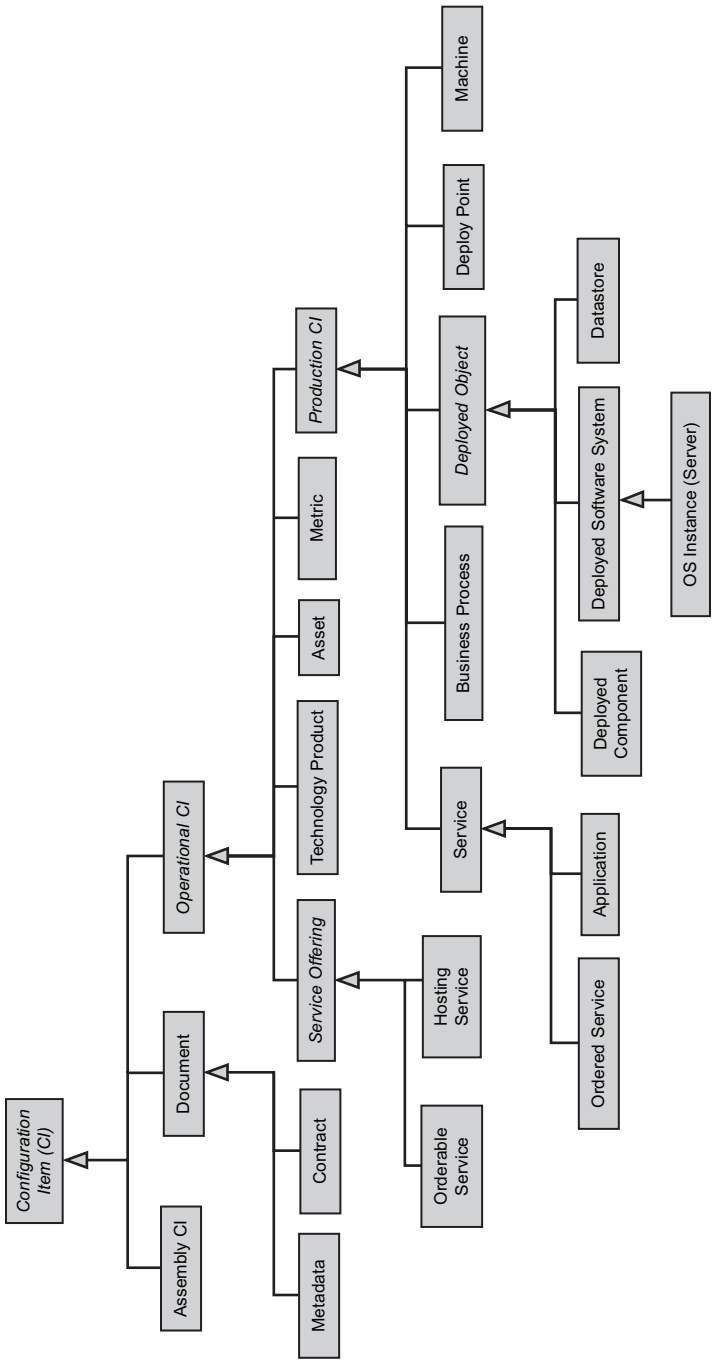


Figure 3.19 Detailed CI taxonomy.

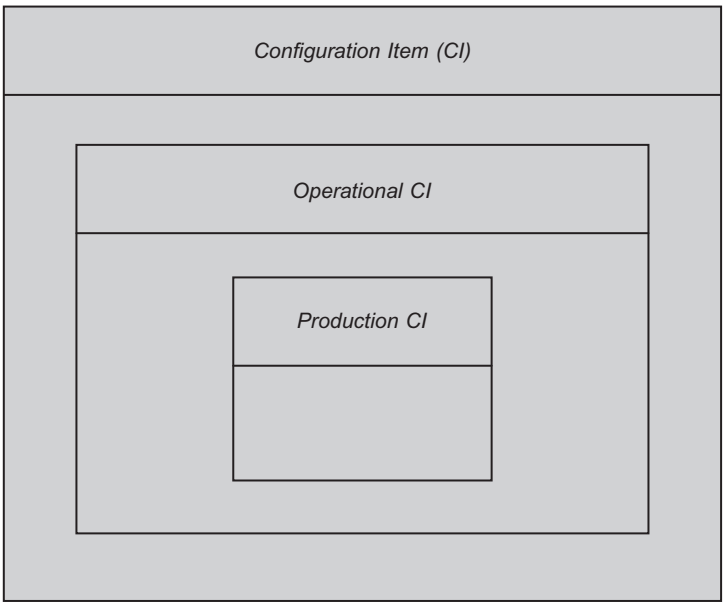


Figure 3.20 CI subtypes.

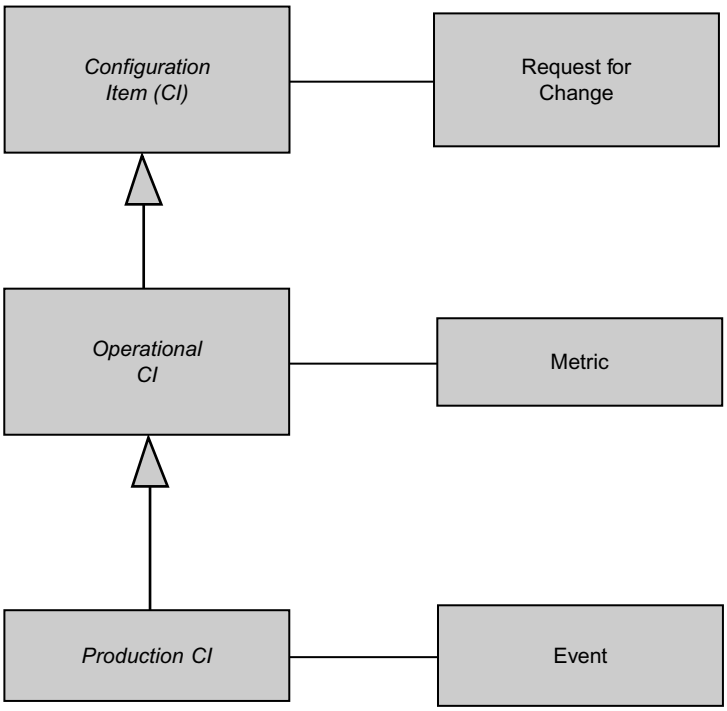


Figure 3.21 CI subtypes and key relationships.

Table 3.1 Logical versus Physical CIs

Logical CI	Physical CI
Application	Component
Process	Datastore
Service	Deploy point
Technology	Document

Logical and Physical Configuration Items

Applications, Processes, and Services in the service catalog–sense are logical CIs. Machines, Components, files, and network-addressable Web services are physical CIs.

CIs can be logical or physical. From the top down versus from the bottom up is another way to think of this distinction: logical are from the top down, physical are from the bottom up.

Physical in this case means no ambiguity about the boundaries of the CI (even if it is only transient bits on volatile storage). Logical means that some consensus is required to set the bounds of the CI.

Applications (especially those built in-house), Processes, and Services in the service catalog sense are the best examples of logical CIs. Machines, Components, files, and network-addressable Web services are physical CIs. Managing logical CIs is challenging and requires a clearly defined process to establish the bounds of this potentially blurry "thing."

Discussion of Logical Applications

Chris: What's the big deal with applications and how they're "logical"? You've been harping on that all day.

Kelly: I found a diagram in some of your system literature.

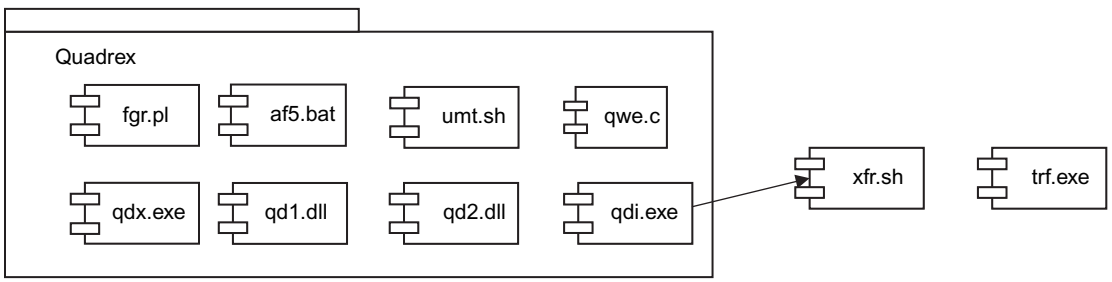


Figure 3.22 Application and boundaries.

It's the perfect example. Those little boxes with "dog ears" are a standard representation (from UML) of software Components. Notice how they are named—that's what you would see on the Servers supporting the application. The functionality as a whole is named Quadrex; that's how you refer to it in meetings and in the halls—but there is no such thing as far as your computers are concerned.

One question: Is "xfr.sh" part of the application? The Quadrex team told me that it's an extract job for data going to the TSI system. The TSI team told me they don't think they support it. Who does? Most organizations have such "gray area" questions, and clarifying the application portfolio's ownership can help reduce the risk of finger-pointing and ineffective response to service outages.

The Base Configuration Item

The next set of definitions focuses on the base CIs, as shown in Figure 3.23.

The base CI is the master category that all CIs belong to. It is any "thing" in the IT environment that requires management (usually defined as being under change control of some sort).

CIs have differing levels of involvement in day-to-day service management and production processes. The base CI includes documentation and the definitions of

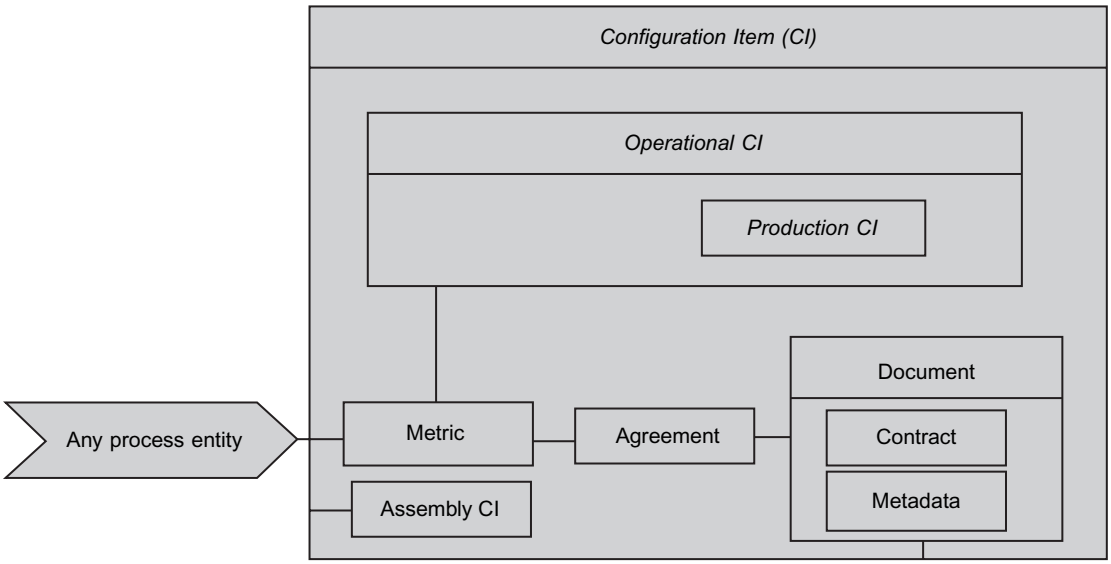


Figure 3.23 Base CIs and relationships.

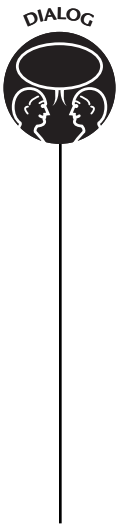
service-level measurements, objectives, and agreements. Any type of CI may be involved in an RFC.

Change control for items that are not Production CIs (not operational or production) may or may not be formalized. For example, the service management group may define Service Offerings, or the asset management group may add new Assets, without going through the highest-formality change control processes reserved for Production CIs.

An Operational CI is something that is involved in day-to-day business processes, that can be measured, and that is a primary entity in the service management workflow.

An Operational CI is distinguished from the other base CI types as something that is involved in day-to-day Business Processes, that can be measured, and that is a primary entity in the service management workflow.

A Production CI refines the concept of Operational CI to include the core CIs that may be involved in Incidents and have Known Errors. (Think data center, or production workstation.) Change control for Production CIs is usually a formal, high-visibility process that is what many enterprise IT people think of when referring to "the change process."



Why Several Categories of CI?

Chris: So, I'm seeing that a Document is a CI—OK. And an Operational CI is a CI? What do the italics in the diagram mean?

Kelly: The italics mean that something can't *only* be an Operational CI or a CI itself. It has to be something *under* the box with italics: in this case, a Service Offering, Technology Product, Asset, or something under Production CI.

Chris: Why do we bother with these detailed types anyway?

Kelly: It's all about being precise. Suppose that we just had one category of CI that included Documents, Service Offerings, and Contracts, as well as Servers and Applications. Servers and Applications can have Incidents and Known Errors—but can a Contract? Not really. This is fundamental information modeling; people can spend their whole careers specializing in describing data structures precisely. Without this precision, your CMDB is at risk.

Assembly CI

CIs require grouping for various reasons, such as supporting a Release, a Service map, or a Service Request. The assembly CI leverages the "owns" and "participates" relationships to support this.

Document

> SLAs, underpinning contracts and OLAs…should be brought under Change Management control….
>
> —ITIL[166]

A Document may be a CI if its existence and content are significant enough to IT service delivery to warrant formal change control. It may apply to any CI or CIs, and any CI may have multiple Documents. There are of course many other types of Documents, and not all are under change control (which means they are not CIs). Another class of Documents that are usually under change control is the class of project Documents. However, this change control is usually at the project level, and ITIL specifically avoids discussing it.

Important types of Documents (not modeled) are Requests for Information (RFIs) and Requests for Proposal (RFPs).

Metadata

The contents of the CMDB are all metadata.

In this model, Metadata is *nonruntime* structured information related to the IT environment. This is a reflexive (self-referential) concept in the CMDB. A clear example would be the relationship between a data model (metadata) and the physical production data structure it represents (Datastore). The contents of the CMDB are all Metadata.

Metadata has a more general computing sense in which it is "data about data." However, because data about data exists throughout IT elements such as file systems and configuration files, this is not a useful definition for this model. There is the conceptual issue of how to distinguish Metadata from general aspects of stored-program computing architecture (taken to the extreme, all processing instructions are data about data).

Metadata can be deployed to an operational context (sometimes by transformation), which makes it runtime. In such cases, the Metadata becomes a Component or a Datastore: for example, a logical data model from which an actual database schema on a running Server is generated. In this case, the database schema as a Datastore CI might be related to the logical data model, as a Metadata CI. Another example would be a BPEL process definition generated from a visual flowchart. When such a transformation happens, the transformed runtime artifact by definition is no longer Metadata. It is computing architecture and impossible to distinguish from general aspects of stored program computing.

Keeping Metadata in synch with the real processing architecture is a continual problem.

Because it is by definition nonruntime, keeping Metadata in synch with the real processing architecture is a continual problem, addressed by tools such as scanners and techniques such as model–database comparison.

Some have called for "real time" or "embedded" Metadata, which would imply continuous introspection into live production infrastructure. The performance and security implications of this are nontrivial, and there are value-adding aspects to offline Metadata (e.g., verbose text definitions and logical dependencies) that will never be directly represented or identifiable in a production infrastructure.

Metadata as a CI is a riddle;[167] it suffers from the same problem noted previously if Change records were considered CIs—the Metadata has Metadata has Metadata, and if all is under change control, the infinite loop can't be resolved and no changes can take place. However, because there is precedent for Documents as CIs, it is conceivable that some Metadata (e.g., as a fixed form or structured project document) may be under change control.

This is one of the more difficult conceptual areas in this book, dealing as it does with "thing" and re-presentation of "thing." Metadata is *re-presentation*. It is not the *thing*.[168]

Contract

A Contract is an Agreement between (usually) two parties with authority in the overall IT service context.

A Contract is an agreement between two parties with authority in the overall IT service context. A Contract may enumerate several formal agreements, based on objectives for measurements of CIs. Contracts are often the subject of intense scrutiny, and their signing is (or should be) a visible event. However, usually a contract management office performs this particular type of change control, and it is not part of the mainstream "change process" as generally understood in most IT organizations.

Contract–Agreement

A Contract may document many Agreements (e.g., SLAs), in turn based on Metrics.

Contract–Asset

A Contract may be the source documentation for the acquisition of certain Assets, especially if the definition is broadened to include invoices.

Metric

A measurement definition is a CI because it represents the criteria on which IT service performance is measured.

A Metric is a defined, specific characteristic of a CI or a process entity, amenable to capture and verification. Metrics are the basis for process control.

(Process entities include all the non-CI entities, e.g., Incident, Problem, and Change.)

Metrics typically vary over time. Specific means that it is one of the basic levels of measurement: nominal, ordinal, interval, or ratio. This conceptual entity encompasses both the definition of the Metric and the implication of its specific instances. Metrics typically nest in a hierarchy, moving from the more technical and specific to the more general and strategic.

A Metric is meaningless without the context of a CI (often a Process, but perhaps a Service). *Metrics have objectives as an associated concept* (not shown in the model). An objective is, with respect to a Metric, what the Metric *ought* to be. This specifically supports the concepts of service-level objective and operational-level objective, where a service provider may have informal service targets that are not the subject of an Agreement.

A measurement definition is a CI because it represents the criteria on which IT service performance is measured.

Metrics may be called for concerning the following, among other IT processes, functions, and characteristics:

- IT financial management
- Availability
- Capacity
- Integrity
- Security
- Disaster recovery
- Performance
- Training
- User support
- Change management

Specific measurement approaches will be discussed in the design patterns section.

Note that the Metric entity is the *definition* of a Metric, such as "unscheduled Changes," "transactions per second," "average response time," or "downtime." Such definitions are not themselves measurable—think about it. But they might be under change control as a basis for contractual agreements.

The ITIL section on IT financial management calls for a resource cost unit; this is a type of Metric applicable to various CIs.[169]

Metrics may use or contain other Metrics; taking this functionality to an extreme will result in the need for mathematical expression management (metric A = metric B × metric C, etc.).

Metrics are described by Metadata. See the Common Warehouse Metamodel's Expressions, Transformation, Information Visualization, and Information Reporting packages for detailed discussion.[170]

The focus in this discussion is Metric as applied to ITSM; Metric also applies more generally to business decision support. A Service may consist of delivering Metrics to an executive dashboard by a certain time every day.

Metrics are directly linked to Strategies; this linkage is essential for applying business performance management principles to IT governance and, for example, building effective digital dashboards.

An agreement is between two parties with respect to a service level, operational level, or some other aspect of a CI.

Agreement

An Agreement is between two parties with respect to a measurement, for example, a service level, operational level, or some other aspect of a CI. A Contract may have many Agreements.

Agreements and Related

Chris: OK, how does this all fit together? Document, Contract, Agreement, Measurement? Seems a little elaborate.

Kelly: Let's walk through a couple cases.

- An email Service where you are guaranteeing 2-day turnaround on 95% of email requests on average, as an SLA to the client
- A consolidated database farm where you are guaranteeing 99.995% uptime as an OLA to your application teams

The email account provisioning is a Service Offering, and each account request is a Service. Both are CIs; therefore, they can both have measurements. The measurement for the Service Offering might be "Aggregate % Turnaround in Days." Each individual Service has associated workflow that tells you the request date/time and the completion date/time. Those measurements are aggregated into the overall Service Offering measurement.

The objective for that measurement might be "< = 2 Days for 95%." (There are precise ways to represent this so that a service management application

(continued)

can accurately calculate it.) However, that objective is just an informal stake in the ground until it is the subject of an Agreement between two parties. And as we all know, if those two parties are within the service provider it is an operational-level agreement (OLA); if one is the client and one is the service provider it is an SLA. That particular SLA might be part of a broader Contract specifying all aspects of the relationship between client and provider. That Contract in turn is a Document and therefore a CI—and hopefully a Contract is under change control. But again, is it managed by exactly the same processes and systems that handle the deployment of software in a data center? Perhaps, but probably not.

Chris: What about the database farm?

Kelly: That's simpler. Let's assume it's a nonorderable Service (it was purpose built for a suite of applications and no more databases will be hosted there). The only thing different from the email case is that it's not a Service Offering; the measurement (e.g., availability with an objective of 99.99%) is on the Service. Aggregation is still necessary at a technical level, however, and that's where you get into the relationship between the Service-level management capability and the lower-level monitoring architecture.

As noted in the ITIL *Service Delivery* volume, agreements may be effectively managed at the corporate, customer, and service levels. See the discussion on role management, which is applicable here (any Party—organization or person—may have an interest in an Agreement).

Note that many components of an SLA would not be discrete measurements: narrative discussions on overall service scope, discussions of continuity management, chargeback formulae, and other aspects. The general problem is that of structured versus unstructured data; unstructured is easier to capture but more difficult to objectively manage, and the converse for structured data.

Configuration Item Dependencies

Allowing CMDB users the use of uncontrolled generic dependencies may result in poor data quality.

The arbitrary dependencies available on the CI concept are risky. They can enable a nonsensical connection, such as a (software) Component containing a (hardware) Machine. Arbitrary dependencies (contains and uses) are useful for CIs of the same type or for grouping CIs into manageable packages. But allowing them generally to be used by CMDB users may result in poor data quality and misalignment among different people's concepts of IT service modeling.

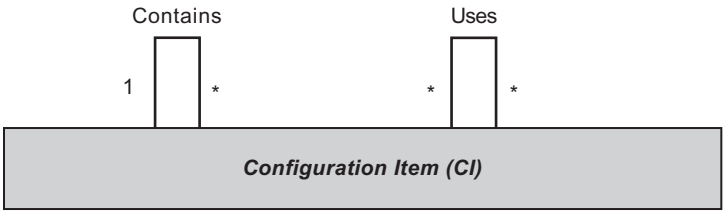


Figure 3.24 CI dependencies.

For further information, see the section on networks and trees (recursive relationships).

We Shouldn't Need Configuration Management Black Belts

From *www.erp4it.com*

More evidence that the theoretical critique of current CMDBs is reflected in people's practical difficulties.

It's been reported to me that a large firm in my area that uses a prominent CMDB tool has determined that its conceptual flexibility is hard to manage. They've had to lock data entry down to a small group of configuration management "black belts."

This is a natural consequence of an overly generic data structure; what these people are essentially doing is building a more precise, de facto consensus information model (metamodel, if you will), which they are enforcing through their group process and joint understanding. This is an unsustainable approach. They are forced into this because the tool does not allow this to be done automatically through declarative constraints, which is how we ought to manage complex data, according to well-established data management principles.

This is why a black belt team emerges when such tools are purchased: a consensus starts to build that, "yes, this service (as in SLA) is a CI, and yes, this hard drive is a CI, but we are not going to directly link the two—instead, we will put the drive in a SAN cabinet, allocate it to a mount point, deploy a database to it, assign the database to an application system, and finally create a dependency between the SLA service and that application." But no automatic constraints enforce such relationships; they are simply embedded in the group consensus that this is the way to do things. Automating such a group consensus is exactly what data architecture (or object-oriented class design) is all about.

(continued)

The scale of the configuration management problem is huge, and to capture and maintain such a mesh of data in a cost-effective way, we need a tool that will enforce sensible data relationships when being used by a variety of staff (e.g., offshore resources).

Again, the fundamental issue here is that CMDB tools vendors have taken the ITIL requirements literally as data schema requirements and are basically delivering simplistic graph metamodels. From discussing the situation with longtime ITIL thought leaders, it's clear to me that this was never intended by those who built the standard.

Usual rant: I don't think that configuration management will ever meet its goals without adopting more explicitly defined metamodel semantics, such as those the OMG (Object Management Group) has been painstakingly building.

Operational Configuration Item

An operational CI refines the base CI concept by including things that are *measurable,* which includes Service Offerings, Technology Products, and Assets. Operational CIs also are directly involved in the day-to-day provision of Services, but the documentation-oriented base CIs are not.

Some Operational CIs are also Production CIs and will be described below. The Operational CIs that are not Production CIs are Service Offering, Technology Product, and Asset.

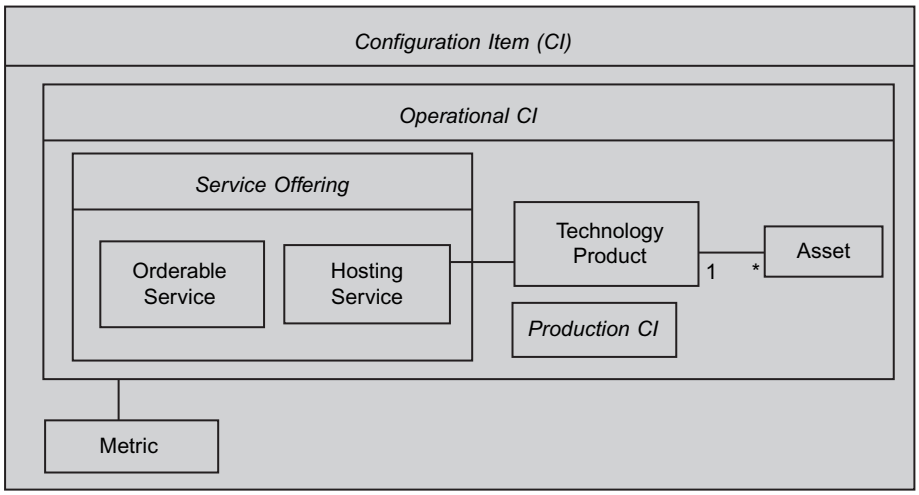


Figure 3.25 Operational CI in context.

Some CIs might not leverage the high visibility change control process that is usually focused on the production data center.

Operational CIs are under change control, but it is a different kind of change management dependent on their specific life cycles. A Service Offering goes through a different process than a change to a production application Server. Although ITIL implies that CIs all participate in a generalized conceptual RFC process, some might not leverage the high visibility change control process with its bias toward production concerns.

For example, a new Technology Product will probably go through some sort of adoption and certification process, perhaps an architectural review led by the IT organization's designated stakeholders for that type of technology. But it probably will not be a subject of change advisory board discussion, unless that Change Advisory Board has the broad ITIL scope.

Asset

An Asset is a financial concept. It shows up on the company's balance sheet and may be depreciable. The Asset concept is often one to one with Machines and Applications in terms of software licenses. However, a Machine may or may not also be an Asset. Another option may be for turnkey systems including several Machines and Deployed Software Systems to be tracked as one Asset.

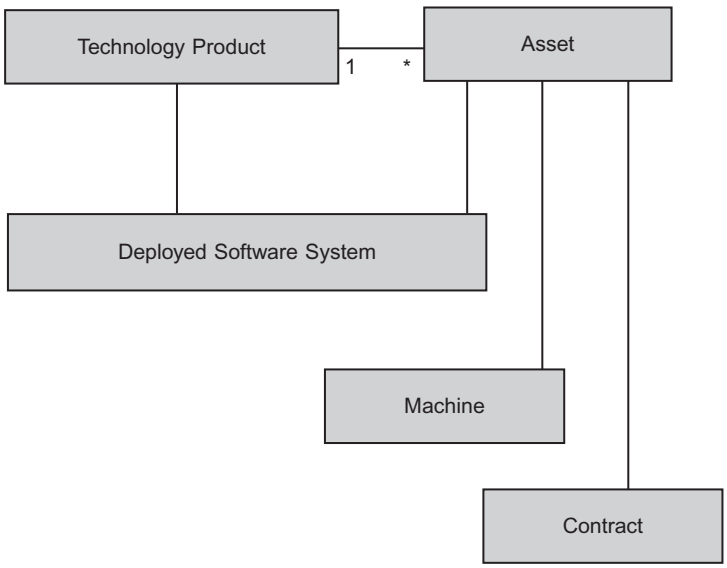


Figure 3.26 Asset context.

Assets and Configuration Items

Chris: All right, you got me. When is a Machine not an Asset? It can be on the loading dock and it should still show up on our books.

Kelly: Remember when we signed the deal with NexQ? Part of the arrangement was that they would locate two of their management servers in our data center. Stuff like that happens all the time nowadays. We track those servers as CIs; they are attached to our network, they are mission critical, and we even pull data off of them. But they aren't ours and don't show up on our balance sheet.

For software, the Asset is more or less equal to the software license. There is little or no industry consensus as to whether to call systems built in-house Assets—they may be built with capital budgets and depreciated, but often the expenditure is simply considered as a Project.

There's increasing awareness that systems developed in-house need to be managed as a portfolio—what relationship this portfolio management concept has to formal asset management is to be determined. Certainly, some of the background and orientation of experienced asset management staff would be valuable to the IT portfolio management objectives. Will asset management ultimately be seen as a subset of IT portfolio management?

Assets should have asset tags and formal identifiers, which should not be equated with serial numbers. Some Assets simply don't have them, and cases have arisen in which serial numbers change but the Asset remains the same, for example, if the serial number is tied to an assembly that is replaceable in the field, such as a machine motherboard.

When Assets are procured, their invoices should be provided in digital form and should enumerate all purchased products by type, model, and serial number. In this way, the incoming invoice can populate a database (asset management or integrated asset/CMDB) directly or with a little translation. One poor practice is when, for example, five Servers are purchased and appear as a single line item—this then requires further analysis and perhaps even physical inspection to determine the actual Servers and their serial numbers (which are often miscaptured when manually examined, rekeyed, or both).

Technology Product

The [IT] organization might hold the maintenance budget flat and force a 5% to 10% productivity improvement. This requirement

would drive IT implementers to design efficiencies into their applications and processes to achieve this goal [which] might motivate IT managers to consider additional criteria when evaluating application concepts, such as asset utilization and projected annual maintenance cost, putting pressure on the organization to simplify the application architecture and minimize the number of new platforms.

—Jeffrey Kaplan[171]

A well-defined Technology Product database, showing dependencies on technologies, is critical for the enterprise's vendor management and technical road map.

The concept of Technology Product is crucial for enterprise architecture and vendor management. A well-defined Technology Product database, with mappings to the specific Applications and Machines that depend on those products, enables tracking the enterprise's status with respect to product obsolescence, portfolio simplification, security issues, vendor support, and overall technical road map. It also helps in Program estimation and is an input into *infrastructural drivers of IT cost.*

The context diagram shown in Figure 3.27 elaborates on the conceptual data model; there are a number of dependent entities not shown on the main diagram.

The Technology Product concept is a combination of the ITIL concept of Definitive Software Library plus the various *types* of hardware devices approved for the environment (note that this is not the same as the ITIL Definitive

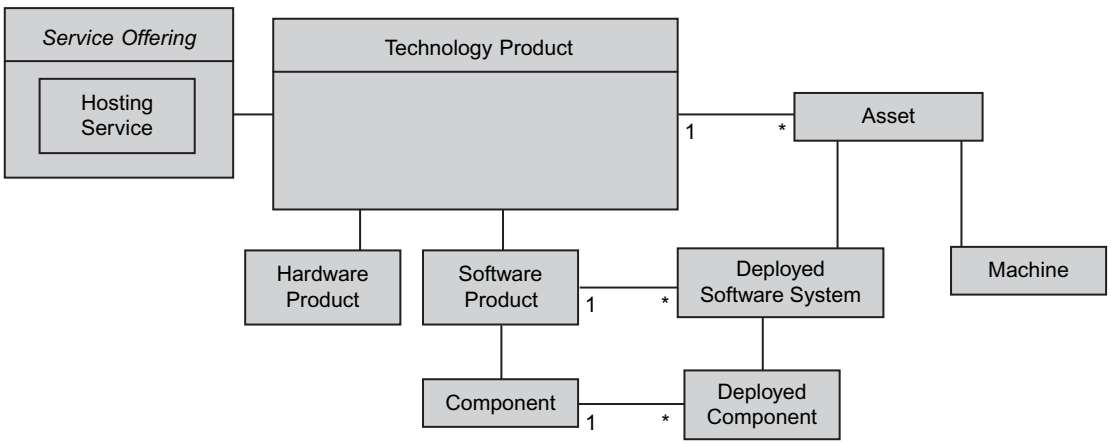


Figure 3.27 Technology Product context.

Hardware Store, a supply of spares). Because of tightly coupled hardware–software solutions (e.g., routers with embedded firmware), it is not feasible to separate Technology Products along strict hardware–software lines, although some kind of categorization taxonomy is required for enterprise architecture purposes.

New Technology Products require acceptance into the environment through some sort of defined process. Often, this may be owned by an enterprise architecture capability.

Technology products have versions; this is a complex problem relevant to many other CI classes. See the versioning discussion in the "General IT Data Architecture Issues" section later in this chapter.

One possible attribute for a Technology Product is a class of use, which might represent various levels of availability or processing power: a class 1 designation might include high availability, for example.

The concept of Technology Product would also be an appropriate place to link the skills sets of IT staff. When a Technology Product is no longer supported, this has an implication for human capital management—are those staff members with strong expertise in the product being retrained?

Technology Product–Hardware and Software Product

It is difficult to make a distinction between hardware and software for purchased Technology Products.

Note that Technology Products may aggregate both hardware products and software products; many purchasable solutions include both, with some level of independence—think of a Cisco router with its upgradeable firmware or a turnkey materials management system based on IBM iSeries (AS/400) computers.

Software products in turn contain Components; software products are logical, and Components are physical. Software products are *by definition* not deployed. Their deployments are represented by the concepts of Deployed Software System and Deployed Component. This representation in particular draws on the concise, elegant Software Deployment model from the OMG's Common Warehouse Metamodel.[172]

Technology Product–Asset

When a Technology Product is no longer supported, are those staff members with strong expertise in the product being retrained?

Technology products type Assets, which in turn are related (often, but not always, in a one-to-one association) with Deployed Software Systems and Machines. Turnkey systems combining both software and hardware will need to be carefully considered here as to data capture approach.

Infrastructural IT Demand Drivers

Chris: Infrastructural drivers of IT cost? You lost me there…

Kelly: We understand when the business comes and asks us to build something. Where we fall down is when Oracle decides to stop supporting Oracle 8, for example. Our business clients typically don't have any awareness of such shifts in the product landscape, but it's a really big deal for us—we have to go without support, pay an expensive (and less-qualified) third party for aftermarket support, or retest all our software on Oracle 9. Our business clients wish that these kinds of costs would just go away, but it's not that easy.

The thing is, we knew 18 months or more in advance that Oracle 8 was going off support. We were kind of in denial about it, partly because we didn't have a good handle on our exposure. Now, with a complete understanding of the technology stacks underlying our apps, we know exactly what our exposure is when Oracle 9 goes off support—we've got 3 big packages and 40 smaller applications, and we've already got the funding for this migration identified in our long-range plan.

Note that these are no different from other business infrastructure issues. Compare to "we have to move, our lease is up" or "our business card supplier is out of business and we must switch suppliers"—the same business drivers drive the same response.

Service Offering

A Service Offering is a defined entry in the enterprise service catalog. It is a measurable and specific offering of the IT organization to external clients. It should be seen as a "logical API," or application programming interface, of the service provider; everything behind it (in theory) may be opaque to the service consumer. Service Offerings are of two major types: Orderable Service and Hosting Service. (In this model, the Project orders the Hosting Service using a Service Request.)

The Project can be seen as the Service Offering of "create new service."

Service Offerings and Services themselves may be created by Projects. In effect, the Project can be seen as the Service Offering of "create new Service."

In ITIL terms, an Orderable Service might be seen as (by definition) a preapproved RFC. Access to an existing Application (sometimes termed a subscription) would be one type of Service Offering.

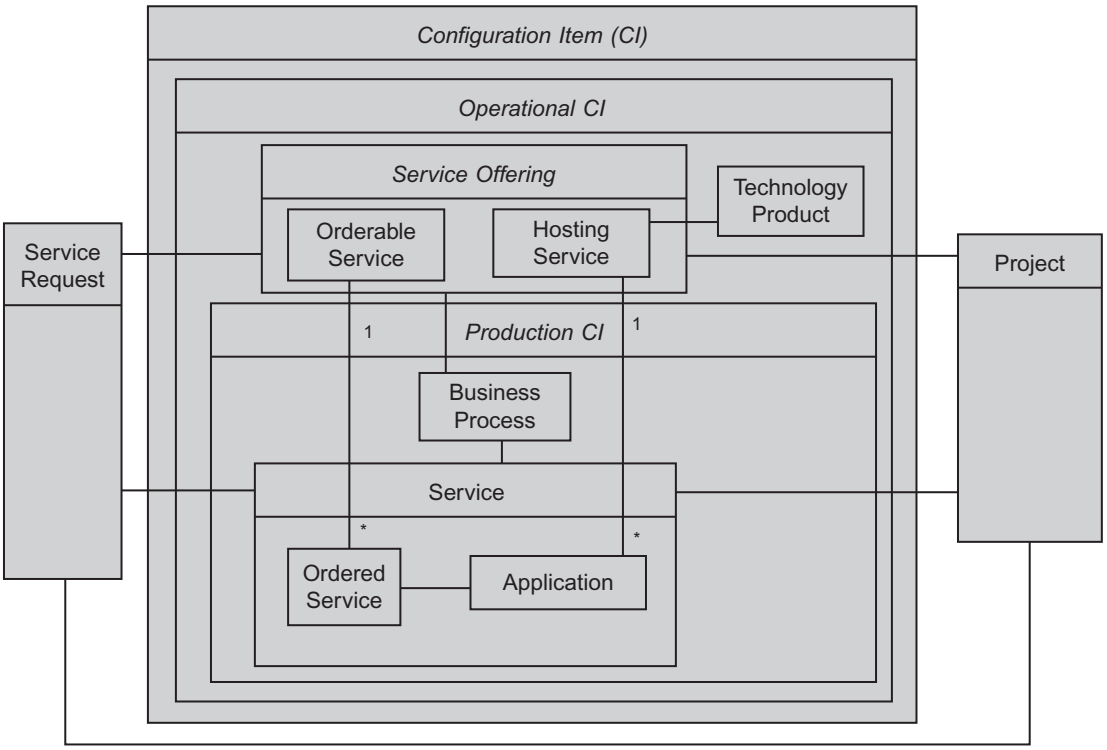


Figure 3.28 Service Offering context.

The Hosting Service is the infrastructure and support services necessary as a platform for an Application.

The Hosting Service is the infrastructure and support services necessary as a platform for an Application. Instances of a Hosting Service are Applications; the Hosting Service is a sort of approved template for how standard Applications are built.

Hosting Services are not preapproved RFCs; they require extensive validation. Ordering a Hosting Service usually implies starting an implementation Project. Hosting Services are based in turn on standard Technology Product stacks.

Projects may involve both Orderable and Hosting Services and their actual Service instances.

Notice that the service *definitions* (Orderable Service and Hosting Service) are Operational CIs. This means that although they can be measured, they do not emit Events and are thus not production concerns. However, their instances are—both the Ordered Service instance and the Application service are production concepts.

Notice the symmetry: A Service Request turns an Orderable Service into an Ordered Service. A Project turns a Hosting Service into an Application.

Service Catalog Confusion

Pat: We're doing a service catalog.

Kelly: So are we. How many services are you going to have in yours, do you think?

Pat: About 20.

Kelly: We're past 500 and counting!

Pat: Seems high.

Kelly: I know you have more than 20; just the other day you said you were managing 45 different SLAs.

Pat: Oh, those are mostly our applications.

Kelly: Aren't those in your service catalog?

Pat: No, of course not. Are they in yours?

Kelly: Yes, of course. They are the major things we're managing for the business. How can they not be in your service catalog? Service-level agreement, service catalog—same thing, right?

Pat: We have something called a hosting service that covers all our applications. Each application is an instance of that hosting service. We manage the hosting services as a different portfolio, but we don't call that our service catalog.

Kelly: I don't see how that can work. We "host" two enormous mainframe applications that are worlds unto themselves, a bunch of midrange stuff, and then dozens and dozens of smaller scale Web apps. I could see the Web apps being instances of a generic hosting service, but what about the bigger stuff?

Pat: Well, as you know we don't have anything quite as huge as yours—lots of medium-sized stuff. We did define several tiers of hosting, based on capacity and availability requirements. What if you took your two biggest applications, kept them as separate service catalog entries, and saw the rest as simply hosting instances? Are the rest of the applications generally comparable?

Kelly: Maybe… I'll have to think about that.

A Service Offering is not a service.

A Service Offering is not a service. The Service Offering is a template, an item *type*—but it is not the item. One Service Offering may result in many actual Services; in other cases, a Service may not even have an Offering (it is a nonorderable service). However, an Offering with no Ordered Services is like a poorly selling retail product; its reason for being is clearly in question. (This is where portfolio management comes in.)

Examples of Service Offerings might be the following:

- Provision new user with a workstation
- Set up new email account
- Set up new user in human resource management system
- The three preceding bulleted examples, all as a package
- Provision new remote store with wide area networking
- Provision Project with new technology stack (e.g., Java 2 platform, enterprise edition) standard container and Oracle database)—notice that this is an internal, IT-to-IT Service

Service Offerings in some cases will reference single or multiple Technology Products that may be composed of other Technology Products (the term "stack" may be used here).

For example, one Service Offering may be "provision HA (high availability) Enterprise Java with RDBMS." This Service might be the configuration and delivery of an enterprise Java application server using WebLogic 8.0 and Oracle 9i, load balanced across enterprise standard servers and managed for failover.

The overall stack record would have dependencies, in turn, on WebLogic 8.0 and Oracle 9i and the necessary server infrastructure to enable HA.

There is risk of making Service Offerings and Services too granular. A distinguishing feature of any Service *Offering* is that it must have a quantifiable price. (Not all *Services* must have a price. They ideally have a quantifiable cost, however.)

A Service in this sense is not a specific technical offering like a Web service; a specific Web service would be a Component and would be linked using the Application entity.

Service–Service Offering

A Service Offering may have many Service instances. See the Service discussion later in this chapter. Also see the "On the Relationship between Service and Application" section.

Service Offering–Business Process

A Service Offering may both support a Business Process and depend on one. Service Offerings in some discussions of ITIL break down into technical versus professional services; orderable professional services can be seen as Business Processes. This reference model assumes that professional services are always based on a process and not functional.[173]

Generally, any Service Offering may require a Business Process to realize it as Service.

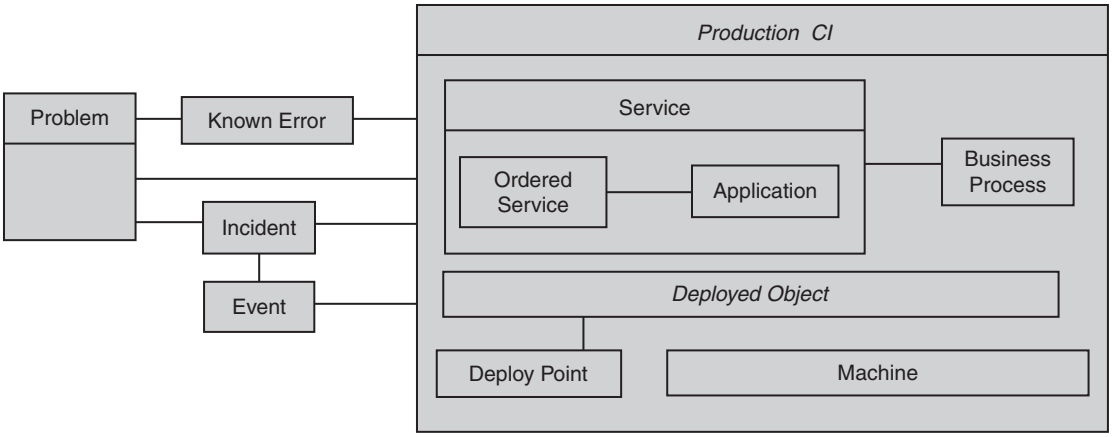


Figure 3.29 Production CI context.

Production Configuration Item

A Production CI is something that's directly involved in the day-to-day delivery of IT services and whose failure or compromise would have an identifiable effect.

A Production CI is where the rubber meets the road. It's something that's directly involved in the day-to-day delivery of IT Services and whose failure or compromise would have an identifiable effect on the customer's value chain. Production CIs are best thought of as the data center and all its components, the networks, and the production workstations attached to those networks. A Service is itself a Production CI, a high-level logical one that serves as a sort of interface by which the consumer interacts with or gains value from the complex underlying IT infrastructure.

Production CIs

Production CIs do not have to be *in* production; just *intended for*. A quality assurance instance of an enterprise application is still a Production CI. It is the fact of being a deployable candidate for operational monitoring that makes it a Production CI.

Production CIs are often logical (Service, Process, and Application). This makes them no less important. Managing the logical CI is one of the most challenging aspects of configuration management; a clear approval and publication process is required.

The concept of "production" can be paradoxical.

The concept of "production" can be paradoxical. As the development life cycle becomes increasingly mature, a developer's workstations and lab servers are seen as "production" assets supporting the Business Process of software development.

A true nonproduction status increasingly must be reserved for pure "sandbox" research and development machines being used to evaluate products and technologies. A workstation being used to develop software upon a standard, proven Java or Oracle technology stack, to tight time frames and deliverables, is a different thing from a prototype workstation brought in to demonstrate the viability of a new 64-bit architecture or experiment with a new encryption product. In short, "development" is "production" to the IT value chain—but not to the business value chain.

Production CI–Event

One distinguishing feature of a Production CI is that it is the only CI type that may raise a monitored Event. Almost without exception, only physical Components, Servers, Machines, automated Processes, or Datastores[174] can raise Events.

Production CI–Incident

Another distinguishing feature of a Production CI is that that is the only CI type against which an Incident can be registered. Incidents can be against logical CIs (e.g., Application), either through a Service Request or through event correlation.

Production CI–Known Error

Another distinguishing feature of a Production CI is that that is the only CI type that may have a Known Error.

Business Process

A Business Process is a defined set of tasks, usually executed in sequence, that results in a specific business objective.

A Business Process is a defined set of activities, usually executed in sequence, that results in one or more specific business objectives (according to process guru Michael Hammer, it must "provide value for the customer"[175]). A Process is generally the intersection point of IT and the business.

Business Processes should be managed as distinct CIs with clear names, identities, and life cycles (e.g., pilot, production, and retired); formalizing their management is a challenge today, and most organizations have an informal process portfolio based on undocumented group consensus. ITIL states that for IT processes "…the process definition itself…should be treated as a CI…";[176] why limit just to IT processes?

It is a hierarchical concept with much ambiguity around granularity; there are various decompositions such as workflow–task–step. At the highest level, a Process is a value chain, and relatively few exist in a given enterprise.

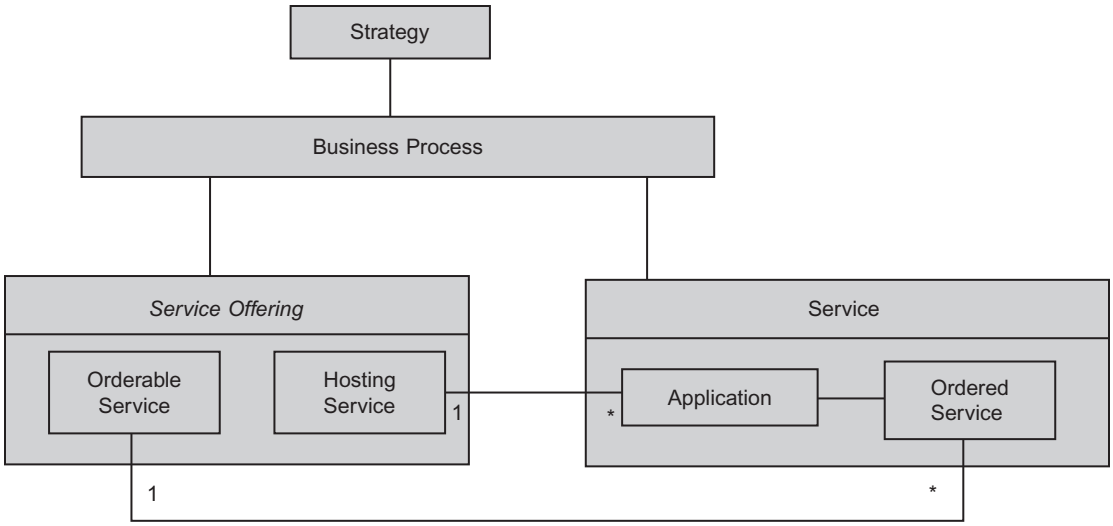


Figure 3.30 Business Process context.

Business Processes may be automated, manual, or (often) both. Many IT processes have critical manual steps, and in an IT organizational culture the importance of these manual steps and the need to make them repeatable may not be appreciated.

Computing Processes

Computing processes (such as those you can see by hitting Ctrl–Alt–Del in Windows NT/2000/XP) are different from Business Processes; they have a specific definition in operating system architectures.

IT processes are Business Processes as well.

This framework does *not* distinguish between "business" and "IT" processes; IT processes are Business Processes as well—just supporting processes, not primary value chain. They are no more special than human resources, property management, or financial processes.

Formally managing a process portfolio results in the interesting metaquestion guaranteed to glaze the eyes of executives: "What is the process to manage the processes?" (Something like, "What is the data about the data?")

For further information, see the literature on BPM cited in "Further Reading." (Note that there is ambiguity in the process management terminology; BPM is sometimes restricted to runtime process management engines. The usage here

is more general, referring to the work of authors such as Paul Harmon, Geary Rummler, and Alan Brache.)

If you are enabling a capacity planning capability in your IT organization, you may have a need for transaction in your data model, for example, to map end-to-end transaction paths. This would be a decomposition or subtype of process.[177]

Strategy–Business Process

Business strategies depend on processes in many or even most cases. Business Processes are a primary vehicle for implementing strategies.

Business Process–Service Offering

Business Processes may depend on routine Service Request fulfillment; this can be seen in part as a decomposition of the process into more specific workflows. Service Offerings in turn may depend on, or be described in terms of, Business Processes (e.g., "Provision new email user").

Business Process–Service

Business Processes depend on IT Services to enable them, typically Applications. IT Services may also require Business Processes.

Service

A service is an instance of a Service Offering.

Service is a general concept with two major subtypes: Ordered Service and Application. Where the Orderable Service may be "provide email to new user," the Ordered Service is "provide email to Peter Baskerville," accompanied by the various workflow steps documenting the provision of that Service from start to finish. (In this case, the Service Offering is a Subscription.)

Services may not depend on automation. The IT organization may provide a purely human-based Process with no Application involved; it may provide a Service based strictly on the availability and performance of an Application, or it may provide both—a Service based on the human execution of a Process backed by automated Applications.

The Service aspect of Applications is distinct from Services focused on provisioning consumers. Provisioning consumers results in many Services for one Service Offering (Figure 3.32).

Service Offerings often require average turnaround times as part of their SLA (e.g., provision email within 48 hours).

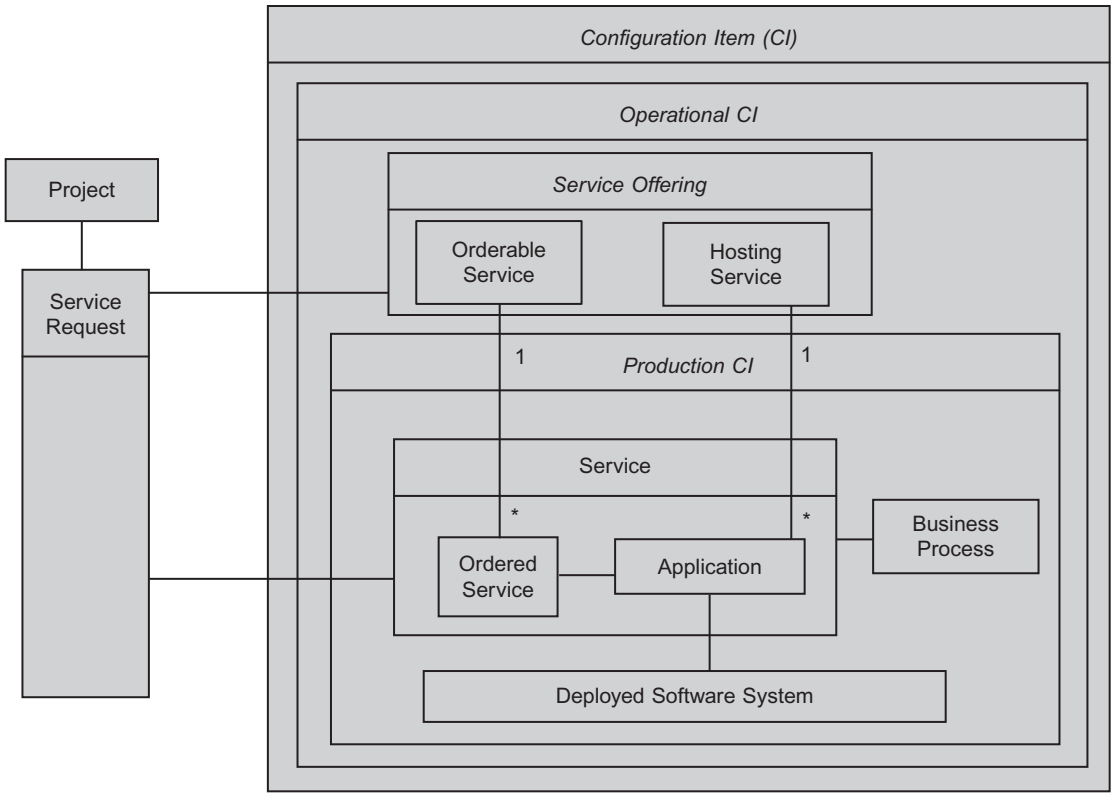


Figure 3.31 Service context.

A Service Offering of access to a given Application may be termed a subscription. However, the following are Application Services:

- Maintain the Quadrex system up with 99.99% availability over 12 months and 99.995% availability during the peak season.
- Complete the X-time batch by 8:00 AM every weekday 99% of the business days.

Another emerging term for Application services are nonorderable services. They are the subject of SLAs based on measured behavior of the Application (e.g., performance and availability).

Another term for Application Services are "nonorderable Services." This means that although they are measured, they are not requested, or to be precise, they are "ordered" through the Demand–Program–Project life cycle—a different service entry point from standard Service Requests. A current consideration in ITSM is the blurry boundary between discrete atomic services such as "order new workstation" and project-based "time and materials" requests such as "Build a new application"—see the discussion on service entry points in Chapter 2 and the "Clarify Service Entry Points" pattern in Chapter 5.

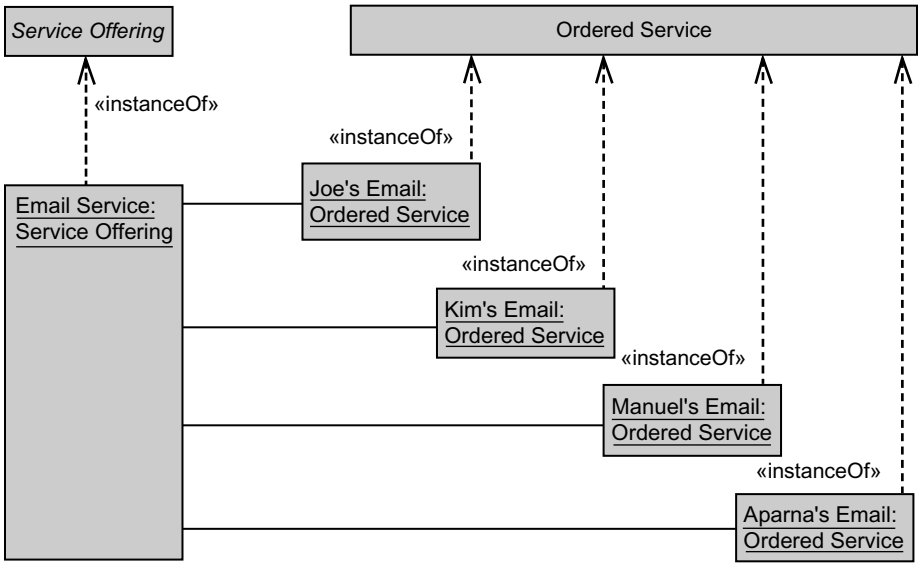


Figure 3.32 Orderable Service Offering and instances.

Their ongoing maintenance is assumed and may be the subject of SLAs, but those SLAs are not based on workflow (e.g., speed of request fulfillment): they are based on measured behavior of the nonorderable Service (e.g., availability). Nonorderable Services do not have a Service Offering entry. Note that for comprehensive service-level management, both Service Offerings and Services need to be tracked. However, Applications may offer subscriptions that *are* Orderable Services.

An Application may play a part in supporting Service Offerings, especially with respect to provisioning (Figure 3.33).

Orderable and nonorderable services are two very different types of entities.

The existence of both Orderable and nonorderable Services has implications for the Service catalog structure. Although a unified report may be desirable from a management visibility perspective, these are nevertheless two very different types of entities and will need to be distinguished in any Service catalog presentation.

Both Orderable and Application services can face inward or outward (see Table 3.2).

Is a Project an Orderable Service? This is a question the IT organization will have to answer. This model treats Projects as distinct from Service Offerings because they are neither preapproved nor fixed in cost.

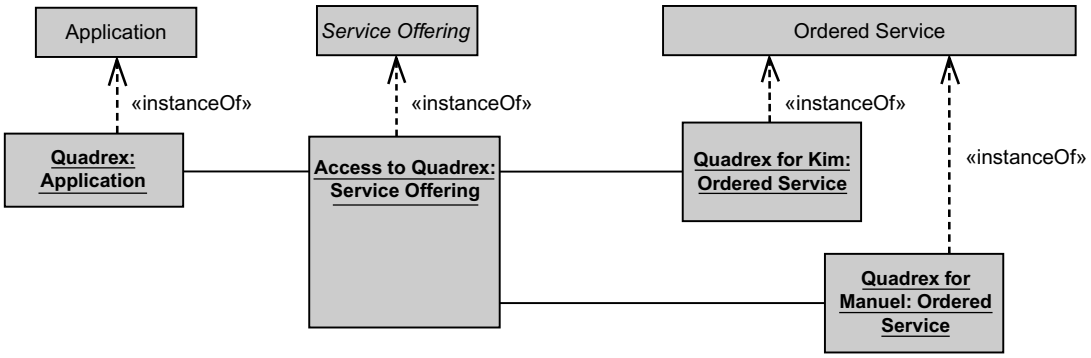


Figure 3.33 Orderable application-based Service and instances.

Services as CIs can contain other Services. This may be useful if several Application services underpin a larger, customer-facing Service concept; however, the Applications themselves should be large grained enough to be recognizable to the business. Smaller-grained, more technical groupings of software are Deployed Software Systems.

As you can see in Figure 3.34, the email Service is underpinned by mainframe and internet email logical Applications, themselves Services. Notice that although

Table 3.2 Service-Type Matrix

Type of Service	Consumer	Internal
Orderable: Fixed cost	New PC (standard configuration)	New server (standardized technology stack)
	New email account (e.g., application subscription)	New database (existing shared database farm with clear pricing model)
	Priced application enhancements (e.g., standard report requests)	
Orderable: Time and materials	New PC (custom configuration)	New server (nonstandard configuration)
	New application project	
	Application enhancements, nonpriced	
Nonorderable (application)	Existing business-facing Service with SLA	Existing infrastructure Service with OLA

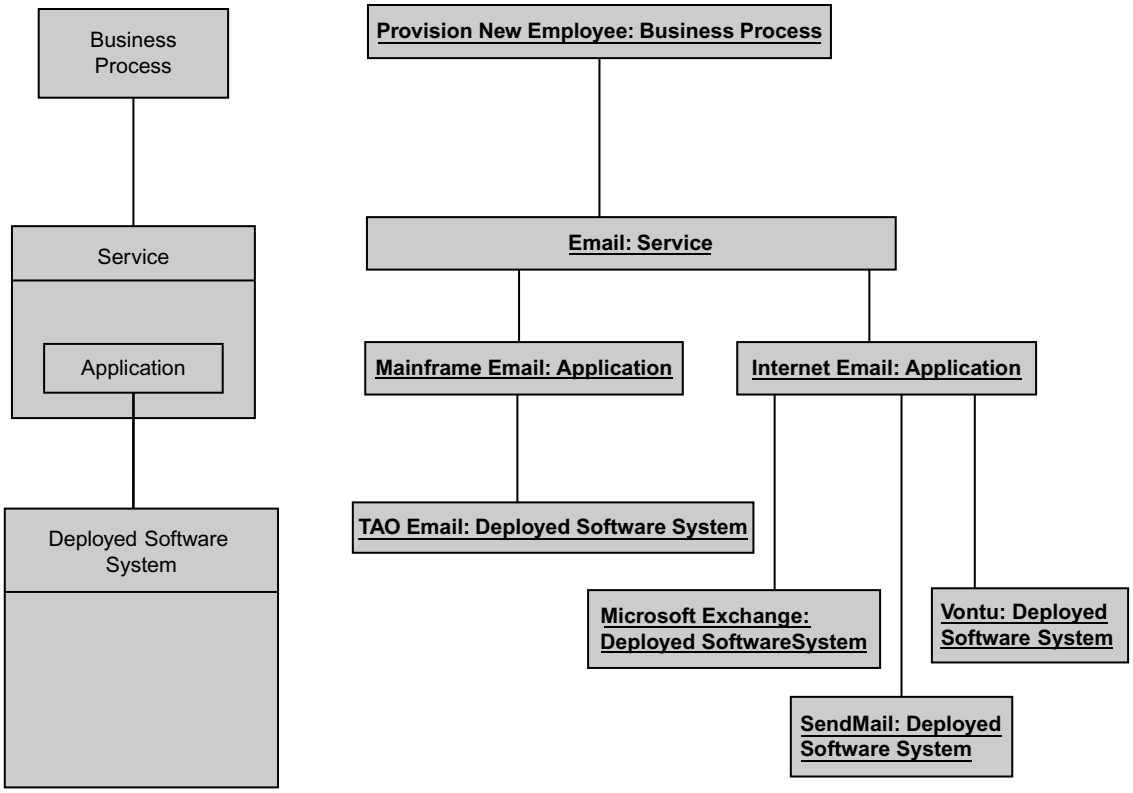


Figure 3.34 Service layering.

the email Service as a whole is the customer value proposition, the underpinning mainframe email and internet email Applications are large grained enough to be recognizable points of investment and support and are themselves managed as Services—not mere technology.

There are many variations on these concepts. In some cases, the Application is the Service—no need for an intervening layer. The critical point is that the enterprise needs to develop a coherent and universal view on these dependencies. It is not acceptable for the architects to have one representation and operations to have a completely different view—although one may be a subset of the other. Naming in particular must be based on common reference data, which in data management circles is known as a master data management problem.

One heuristic for the highest-level business-facing Service concept is that it be traceable directly to a quantifiable business value chain. Understanding the revenue dependencies of a Service is essential for correctly prioritizing the IT organization's activities, but too often this information is locked only in the heads of the most senior executives. It should be broadly available and transparent (within judicious security boundaries).

The highest-level business-facing IT Services are privileged and should be easily separable from lower-level internal Services. But both are distinct from mere Deployed Software Systems, which are purely technical in nature and do not, for example, ever have SLAs or OLAs.

API as Metaphor for ITSM

The API is a key concept to object- and component-oriented development; the implementation details of a software component are encapsulated behind a defined set of gateway operations (Figure 3.35).

The idea is that 1) the only way to access the program's functionality is through the interface and 2) it is no concern of the user how the program does its job; it can be radically revised as long as the interface still exhibits the same behavior.

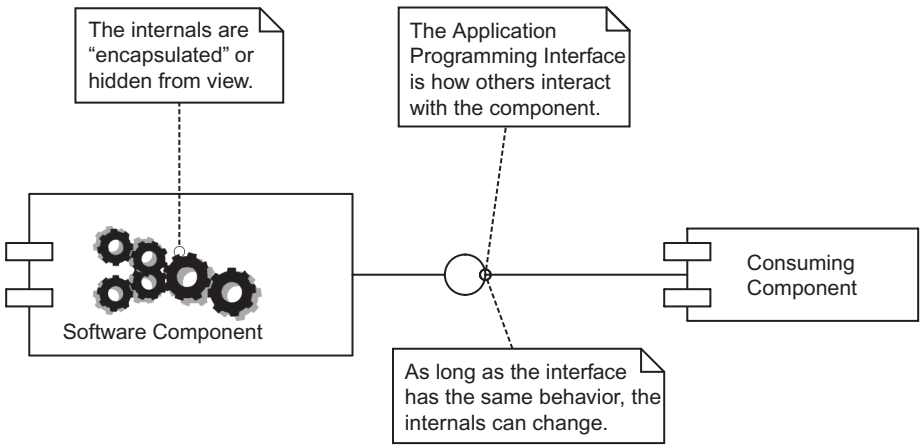


Figure 3.35 Components and interfaces.

(continued)

This is a perfect analogy for Service Offerings and Services. To carry it further, the Service Offering is the API definition, and a Service is a particular invocation of the API.

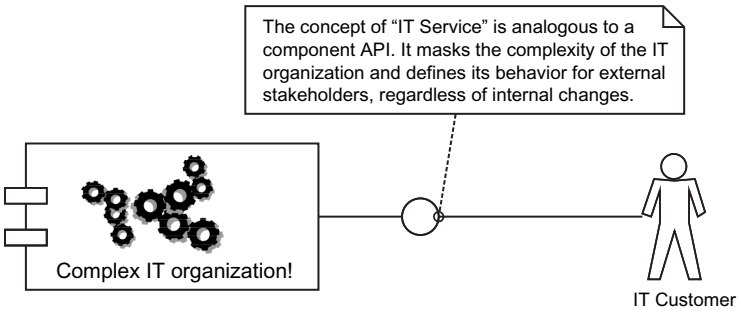


Figure 3.36 Service as API.

Application

An Application is a logical grouping of software Components. It is a consensus concept and must be carefully crafted so that it is neither too abstract nor too granular.

This is also known as product, software, software service, or middleware.

An Application is a logical grouping of software Components managed as a Service in the ITSM sense. Technologists may liken it to a "namespace." It is a consensus concept and must be carefully crafted so that it is neither too abstract nor too granular. Some rules of thumb that may be useful:

- An Application should be recognizable to a senior business manager. It is first and foremost a *portfolio* concept.
- Applications should be assigned to financial management structures. They should have clear executive ownership.
- Applications may be instances of a Hosting Service if the Organization has formalized these as Service Offerings.
- An Application usually will have been the sole product of a Project, but subsequent Projects may be managed to enhance it. (Not all projects result in the creation of an application.)
- An Application may be externally hosted (i.e., Software as a Service).
- Databases are not necessarily owned by any one application.

All CIs owned by the Application should be named using its identifier.

- Applications should have a unique human memorable identifier, ideally a three- or four-letter acronym. All CIs owned by the Application should be named using that identifier as a basis for a naming standard. (Vendor-delivered software is not renamed but should still have an identifier assigned for security identification.)

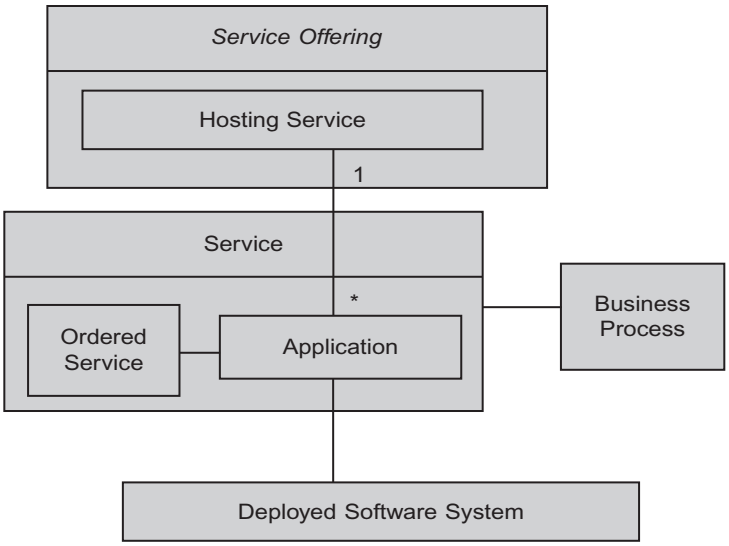


Figure 3.37 Application context.

Events emitted by the Application's Components should have this identifier, if possible.

- The same Application may have different informal names in the Organization; therefore, an aliasing capability is essential to manage the portfolio and eliminate redundancy while supporting legacy terminology.
- Applications in this model are specific instances. If an organization has two instances of Oracle Financials (e.g., for two different operating companies) supported by two different support teams, that should be two entries in the portfolio. Oracle Financials would also have one record as a Technology Product for each major version.

If no one wears a pager for it, it may not be an application.

If no one wears a pager for it, it may not be an Application, as Applications are subtypes of Service. If an Application is not part of an identifiable Service, it might be a Technology Product. For example, if an IT organization uses WebSphere Application Server for multiple different applications, WebSphere might not be in the Application portfolio—it would be a Technology Product (possibly part of a stack) and Deployed Software System on which Applications depend. However, if a shared WebSphere server farm is managed as an entity with perhaps an OLA by an infrastructure team, then that should be in the Application portfolio.

One problem with a strict application taxonomy is that actual applications often fall into more than one category.

Applications may have various types, with a common distinction being between "business" and "infrastructure." "Customer facing" versus "back office" is another sustainable distinction. Figure 3.38 shows a simple Application classification; more elaborate taxonomies are possible, but complexity may be hard to maintain, especially in terms of sustaining mindshare and driving effective use. There are vendors of in-depth classification taxonomies that may be useful in some cases. One problem with a strict application taxonomy is that actual applications often fall into more than one category.

Note in Figure 3.38 the question as to whether an ITSM Application is a business-facing or infrastructure Application. This is more than an academic distinction, as it may affect which major organization supports the application. Classifying such applications as "back office" is more in alignment with the IT Enablement

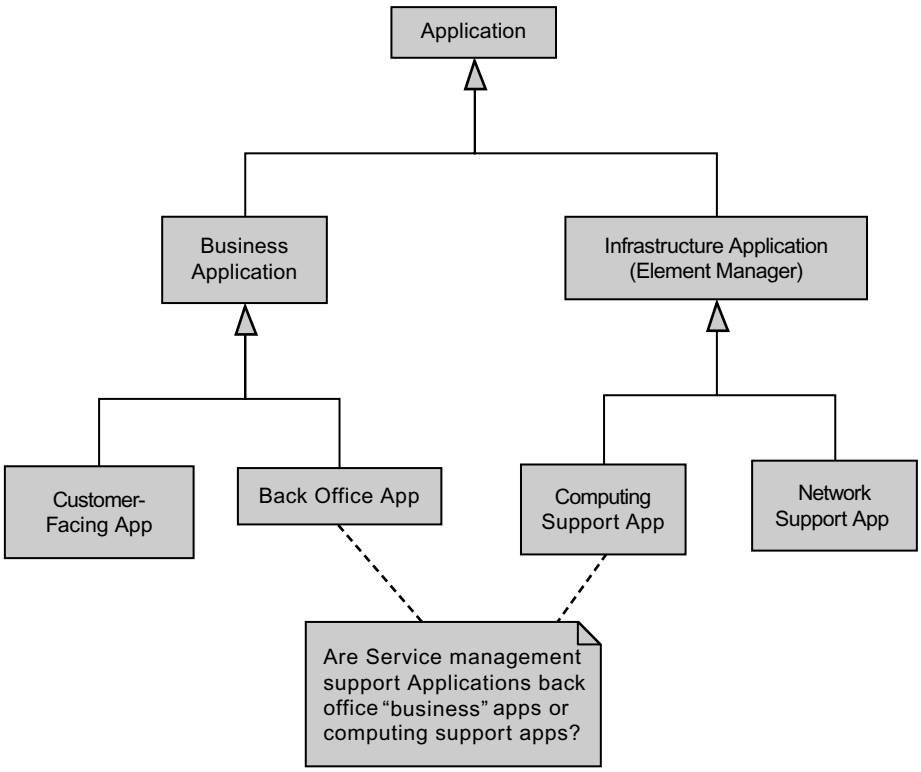


Figure 3.38 Sample application taxonomy and a key question.

Capability pattern. With this approach, all infrastructure Applications are focused on element management and may be managed by their own team. (See the "IT Enablement Capability" pattern in Chapter 5 and the discussions of element management.)

Applications as Portfolio

The Application portfolio is probably the most important set of CIs to baseline for a data center–focused ITSM initiative.

The Application portfolio is a key set of CIs to baseline for an ITSM initiative concerned with the data center. Physical devices will be seen as highest priority, but these usually have some attempts at management; the master list of applications, on the other hand, often does not receive explicit management.

Some CMDB efforts fail because they attempt to start with the concept of physical binary Component, which (while straightforward to harvest) is too granular and hard to manage for most organizations. The logical concept of Application provides a bridge between the overwhelming details of the technology and the business it supports.

A defined process must be implemented for identifying that something is to be tracked as a formal Application, for example, requiring the agreement of an architect and an IT line manager. Proliferation of Application identifiers (which can happen if a nonarchitectural, technical team is allowed to assign them) is a bad practice because it prevents the accurate rollup of IT operational data into larger, business-aligned hierarchies for IT performance reporting.

A defined process must be implemented for formally identifying Applications.

This model does not distinguish between Application and middleware. It's assumed that the Application entity if implemented would have a "type" attribute and this distinction could be handled at that level. Both Applications and middleware behave similarly in terms of the relationships to other entities, and the boundary between them can be blurry.

Middleware can be both a Service and a Technology Product. A middleware "hub" operated as a shared enterprise service is an Application, probably infrastructure, as well as a Technology Product and instance of a Deployed Software System. A middleware product used as a building block by many different service providers (e.g., application teams) is only a Technology Product.

Middleware as a Service, however, generally would not be business facing.

Application identifiers should be visible on all CIs where appropriate, in particular on Web pages and other graphical user interfaces. There is currently a problem in the industry with inaccurate CI identification: users do not necessarily know what Application they are even interacting with. Firm labeling standards for all Application interfaces would be a big help. This is

nothing new; on older mainframe green screen systems, the system and screen identifiers would typically appear in a corner. New distributed systems with less rigorous graphical user interfaces development standards were a step backward in this concern; off-the-shelf packages could easily add this as a configurable functionality.

Disparate Application Portfolios

A Fortune 100 corporation established an Integration Competency Center, which began to track the difficult subject of application interdependencies. The group tasked with this goal realized the first priority was to establish a definitive list of applications. (How can you define *relationships* between "things" when you are not sure what the "things" are?)

The application support and maintenance team had a list, but it only included applications that had been formally "turned over" and some key applications had never gone through this process. It also had poor data quality, with applications listed for which no physical evidence or owner could be found and other applications listed twice (by different names).

The production control group was responsible for assigning "system codes," three-character identifiers associated with the logical application concept. However, they never had strong criteria for doing so, and as a result the codes tended to proliferate, with one logical application sometimes having many codes. In other cases, one code would be used by a large application area for all applications.

The distributed server engineering group had a list of distributed applications and their dependencies on servers, but it did not include mainframe applications and had no defined process for maintenance.

A consulting group was brought in to reinventory all the applications, and this resulted in one more list. Lists were also compiled for compliance and disaster-planning activities. It became clear that there was significant waste and redundancy occurring.

The Integration Competency Center declared itself system of record for the application portfolio and defined a process for maintaining applications and their stakeholders and dependencies. The enterprise architecture, compliance, and security teams began to partner on these processes, which helped enable tighter controls. The application identifier assignment was seen as a key component and added to the mix, with tighter policies aimed at ensuring "one application, one code." This list then served as the basis for first-generation configuration management; databases and servers were linked to the applications and the capability took off from there, becoming recognized as a valuable IT asset.

Application–Application

Applications have many interrelationships between each other, which should be documented in the repository or CMDB. Approach issues to be sorted out here include the distinction between Application-to-Application dependencies (i.e., at the API layer) and Datastore-to-Datastore dependencies (i.e., the extract, transform, and load domain). Another issue is the danger of capturing trivial dependencies, for example, the near-universal dependency of all distributed computing on the TCP/IP system infrastructure (which should be captured as an infrastructure Application or Service in the repository).

Application–Component

Applications contain Components. For accountability, all Deployed Components should be owned by one and only one Application (although they may be used by many).

Application–Datastore

Application-to-data dependency is one of the most important production dependencies to understand.

Applications are collections of processes and algorithms at their core. They depend, in turn, on Datastores such as relational databases or flat files. Application-to-data dependency is one of the most important dependencies to maintain for CIs in the data center; many organizations spend considerable resources continually reanalyzing this dependency. One immature approach is to simply document the dependency of an Application on a database Server (without specifying catalog or database); however, database Servers are often large, shared assets and the database administrators need to know *exactly which database,* or schema, is serving an Application. (This is also needed for regulatory compliance.)

Application–Deployed Software System

Applications depend on Deployed Software Systems. The distinction between the two is subtle but crucial. Deployed software systems are all software Components that support the Application. They include the actual software Components embodying the logical Application, as well as application servers, DBMS engines, operating system services, middleware, and so forth. They should not be business visible.

A sign of an immature environment is when Projects are confused with Applications.

On the Relationship between Project and Application

A sign of an immature IT enablement environment is when Projects are confused with Applications. Projects have a defined life cycle, typically measured in months.

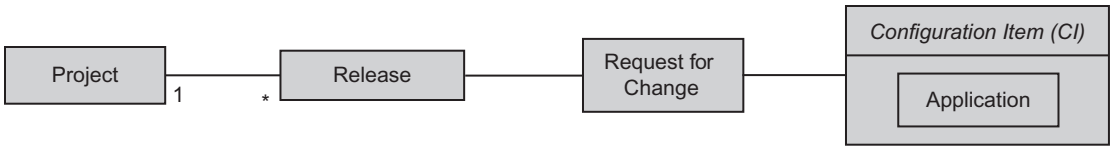


Figure 3.39 Project, Release, and Application.

Applications have an indeterminate life cycle, typically measured in years. One Application is usually the subject of multiple Projects; the first Project creates and deploys it, and subsequent Projects enhance it. *It remains the same Application throughout*, unless a conscious decision is taken to manage a major new version as a distinct new Application. There are various approaches here; the important point is that they be managed and agreed to.

The relationship between Project and Application in the model is mediated through Release and Change (Figure 3.39).

This is a purist approach, and it may be desirable for your IT enablement tooling to simply relate Project and Application—there's quite a bit of value there, even if you haven't sorted out Release yet (Figure 3.40).

For example, if an Application has a known Risk having to do with regulatory compliance, the Project making changes should be held to high standards for process adherence and software quality. That kind of focused emphasis is difficult to achieve consistently without a rich and well-managed IT enablement system that clearly distinguishes between Application and Project. It also speaks for the integration of demand management with ITSM tools to more objectively assess risk and impact (cf. the generalized ITIL Change concept).

Application–Process

> The alignment between the IS [information system] view and the customer view gains value when IS is able to identify the relationship between the technologies and the business processes they support.
>
> —ITIL[178]

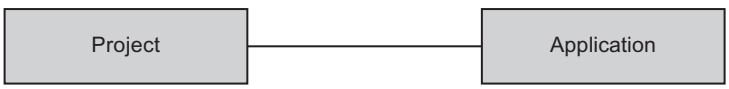


Figure 3.40 Project–Application direct relationship.

Processes are supported by Applications (as Services) in a many-to-many relationship. For example, the pricing process at a large retailer may involve a merchandising system and a point of sale system, provided by different vendors. The merchants set the prices, which are then replicated down to the point of sale terminals. Value is not derived from the process until it runs from end to end, so one process depends on two Applications.

Similarly, it is common for one Application to support two distinct processes, such as a customer relationship management system that supports both operational customer interactions and analytic planning purposes.

Processes can be decomposed into constituent steps, depending on the granularity of the analysis required. One constituent of a process would be a transaction, and understanding the major transactions supported by an Application and/or an underpinning Deployed Software System is useful for portfolio management, capacity planning, financial chargeback, and other purposes.

Deployed software systems increasingly may directly support processes as well, especially in the emerging world of SOA. There may be no concept of an Application—just process choreographies interacting directly with technical services. This is an emerging area and this representation is preliminary.

See also Figures 3.32–3.34 and 3.37 and related discussions.

On the Relationship between Service and Application

Although the IT industry has traditionally made a distinction between Application Development (creating a service) and Service Management (delivering the service), that has not always worked well.

—ITIL[179]

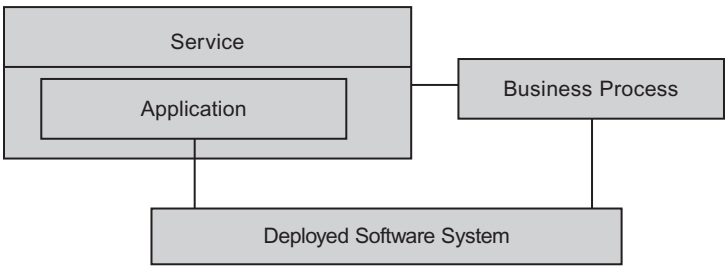


Figure 3.41 Service, process, and Application: complex and subjective.

In many large IT organizations, the "application" team is concerned with service issues and supports people and process, as well as technology.

The relationship between Service and Application is subtle, so subtle that many Organizations may wish to *not* distinguish the two. ITIL is strong on distinguishing the two because its view of Application is technical—it's simply the binary software executed for the customer. However, in many large IT organizations, an Application team is concerned with customer service issues and effectively is supporting a Service or system—not just technology but people and process as well. Such customer-oriented application management teams would be surprised to learn that they are "invisible to the Customer," as ITIL states.[180]

There is great variability in the industry: in some organizations the application teams are indeed merely technical, and in yet other organizations there is no consistency. Some application teams are truly service managers, and others are merely technicians. The inconsistencies erode IT credibility.

However, at least for a first cut inventory, *the enterprise Application can serve as a reasonable surrogate for a Service.* This starts to break down in enterprise applications that are so large they support multiple distinct Business Processes and have multiple stakeholders (perhaps expecting different SLAs). An example might be an ERP system for which the operational customer negotiates 99.99% uptime and a planning group negotiates decision support batch completion by 8:00 AM every day. (Of course, an overall contractual SLA may have multiple specific agreement points in any case—the distinction here is that there are two different customers expecting notification for different types of service breach.)

Conversely, if a set of smaller Applications has been developed with all managed by the same team, these distinct pieces of functionality may be managed increasingly as a unitary Service.

For example, an organization may have a legacy email system on its mainframe and a distributed email system such as Microsoft Exchange. Both may be supported by the same team, and a request for "email access" may result in the customer receiving accounts in both environments. Nevertheless, they should remain two distinct entries in the application portfolio so that there is visibility into the portfolio's complexity and enterprise progress toward simplification (e.g., stopping support for the mainframe email system).

Service versus Application

One way of managing the distinction is linguistic. Where the Application is "Oracle HRMS," the Service might be "human resources application management."

Where the Application is "Oracle HRMS," the Service might be "human resources application management."

This has an advantage of conceptually decoupling the Service to some degree from the Application; however, the added value of this linguistic distinction may be suspect, if all involved (wink, wink, nudge, nudge) know that it simply translates into the same set of services the Oracle HRMS team has provided all along.

The introduction of a layer of abstraction also poses maintenance issues: now *two* logical CIs that are hard to manage must be maintained, with a mapping between them.

See further discussion under the Service Request description (e.g., Figure 3.44) and in the ITIL *Service Delivery* volume under "Service Level Management: What Is a Service?"

What's an Application Manager to Think?

Natalie is an application manager for a large midwestern manufacturer. Her responsibilities include both the development of new functionality for her system (the enterprise customer relationship management system) and its ongoing operations. One day she is called into a meeting at which a senior ITIL consultant is discussing service management.

Gary: The thing you folks need to do is get out of a technology-centered approach to interacting with the business. The business doesn't care about things like "applications"!

Natalie: Excuse me, why do you say that?

Gary: Well, it's clear. The business doesn't know what an application is. You shouldn't even talk about it with them. What they need is a service!

Natalie: I'm not providing a service?

Gary: Not if you are calling yourself an application manager. All that application managers do is build technical stuff.

Natalie: Hmm. I just got out of a meeting with the senior VP for marketing. We were talking about my application's availability level. We even used the term SLA. But this term "service" you're throwing around, we don't talk in quite the same way.

Gary: That's because you are too technical in your approach. See, you need to get out of the bits and bytes and talk in business terms!

Natalie: Like discussing the business objectives of the next major release with the SVP? How the application—excuse me, service—is going to help improve customer retention and sales force productivity?

(continued)

Gary: Right… Say, I thought you said you were just an application manager.
Natalie: I did… Oh, never mind….

Relationship among Service Offering, Service Request, and Service

Now that I have introduced all of these concepts, I will examine how they work together and hopefully clarify why we need them.

The concept of service is tricky; it is used quite freely in the ITSM literature. It's therefore not surprising from a data perspective to find that the term is badly overloaded and requires considerable clarification, including five distinct entities in this discussion. This is not even including "service" as used in SOA (Figure 3.42).

Figure 3.43 shows the interrelationships of the service-related entities for a simple scenario of email provisioning. Note that email provisioning in this enterprise consists of configuring the user's accounts on two different email systems, a good example of one Service being supported by two Applications.

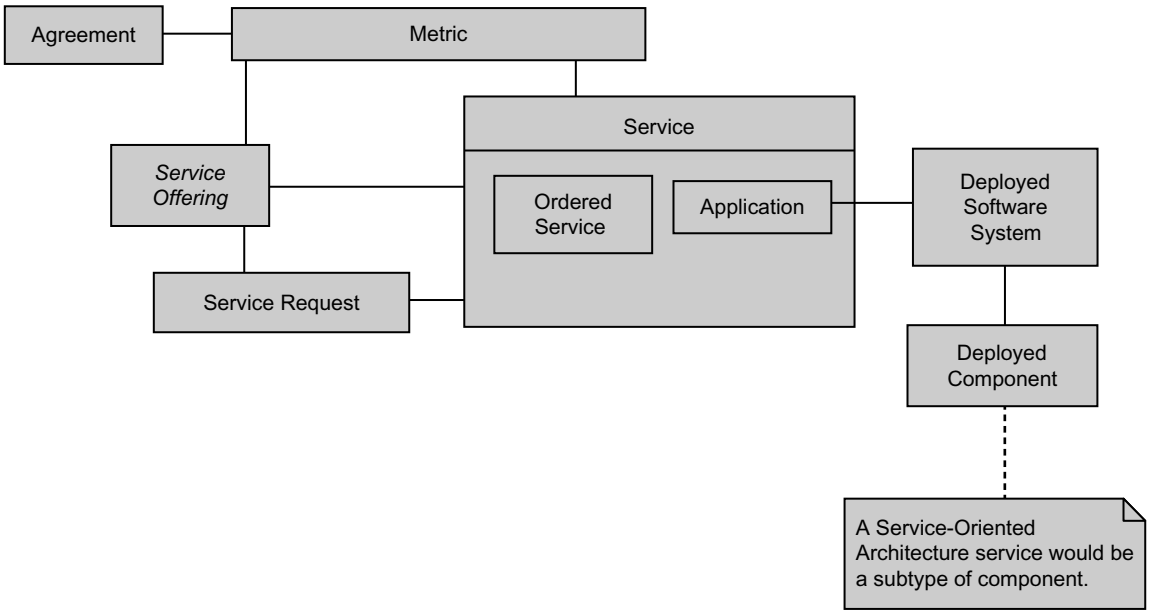


Figure 3.42 Service context: expanded.

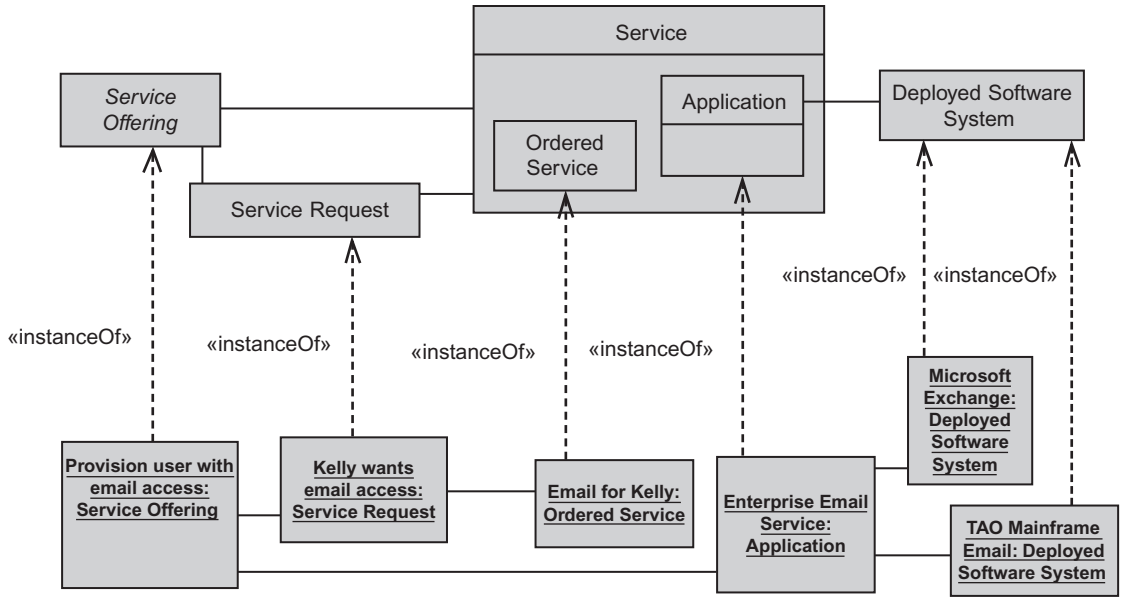
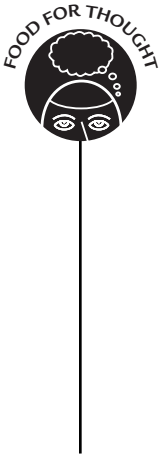


Figure 3.43 Service instance example.

Again, an individual provisioning of the email service to a customer might be called a subscription.



Why Is This So Complicated?

Well, it's really not. It's just unfamiliar. Think about ordering a book from your favorite online retailer.

Say that I log into my favorite online bookseller. It offers in general a Service of selling books, but that is not what I am *ordering*. I am ordering *one book* in the bookseller's equivalent of a service catalog. However, the ongoing performance of that bookstore is a Service as well—a nonorderable Service. (It's as simple as a store keeping its doors open—you don't *purchase* that, but it's *necessary* if you are to enter the store and see what's on the shelves.) The bookstore Service itself is supported by underlying Applications; for example, its own order management system and a delivery logistics system that might be outsourced (e.g., to UPS). I need all of these things to get my book.

(continued)

The one thing that seems a little elaborate is the distinction between Service Request and Ordered Service. However, this is necessary because of the ongoing production nature of ordered IT Services; the bookstore delivers my book and doesn't care about supporting it once I have it, but an IT organization delivers a computer (or email account or disk storage) and then has to provide ongoing support for it.

Deployed Object

A Deployed Object is a Deployed Software System, a Component, or a Datastore. Figure 3.45 attempts to represent an extremely complex space concisely. More elaborate representations are possible,[181] but these core concepts can serve as a basis.

Deploy Point

A deployable object is tied in turn to a Deploy Point, which is usually a file system directory.

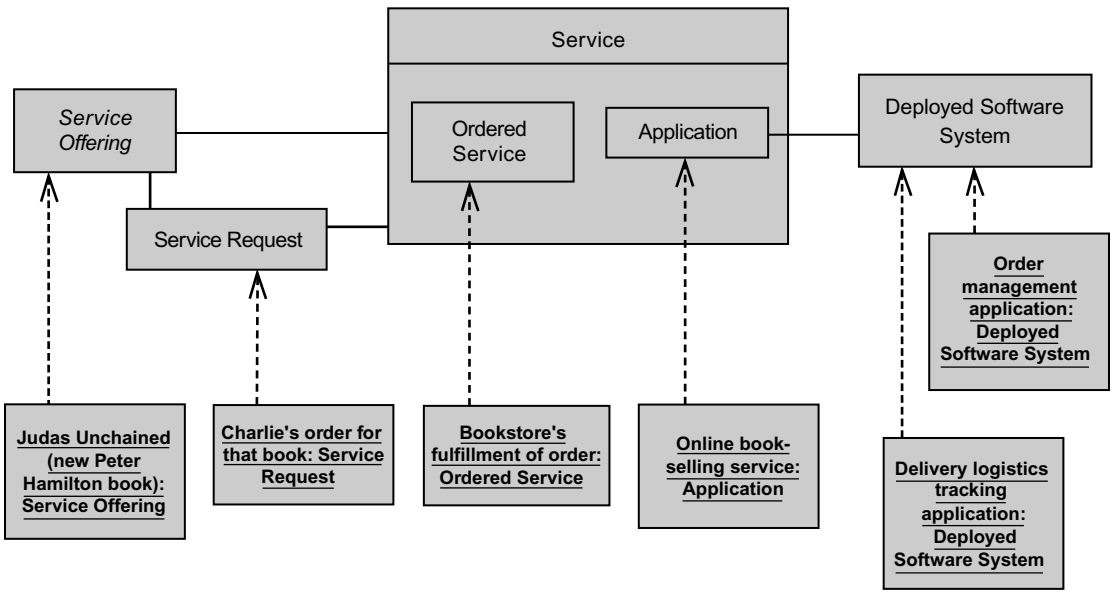


Figure 3.44 Book order as Service example.

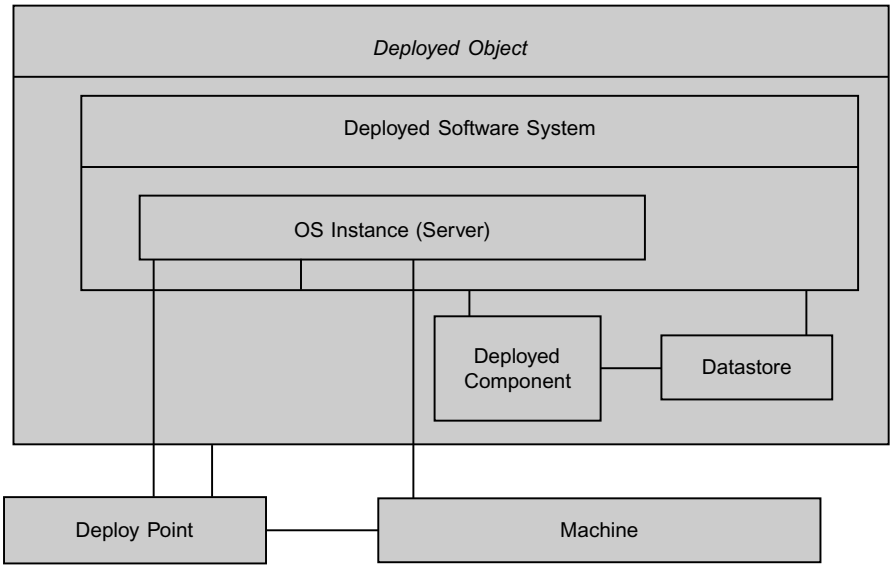


Figure 3.45 Deployed Object context (applies through the end of this section).

A Deploy Point is a major type of CI.

The concept of Deploy Point as a type of CI is an innovation proposed in this analysis and comes from my experience with configuration management and supporting an integration competency center. There are several reasons for this:

- The need to identify "root" directories to facilitate interaction between infrastructure and applications teams (root in this sense not being the base file system object but the top directory allocated to the application team)
- The sensitivity of certain directories when used as exchange points for moving data
- For configuration management approaches that do not enumerate distinct Components but rather perform broad integrity checks across large blocks of storage
- Capacity management of centralized storage and its traceability to application services

The application root directory is a key interaction point for the infrastructure team managing the server and the application team.

The Application Root Directory

A large, complex application may have dozens or hundreds of directories, in some cases appearing and disappearing dynamically. However, with few exceptions the application's scope of activity is constrained to one or a few master directories that

contain myriad subdirectories used by the application. These master directories are a key interaction point for the infrastructure team managing the server and the application team (assuming that the IT organization has moved toward the best practice of segregating these teams and moving the application teams out of the business of server management).

Shared libraries complicate this arrangement, but multiple applications updating shared libraries have been proved to be poor practice in Microsoft Windows. This touches on core computing issues around component reuse and operating system services and architectures, and it will never be a simple matter. Arguably, the move toward server virtualization is in part a response to the complexity of managing shared libraries in a single operating system instance.

The Shared Exchange Directory

Shared directories that facilitate Application interaction are important points of control and need to be treated as CIs.

A problematic design pattern in integration architectures is the shared directory. This is typically a directory in which one application deposits files and another picks them up for further processing or to consume their information.

The trouble with shared directories is that sometimes the consuming application will be built with logic that states, "Do X for all files in the directory." Thus, if an incorrect file is placed in the directory, unexpected results may occur. (An architecture of this nature resulted in the complete failure of the replication feed for all pricing data at a major retailer, costing many hundreds of thousands of dollars and spurring an interest in configuration management.)

Shared directories that facilitate application interaction are therefore important points of control and need to be treated as CIs.

This whole concept may seem obvious; the key point being made here is that *these directories should be explicitly tracked as CIs in the CMDB*, and the stuff they contain is not necessarily individually tracked.

Deployed Software System

A Deployed Software System[182] is a more technical concept than an Application. It is a specific set of computing Components that can be managed as a unit. Applications (which in this model are seen as subtypes of Service) depend on Deployed Software Systems.

Deployed software systems are often the instantiations of Technology Products. They are the *real, running instances.* They support Applications, which in turn figure in SLAs, may have Incidents, and so forth. Technology Products in contrast

show up on invoices and Contracts, and the complete list of *software* Technology Products is the Definitive Software Library.

As you can see in Figure 3.46, Applications as Services depend on a great deal of technology they do not own. Maintaining these relationships is essential for understanding the effect of external forces on the IT organization.

One result of the model's distinction between Technology Product and Application is the apparent duplication in some cases of information across the Technology Product, Deployed Software System, and Application entities, which in simple cases may all have the same informal name.

Technology Product includes "undeployed software" generally, and this is useful in the case of both externally and internally developed products, especially those that have multiple production versions. Again, if a piece of software is to be considered part of the Definitive Software Library, it must be registered as a Technology Product.[183]

A question to consider is whether the custom module entity in Figure 3.46 should also be a Technology Product.

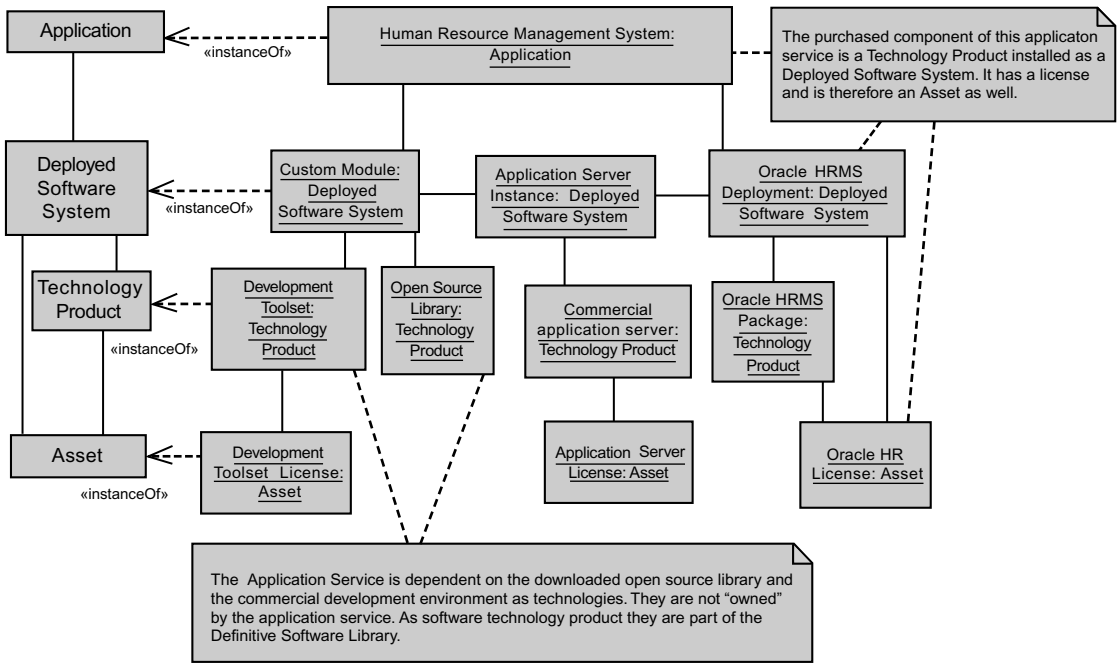


Figure 3.46 Application Service, Deployed Software Systems, Technology Products, and Assets.

Deployed Software Systems do not have SLAs or OLAs. Those concepts are reserved for the Application entity as a subtype of Service.

Operating System Instance (Server) and Machine

Servers and Machines are not the same thing.

A precise definition of Server versus Machine is increasingly critical. Server is becoming an ambiguous term because of virtualization, but as one of the most commonly heard words in IT, it must be addressed in this model, which sees Server

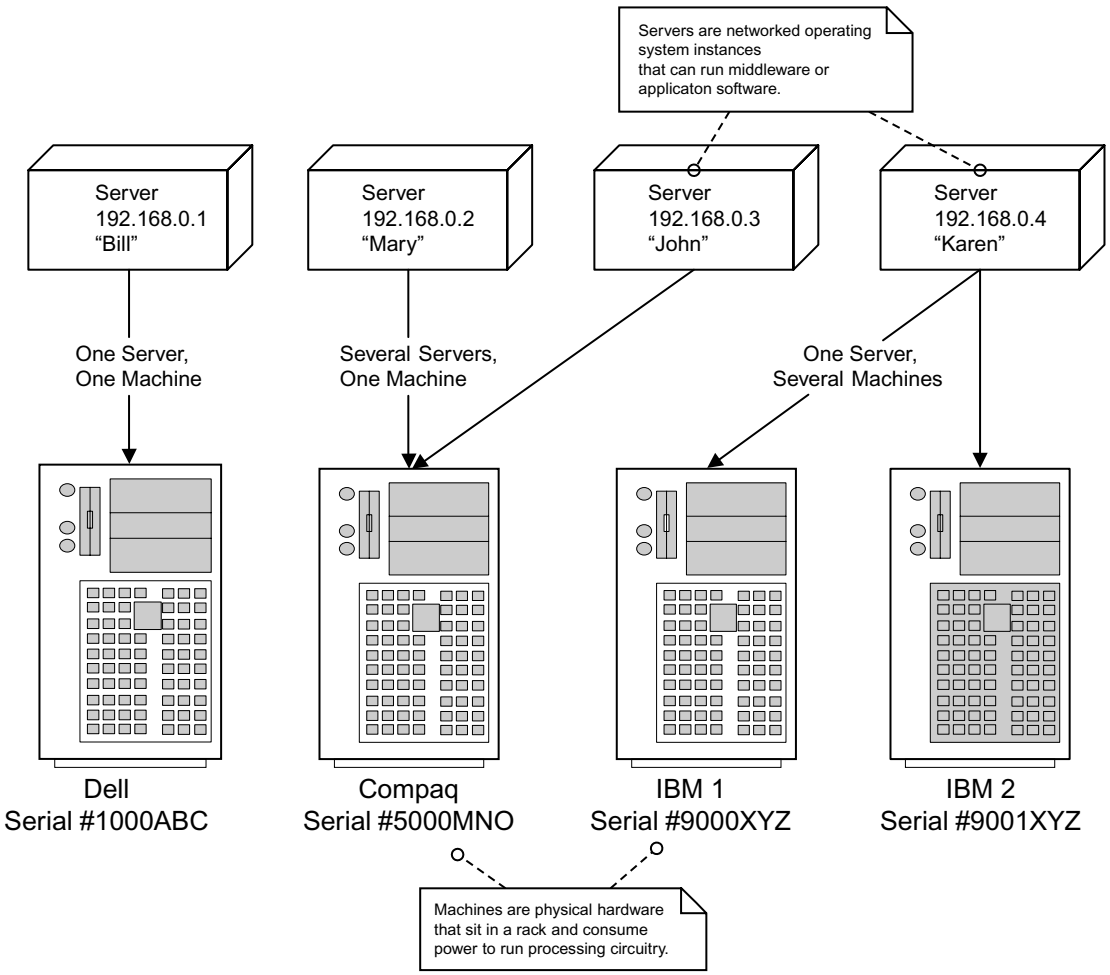


Figure 3.47 Machines and Servers.

(and workstation) as an *operating system instance, almost always networked.* An operating system instance is a special case of a Deployed Software System.

A Machine is a physical computing device that can be equated to an Asset. One Machine may host multiple Servers (virtualization and partitioning), and one Server may be hosted by multiple Machines (failover and load balancing). Server is the bits and the process (often linked to a software license as an Asset); Machine is the atoms and the serial number, linked in turn to a physical Asset tag.

Machines may have subassemblies, including well-recognized components such as disk drives and memory chips but also including full computing devices (blade systems).

Common asset management solutions are just beginning to support these requirements, and in many companies the reality of the computing infrastructure has already outstripped their asset management solutions' capabilities.

Component

A Component is a physical piece of executable code that can be objectively inventoried.

A Component is a physical piece of executable code. Even though it is only magnetic bits and bytes, it is common practice to call a Component "physical." Calling it "physical" in this context means that there is no disagreement about what and where it is; Components are unambiguous assets that can generally be objectively inventoried without debate about their boundaries.

UML "Component"

During the writing of this book I became aware that the UML definition of Component had changed considerably between UML 1 and 2. This book retains the UML 1 sense of the word; the new UML term is "artifact," which I find too general and nonintuitive—it is not a commonly heard industry term in IT operations.

Again, the purpose of this conceptual model is to rationalize commonly heard industry terminology, not to develop a completely precise model, which would require the use of less familiar terms (such as artifact) in support of more rigorous normalization.

The use of Component here is not in a pure object-oriented sense. In the object-oriented world, a Component also has a well-defined interface that encapsulates its behavior and provides an effective contract for anyone who chooses to use it. However, Component as defined here applies to any piece of executable code, regardless of whether it has a well-defined interface.[184]

A Web service, shared object, or other similar addressable, distinct piece of functionality *in this model* is a Component—not a Service. This is quite a point of confusion because of the overloading of the term Service.

Modern discovery tools discover Components in many cases through their associated computing process evident in the operating system. (The concept of computing process is not represented in the model—this is *not* a Business Process.) Computing processes have interesting technical metadata, including the specific command line used to invoke the process by launching an executable. This area moves into more technical concerns out of scope for this conceptual model.

Component Relationships

Components, like Applications, can be related to Datastores and Deploy Points. However, doing dependencies at this level for the general case of a large enterprise IT organization is usually not practical or useful given current industry capabilities—the objects and their dependencies would quickly amount to millions, and the information might not even be available in many cases (e.g., packaged software). Instead of inventorying all the detail of Components, some configuration management approaches focus on overall integrity checks across large blocks of storage. In such cases the deploy point becomes a fundamental CI to manage.

Capturing Component-level dependencies is a recommended best practice for all aspects of EAI.

Capturing Component-level dependencies *is* a recommended best practice for all aspects of EAI.

Datastore

The most well-known example of a Datastore would be a relational database catalog.

A Datastore is a distinct, addressable source of data, usually structured. The most common examples would be database catalog (sometimes imprecisely called an "instance"; this model uses it in the DB2 sense of a query space containing schemas) and flat file; message queues may also be represented here (Figure 3.48).

A Datastore should have one and only one data definition. As a Deployed Object it depends directly on Servers and their underlying Machines. Note that as a CI it can depend on and contain other Datastores. Again, generalized CI containment is frowned on in the model—you don't want Datastores containing Machines!

A database would further decompose into the well-known stack of schema, table, and column (Figure 3.49). Metadata attributes specify the data types, lengths, and so forth of the columns.

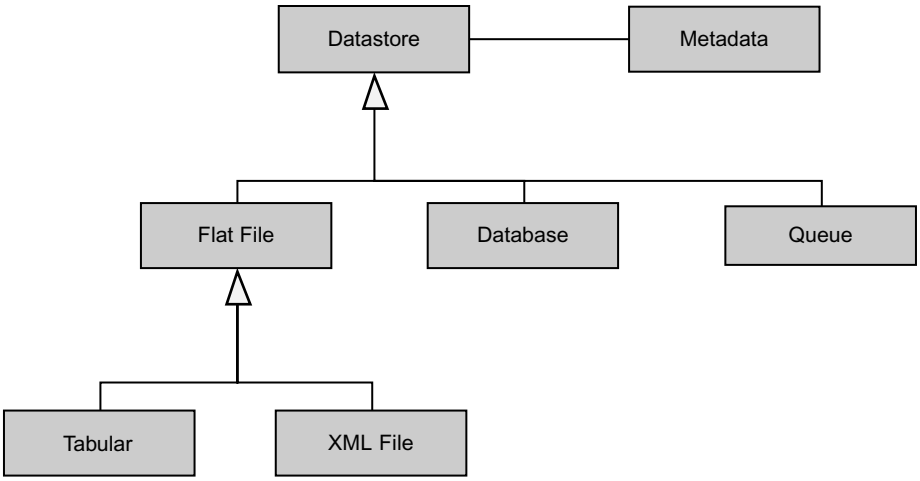


Figure 3.48 Subtypes of Datastore.

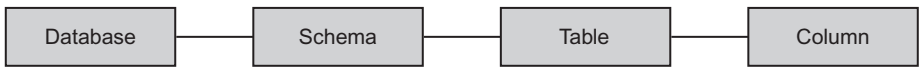


Figure 3.49 Simple data dictionary.

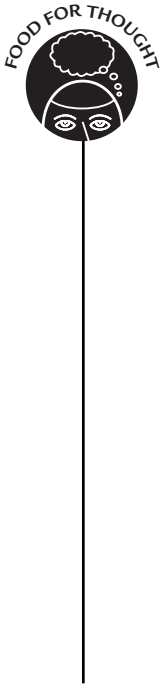
Datastores should have data definitions, which are by strict definition *Metadata*—data about the data.

The data definition tells you whether a given Datastore contains customer or supply chain information. More elaborate representations exist: distinctions between entities, attributes, tables, and columns; the structure of keys and indices; inheritance; and other fundamental information modeling concepts. Making sense of these elaborations requires attention to the issue of what is a Datastore (physical CI) and what is Metadata (its offline representation in a structured format). As noted in the Metadata section, this is one of the more difficult conceptual areas in the book.

See the OMG's Common Warehouse Metamodel and other metamodels and the work of David Marco, David Jennings, and Dave Hay (among others). References are noted in "Further Reading."

Datastores are often equated with their relational database management system (RDBMS) instances in casual architectural sketches. Precisely, an RDBMS is an instance of a Technology Product installed as a Deployed Software System, and the Datastore is merely a passive container managed by the RDBMS. However,

this level of precision is sometimes not necessary in earlier phases of configuration management.



Process and Data

The separation of process and data has both a conceptual and a physical driver. Conceptually, it is convenient to think of data as orthogonal to process, a distinction carrying through into fundamental computer science. Practically, the distinction of data and process has been reinforced by the "access time gap": the difference between real-time, processor-driven access to solid-state memory (the province of programming languages) and slower media such as hard disk and tape (the focus of data management as it's evolved over the past 50 years).[185] This distinction is eroding because of advances in hardware capabilities and economics (solid-state memory continues to decline in price, making "in-memory databases" increasingly common). It is also eroding because of ongoing efforts to incorporate persistence semantics directly into higher-level computing languages and eliminate the "object–relational impedance mismatch." The continuing amalgamation of data into the processing realm will have implications for configuration management practice.

However, data reuse, capacity, and regulatory drivers will push the continued distinction of data as a separate asset from (or at least a manageable and distinct subcategory of) purely processing elements. How this plays out for the CMDB of the future will be an interesting question.

See also Figure 5.8, "Metadata-based risk management."

Location

A Location is the physical site at which a Machine may be located. The Location–Machine relationship can be elaborated for the purposes of facilities management, including concepts such as rack and grid. Power and HVAC systems present significant information modeling challenges that will not be directly addressed here.

An Iterative and Incremental Approach to Configuration Data Maturation

An Iterative Approach

"Love the reference data model. We're not going to get it done for years. What to do in the meantime?"

"Well, let's look at how to build it up over time."

Immediately attempting the full scope of the reference models outlined in this book would be sure to fail.

Depending on the business objectives the configuration management capability is to meet, it's strongly recommended that the architects consider its evolution incrementally and iteratively.

The configuration management problem is a large and varied challenge, and different patterns and approaches will be discussed in subsequent sections. From a data perspective, I describe a maturation process.

Stage 1

First, the association of Applications to Servers is often the top priority when assessing the business value of configuration management. This is a relatively simple data structure (Figure 3.50).

Note that in this data structure there is no distinction between Applications and Deployed Software Systems or between Servers and Machines. The Application dependency on the Server may be due to a database, but that is not called out as a separate entity, so certain data privacy requirements will be poorly handled. The challenges of tracking Technology Products as distinct from IT services will not be met, nor will the issue of Server virtualization be covered.

However, as an incremental step, it is a solid achievement and may present significant challenges in itself.

Stage 2

This adds the concept of Datastore to the model. Databases are now called out specifically but are simplistically related to Servers (Figure 3.51).

A Datastore requires an intervening DBMS deployed to the Server, but this can be disregarded at early stages of configuration management. There is now potential to tie in Metadata, for example, as relevant to data privacy issues.

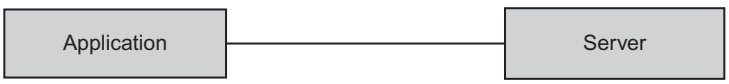


Figure 3.50 Configuration iteration 1.

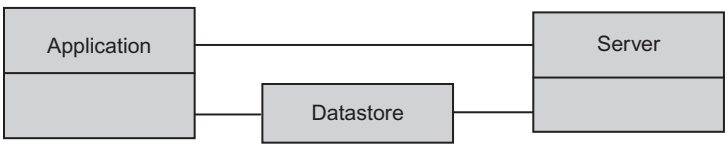


Figure 3.51 Configuration iteration 2.

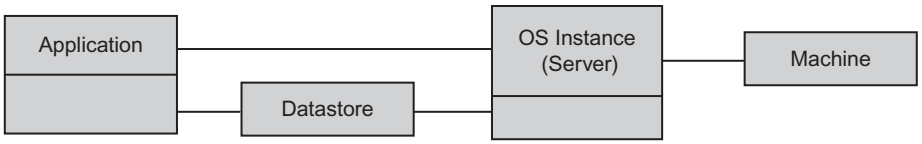


Figure 3.52 Configuration iteration 3.

Stage 3

This stage distinguishes between the Server as a logical instance of an operating system, as distinct from a physical Machine (Figure 3.52). Being fully mature in this area may require further elaboration, as there may be host and guest operating systems and machines containing machines. (Technically, you may have to institute a recursive relationship on the operating system instance and Machine entities. This is tricky to manage consistently, especially if multiple engineers are inputting data manually. Fortunately, this level of the stack is amenable to discovery tools, not that they are all that mature as of this writing.)

Stage 4

This distinguishes between Application and Deployed Software System (Figure 3.53).

This is a *big* job, probably one that requires discovery tools to get it right. It also can become annoying, as now you have to navigate through the Deployed Software System concept to reach the Server. (It's possible to still relate Application directly to Server, but the potential for ambiguity arises and it's not recommended. See your local data architect if you want an in-depth discussion.)

Note that each of these iterations will require data refactoring; see the refactoring literature for assistance here.[186]

Further Stages

Deploy point and Component might be considered next, and generally there are many options once this basic framework has been built. Depending on the organization's priorities, they may include more focus on networking, storage, metadata, messaging, or many other concerns.

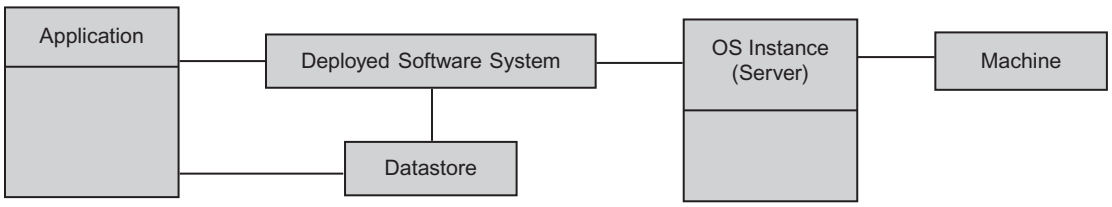


Figure 3.53 Configuration iteration 4.

This discussion only scratches the surface of the iterative approach to the ITRP problem domain. However, it's highly recommended that you approach your project in this way, because immediately attempting the full scope of the reference models outlined in this book would be sure to fail. Implementing an iterative approach within the constraints of vendor products will be particularly challenging but still more likely to succeed than a "boil the ocean" approach.

3.5 Process and Workflow: A Data Perspective

In this data-centric section, I haven't talked a lot about workflow and process. Let's turn to these from the data perspective.

The CRUD Matrix: An Old Standby

The CRUD, or create–use matrix, tells us the relationship between data and process.

A well-known technique for understanding data's relationship to process is the unfortunately named CRUD matrix. CRUD stands for the following:

- Create
- Read
- Update
- Delete

I'm going to modify the old CRUD standby to the following matrix:

- Create
- Use
- Aggregate

Note the following about this modification:

- Use includes both read and update.
- Delete isn't really of interest for high-level architecture.

An aggregate usage always means a Metric is being derived.

- Aggregate means that a given process depends not on *single instances* of a given data entity but rather on *summarizations* such as counts and averages. An aggregate usage always means a Metric is being derived and often implies some sort of underlying data mart or warehouse capability, which is important to know when considering systems architectures.

Creating such a matrix is a key reason for doing a conceptual data model. With the data on one axis and the processes on the other axis, the intersections are used for understanding how the data and process relate; it's an important alternative to spaghetti process models. Table 3.2 shows a high-level create–use–aggregate matrix for the book.

Table 3.3 Data and Process Cross-Reference

Processes				Entities																																		
				Strategy	Idea	Demand Request	Program and Project	Release	Request for Change	Service Request	Event	Risk	Incident	Problem	Known Error	Orderable Service	Hosting Service	Service	Ordered Service	Application	Technology Product	Business Process	Deployed Software System	Component	Deploy Point	OS Instance (Server)	Location	Machine	Datastore	Asset	Assembly CI	Measurement Definition	Agreement	Contract	Account
	Primary Value Chain	**Manage Demand**	Manage Customer Relationship	C	C		C	U		A		U	A	U	U	U	U	U		U		U									A	C	C	U
			Fulfill Demand Requests	U	C	C	C							U		U	U	C		C		U				U						C	C	U
		Developed Solutions	Manage Project		U	U	U	C	C			U					U	U		U		U	U	U				U		U		U		U
			Manage Requirements	U	U		U	U				U		U			U	U		U	U	U	U								U	U	U	
			Design and Build Solution				U	U				U		U	U		U	U		U	U	U	U	C	C	C			C		C	C	U	
			Ensure Solution Quality				U	U	U			U						U					U	U	U	U		U	U		U	U	U		
		Support Services	Manage Releases				U	C	C			C	U	U			U	U		U	U		U	U	U	U	U		U		C				
			Manage Production Change				U	U	U			C	U	U	U		U	U		U	U	U	U	U	C	C	U	C	C	C	C	U	U	U	
			Manage Production Configuration				U	U	U								U	U		U	U		C		U	U	U	U	U	U	U				
			Fulfill Service Requests							C			C		U	U	U	U	C	U		U						U			U			U		U
			Sustain Services					U	U	U	C		U					U		U		U		U	U	U		U	U			U	U	U	
			Resolve Incidents and Problems					U	U	U	U		C	C	C	U		U	U	U	U	U		U	U	U	U	U	U			U	U	U	
	Supporting Activities	**Manage Architecture**	Develop IT Strategy	C	C	C	A	A	A	A		U	A	A	A	A	A	A	A	A	A	U	A	A	A	A		A	A	A		A	U	U	A
			Manage IT Portfolio`	U	U	U	C	A	A	A		U	A	A	A	C	C	U	A	C	C	U	A	A	A	A		A	A	C		A	U	U	U
			Manage Capacity	U	U	U	U	U	A	A	A	U					U	U	U	U	U	U	A	A	U	A	U	A	U			A	U		U
			Manage Availability	U		U	U	U			U	U	A	U	U		U	U	U	U	U	U	U	U	U	U		U	U		U	A	U		
			Manage Service Levels			U	U			A	A	A	A	A	A	U	U	U	U	U		U	U			A					U	C	C	C	U
			Manage Process	U		U	U			A						U		U	U	U		C							U						
			Manage Data	U		U	U											U		U		U							U						
			Manage IT Finances	U	U	U	U			U		C				U	U	U	U	U	U	U	U			A	U	A		C			U	U	C
			Manage Sourcing, Staff, and Vendors	U		U	U			A							U	U	U	U	U						U	C					U	C	U
			Manage Risk, Security, and Compliance	U	U	U	U	U	U			C	A	U			U	U	U	U	U	U	U			U	U	U	U	U		U	U	U	
			Manage Facilities and Operations			U	U	U			U	U		U			U	U	U	U	U			U	U	A	C			U			U	U	U

The Matrix

This is a "reference matrix" based on my industry experience and research. It's presented as a method example more than a normative reference (although I did devote considerable thought to it).

If you are rationalizing your internal IT systems, consider doing your own matrix for both your current and your desired target states. Don't just take this version as gospel. Map it out yourself.

Another 100 pages could have been devoted to analyzing every cell, elaborated out to all intersection entities. As the academics say, this will be "left as an exercise for the reader." It will be different for every organization. The primary goal of this section is to demonstrate the analysis principles.[187]

A Document and its subtype of Metadata can be created by any of the process areas, and the IT enablement process area, because it is a miniature of the entire value chain, similarly can create and use anything—hence they are not shown.

A matrix like this is a distilled view of information that could also be drawn in dozens of diagrams. For example, Incidents and Problems go through a life cycle that may feed back into the demand process (Figure 3.54).

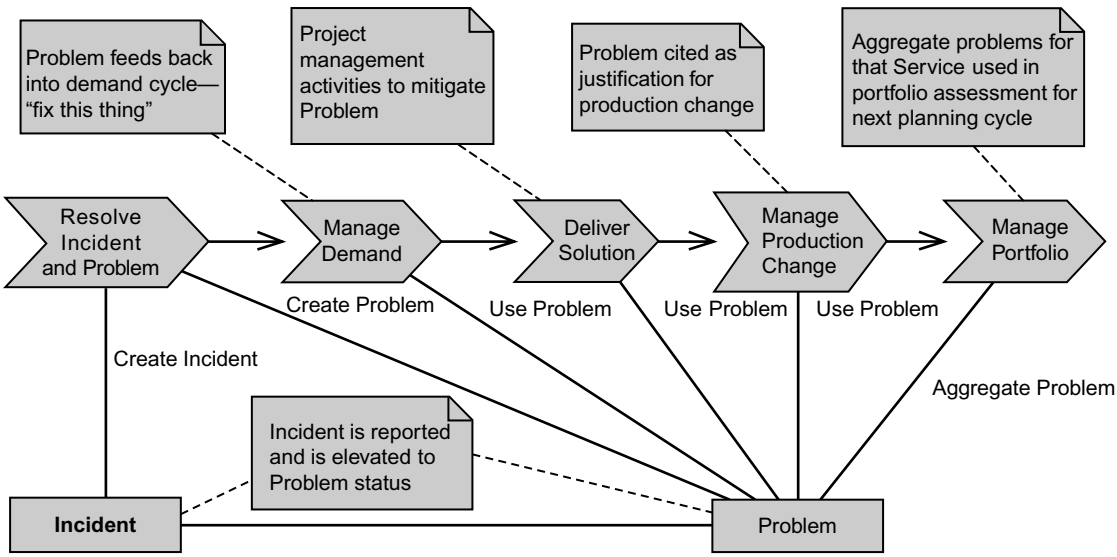


Figure 3.54 Graphical representation of a process or data create–use matrix.

This create–use matrix is presented as a starting reference model. There are lots of interesting questions generated by such a matrix.

Is a Problem created in the Incident process, or is it created in the Problem process? (Incident Management refers one or more Incidents to Problem Management for further analysis, but Problem makes the call as to whether to create a new Problem record.)

An RFC can be created by the release manager in the system development process or by some team attempting to respond to an Incident. When an entity can be created by more than one process, this deserves special attention. Ditto for Service Offering, Process, and Contract. Contracts might be created as the result of outsourcing service agreements, for vendor product purchases, or between the IT organization and its clients—three different origination processes.

Notice how many processes use the Application entity. This is typically one of the most poorly managed entities in all of IT governance.

The primary value chain activities are the most reliable data origination points.

The primary value chain activities are the most reliable data origination points. Although data also can originate in the supporting processes, these processes may be underfunded and not scrutinized effectively for quality. Therefore, it's a best practice to focus on core value chain activities and the data that they produce and consume.

For example, asking the risk management or business continuity activities to generate a list of all Business Processes dependent on IT is bound to fail. That is core IT value chain data, and the systems underpinning those processes should have the process dependencies documented as part of their construction and release.

If a supporting process needs data to achieve its mission, efforts should be made to capture that data as part of the primary value chain activities. If resistance is met, either the matter should be escalated or the supporting activity's need for that data should be questioned and perhaps abandoned.

Intersection Entities and Process

Most entity relationships in the conceptual data model are many to many. As noted later in the material on intersection entities, these relationships must be resolved with an intermediate table. Such intersection entities require the same CRUD analysis as the major IT concepts, and some of the most challenging problems emerge in attempting to manage them.

For example, an Application may have many Servers, and vice versa (Figure 3.55).

Figure 3.55 Application to Server.

Table 3.4 Intersection Entity Analysis

	Manage Application Portfolio	Provide Infrastructure	Document Application Dependencies
Application	C		U
Application/Server			C
OS Instance (Server)		C	U

(See the earlier section "An Iterative and Incremental Approach." Note that this example is actually using the third iteration for simplicity.)

When analyzing process to data, include all three entities as in Table 3.3. Note that in this example the processes are more granular—the process framework as presented in this book needs to be drilled down further to enable this level of detailed analysis.

Workflow

One requirement for IT enablement tooling in general is rigorous tracking of all changes to any entity: *who* changed *what, when.* There are a surprising number of tools that do not do this and should be ruled out as possible product choices for any enterprise. Common terms will be "effective dating," "timestamping," and/or "audit trail" (use these in vendor discussions).

Timestamping of status changes is how SLAs are monitored for things like Incident, Service Request, and Problem resolution.

Business Process meets the entity through these techniques, especially when audit trails are collected on the changing roles and responsibilities for an entity (see the "Role Management" section earlier in this chapter). A trail of who "owns" an Incident and where it has been referred is a feature of most incident management tools; this is a specific example of the general principles here. Timestamping of status changes is (in part) how SLAs are monitored for workflows like Incident, Service Request, and Problem resolution.

Similarly, IT enablement tooling should manage audit trails on other entities and their Role assignments:

- Who have the application managers been for this Application?
- What Projects have built upon this Application? Who has been on these Projects?
- Who has approved this Change?

3.6 General IT Data Architecture Issues

Mapping the Business to IT

The goal of mapping IT to the business is implicit throughout the data model; one representation of often-encountered concepts can be seen in Figure 3.56.

If the preceding concepts (or equivalents) are understood and formally inventoried, with dependencies mapped and maintained, this can be of great service in understanding business–IT alignment. (Business–IT alignment is also a matter of perception, which no amount of data can address.)

Some of these concepts are highly subjective and require clarification for a given organization's context and culture. There are various methodologies, out of scope for this book. See Appendix A for a detailed discussion of function *vis-à-vis* process. Capability is another concept sometimes encountered.

Enterprise architecture efforts should map IT services to the enterprise value chain, including quantified revenue data.

Such analysis is typically the domain of enterprise architecture. It can degenerate into ivory tower efforts and must remain aligned with business objectives. Enterprise architecture efforts would be well advised in particular to analyze and document the role of any particular IT Service, Business Process, or Function with respect to the enterprise value chain, including quantified revenue data. Mapping architecture to the enterprise financial model is not often done and would help the enterprise architecture practice immeasurably if undertaken. Such data has

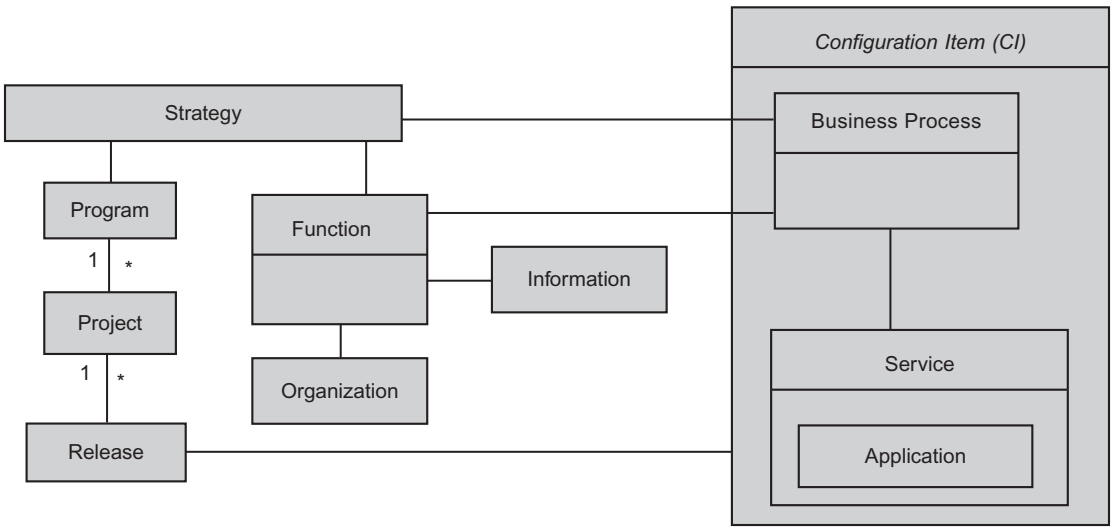


Figure 3.56 Essentials of Business–IT mapping.

applicability in ITSM efforts such as prioritizing Incident and Problem resolution and continuity strategies. Unfortunately, many enterprise architects do not have the requisite background.

Versioning

For a given product, any unique combination of the base software plus patches is a version.

Versioning is a challenging area in IT data management, especially with respect to application software. Technically, for a given product, any unique combination of the base software plus patches is a version. For many complex enterprise software products, patches are applied on an as-needed basis—they are not cumulative, so the number of potential combinations can be large. This means that naïve approaches to tracking IT components (such as a simple version field on a CI) are not robust enough.

Fully elaborated patch and version management should be considered an element management problem area and left to the specialized tools emerging (e.g., provisioning systems) optimized to handle this complex domain. The consolidated CMDB is probably best served by keeping version and patch management information at a relatively high level, with traceable links to the provisioning or patch management systems if that level of detail is required.

CMDBs and metadata repositories also run into some conceptual issues with versioning and life cycle state; there is a need to distinguish between the following:

1. The life cycle state of the *object in question*—for example, purchased, in service, or retired—and the relevant versions
2. The life cycle state of the *CMDB record pointing to the object*—for example, planned, discovered, or confirmed

As noted in the discussion on the Metadata entity, this is a core problem of "thing" versus "re-presentation of thing." This is further discussed in the "Configuration Management" section in the next chapter.

Related to the concept of versioning is current versus target analysis. An enterprise architecture is essentially a set of high-level dependencies distinguished from an operational service model by 1) how low in the technology stack it extends and 2) the presence of future-state data.

An ideal solution would be a robust as-is model of the IT configuration (including logical concepts such as Process, Service, and Application) upon which future-state scenarios could be based, modeled in an area logically separated from the critical current-state data. These scenarios, once elaborated, can be compared with the current state and change initiatives derived.

Collaboration

Any entity in the model might serve as a basis for collaboration. The ability to have a threaded discussion on any item would be highly desirable, as would be the ability to easily exchange links (e.g., Uniform Resource Identifiers).

Portfolio

A portfolio is a collection of objects with like attributes across which meaningful comparisons can be made for decision-making purposes. It has a further connotation of a financial resource pool or account of some sort, but portfolios can also be measured and managed on nonfinancial bases.

There is no portfolio entity.

As discussed in Chapter 2, there are various approaches to portfolio segmentation. Portfolio is not a straightforward concept to model; there is not a single abstract portfolio entity. It is better to conceive separate portfolios based on the objects to be comparatively managed:

- Project portfolio
- Service portfolio
- Application portfolio
- Technology product portfolio
- Asset portfolio

These classes of objects might further be distinguished into different portfolios based on an organization (i.e., as a Party) having a defined portfolio interest in them. For example, organization A might have 15 projects in their portfolio, and organization B has 23. These are truly separate portfolios with different assessment metrics.

Each class of item has different metrics. For example, a Service portfolio may have comparison metrics based on SLA adherence, perceived quality, intensity of use, and life cycle of underpinning technology, and a Project portfolio might have metrics based on business alignment, anticipated return on investment, and so forth.

Granularity is a key issue in portfolio management. Some theorists call for an ideal of "no more than 30 to 50" applications,[188] for example, but the number of applications in a large company may easily top 1000 (depending on the methodology by which they are counted). This is a classic rollup or aggregation issue amenable to the same techniques used to construct dimensions for business intelligence purposes.

Should Applications Be Managed as Projects?

> Every product and every activity of a business begins to obsolesce as soon as it is started. Every product, every operation, and every activity in a business should be put on trial for its life every two or three years. Each should be considered the way we consider a proposal to go into a new product, a new operation, or activity—complete with budget, capital appropriation request, and so on. One question should be asked of each: "If we were not in this already, would we now go into it?" And if the answer is "no," the next question should be: "How do we get out and how fast?"
>
> —Peter F. Drucker[189]

Having emphasized earlier that an Application and a Project are different things, I want to contradict this. How do you track TCO for Applications? This question gets to the heart of IT portfolio management, which in some representations has a project-centric bias—project in the sense of having a defined end date. But what if we relaxed that requirement and accepted the concept of application as a sort of open-ended Project? (This will give anyone schooled in formal project management pause; having a defined end date is typically seen as essential to the definition of a Project.)

The portfolio of base activities should include the application portfolio as a subset.

However, pragmatically, the time-tracking tool may be first brought in to support project management. Implementing a separate time-tracking tool for nonproject staff hours (e.g., time spent supporting the operation of an Application) clearly makes no sense, so the list of chargeable elements in the time-tracking tool needs to include both Projects and other activities. The portfolio of base activities thus should include the application portfolio as a subset. (It won't be a complete match because there are base activities that don't correspond to either Projects or Applications).

The overall population of "buckets" thus should look like this:

- True, defined-scope Projects (typically incremental, sometimes base)
- Ongoing maintenance activities tied to defined Applications or Services
- Other valid activities (e.g., training)

Without integration, visibility into project versus maintenance activities will remain elusive, and integrated staff resource planning will remain difficult.

This can present practical consequences if a project management office controls the time-tracking tool; its members may not understand the concepts of IT Service or Application well and may implement charging structures that do not align with the IT Service portfolio. Determining the master system of record for steady-state elements (i.e., the Application or Service portfolio) to be used as a basis for time tracking and where necessary building data feeds will be critical.

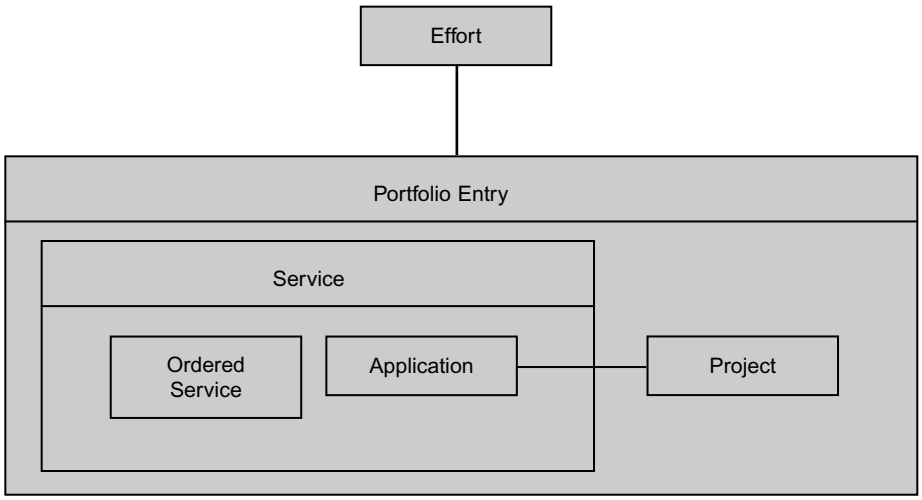


Figure 3.57 Effort tracking based on portfolio entries.

Figure 3.57 shows a conceptual fragment illustrating the commonality of service, Application, and Project within an overall portfolio management structure. Note that, although they are similar elements in this representation, for other purposes they are radically different concepts, Service being a CI with an indeterminate life cycle and Project being a defined-scope, finite effort.

Because effort can be expended on either, their data must be rationalized and integrated to some degree; there would be several technical means of doing this. Without this integration, visibility into Project versus maintenance activities will remain elusive and integrated staff resource planning will remain difficult.

The concepts of Program and product are sometimes used in Project portfolio management in solving these issues. An Application might be seen as a longer-lived Program in the project management sense (not the computing sense).

See the "Justify Change" pattern in Chapter 5.

Intersection Entities

The intersection entities are where the devil emerges from the details.

This is a high-level conceptual data model. Most of the relationships (all the unadorned lines) are of the many-to-many type. For example, an Application may use many Servers and a Server may support many Applications (Figure 3.58).

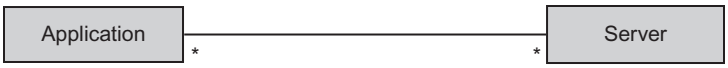


Figure 3.58 Unresolved many-to-many relationship.

Figure 3.59 Resolved many-to-many relationship.

(This is from iteration 1, so it doesn't track with the full reference model.) To turn these language concepts into an operable system, an *intersection entity* is required[190] (Figure 3.59).

If you look at the main data model and imagine all many-to-many relationships being elaborated with their intersection entities, you'll see that it would be far too complex to represent as one diagram. That's the beauty of a well-scoped conceptual data model; it should be able to represent a substantial problem domain on one page.

The intersection entities are where the devil emerges from the details. For example, it is likely that your database administration team has a list (or at least a spreadsheet) of all the team's databases. Perhaps you have an application management group with its own spreadsheet. Therefore, you might be able to say that you can populate the Application and Datastore entities. But who is responsible for the relationship, as represented by the Application–Datastore entity? Questions of this nature permeate the problem of configuration management. As with any entity, documented processes are required for the creation, reading, updating, and deleting of data in the Application–Datastore intersection entity. Would it be your application team? Your database administration team? A separate team of configuration analysts?

The current state of most IT organizations is much less formal. What you often see is uncoordinated spreadsheets, which do not handle the challenge of many-to-many data well.

Spreadsheet Silos

Chris: What's so bad about people maintaining their own spreadsheets?

Kelly: Well, let's look at your Organization. Here are some extracts from spreadsheets maintained by your application support, database, and server teams:

(continued)

Server team:

Server name	Notes
WNAPPL01	Supports FirstTime and X-time Batch
FRED	?
UXPLV01	PLV server. See Scott Armstrong
WINWEB03	External Web server
UNXDB001	PLV databases
WINDB2	SQL Server
TXEMLA	Email server
QDXAPP02	Quadrex App server

Applications team:

	Servers	Databases
Quadrex	QDXAPP02 UNXDB001	Oracle
X-Time	WNAPL01	SQL Server
PLV	UXPLV01 UNXDB001	Oracle

Database team:

Database	Server	App
PDBX01	UNXDB001	Quadrex
LVDBX01	UNXDB001	PLV/X-Time
ARGDBX02	WINDB2	Argent
GDBX01	WINDB2	GuardSys

Chris: Ouch. This data makes my head hurt.

Kelly: Well, stick with me. There are some serious issues here. Let's focus on Quadrex. The server team knows that Quadrex uses QDXAPP02 as an application server but doesn't seem to realize that Quadrex also uses UNXDB001 through its use of the PDBX01 database. They think that UNXDB001 is only used for PLV. (Perhaps there was surplus capacity on that server and Quadrex came later.)

The application team knows that Quadrex is using QDXAPP02 and UNXDB001, but it doesn't have the level of detail that the database administrators do, that Quadrex is using specifically the PDBX01 database on that server. Quadrex does not own that server—the PLV team is also using it. This is important from a cost allocation and support impact standpoint.

Chris: Actually, no application team "owns" their server according to our VP for systems engineering, even if that server is currently allocated 100% to them. It's a "hosting" relationship. But some of them haven't quite bought into that point of view.

Kelly: Right… Common argument nowadays! Now, the database team knows that Quadrex is using the PDBX01 database on UNXDB001—but isn't tracking Quadrex's use of QDXAPP02, as that is an application server that they don't manage. Finally, notice that someone fumble fingered the Quadrex name on the first row of the database administration spreadsheet, misspelling it "Qaudrex." This means that when we go to consolidate all this data into one database, we're going to have to manually identify and clean that up.

Chris: Why didn't the database administrators pick from a list of application names?

Kelly: Has that list been shared with them? Do they agree with how those applications are represented? Is there confidence in the process for keeping the list up-to-date? (For that matter, is there even a process?) Do they have a technical approach on how they can integrate that list from another system? Excel can pull a list from a live database, but you start to get into advanced features—too far down that road and you're looking at real system development.

The same issues need to be thought through for every many-to-many relationship, such as the following:

- Event–Incident–Problem
- Application–Technology Product
- Application–Process
- Change–CI
- Change–Incident

The complexities of doing this are why vendor products are recommended, but it's not impossible to build your own.

This is also the most critical area to review the vendor product—a common vendor mistake is to put in a one-to-many relationship where a many-to-many relationship is required. For example:

- A Problem might be addressed by several Releases, but your problem management tool only allows you to identify one Release that fixes it.
- A Datastore may be shared by many Applications, but a configuration management tool only allows you to identify it with one.
- A Machine may support multiple Servers, but your asset management tool only allows you to associate it with one.

The purpose of asking for a data model is to assess the business rules that the application is based on.

These are the kinds of details critical to review in assessing any vendor product—and it all starts with having good, specific, clear requirements for what you need to track and how it needs to relate. Even when purchasing a vendor product, a conceptual data model is needed. (Emphasis on conceptual. The physical data model is irrelevant; the purpose of asking for a data model is to assess the business rules that the application is based on—not to assess their technical architecture.)

Networks and Trees

IT configuration management data (or metadata) presents unique problems compared with the data that IT manages on behalf of its partners.

Metadata, or IT configuration management data (this book sees them as synonymous), presents unique problems compared with the data that IT manages on behalf of its customers. Financial, logistics, and human resources data has deep roots in paper-based history; a purchase order or hiring authorization message can be traced directly to its origins in the forms once routed by interoffice mail to "IN" baskets throughout preelectronic corporations.

One difference is the "recursive relationship," a common occurrence when managing IT data.

If you look at a sales journal or a stack of invoices, you will generally see consistency: the data model is the same for all the information. The data also has limited interconnections: one invoice does not typically reference another in simple models; invoices do not have *dependencies* on one another. An invoice references common customer lookup tables and product tables, resulting in data models that are relatively straightforward to understand (Figure 3.60).

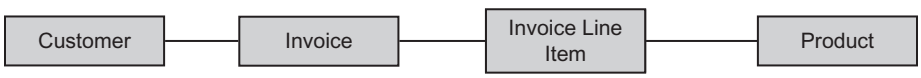


Figure 3.60 Basic data model.

Graph data can rapidly become complex to the point of incomprehensibility.

With configuration management, everything becomes more complex. Applications depend on other applications, data flows from one database to another to yet a third, and network devices are by definition embedded in a web of interconnections. The data (and its required modeling) starts to take on new characteristics. In mathematical terms, it becomes graph-based; that is, it looks as shown in Figure 3.61.

This kind of data presents well-known problems in storage, querying, and presentation because it requires "any-to-any" data models and can rapidly become complex to the point of incomprehensibility.

This kind of data is not typically encountered in the business-centric systems that are successors to forms-based paper processes. It *is* encountered in manufacturing and supply chain systems in the well-known "bill of materials" problem. It is the kind of data stored by CMDBs and metadata repositories, when they move into managing technical metadata such as interconnections among network devices, integration flows, and so forth. It is also seen in computer-assisted design and manufacturing tools, and CASE tools.

The recursive relationship enables complex data. This is a relationship when one type of thing can be connected to other instances *of the same thing*. There are two basic types of recursive relationship:

- Tree
- Network

The tree relationship is a relationship where one thing "contains" other things. A taxonomy is a tree; so is a hierarchy. Common examples of trees in ITSM are CIs containing other CIs, organization hierarchies, and process steps

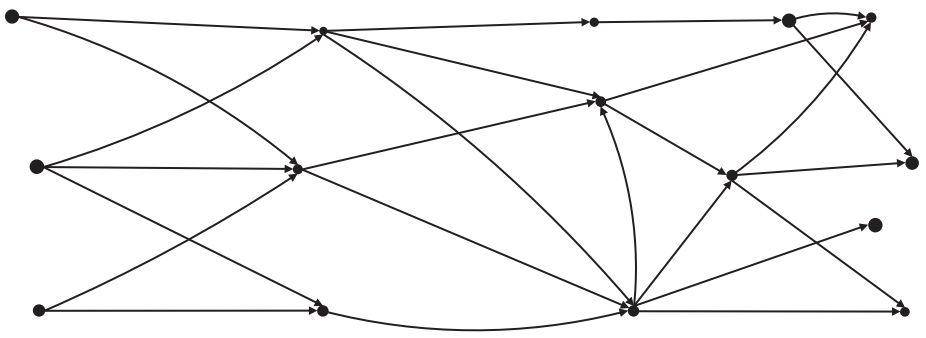


Figure 3.61 Graph-based information.

decomposing into finer-grained activities and tasks. A tree often looks as shown in Figure 3.62.

Notice how it is always possible to say that one box owns and/or is owned by others. A tree can be recognized in a data model by the notations in Figure 3.63.

A common strategy of data architects when dealing with treelike structures is to fix the number of levels.

While simpler than networks, trees can be troublesome to report on if they are of indeterminate depths; that is, if one branch of the tree is five levels deep and the other is only three, it's hard to create a consistent, sensible report. A common strategy of data architects when dealing with treelike structures is to *fix the*

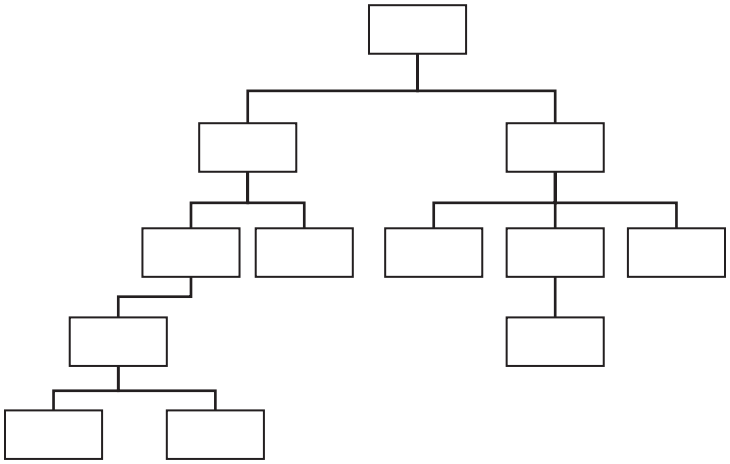


Figure 3.62 Indefinite-depth tree.

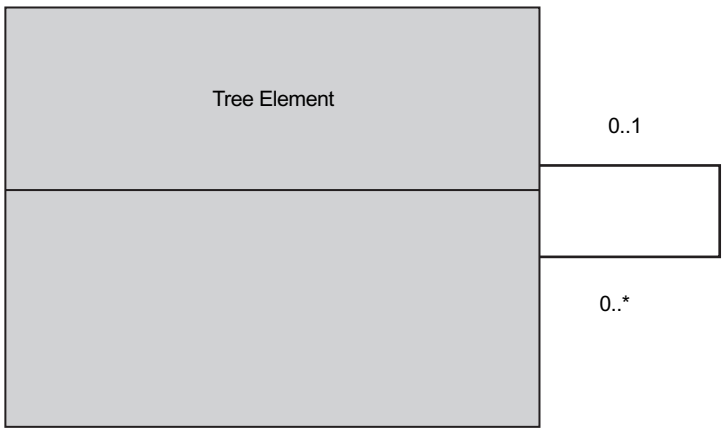


Figure 3.63 Tree data model.

levels and establish that all branches of the tree have the same number of levels (Figure 3.64).

But this may have problems in dealing with the real world—what if the organization (or whatever) is just not structured that way? Organizations may decide to structure themselves, and adapt their business processes, to fixed-level hierarchies, as you see in retail Organizations with their typical store–district–region hierarchies.

A network is characterized by things related to other things.

A network is characterized by things related to other things, not necessarily containing other things. A diagram of a redundant wide area network, an Organization chart with "dotted-line" relationships, or a mapping of how systems interrelate would probably be a network. A network often looks as shown in Figure 3.65.

Although there are treelike structures in it, the difference is that it is no longer possible to say that one box owns or is owned by others. A network can be recognized in a data model by the notations in Figure 3.66.

This is also often called the "any-to-any" relationship.

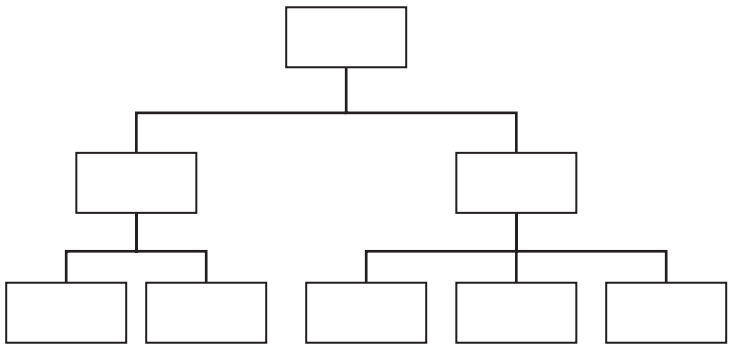


Figure 3.64 Fixed-depth (level) tree.

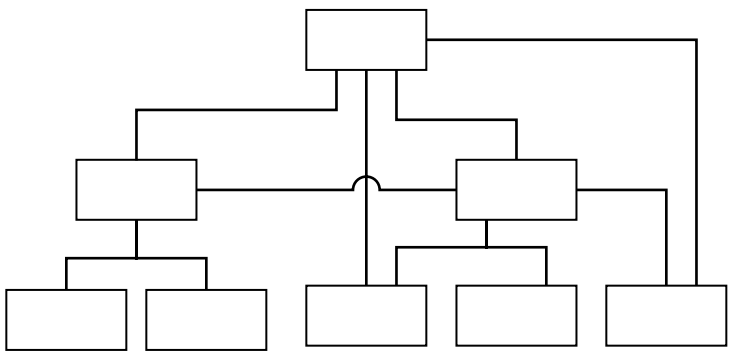


Figure 3.65 Network (no longer a tree).

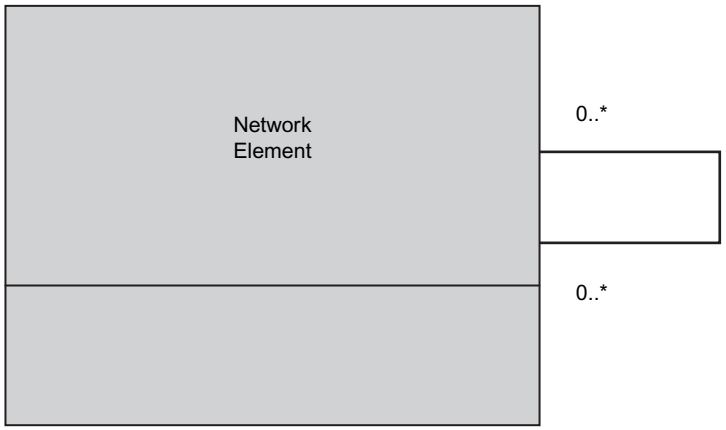


Figure 3.66 Network data model.

One issue in service dependency mapping is that a service map is often presented as a tree but in reality is a network because infrastructure elements often support more than one Service.

Trees and networks make ITSM data much harder to deal with compared to sales or financial data. Why is this? Start with a (now somewhat dated) picture of my son (Figure 3.67).

[n.b.: Thanks, yes he's cute. He's a happy boy :-).] What I want to draw your attention to is the Skwish toy he's holding (Figure 3.68).

Now, a regular business data is like a deck of cards (Figure 3.69).

You can say,

"Show me all the red cards between 3 and 8."

"Show me all the jacks."

"Show me all the hearts and spades."

It's a pretty simple problem. The hearts don't have much to do with the spades, and there's not a lot of ambiguity.

What if you say "show me everything connected to the small red sphere"? What do you mean by that?

The Skwish toy represents interconnected, indefinite-depth, recursive data. It's troublesome. You can say, "show me a small red sphere," but what if you say "show me everything connected to the small red sphere"? What do you mean by that? The whole toy? Or just things immediately connected to the red sphere? By elastic? By wood? Where do you draw the line?

What does this have to do with reporting for ITRP (and ITSM)? Much mainstream business reporting is of the deck-of-cards variety. You can handle this with the same tools your business users use: relational databases and reporting or business intelligence tools such as Crystal, Brio, Microstrategy, and Actuate.

Figure 3.67 Keane Betz and Skwish toy.

Using these well-established techniques, one can answer all of the following questions (assuming the data is consolidated into a data mart):

- What Services do I have?
- Have I met my service levels for a Service?
- What is the history of changes associated with a CI?
- How many Projects do I have running right now?
- Which Projects contributed to building this system, and what did they cost?
- What does this system cost to run?

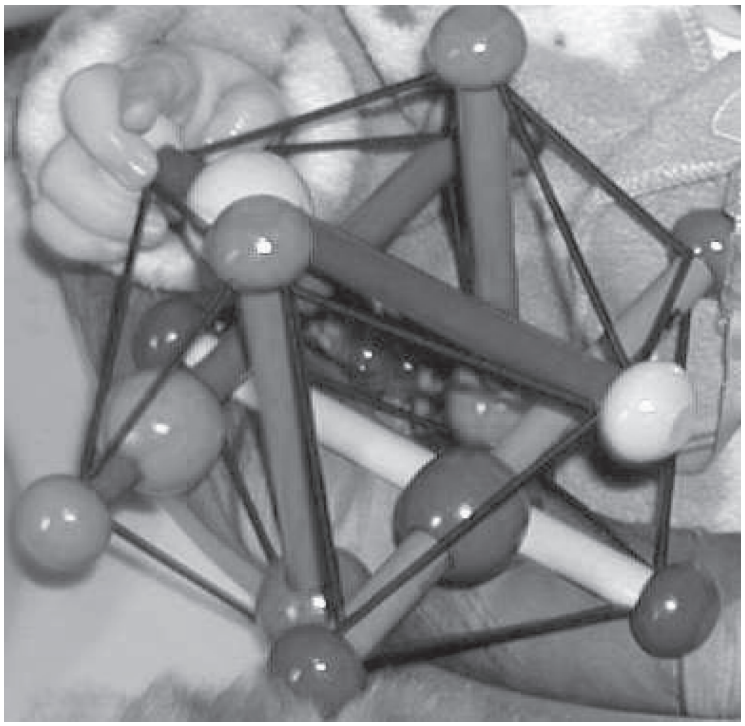


Figure 3.68 Skwish toy: network example.

Relational databases and query tools don't handle recursive data well.

But those tools don't handle reporting on recursive data. Although a relational database will *store* recursive data just fine, with relational databases and query tools, it's hard to answer the following questions:

- What is this Service dependent on (other Services, Applications, hardware, network)?
- What depends on this infrastructure piece, directly or indirectly?
- Is the Project on schedule? On budget? (This requires traversing an unknown depth of project tasks and subtasks—obviously, project management tools do it, but a customer is hard pressed to deal with this data in raw form. A project management office, in configuring the project management tool, may "fix the levels," only allowing, for example, four levels of project, phase, task, and subtask.)
- For a Project, which tasks are on the critical path? (Ditto.)
- What is the complete lineage of this data item in this report? Where did it come from? What systems did it flow through? (An important compliance issue.)

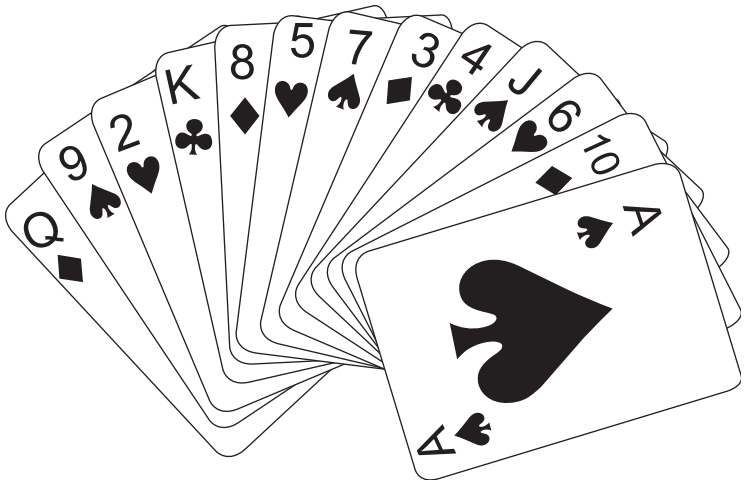


Figure 3.69 Deck of cards.

- What are all the downstream destinations for this data element? What middleware infrastructure does it flow across? (Important security questions.)

Basically, if you have language like "direct or indirect dependency" in a requirements specification, you probably are into the Skwish type (tree or network) problem. The problem is that although the theorists have been kicking this around for a while, no standard approaching SQL has been implemented across multiple platforms.[191]

Recursion in internal IT data is an immediate challenge to the application of business intelligence–based performance management principles.

Practical Use of Recursion

The recursive relationship can easily be abused and can enable nonsensical connections.

The recursive relationship can easily be abused and can enable nonsensical connections. One of the problems with the CI concept as framed by ITIL is that it calls for any-to-any relationships between CIs generally. (Actually, it calls for both the "contains" and "uses" relationships for any CIs.) However, some connections don't make sense. For example, a Datastore should not "use" a wide area network circuit, and a RAM chip would have nothing to do with an XML schema—yet some configuration management tools allow the customer to put in such relationships. Being more precise is why we go to the trouble of building a data model.

It is usually the case, however, that any CI *of a given type* can both use and contain other objects of the same type, especially in a high-level conceptual data model such as this.

For example, a Server might contain hard drives; both would be types of machine. A Machine might be connected to other Machines using a network. A Process can both contain and depend on other Processes. Datastores contain other Datastores, and with mechanisms like linked databases they may depend without owning. A Deploy Point (i.e., a file system directory) can certainly contain other Deploy Points and through mechanisms like directory linking (common in Unix) can depend on them without owning.

It's OK if the configuration management tool allows the any-to-any relationship as a controlled administrative option.

Finally, it's also the case that the IT world is not well understood and new dependencies present themselves. Therefore, it's OK if the configuration management tool allows the any-to-any relationship as a managed, controlled administrative option. It's important to be clear about how this differs from bad practice CMDB tools: in the recommended approach, an administrator can decide that "well, we do need to track a dependency between XML Schemas and RAM chips." They specifically allow *just this additional dependency* to be permitted by the tool and created by customers. In a poorly engineered tool, the *user* gets to decide what is related to what. That is a recipe for chaos.

Partitioning the Data Model

There are no vendor products on the market that cover the entire scope of this conceptual data model. The IT organization will therefore need to integrate two or more products. These integration points can be understood by simply drawing boxes around the entities, representing systems of record, and then observing where those boxes are crossed by relationship lines—that is where interfaces must be built.

For example, if service request management is handled by a different system than service management (a common industry pattern), some Service Requests may result in true, formal RFCs (Figure 3.70). This in turn requires some sort of interface between the two systems to handle the relationship between Service Request and Change. The interface may be one of several types:

- Service requests requiring RFCs are moved from the Service Request management system and automatically moved to the service management system.
- The Service Request is assigned an unambiguous identifier, and this is manually entered into the RFC system (potential for human error).

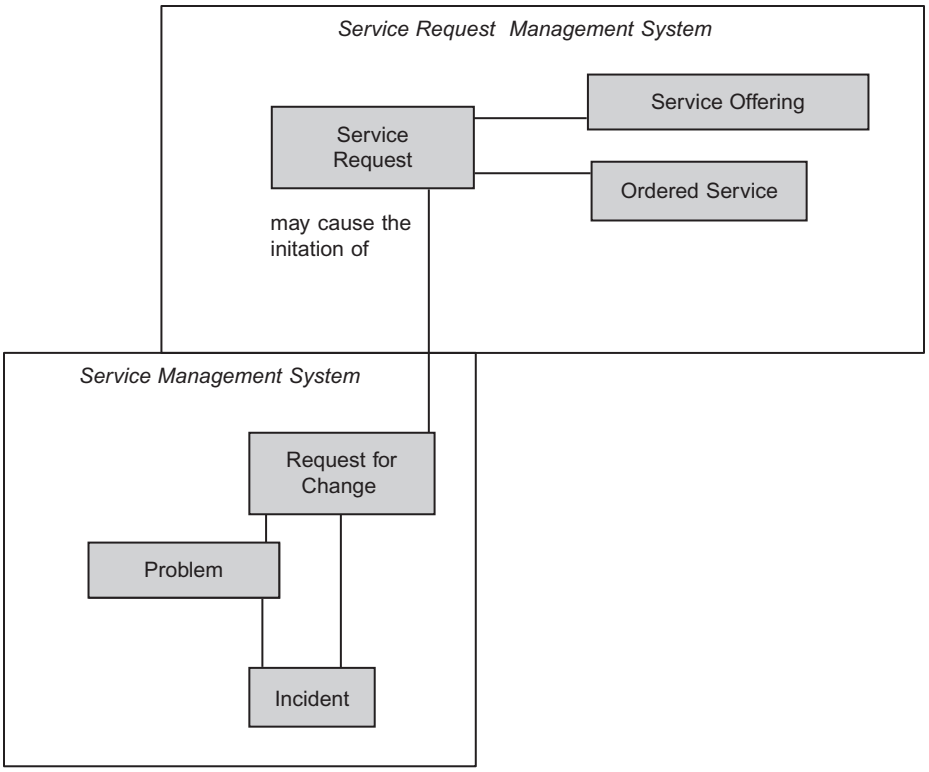


Figure 3.70 Partitioning data across systems.

- The Change is created and its identifier is manually entered into the Service Request (again potential for human error).

If no cross-reference is created, the Service Request is at risk if RFC approval is needed to complete it.

Don't underestimate the costs of integrating several best-in-class systems.

The creation of data silos that do not interoperate is one of the most pervasive architectural failures in modern IT systems, and it is recommended you don't do it to yourself. But interfaces are expensive to build and run, so don't underestimate the cost of integrating several best-in-class systems.

Data Implications of Operational versus Portfolio Configuration Management

As noted previously, there are two types of configuration management: operational and portfolio. (Drift control is a type of operational configuration management.)

Portfolio configuration management is where data models are applicable. Operational configuration management, with its bit-level concern for change, can't be easily translated into a metamodel—there are too many variations. Data models would be necessary for all file formats for starters, which is impossible.

Operational configuration management therefore becomes concerned with the Deploy Point concept, which is a defined block of storage across which integrity can be ensured and changes detected. It also might be applied individually to the Document, Component, and/or Datastore concepts, but this may be inefficient in the case of large-scale Applications with dozens or hundreds of Components and Datastores.

For example, an Application may be deployed to a given Server, situated in a Deploy Point. Portfolio configuration management tracks the fact of deployment and the Application's other dependencies on databases, other Applications, and so forth.

The operational configuration management tool runs a nightly scan on the Deploy Point (filtering out data processing directories) and through analyzing checksums identifies if anything has changed. (A simple listing of files will not do; their size and internal characteristics need to be examined and compared.) Although this could conceivably be integrated with a portfolio CMDB, there are also tools that decouple the change detection management from the portfolio and dependency management problems.

3.7 The Business Case

Making the business case for data architecture and analysis is notoriously difficult. It is often seen as overhead on projects, busy work to be gotten out of the way so that the real work of system construction can commence. The problem with this attitude is that the data model is a fundamental consensus point for many (if not most) complex IT undertakings. Hammering out the shared definitions of the major "things" in the problem domain is essential for project efficiency and effectiveness. Without consensus on the data model, the project runs the risk of integration problems, unfulfilled expectations, conflicting reports, and so forth.

Building a data "view" on the IT enablement problem domain can assist with identifying and aligning conflicting or redundant processes, identifying opportunities to reuse shared data, and minimizing the capacity consumption of internal IT enablement systems.

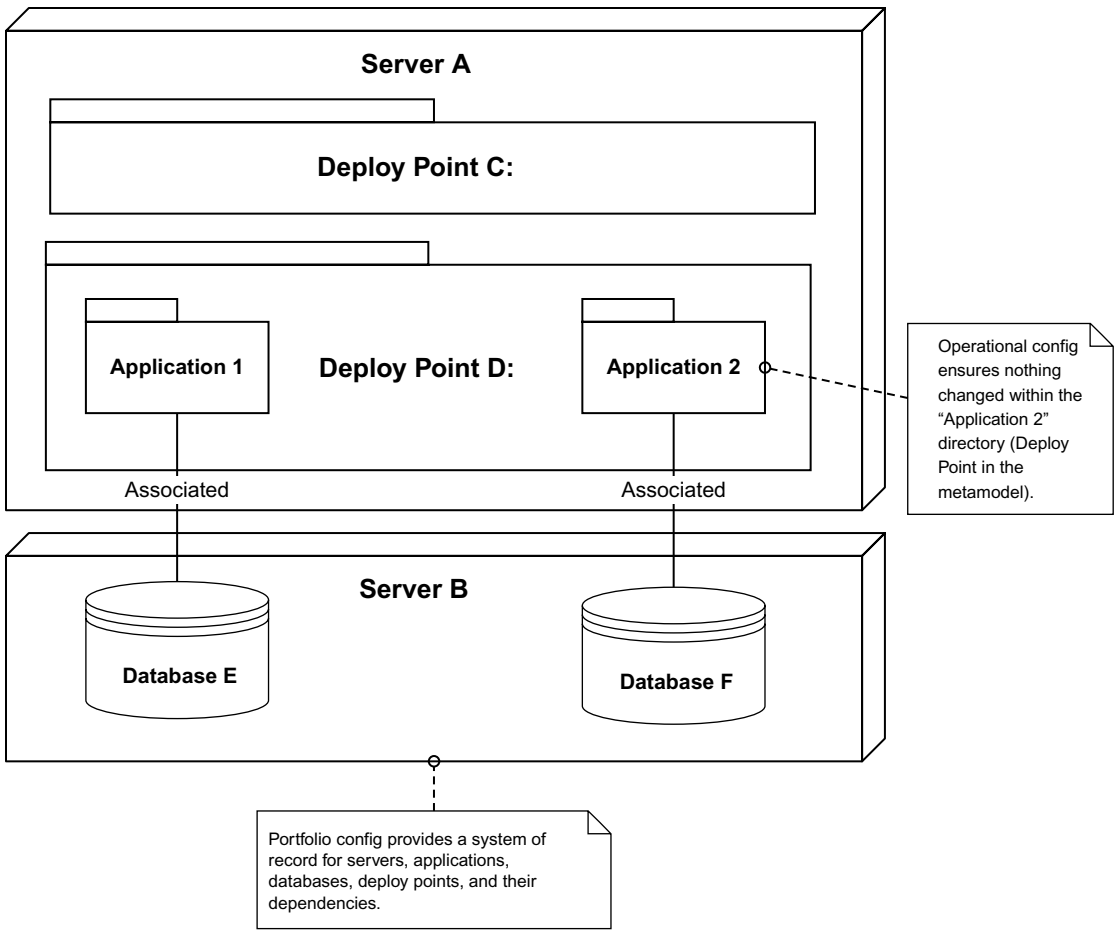


Figure 3.71 Distinction between Operational and Portfolio Configuration Management.

3.8 Making It Real

- Do you have an inventory of all the datastores containing internal IT data? What major subjects and entities are contained in each?
- Does every data element in your production IT enablement infrastructure have a defined maintenance process? Are data quality metrics defined and measured?
- Do any represent "multiple master" situations in which the same data is being maintained in two different places by two different processes or owners? (The core list of Applications is a common problem. Application to Server dependency is another example.)

- Do you have the ability to formulate dependency queries? That is, can you report on an Application dependency chain such as Application A is dependent on B is dependent on, and so on, to any number of links in the chain? Can you constrain and filter this query to make it usable (limit the number of links, limit it to only certain data topics, etc.)?
- For a given element of infrastructure, can you identify instantly what business Service or Process it supports?

3.9 Chapter Conclusion

The definition and normalization of conceptual entity models is an essential part of any full process analysis.

One of the unfortunate extremes encountered with today's process-centric thinking is the idea that data is some mere technicality whose consideration can be deferred to vendors or developers. The definition and normalization of conceptual entity models is an essential part of any full process analysis and is a key bridge between the process framework and the technical systems supporting it.

This discussion of the essential, process-independent data concepts has clarified the core concept of CI, essential to the maturation of IT enablement. This concept will reappear often in the system and pattern discussions. The management and interactions of these data structures are concerns to be further elucidated in the following material.

The objective of this chapter was to create a controlled vocabulary.

Again, the objectives of this chapter were to create a controlled vocabulary reflecting current IT management discourse and to start to explore some of its implications. The objective was *not* technical design, although the degree of precision required in the vocabulary analysis required the use of modeling notations and matrices.

3.10 Further Reading

This discussion is by no means the first coverage of internal IT data analysis. The most sophisticated efforts are generally found under the heading "metadata" and "metamodeling." Significant work has been done by the OMG, the Distributed Management Task Force, the Tele-Management Forum, and authors such as Adrienne Tannenbaum, David Marco, Michael Jennings, and David Hay. This chapter, as with the process framework, sought to distill much of this effort down into a digestible chapter focused on essentials and terminology in common use.

An early systematic matrix of IT data to process is seen in *A Management System for the Information Business* (IBM 1980). However, the data representation was not as a normalized conceptual data model but rather as data "classes" corresponding to what would be called subject areas today. Although this work was reportedly an input into ITIL, the rigorous data-based approach was lost to the overly general CMDB and CI discussion.

For data modeling generally, see Reingruber and Gregory (1994), Teorey (1994), Hay (1996), Carlis and Maguire (2001), Halpin (2001), and Simsion and Witt (2005). For the OMG specifications on metadata and metamodels, see OMG (2002a and 2002b) and *www.uml.org*.

For the DMTF work, see the Distributed Management Task Force (2000, 2002a, 2002b, and 2003) and Bumpus (2000). Other views of IT domain (meta)data models can be seen in Marco and Jennings (2004) and Hay (2006).

Data Center Markup Language is another standards effort, notable for its advocacy of Semantic Web technology to solve the CMDB data Problem. Although the Semantic Web approach will present significant skills challenges, the promise of that standard seems to fit well with CMDB data requirements: partial knowledge, multiple representations, discovery based, and so forth. I encourage you to investigate this avenue—but the learning curve for any hands-on implementation will be significant.[192]

For IT metrics and measurements with respect to SLAs, see Ruijs and Schotanus (2002), Brooks (2006), and Aitken (2003), Chapter 5. For the classic discussion of dimension management and modeling, see Kimball (1998) and Kimball and Ross (2002). Dennis Gaughan of AMR Research (paid subscription required) is actively developing a comprehensive metrics hierarchy for IT.

For an overview of the history of transaction-based processing in the context of end-to-end response management, see Tsykin (2002). For further information on the Integration Competency Center concept, see Schmidt and Lyle (2005). For discussion of the Application–Service relationship, see the ITIL *Application Management* volume (Office of Government Commerce 2002a). For further information on events and application management, see Sturm and Bumpus (1999) and Bumpus (2000).