

A data architecture for IT Service Management

Charles T. Betz

Chris: One thing that has us all puzzled is exactly how the ITIL/ITSM 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 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?

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 very relevant even if you are purchasing products. There's 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 really 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," while another vendor may call the same two things a "Template" and a "Service Instance" In our conceptual data model (which we also call a **reference model**) we call them "Service Offering" and "Service." It doesn't matter what the vendors call them, but you need to understand that any competent service request management solution should have both concepts. Doing the data model helps us understand our requirements better and communicate them to the vendor.

IT Service Management is running into the limitations of pure process-centricity. When one is architecting systems the concept of data is critical.

The Problem

Process-centric thinking is a hallmark of modern business practices. IT Service Management, and the ITIL specification, are process frameworks. They focus on overall functional capabilities and the sequences of activities which **add value for the customer** (internal or external). This is well and good, and one should always start with the business process in such matters. However, the ITIL/ITSM movement is starting to run into the limitations of pure process-centricity. When one is architecting systems (defined as combinations of people, process, and technology) the concept of **data** is critical. Data has been either absent, or at best a second-class citizen in much of the ITSM literature and training. The consequences of this are clear from an enterprise architecture perspective: processes can't be fully optimized, because the "things" which 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 – especially with the ITIL concept of Configuration Item – two very different things may be lumped inappropriately together in a given process context.

Without a, product-independent data perspective, IT Service Management 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 really exists. Without a sound, product-independent data perspective, IT Service Management and its implementers will be somewhat hostage to product vendors.

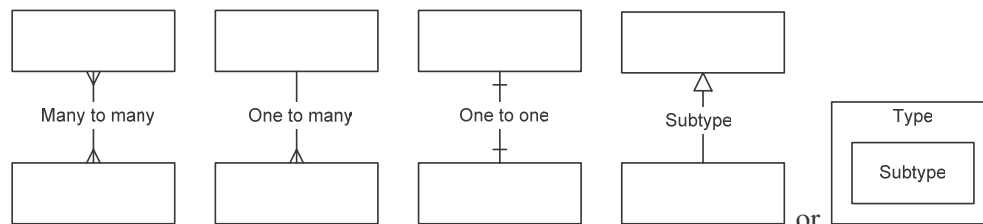
A Conceptual Data Model

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 terms are "conceptual model," "logical model," "domain model," "ontology," "class model," and so forth.)

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

- A Configuration Item may have many Changes, and a Change may have many Configuration Items. (Many to many.)
- A Machine may be related to an Asset, and an Asset may be related to a Machine. (One to one).
- A Configuration Item may be a Service, Process, or Application. (Subtyping)

These relationships are visually represented as follows:



Using these tools, we can start to carefully structure the relationships between the various loosely-used terms of IT governance: ¹

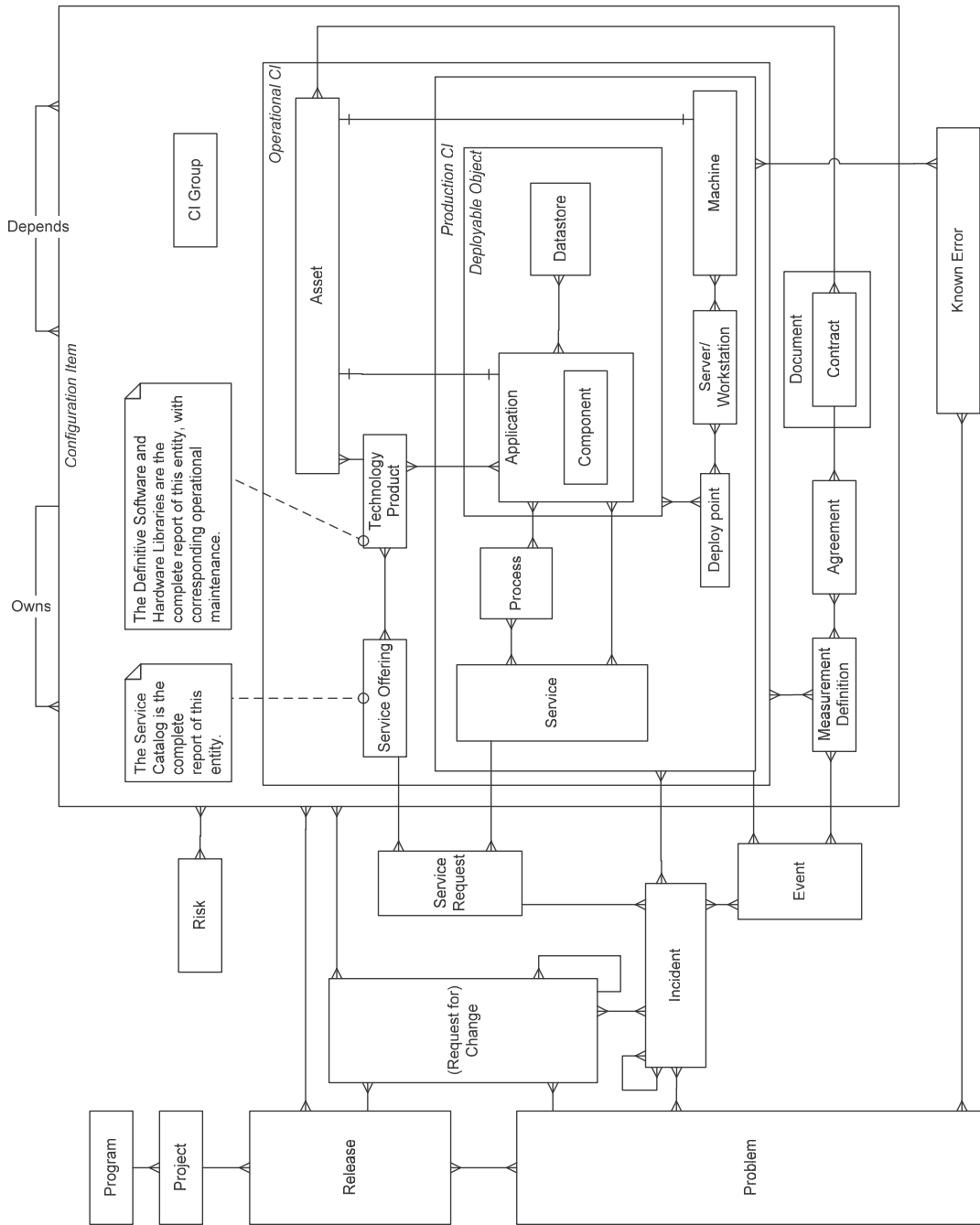


Figure 1. IT Governance information model

¹ I use containment to indicate inheritance, and the crow's foot to indicate cardinality. At this level of modeling I am not concerned with composition vs. aggregation, or optionality. I use my own Visio idiosyncratic notation partly for convenience, and partly for intellectual property reasons. For further information see any book on data or class modeling.




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

This is a conceptual data model. **It is primarily about refining language and concepts.** It deliberately omits a number of data structures that would ultimately be necessary to realize a solution.²

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 crow’s foot on either or both ends  or a cross hatch  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.

Configuration Item & subtypes

Configuration Item (CI) is one of the most necessary yet problematic concepts in all of IT governance. It must be very carefully defined and managed. Here is the ITIL specification:

“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...”

² The omitted data structures are generally intersection entities and dependent entities that elaborate on the core concepts. Process focused entities are also omitted. Some notes on possible approaches for elaborating this into a full logical data model are covered in the data definitions.

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

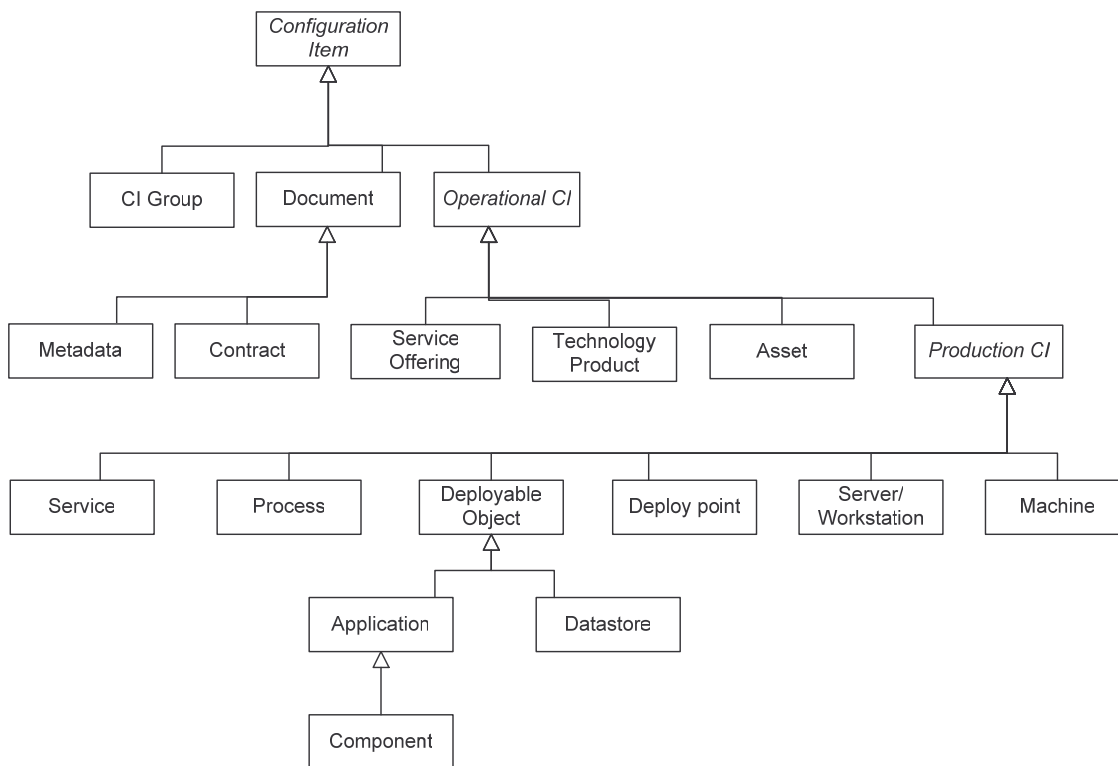
This is a very general representation, and one issue in the industry is that some vendors have interpreted this specification to allow their end users far too much freedom in defining configuration items and their relationships. More rigor is necessary. This analysis refines the ITIL representation and makes it more specific.

A CI itself is a **managed, specific object or element in the IT environment**. A CI by definition is under change control and the RFC process. That means that certain things are NOT CIs, for example:

- Events
- Incidents
- Requests for Change
- Projects

CIs may be logical or physical, deployed or undeployed, but always 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. Here is one representation:



:

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

Kelly: The italics mean that something can't **only** be an Operational CI, or a Configuration Item 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 anyways?

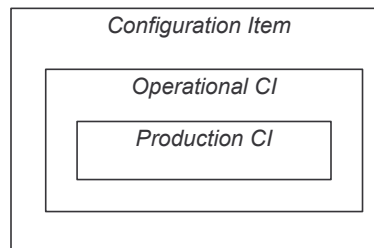
Kelly: It's all about being precise. Suppose that we just had one category of CI, which included documents, service offerings, and contracts as well as servers and applications. Servers and applications can have Incidents and Known Errors – but can a Contract? Not really. This is basic information modeling; people can spend their whole careers specializing in describing data structures precisely. One of the problems of CMDBs is that they didn't really take this side of things seriously at first, and so many early CMDB attempts weren't successful.

Servers and applications can have Incidents and Known Errors – but can a Contract?

The major types of CIs are:

- Configuration Item (base)
- Operational Configuration Item
- Production Configuration Item

They are “nested”:



which means that an Operational CI is also a Configuration Item, and a Production CI is also an Operational CI as well as a base Configuration Item.

This architecture proposes three major categories of Configuration Items: base, operational, and production.

The base **Configuration Item** 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). Also, a Configuration Item typically has an indeterminate lifecycle, unlike a Project or an Incident which are defined and tracked partly in terms of their closure.

Configuration Items have differing levels of involvement in day to day service management and production processes. The base level CI includes documentation and the definitions of service level measurements, objectives, and agreements. Any type of CI may be involved in an RFC.

Change control for items that are purely Configuration Items (not Operational or Production) may or may not be formalized – this is discussed below for the various item types.

An **Operational CI** is distinguished from the other CI types (Document and Group) as something **involved in day to day business processes, and that can be measured and is a primary entity in the Service Management workflow**. Changes to operational CIs (that are not also Production CIs) may be managed strictly by a functional group. 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 processes.

An **Operational CI** is something **involved in day to day business processes, and that can be measured and is a primary entity in the Service Management workflow**.

A **Production CI** in turn 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 most enterprise IT people think of when referring to “the change process.”

Configuration Item dependencies

The CI concept in this model recommends that arbitrary dependencies (owns and uses) be allowed only between CIs of the same type, or for purposes of grouping into manageable packages. Without this constraint, it would be possible for a (software) Component to contain a Server. Preventing such nonsense is the reason why we do logical and physical data models, and if your vendor or internal partners don't seem to get this issue, start raising flags.

However, it is possible for any CI of a given type to depend on or own a CI of that same type. This is discussed in the section for each type.

For further information see the section on recursive relationships.

Logical and physical CIs

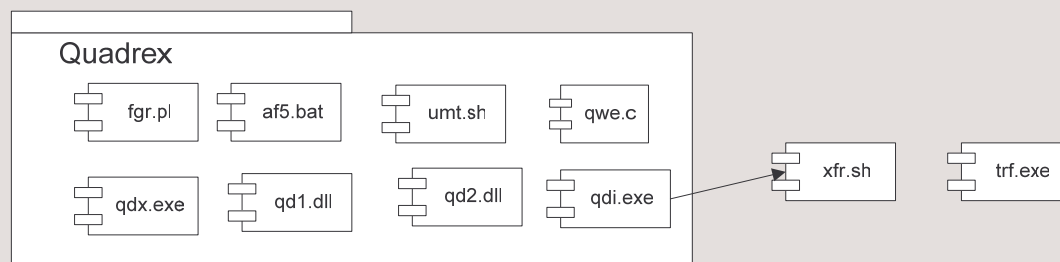
CIs can be logical or physical. Physical in this case means no ambiguity about the boundaries of the CI (even if it is only transient bits on volatile storage). Logical means that some consensus is required to set the bounds of the CI. Applications (especially in-house built), processes, and services 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 clearly defined process to establish the bounds of this potentially blurry “thing” and get buy-in that the boundaries are correct.

Applications (especially in-house built), 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.

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

Kelly: I found this diagram in some of your system literature:



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

actually 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 really no such thing, as far as your computers are concerned.

CI Group

Configuration Items require grouping for various reasons, such as supporting a Release or a Service Request. The CI Group leverages the Owns and Participates relationships to support this.

Document

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

Contract

A Contract is an agreement between 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 Configuration Items. Contracts are often the subject of intense scrutiny, and their signing is (or should be) a very 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.

Measurement

A Measurement definition is a Configuration Item because it represents the criteria on which IT service performance is measured.

A Measurement is a defined, specific characteristic of a Configuration Item. Specific should mean countable or otherwise deterministically translatable into some form of scale or categorization. This conceptual entity encompasses both the definition of the measurement as well as implying its specific instances. A Measurement is meaningless without the context of a CI. **Measurements have objectives as an associated concept** (not shown in the model.) An objective is with respect to a Measurement – what the measurement 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 Configuration Item because it represents the criteria on which IT service performance is measured.

Agreement

An Agreement is between two Parties with respect to a Service Level, Operational Level, or some other aspect of a Configuration Item.

An Agreement is between two Parties with respect to a Service Level, Operational Level, or some other aspect of a Configuration Item. A Contract may have many Agreements.

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 can precisely 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; if one is the client and one is the service provider it is a Service Level Agreement. 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 – one would hope 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 non-orderable 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 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.

Operational CI

An **Operational CI** refines the base CI concept by including things that are **measurable**³, which includes Service Offerings, Technology Products, and Assets. Operational CIs also are directly involved in the day to day provision of services, while 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.

³ Note that the Measurement entity is the **definition** of a measurement, such as “transactions per second,” “average response time,” or “downtime.” Such **definitions** are not themselves measurable – think about it. But they might well be under change control as a basis for contractual agreements.

While ITIL implies that CIs all participate in a generalized conceptual RFC process, they might not leverage the high visibility change control process which 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 lifecycles. A Service Offering goes through a different process than a change to a production application server. While ITIL implies that CIs all participate in a generalized conceptual RFC process, the reality is that they might not leverage the high visibility change control process which is usually focused on the production data center.

For example, a new Technology Product will probably go through some sort of adoption and certification process, perhaps led by key stakeholders for that given technology domain. But it probably will not be a subject of Change Advisory Board discussion, unless that CAB has a broader scope than usual.

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. However, a Machine may or may not also be an Asset.

Chris: Alright, 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 Nexatel? 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, 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 in-house built systems "assets." Generally this is **not** done, but there's increasing awareness that they 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 IDs, and should not be tightly coupled to serial numbers – cases have arisen where serial numbers change but the asset remains the same, for example if the serial number is tied to an assembly that is field-replaceable or is replaced under special circumstances, such as a server frame or motherboard.

Technology Product

A Technology Product is another name for the combined Definitive Software & Hardware Libraries (or at least their data indexing). 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 are dependent on those products, enables tracking the enterprise's status with respect to product obsolescence, security issues, vendor support, and overall technical roadmap. It also helps in program estimation and is an input into non-functional drivers of IT cost.

Chris: Non-functional drivers of IT cost? You lost me there...

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

Kelly: We understand when the business comes and asks us to build something. Where we fall down is when (Oracle CEO) Larry Ellison 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 either have to go without support, pay an expensive (and less-qualified) third party for aftermarket support, or re-test 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 in 2007 – we've got 3 big packages and 4 smaller applications, and we've already got the funding identified in our long range plan.

And don't even get me started on the Windows server patching costs – but again, with our mapping of Technology Products to Applications and Machines, we have a much better handle on our exposure there as well. We know that every 30 days we have to allocate 1 FTE per 10 servers for a week to patch and reboot, and we are able to notify those application teams systematically as we do that. Much better customer service.

Note that Technology Products may include both hardware and software. **It is difficult to make a distinction between the two in the purchased product space, as 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).

It is difficult to make a distinction between hardware and software for purchased technology products.

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”⁴ of the service provider; everything under it (in theory) may be opaque to the service consumer. 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 to the Service entity.

⁴ Application Programming Interface. This is a key concept to object and component oriented development; major characteristics of a computer program or module are *encapsulated* behind a defined set of gateway operations. 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. This is a perfect analogy for Service Offerings and Services. To carry it further: the Service Offering is the API *definition*, while a Service is a particular *invocation* (sometimes *very* long-running – the metaphor is not perfect) of the API.

A Service Offering is not a Service. One Service Offering may result in many actual Services; in other cases, a Service may not even have an Offering (it is a non-orderable 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 non-orderable service). However, an Offering with no ordered Services is like a poorly-selling retail product; its reason for being is clearly in question.

Examples of service offerings might be:

1. Provision new user with a workstation
2. Set up new email account
3. Set up new user in HRMS system
4. Complete bundle of 1, 2, and 3.
5. Provide wide area networking to new remote store.
6. Provide new tech stack instance to project (e.g. J2EE standard container and Oracle database).

Service Offerings in some cases will reference single or multiple Technology Products which may be composed of other Technology Products (a common term is “stack.”) For example, one service offering may be “Enterprise Java application hosting using Weblogic 8.0 and Oracle 9i, with high availability.” The overall stack might be called “HA Enterprise Java with RDBMS,” with dependencies in turn on Weblogic 8.0 and Oracle 9i and the necessarily 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, however.)

Service – Service Offering

A Service Offering may have many Service instances. See the Service discussion below.

Production CI

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 impact.

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 impact. 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 end user interacts or gains value from the complex underlying IT infrastructure.

The concept of “production” can be a little paradoxical. As the development lifecycle gets more and more mature, one can see developer's workstations and lab servers as “production” servers supporting the business process of software development. A true non-production status increasingly must be reserved for pure “sandbox” R & D machines being used to evaluate products and technologies. A developer workstation being used to develop assets upon a standard, proven Java/Oracle technology stack, to tight timeframes and deliverables, is a very 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.

Production CIs are often logical (Service, Process, 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.

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, or Datastores 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.

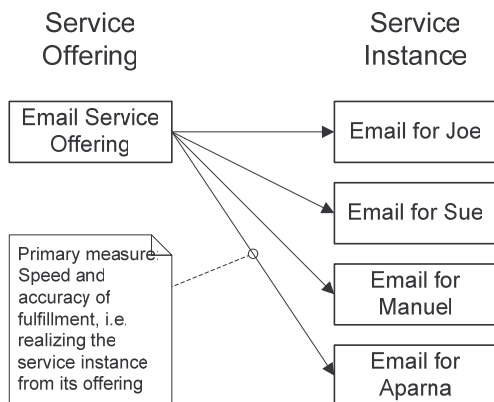
Service (and the Service-Application relationship)

A Service is an instance of a Service Offering. Where the Service Offering may be “Provide email to new user” the Service is “Provide email to Peter Baskerville,” accompanied by the various workflow steps documenting the provision of that service from start to finish.

A Service is an instance of a Service Offering.

Services do 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 finally it may be 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 end users. Provisioning end users results in many Services for one Service Offering:



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

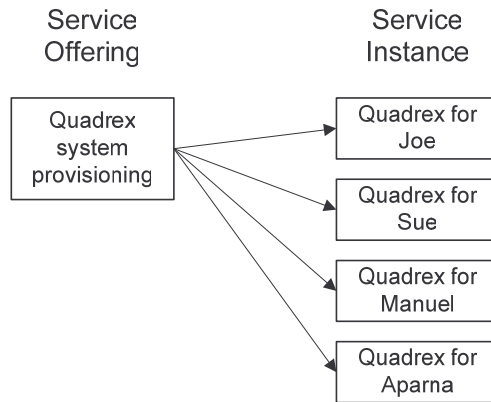
However, the following are Application services:

- 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 morning 99% of the business days.

Another emerging term for Application services are **non-orderable services**. They are the subject of service-level agreements based on measured behavior of the application (e.g. performance & availability).

Another emerging term for Application services are **non-orderable services**. This means that while they are measured, they are not requested⁵. Their ongoing maintenance is assumed and may be the subject of service-level agreements, but those service level agreements are not based on workflow (e.g. speed of request fulfillment) – they are based on measured behavior of the non-orderable service (e.g. availability). Non-orderable services do not have a Service Offering entry. Note that for comprehensive service level management, both Service Offerings and Services need to be tracked.

An Application may play a part in supporting service offerings, especially with respect to provisioning:



Process

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

A Process is a defined set of tasks, usually executed in sequence, that results in a specific business objective (according to process guru Michael Hammer, it must “provide value for the customer”). Processes should be managed as distinct CIs with clear names, identities, and lifecycles; formalizing their management is a major challenge today and current reality in most shops is a very informal process portfolio based on undocumented group consensus. It is a hierarchical concept with much ambiguity around granularity; there are various hierarchies such as Workflow/Task/Step and so on. Formally managing a process portfolio results in the interesting meta-question, guaranteed to glaze the eyes of executives: "what is the process to manage the processes?" (Something like "what is the data about the data?")

Process-Application

Processes are supported by Applications 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 we have one process depending on two applications.

Similarly, it is common for one application to support two distinct processes, such as a CRM system that supports both operational customer interactions as well as analytic planning purposes.

⁵ Or to be precise, they are “ordered” through the demand/program/project lifecycle. A current debate in IT Service Management 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.”

Process is the fundamental unit of management for the Business Process Execution Language (BPEL) spec. For further information see the literature on Business Process Management.

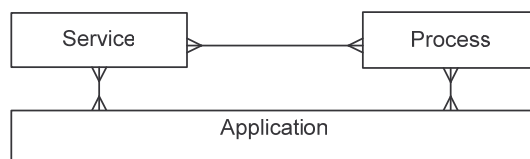
Application

An Application is a logical grouping of software components. Technologists may liken it to a "namespace." It is a consensus concept and must be carefully crafted so it is not too abstract or granular. Some rules of thumb that may be useful:

- An application should be recognizable to a senior business manager.
- Applications should be assigned to financial management structures. They should have clear executive ownership.
- 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.)
- All physical binary software components should be owned by one and only one logical application (i.e. as a namespace).
- Databases are not necessarily owned by any one application, however.
- Applications should have a unique and distinct human readable identifier, ideally a three or four letter acronym. All Configuration Items that are owned by the application should be named using that identifier as a basis for a naming standard.
- The same application may have different 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.
- If no-one wears a pager for it, it is not an application. Applications should be part of a Service. If an Application is not part of an identifiable Service, it might be a Technology Product instead. For example, if an IT shop uses Websphere Application Server for multiple different applications, Websphere itself might not be in the application portfolio - it would be a Technology Product (possibly part of a stack) on which Applications depend. However, if a shared Websphere service is managed as a unitary entity with an SLA by an infrastructure team, then that should be in the Application portfolio.
- Applications support Processes and Services (many to many). This triad of many to manys is a complex structure and needs to be refined to reflect an organization's culture and practices.

An Application is a logical grouping of software components. It is a consensus concept and must be carefully crafted so it is not too abstract or granular.

All CIs owned by the application should be named using its ID.



The application portfolio is probably the most important set of Configuration Items to baseline when an ITSM initiative is moving into the data center & production IT.

The application portfolio is probably the most important set of Configuration Items to baseline when an ITSM initiative is moving into the data center & production IT. Many CMDB initiatives fail because they attempt to start with the concept of physical binary Component, which (while it is straightforward to harvest from servers) is too granular and hard to manage for most organizations. The logical concept of “application” provides a key bridge between the overwhelming details of the technology and the business drivers it supports.

A defined process must be implemented for identifying that something is to be tracked as a formal "application." and one practice that is known to work is requiring consensus between an enterprise architect and an IT line manager that something should be assigned an application identifier, and included in the portfolio. Proliferation of application IDs (which can happen if a non-architectural, technical team is allowed to assign them) is a bad practice, as it prevents the correct rollup of IT operational data into larger hierarchies for scorecard reporting.

A defined process must be implemented for formally identifying "applications."

Application IDs 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 in! Firm labeling standards for all application interfaces would be a big help. This is nothing new; on older mainframe greenscreen systems the system and screen ID would typically appear in a corner. New distributed systems with less rigorous GUI development standards were a step backward in this concern.

Applications have varying types. A typical taxonomy might include:

- Business Application: production, transactional, business or customer facing
- Infrastructure Application: production, not directly business/customer facing
- Decision Support Application: based on a data warehouse/enterprise reporting infrastructure
- Desktop Application: productivity applications such as MS Office

For further information see the literature on Application Portfolio Management.

Application - Component

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

Application – Datastore

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

Applications are collections of processes and algorithms at their core. They are dependent in turn on datastores such as RDBMSes 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 re-analyzing 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 frequently large shared assets and the database administrators need to know **exactly which database** is serving an application. (This is also needed for regulatory compliance.)

Application – Technology Product

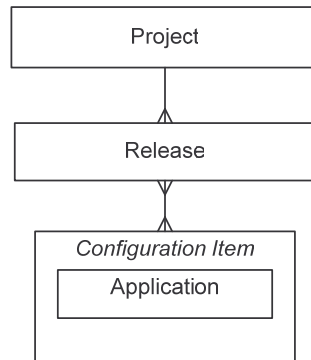
Applications depend on Technology Products. The distinction between the two is subtle but absolutely crucial. Bear in mind that Applications are **real, running instances**. People wear pagers for applications, they figure in SLAs, may have Incidents, and so forth. Technology Products show up on invoices and contracts, and the complete list of software Technology Products is the Definitive Software Library.

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 lifecycle, typically measured in months. Applications have an indeterminate lifespan, 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.

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

The relationship between Project and Application in the model is mediated through Release:



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. 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.

Component

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 as to what and where it is; components are non-ambiguous assets that can generally be objectively inventoried without debate as to their boundaries.

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

The use of Component here is not in a pure OO sense. In the OO world, a Component also has a well-defined interface which 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 an interface.

Components inherit from Applications and therefore 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 given current industry practice – 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 (e.g. checksums). In such cases the Deploy Point becomes the fundamental CI to manage.

Capturing component level dependencies is a recommended best practice for all aspects of enterprise application integration.⁶

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 due to the overloading of the term “Service.”

Datastore

A Datastore is a distinct, addressable source of data, usually structured. The most common examples would be database catalog and flat file; message queues may also be represented here. A Datastore should have one and only one data definition. As a deployable object it is directly dependent 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 – we don’t want Datastores containing Machines!

Datastores should have data definitions, which are by definition **metadata** – data about the data. The data definition tells you whether a given Datastore contains customer or supply chain information. Metadata generally is another term for everything we represent in our data model.⁷

Server/Workstation and Machine

It is **crucial** to have a precise definition of “server or workstation” and “machine.” Server/workstation is an **operating system instance, almost always networked**. Machine is a **physical computing device** which 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; Machine is the atoms and the serial number, linked in turn to the Asset tag.

The current failure of common Asset Management solutions to recognize this distinction is an industry problem.

Deployable Object

A Deployable Object is either a physical Application or a Datastore. This based on the fundamental computing theory distinction division between algorithms and data structures. In modern distributed systems, both have identifiable physical locations in a directory structure or analogous storage location.

⁶ See *Integration Competency Center*, John Schmidt and David Lyle (Informatica 2005).

⁷ I considered adding “Metadata” to the data model as a subtype of Document. There are some fascinating, frustrating hall-of-mirrors issues in considering Metadata itself as a CI. Decided not to go there.

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

Servers and Machines are NOT the same thing.

Deploy Point

A Deployable Object is tied in turn to a Deploy Point, which is usually a file system directory but might also be a management infrastructure such as a queue manager or a relational database management system.

The concept of Deploy Point as a major type of Configuration Item is an innovation proposed in this analysis, and comes from the author's experience with configuration management and supporting an Integration Competency Center. There are several major reasons for this:

- The need to identify "root" management concepts to facilitate interaction between infrastructure and applications teams.
- The sensitivity of certain directories when used as exchange points for moving data
- The need to manage RDBMSes and message queuing managers, which don't sit comfortably as "Applications."

Deploy Point is a major type of Configuration Item.

The application root directory

A large, complex application may have dozens or hundreds of directories, in some cases appearing and disappearing dynamically. However, it's a best practice to constrain the application's scope of activity to one master directory which serves to contain the myriad of sub-directories used by the application. This one master directory is a key interaction point for the infrastructure team managing the server and the application team (assuming that the IT organization has moved towards the best practice of segregating these teams and getting the application teams out of the business of server management.)

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

Shared libraries complicate this arrangement, but multiple applications updating shared libraries has been proven to be a very bad practice in Microsoft Windows, so hopefully the root directory concept will continue to gain strength as a management approach. This touches on core computing issues around component re-use and operating system services and architectures, and will never be a simple matter.

The shared exchange directory

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

Shared directories which facilitate application interaction are important points of control and need to be treated as Configuration Items.

The trouble with shared directories is that often the consuming application will be built with logic that states "Do X for all files in the directory." Thus, if an unexpected 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 which facilitate application interaction are therefore important points of control and need to be treated as Configuration Items.

IT Governance entities

Service Request

A Service Request is a logged interaction between an individual and the Service Desk requiring followup. Service Requests may have various Types such as:

- Hardware/Software Request
- Incident Report (i.e., the request is “Resolve this incident”)
- Configuration Change Request
- Security Request

and so forth.

A critical distinction is that between Service Request and project initiation. The service management architects will need to pay close attention to the differences between 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 at all but should be routed to a project initiation process. Alternatively, one might view the gateway into the project initiation process as a Service Request and drive to a more generalized approach.

Program

A Program is an ongoing, large-scale organizational commitment and corresponding investment towards meeting a major goal or objective of the enterprise. A Program typically consists of one or more Projects.⁸

Project

A Project is a defined set of manageable activities to achieve a well-specified mission, with explicitly allocated resources (time, money, staff), executed and measured within the scope of those resources. Projects may (or may not) be part of larger Programs. A Project has one or more Releases.

(Request for) Change

A Change is a work order or authorization to alter the state of some Configuration Item. Changes, like transactional logical units of work should be

- Atomic
- Consistent
- Isolated
- Durable

(For further information on the ACID model see any text on transaction processing.) In the context of enterprise IT, an **atomic** change is “all or nothing;” either the change goes in and succeeds, or it is rolled back completely. If a change might have some functionality that would be rolled back, while other functionality would stay, it should actually be framed as two changes. (Note that ITIL does allow for “partial rollback” but clearly indicates this is non-preferred.)

A Consistent change means that the functionality, when deployed, leaves the system in a stable state. Old functionality no longer needed by the new version of the system should be removed as part of the change. New functionality should integrate seamlessly with the previous functionality without undesired or unexpected impact.

⁸ It is possible to model higher level planning concepts such as Mission, Objective, Goal, Risk, External Driver, and so forth. This was not attempted in this analysis.

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

An Isolated change means (in theory) that it can go in without impacting other changes or major system functionality. This would be very 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; all instances of the new software in all deployment locations persist, and older software is not inadvertently re-installed (e.g. during a system restoration process). This requires attention to the Definitive Software Library.

Change-Release

A Release may have a number of Changes associated with it, but a Change should only apply to one Release.

A Release usually affects multiple CIs; however, CIs can be grouped together. This is the purpose of the CI Group, which can (at least) have types of Release Group and Technology Stack.

Change-CI

This is perhaps the most important relationship in all of IT Service Management. Very simply, a Change by definition affects 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 1,500 individual Configuration Items, yet this is what a simplistic approach will arrive at (e.g. in putting in an initial release of a major software package with many separate binary assets).

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 changes. Whether or not to inventory those binaries in the CMDB at all is one of the most critical approach decisions the ITSM implementer faces. For high security organizations, this may be done, but it is questionable if lower-criticality IS organizations truly require it, especially in a world of purchased software where the physical architecture of a software product is less and less a concern for the package vendor's customers.

Whether or not to inventory all binary software components in the CMDB is one of the most critical approach decisions the ITSM implementer faces.

Alternatively, the concept of CI Group (which is also a CI) can be used. An Application plus its Datastores and Deploy Points might be a logical CI Group. 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 other forms of CI Groups. 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 (e.g.) these 50 components.

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 intersection entity resolving Change-Incident should probably have a Type attribute, so we understand which caused which.

Production change and the software development lifecycle

RFCs in this architecture, and the concept of Change more generally, are not applied to project deliverables, 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...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.”

Release

Release is the gateway from the software development lifecycle into the IT Service Management world. It is one of the most important concepts for which to develop an enterprise approach. A Release is (narrowly defined) a distinct package of new 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.”*

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

The concept of CI Group may be helpful in supporting a Release’s various elements.

Note that Release Management as an overall capability includes the planning and harmonization of all Releases in the environment, not just managing Releases for an individual Project or Program (the enterprise Release Managers should interface with the program/project Release Managers).

Event

An event is raw material. It is any operational signal emitted by any production CI. Only a very small fraction of Events are meaningful to IT Service Management, and an even smaller fraction result in Incidents. Events are the raw material of Measurements, which in turn drive Agreements and Contracts.

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.” ITIL also implies that a Service Request is a type of Incident, which seems perverse (a Service Request might be for new capability, in which case it is **not** an interruption unless you are trying to build a culture of hostile customer service!) This line of thinking is **not** supported in the metamodels – ITIL editors please take note.

Service Requests may be tied to Incidents via the Configuration Item against which the Incident is reported. In this interpretation, Incidents are independent of their mode of

Release is the gateway from the software development lifecycle into the IT Service Management world

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

detection; this is necessary to support Incidents that may be detected through enterprise monitoring without ever being reported through the centralized Service Desk.

To refine this further, it seems that an Incident has to also be **experienced** by a person. (If a tree falls...). 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 be in respect to an Incident. Incidents (especially when generated from monitoring tools) often require correlation and root cause analysis, which are supported through the recursive relationships⁹ relating Incidents and Events to each other.

Problem and Known Error

In ITIL, a Problem is “the **unknown** underlying cause of one or more Incidents” while a Known Error is “a Problem that is successfully diagnosed and for which a Work-around is known.” However, this leaves a rather large hole for problems with known underlying causes that nevertheless have no workaround, so the ITIL spec won’t do as a data definition. 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 as well.¹⁰

ITIL further states that “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.”

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. It is a possible best practice that Problems are generally handled by Releases, while Incidents are handled directly by RFCs. 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.

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

Risk

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 configuration items such as Application and Process, and not looking to a common system of record for this critical reference data.

⁹ See below for detailed discussion on recursive relationships.

¹⁰ Comments appreciated: should the model restrict Incidents to only having one Problem?

Cross cutting data architecture issues

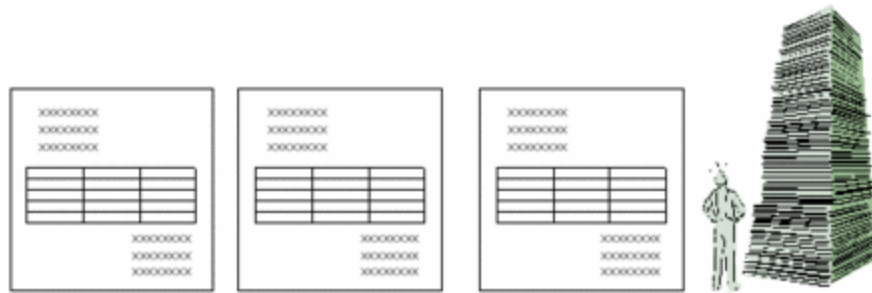
IT configuration management data (or metadata) presents unique problems compared to the data that IT manages on behalf of its partners.

Metadata, or IT configuration management data (this architecture sees them as synonymous) presents unique problems compared to the data that IT manages on behalf of its partners. Financial, logistics, and HR data has deep roots in paper-based history; a purchase order or hiring authorization message can be traced directly back to its roots in the forms once routed by interoffice mail to in baskets throughout pre-electronic corporations.

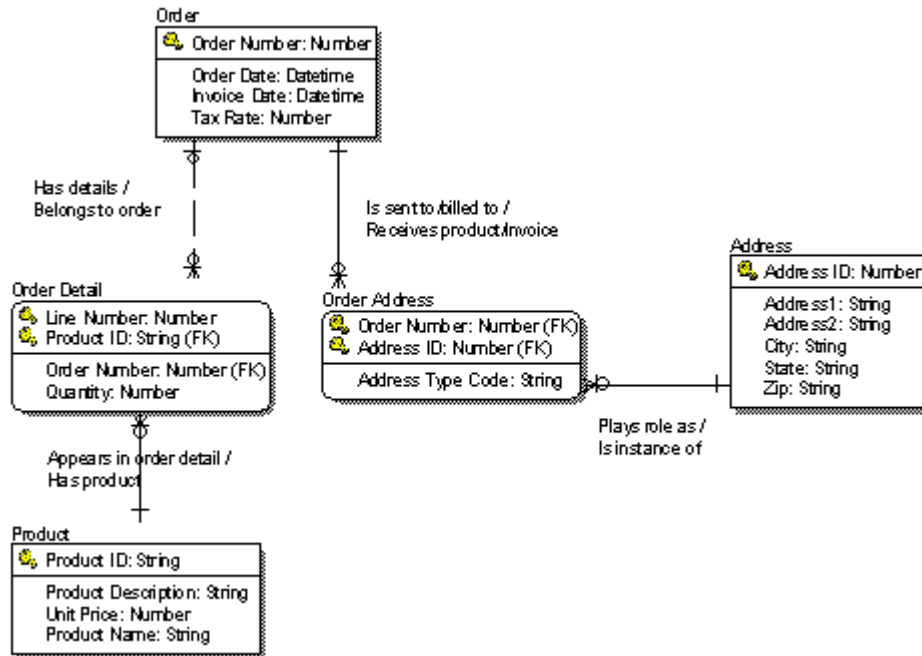
Recursive relationships

The “recursive relationship” is a common occurrence when managing IT data.

When one looks at a sales journal, or a stack of purchase orders, one generally sees consistency: the data model is the same for all the information.

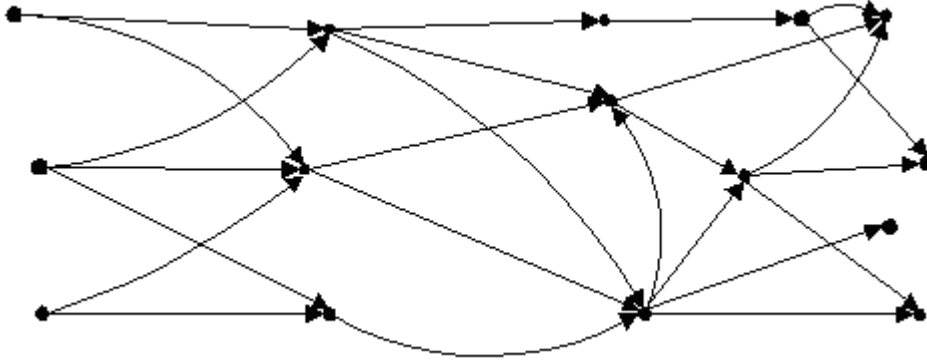


The data also has limited interconnections. A purchase order may reference common employee lookup tables and product tables, resulting in data models that are relatively straightforward to understand:



With metadata, everything gets much more complex. Data metadata is the most tractable; tables (or entities) have columns (or attributes) and therefore building simple data dictionaries is straightforward. But when one moves beyond this into technical metadata (i.e. configuration management) the data starts to take on new characteristics. In mathematical terms, it becomes graph-based; that is, it looks like this:

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



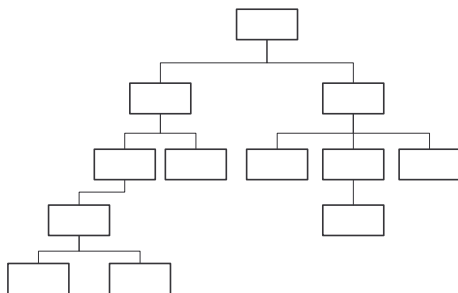
This kind of data presents well-known problems in storage, querying, and presentation, as it requires "any to any" data models and can rapidly become complex to the point of incomprehensibility.

This kind of data is not typically encountered in business-centric systems that are the successors to forms-based paper processes. It is the kind of data stored by configuration management databases and metadata repositories when they move into managing technical metadata such as interconnections between network devices, integration flows, and so forth.

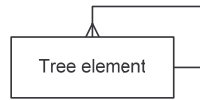
The **recursive relationship** enables complex data. This is a relationship when one type of thing can be connected to other instances of the same thing. There are two major types of recursive relationship:

- 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 IT Service Management are Configuration Items containing other CIs; organization hierarchies; process steps decomposing into finer grained activities and tasks, and so on. Here is how a tree often looks:

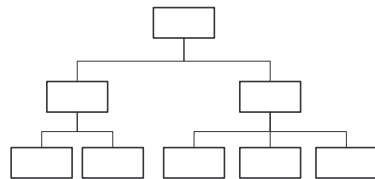


Notice how it is always possible to say that one box owns and/or is owned by others. A tree can be recognized in a data model by the following notation:



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

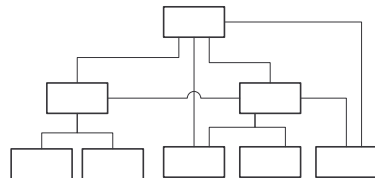
While simpler than networks, trees can be troublesome to report on if they are of indeterminate depths; that is, if one branch of the tree is 5 levels deep and the other is only 3, it's hard to get a consistent, sensible report. A common strategy of data architects when dealing with tree-like structures is to **fix the levels** and establish that all branches of the tree have the same number of levels:



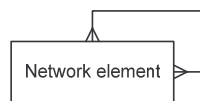
but this in turn may have problems in dealing with the real world – what if the organization (or whatever) is just not structured that way? In fact, organizations may actually decide to structure themselves, and adapt their business processes, to fixed level hierarchies, as we see in retail organizations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

A network is characterized by things that are related to other things

A network is characterized by things that are related to other things, not necessarily containing. A diagram of a redundant wide area network, an org chart with “dotted-line” relationships, or a mapping of how systems interrelate, would probably be a network. Here is how a network often looks:



While there are tree-like structures in it, the key 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 following notation:



This is also often called the “any to any” relationship.

Trees and networks make IT service management data much harder to deal with, compared to sales or financial data. Why is this? Let's start with a picture of my son:



Thanks, yes I know he's cute. He's a happy boy :-). What I want to draw your attention to is the Skwish™ toy he's holding.



Now, regular business data is like a deck of cards:



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"

and 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 whole toy? Or just things immediately connected to the red sphere? By elastic? By wood?

The Skwish toy represents interconnected 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 ERP for IT (and IT service management)? Much reporting is of the deck-of-cards variety. You can handle this with the same tools your business users use: relational databases and reporting/BI tools such as Crystal, Brio, Microstrategy, Actuate, and others.

Using these well-established techniques, I 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 configuration item?
- How many projects do I have running right now?
- What projects contributed to building this system, and what did they cost?
- What does this system cost to run?

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

But those tools don't handle reporting on interconnections. With relational databases and query tools, it's very 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? (Requires traversing an unknown number of project tasks and subtasks – obviously, project management tools do it, but an end user is hard pressed to deal with this data in raw form.)
- For a project, what 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? (A very important compliance issue.)
- 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 spec, you probably are into the Skwish type (tree or network) problem. The problem is that while the theorists have been kicking this around for a while, no standard approaching SQL has been implemented across multiple platforms.¹¹

Recursive relationships in the data model

The recursive relationship can easily be abused, and enable nonsensical connections. One of the major problems with the Configuration Item 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

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

¹¹ See Oracle's CONNECT BY operator

example, a cell phone should not “use” a database trigger, and a RAM chip would have nothing to do with an XML Schema – yet some configuration management tools allow the end user to put in such relationships. Being more precise is why we go to the trouble of building our data model – in an “any to any” world, you don’t need a data model’s specific lines, because anything can be hooked to anything.

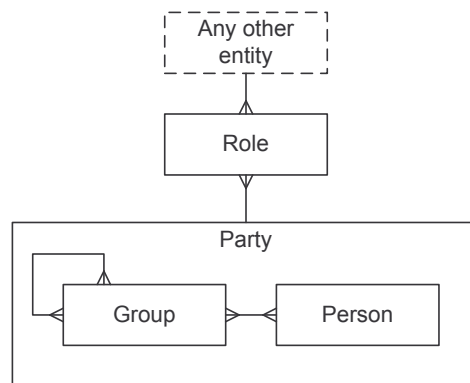
It is usually the case though that any configuration item **of a given type** can both use and contain other objects of the same type, especially in a high level conceptual data model such as this.

For example, a server might contain hard drives; both would be types of machine. A machine might be connected to other machines via a network. A process can both contain and depend on other processes. Datastores contain other Datastores, and with mechanisms like linked databases also 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 also depend on them without owning.

And 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 on 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 end users. In a poorly engineered tool, the **user** gets to decide what gets related to what. That is a recipe for chaos.

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

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 (Persons or Groups composed of other Parties) have Roles with respect to any Entity in the model.

Party/Person/Group

A Party is either a Group or a Person; Persons 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, as business users do not understand it. They understand concepts like “administrator” or “steward.” However, these are **roles**. (These are very well understood issues in data modeling.)

Roles

Here are some example role types and the entities they might interact with. Note that ITIL does go into some depth around this, so it 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 request a change to an existing Service.
Support Group	Usually Application	A Support Group would usually be a Group that is associated with one or more Applications. Sometimes, a Support Group might be associated with a Technology Product (for example, a Windows Engineering group).
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 that are associated with the CIs in question, as well as overall change coordination from a central enterprise group. Frequently, the Change Approval group may have standing representation from major Technology Product areas (e.g. Unix engineering, network engineering, etc) or other operational capabilities (e.g. Security)

Here is a common role type that may be problematic:

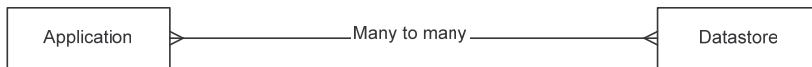
Change Approver	Any CI	ITIL calls for just a Change Approval Board. However, different CIs have different stakeholders. A better practice is to allow flexibility in the definition of the acting CAB (here called a Group) by CI. This however will require more maintenance.
-----------------	--------	---

		<p>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 better understood use of Change Approver is with respect to Production CIs.</p>
--	--	--

Intersection entities

This is a high level conceptual data model. Most of the relationships are of the “many to many” type. For example, an Application may use many Datastores, and a Datastore may be used by many Applications:

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



In order to actually turn these language concepts into an operable system, an **intersection entity** is required:



If you look at the main data model and imagine all the many to manys 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 your database administration team has a list (at least a spreadsheet) of all their databases. Perhaps you have an application management group with their 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 a defined entity, documented processes are required for the creation, reading, updating, and deleting of data in the Application/Datastore entity. Would it be your application team? Your DBA team? A separate team of configuration analysts?

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

The current state of most IT organizations is much less formal. What we often see is uncoordinated spreadsheets.

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:

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	WNAPPL01	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.

The application team knows that Quadrex is using QDXAPP02, and UNXDB001, but doesn't have the level of detail that the DBAs 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. But some of them haven't quite bought into that point of view...

Kelly: Right... Common argument nowadays! Finally, 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 DBA spreadsheet,

misspelling it “Qaudrex.” This means that when we go to consolidate all this data into one database, we’re going to have manually identify and clean that up.

Chris: Why didn’t the DBAs 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 leverage that list? It can be done in Excel, but you start to get into advanced features – too far down that road and you’re looking at needing a real database-based system.

The same issues need to be thought through for every many to many relationship:

- Event/Incident/Problem
- Applicaton/Technology Product
- Application/Process
- Change/CI
- Change/Incident

And so forth. 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 frequent vendor mistake is to put in a one to many where a many to many 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. Or a Datastore may be shared by many Applications, but a configuration management tool only allows you to identify it with one. These are the kinds of details that are 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 if you’re buying a vendor product, the data model is needed.

Process and workflow

In this data centric presentation, we haven’t talked a lot about workflow and process. Let’s turn to these from the data perspective:

The CRUD matrix: an old standby

A very well known technique for understanding data’s relationship to process is the unfortunately named CRUD matrix. CRUD stands for

- Create
- Read
- Update
- Delete

The CRUD, or Create/Use matrix, tells us the relationship between data and process.

A simpler version is a Create/Use matrix, whis is what is presented here.

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:

		Processes																									
		Portfolio Management						System development						ITSM/ITIL													
Entities		Demand Management	Project Portfolio Management	Application Portfolio Management	Risk Management	Contractor Management	Asset Management	Vendor Technology Product Management	Project Initiation	System Requirements	System Integration	System Analysis	System Design	System Build/Integrate	System Release	Incident Management	Problem Management	Configuration Management	Change Management	Release Management	Service Level Management	Financial Management for IT	Capacity Management	Continuity Management	Availability Management	Service Request Management (Help Desk)	
	Program	C	U		U	U	U	U																			
	Project		C	U					U																		
	Release														C						U						
	Problem		U	U												U	C				U						
	Risk	U				C		U		U	U	U								U	U	U	U	U	U	U	
	RFC					U									C		C		U	U	U	U					
	Incident															C	C	U									C
	Service Request															U	C			U							C
	Event															U	U					C					
	Known Error															U	C										U
	Service Offering														C							U		U	U	U	C
	Service															U		U	U								C
	Technology Product							U	C	U			U					U	U			U	U				U
	Process	C				U				C	U	U	U		C	U	U	U	U	U	U	U		U	U	U	U
	Application	U	U	C	U		U	U		U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
	Component												C	U	U	U	U	U	U	U	U	U	U	U	U	U	U
	Deploy Point				U								C	U	U	U	U	U	U	U	U	U		U			
	Server/Workstation				U		U	U			U		C	U		U	U	U	U	U	U	U	U	U	U	U	
	Machine						C	U		U			U	U	U	U						U	U				
Datastore			U	U	U							C	U	U	U	U	U	U	U	U	U	U	U	U	U		
Asset				U		C	U															U	U			U	
CI Group							C							C				C	U	C	U	U					
Measurement					U																C						
Agreement					U																C						
Contract	U	U	U	U	C	U	C		U	U					U	U				C	U	U	U	U	U		
Document	C	C	C	C	C	C	C		C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	

This Create/Use matrix is presented as a starting reference model. There's lots of interesting questions generated by such a matrix; the cells highlighted in gray show some of these:

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 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.

Measurement and Agreement look like very weak entities; they are subsidiaries of Contract and are not widely used by other processes. To make the conceptual model cleaner, we might consider removing them entirely.

A Document can be created by any of the process areas – and when this is true, the value of having the entity in the CRUD matrix becomes questionable.

These weaknesses are purposefully left in for demonstration purposes; one objective of this document is to enable the reader to build their own analysis.

Workflow

Entity lifecycles and audit trails

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. Technical terms will be **effective dating** and **audit trail** (if your vendor gives a blank stare when you mention these, look elsewhere).

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 above). A trail of who “owns” an incident and where it has been referred is a key feature of most incident management tools; this is a specific example of the general principles here. Effective dating of status changes is (in part) how SLAs are monitored for things like Incident, Service Request and Problem resolution.

Effective dating of status changes is how SLAs are monitored for things 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 them?
- Who has approved this Change?

Reporting

One criteria for evaluating IT enablement tooling is the quantity and quality of reporting available. Some reports that should be available with push-button ease from an integrated (ERP-like) IT management system would be:

Operational reports

- Applications using Datastores and Servers
- RFCs and their Configuration Items
- Projects affecting an Application
- Changes planned for a given Release
- Releases affecting mission-critical (i.e. Risk-identified) systems

- Application Dependencies on Technology Products
- Applications accessing Confidential (i.e. Risk-identified) Databases on Servers
- Applications and their Stakeholders
- Incidents resulting in Problems
- Problems by Application
- Problems by Application, correlated to Project Release (Quality Assurance report)
- Average Time to Resolution of Service Requests
- First Call Resolution for Service Requests
- Servers due for Lease Refresh, Dependent Applications, and Owners
- Applications dependent on other Applications
- Services and their Dependencies
- Application Transmission of Data to other Applications
- Data Definitions for Datastores
- Services dependent on External Providers
- External Connections due for Stakeholder Validation
- Total Cost of Acquisition for Application
- Allocated Infrastructure Cost for Application

Exception reports

Exception reports are critical checks on processes; they often are the key metric defining process success.

Exception reports are critical checks on processes via analysis of data quality; they often are a metric indicating process success.

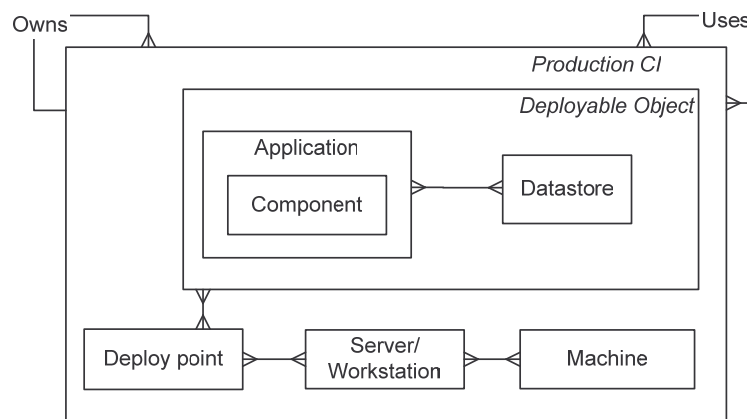
- Service Request/Incident/Problem aging over X days
- Applications with no known Stakeholders
- Applications/Services with no known Supporting Dependencies
- Servers with no known Applications
- Servers with no known Machine

These are only examples – a robust integrated IT enablement capability will have hundreds of reports.

Complex dependency capture

The core of configuration management

The following entities are perhaps the core of production configuration management:



There are many variations and refinements on this basic problem area, for example the DMTF has extensive specifications and large parts of the OMG's modeling languages also cover it. The Fundamentals of Integration Metadata series on the www.erp4it.com site goes into great detail on enterprise application integration configuration management.

A typical pattern in many organizations is for this data to be very fragmented, into small, incomplete spreadsheets and databases. Often, there is no defined process for keeping these data sets up to date, and each point solution takes a particular point view at the expense of related data. (See the story in the Intersection Entities section above.)

Centralized attempts to maintain this data have proven ineffective as well. There are three major areas that historically have attempted this mission:

- Metadata management (the oldest)
- Enterprise Architecture
- Configuration Management (most recent)

The trouble in all of these cases starts when a centralized team attempts the data maintenance, with little or no distribution of process steps and (often) no defined process at all. History has not been kind to such attempts, which usually wind up abandoned.

Central maintenance of complex dependencies usually fails.

It is therefore critical to distribute the maintenance of complex system dependencies to the application and support teams. This is not as hard as it sounds:

First, it is not generally recognized in the IT service management discussions that complex system dependencies are carefully mapped by development teams. The problem is that these mappings are too often done in a format (e.g PowerPoint or Visio) that cannot be consumed by a configuration management capability. This is simply a matter of standardization and tool alignment. (See the Model Driven Configuration Management article on www.erp4it.com.)

Second, an elegant closed-loop process with mutual incentives can be defined between the application and engineering teams. Most organizations have split server (data center) engineering and operations from the application development and maintenance teams; this is increasingly required to meet audit objectives regarding separation of concerns. The data center team is often hard pressed to understand the dependencies on a given device (the classic "Can of Coke" problem – if I spill one on a machine, what business process is affected?) This leads to continuous issues when servers need to be patched for cross-cutting system reasons (e.g. anti-virus measures), or when 20% of the servers in a data center must be retired for lease refresh reasons.

The application teams find themselves reacting to unplanned outages, reboots, and having to scramble because a server must be decommissioned. Neither side in this equation is particularly happy with the other.

The solution (which has been seen to work) is to require the application team to document their dependency on the server. The understanding must be that if the application team does this, it will be notified of server impacts. If it does not do this, the application team has no expectation of notification. Since application teams are generally more customer-facing than the engineering/data center teams, they have the higher incentive to do this.

The trouble with relying on discovery is that it can only tell you what is there – not whether it should be there.

A note on discovery

Tools have always existed that can inspect an IT processing environment and analyze the inventory of programs, files, processes, and the like. These tools are becoming an important point of discussion as Configuration Management becomes more mature. Too often, however, they are presented as a “silver bullet.” The trouble with relying on discovery is that (even at its best) it can only tell you what is there – not whether it **should** be there. (Just like your bank statement – it tells you what the transactions were, but did you intend them?)

A major capability any discovery tool requires for relevance in enterprise IT is the concept of fingerprinting or footprinting. This is an ability to infer from the presence of some physical component that a logical application dependency exists. For example, if we see the executable file qdx.exe on a server, we can infer that the logical application Quadrex is dependent on this server. (However, it cannot answer the question whether Quadrex **should** be there.)

Footprints must be maintained – if a new Quadrex module consisting of newly-named executables is deployed, the discovery system’s database will need updating. This needs to have an explicit process step, probably as part of the overall Release process.

Finally, discovery tools are limited in their ability to detect application to application data transfers, which are some of the most critical dependencies in a large enterprise. Due to the wide variety of means via which application data can be transferred, no discovery tool yet exists that can comprehensively map all the different flows: FTP, file shares, middleware, ETL, EAI, Web services, shared databases, and so forth. Files renamed en route, parameter-driven EAI adapter architectures, dynamic binding of process to data resources, and similar challenges make this an extremely difficult configuration management challenge; for further information, see the Fundamentals of Integration Metadata series on www.erp4it.com.

It should however be a goal for the enterprise’s technical architects to specify an integration architecture that is as transparent as possible, so that this notorious problem becomes easier to manage. Providing greater visibility here is one of the key justifications for centralizing an enterprise Integration Competency Center.

Final issues/open loops

This model attempts to handle both workstation/desktop computing and data center computing in one. I question whether they should be completely separated.

The model does not go into the higher levels of IT governance (Mission, Objective, Strategy, External Influence, Risk).

The model does not cover financial data explicitly. (The mapping of Application into the enterprise's financial structures is a key enabler.)

Acknowledgements

Many people have helped me understand this complex problem domain. In particular many insights have come from interactions with Chris Capadouca, Pete Rivett, Curt Abraham, and Doug Jackson. Apologies if I have overlooked anyone else!

Intellectual property notes

Copyright © 2005, Charles Thomas Betz. All rights reserved.

This material is entirely owned by Charles Thomas Betz. It was either developed before July 2004, or after June 2005. It has been entirely written on my own time and (unless attributed) no text or figure has been directly copied from any other fixed form. **No text or figure under any circumstance has been extracted from proprietary, unpublished work products (mine or others') for any employers.** Terms of art, common turns of phrase, basic conceptual structures, and the like are emergent properties of the language and industry practice. This material will thus unavoidably have similarities to other industry artifacts; however, **all has been re-considered, re-authored and re-presented in a new fixed form.**

Substantial care has been taken to present this material in a highly generic form, applicable to all IT organizations in all industries, and from which no inferences can be made about the particulars of my employment situations.

“Employees cannot be compelled to ‘wipe clean the slate of their memories.’”
Moss, Adams & Co. v. Shilling, 179 Cal. App. 3d 124, 129 (1986)