Architecture Governance

Logical Architectural – Diagramming Guidelines

UML 2.0 style guide

October 2008

Table of Contents

1 INTRODUCTION.....1
1.1 Overview.....1
1.2 Context.....1
1.3 UML.....2
1.4 Scope and Purpose.....3
1.5 Acknowledgements.....5
1.6 Diagram Guidelines and Drawing Conventions.....6
2 TELLING THE (FUNCTIONAL) ARCHITECTURE "STORY".....8
2.1 The Setting – The Perspective View.....10
2.2 Outlining the Plot – System View 1.....13
2.3 Developing the Characters – System Views 2-n.....15
2.4 Organizing the Dialogue - Sequence Diagrams.....17
2.5 The Conclusion – Deployment View.....19
2.6 Character "Bios" - The Glossary.....22
2.7 The Afterward – Deviations from the Ideal.....23
2.8 Additional Diagrams.....24
2.8.1 Use Case Diagrams.....24
2.8.2 Activity Diagrams.....24
2.8.3 Data Model Diagram.....24
3 SYMBOLS AND STEREOTYPES.....26
3.1 System View Diagram Symbols.....26
3.2 Data Flow Diagram Symbols.....27
3.3 Deployment View Symbols.....29
3.4 Stereotypes.....30
3.5 Module Behavior Stereotypes.....30
3.6 Architectural Deviations.....32
3.7 Interface Descriptors.....32
4 EFFECTIVE DIAGRAMMING.....34
4.1 Interface Pattern.....34
4.2 Shared Component Library.....34
4.3 Controller Pattern.....35

5 USING VISIO 36
5.1 Shapes and Stencils 36
5.2 Connections 37
5.3 Zooming 37
5.4 Aligning Objects 37
5.5 pro.lankoenig.com stencils 37
TABLE OF FIGURES 42
BIBLIOGRAPHY 44
INDEX 45

1 Introduction

1.1 Overview

These *Architecture Diagramming Guidelines* are intended to facilitate the communication of technology architecture. This can either be as a part of an Enterprise Architecture Governance process or simply in the flow of information from business requirements to systems architecture to technical design to implementation to operation.

Whatever the reason, communicating architecture is important to any large technology infrastructure and core to any Enterprise Architecture Governance Process. In order to communicate and concept, particularly complicated ones, both the presenter and the audience must speak a common language and that language should be descriptive enough, flexible enough and precise enough to get the point across..

They say "a picture is worth a thousand words" and where technology architecture is concerned, the old adage is pretty much true. As opposed to inventing something brand new, these Guidelines start with an existing, fairly widely used diagramming language called the UML (Unified Modeling Language).

UML is widely used, extensible and flexible. But with flexibility comes a degree of imprecision. In addition, UML is generally used as a language for communicating design rather than architecture and many of its constructs are moving in the direction of "model driven design" or models that are precise enough so that code can be generated directly from the model (or nearly so).

Architecture on the other hand is more abstract than design, which is more abstract than implementation. As such, we intend to use diagrams to draw out larger relationships between systems and reproducible patterns, rather than designs and implementations. We can use the flexibility inherent in UML quite effectively as long as we define which UML constructs we intend to use, how we intend to use them and what they mean in our context.

1.2 Context

When we define architecture, whether it is the architecture of a service, an application or a combination thereof, we divide the architecture into four main parts:

1. **Information Architecture** – A systems agnostic view of content as it exists across the technology landscape. The scope covers: entities, relationships, data sets, categorization (ontology), content organization, flow, search, navigation, etc.
2. **Functional Architecture** – The modules (organized logically not physically) across the technology landscape proving the features and functions. Often this is decomposed into *Services* and *Solutions.* But even on a smaller scale there is usually some separation between shared system-level modules and product specific modules and the glue that binds them.

    Service – Coarse grained reusable infrastructure that is live and operational and accessed via loosely coupled interfaces. A service is responsible for handling load and scaling itself as appropriate. You don't build a new one when you run out of capacity

    Solution – Application, Product, etc that either monolithically delivers required business benefit (architecturally bad ☹) or aggregates multiple services

together in order to do so, adding the necessary business logic and glue (architecturally good ☺)

3. **Physical Architecture** – The systems (e.g. computers, disk arrays, back-up units, etc), networks (switches, routers, Load balancers, DNS, etc) and houses (data centers, power, cooling, etc) that represent the physical manifestation of the production environment.
4. **Business Process Architecture** – The processes that support the underlying business model for which the technology was built. This could be related to getting "hits" on a website to support an advertising model or it could be a full blown order processing/shopping cart/authentication/authorization/billing process supporting a paid subscription. Whatever the Business model is, there should be a Business Process Architecture defined to support it (no matter how simple).

In this context, the following diagramming guidelines (UML style guide) apply solely to the aspect of Functional Architecture.

1.3 UML

UML was intended to be a visual language for capturing software designs and patterns. We use version 2.0 of UML, which made quite a few changes from the original version (1.0).

Physically, UML is a set of specifications from the OMG. There are four individual specifications, all available on the OMG web-site at http://www.omg.org

Even though UML was born to capture design concepts, we are using it to capture architecture concepts because it is extremely flexible, designed to be extensible and has an extensive vocabulary for expressing behavior, relationship and flow.

It is useful to think of a system as a 3-dimensional entity and an architecture diagram as a 2-dimensional "view" of that 3-dimensional entity. Therefore to get a full picture of the entity in question, one requires multiple views of it. UML provides for multiple views so that the full model [1] can be represented.

This 3-dimensional object is represented through three separate 2-dimensional projections or *Views* of the object.

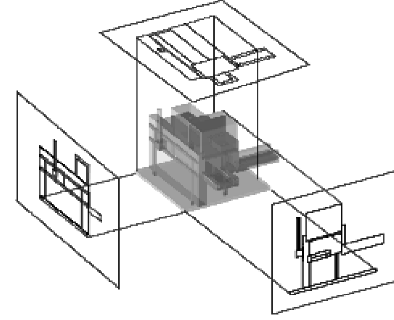


Figure 1 - 3-D projection

A single 2-D view does not adequately depict the full object, but all of the views taken together, give a good understanding of the 3-D object. Similarly, the individual *views* in a set of architecture diagrams must be understood in the context of each other to tell the full story.

[1] Model – "UML 2.0 in a Nutshell" defines as – a means to capture ideas, relationships, decisions and requirements in a well-defined notation that can be applied to many different domains.

UML provides for multiple different diagram styles in two groupings:

- Structural Diagrams – To represent how things relate
- Behavior Diagrams – To represent how things act

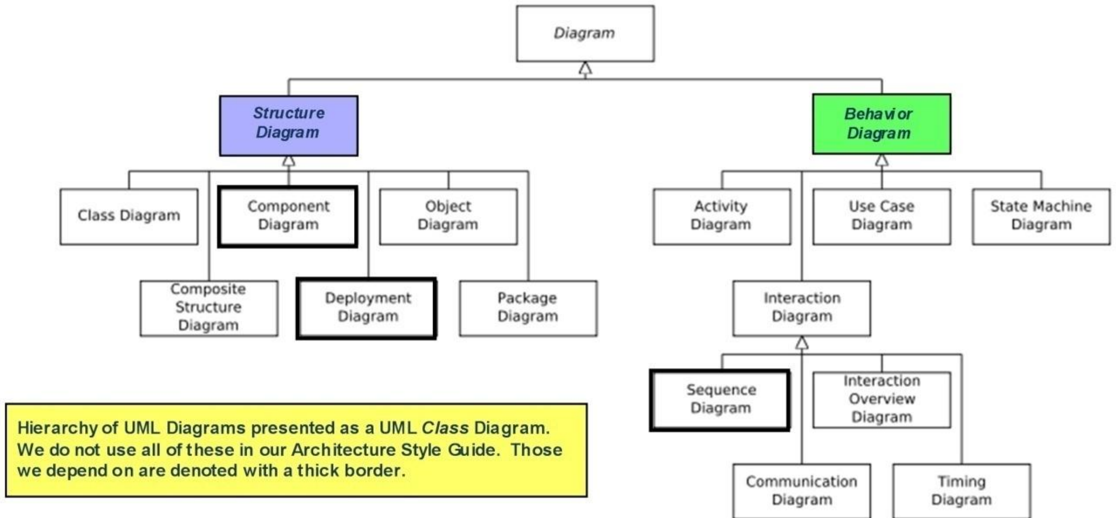


Figure 2 - UML Diagram Model

In our usage (i.e. the style guide for representing functional architecture), we will only be using three[2] of the available UML diagramming styles: Component diagrams, Sequence Diagrams, and Package Diagrams (these are noted by thick outlines in Figure 1)

1.4 Scope and Purpose

The purpose of the architectural diagrams provided in these guidelines is to provide a complete functional architectural overview of the system from a logical perspective. What that means is that we are trying to understand the modules and how they interact in order to provide the features and functions of the system, but not how these modules are arrayed onto computer systems across networks in order to accomplish that task. This latter problem domain is delegated to a set of *physical diagramming* guidelines which are defined specifically for that purpose and covered in a separate set of guidelines. These diagrams are also not intended to be a detailed design, but as with any well defined architecture, should be both a high enough level to get the full perspective and detailed enough to bind the design and place it in context.

In effect, if you were to compare the processes of architecture and design to the output of your favorite mapping software or web site, the architecture would be like the overview map allowing you to place the detailed maps into context with the whole and with each other.

It is really up to individual organizations as to whether they annotate the diagrams with textual documentation to explain them or not. Many will choose to do so, many will

[2] Just as the UML is intended to be extende3d, so is this style guide. We err on the side of simplicity "less is more". But if an important concept needs to be communicated and these guidelines do not cover it, then by all means they should be extended. And if UML already has a mechanism for `communicating in the form of a diagram, then by all means, use it.

choose not to. As long as the diagrams are kept accurate, the bulk of the information can be conveyed in a meaningful way. If by virtue of writing hundreds of pages of documentation to accompany the diagrams, an organization makes the architecture too costly to maintain, then the purpose is defeated. Non-updated diagrams, that fall out-of-line with reality are like a “dead language”; beautiful as they are, have no real value.

On the other hand, if the diagrams do not “speak” for themselves, without accompanying text, then likewise, the communication problem is not solved. It is important that the communication mechanism be tuned to match the use and style of the organization using it.

One mechanism that has worked in the past is to write very little accompanying text, but to have the architect present the architecture to his/her peers much the way a Masters student would present a thesis for peer review. Everybody learns and knowledge is transferred. But once again style must fit culture.

While the diagramming style is not meant to imply a specific architecture process, or specific architecture tools, we present the diagrams from widest scope top narrowest scope. This may be accomplished following a hierarchical decomposition process or not. We are not trying to prescribe the process followed to achieve the architecture, only to recommend a style for communicating it.

We start with a black-box view of the entire system being described.

> A black-box view shows the interfaces a module requires, the interfaces it provides and any other detail required to describe the *guaranteed* behavior of the module. It does not specify anything about the internal implementation of the module(Pilone & Pitman, 2005, p. 60).

The first view, by virtue of being a black-box view of the entire system, we call the *perspective view,* because as the name implies it provides perspective into the environment in which the system operates. This view defines all the other systems both inside the company and outside that the system being described interacts with and the interfaces (and contracts) defining that interaction[3].

Given a black box view for one or more modules, we can then use a white-box view to look inside.

> A white-box view shows the details of how a module realizes the interfaces it provides (Pilone & Pitman, 2005, p. 63).

A white box view of a module shows how the module realizes its interfaces by presenting the contain modules (sub-modules) as black-boxes, with their interconnections. For an example, see: Figure 6 - System View 1, on page:13. These sub-modules may be further decomposed in white-box views in subsequent diagrams. This decomposition process can continue as far as it needs to go. Eventually the white-box decomposition must be illustrated as a UML class diagram. This is when the architecture process has `clearly transitioned into the design process and is therefore no longer the subject of this guideline.

The architecture diagrams and communication proceed from the outer level, progressively opening up the black boxes and looking inside only to find more black

[3] We have found the perspective view to be of enormous value as it is the interactions between systems owned by different groups and the different understanding each group of the contract between them as the root cause of a sizable number of real-world issues, and also something often ignored.

Copyright © iankoenig.com

boxes, until eventually, we have gone far enough. It's the architecture equivalent of the Russian Matryoshka Doll.

Figure 3 - Matryoshka Doll

An oft asked question is: "How far do you go in the decomposition. Do you have to go to the edge of the design process?" The only answer is, "Go as far as you need to and no further".[4]

We use the UML Component view primarily to represent the modules encapsulating logical functionality and the ball/socket icons to represent the interfaces by which they are connected.

Once we have laid out the modules and their relationships via encapsulation and interfaces in sufficient detail to understand the components of the system, we represent data flow using UML sequence diagrams (a type of behavior diagram). It is generally very useful to distinguish interconnection / relationship from flow using UML structure diagrams to represent the former and behavior diagrams to represent the latter. Do not mix the two.

1.5 Acknowledgements

We would not have been able to write this without building on the works of those who came before us, including some people we worked with at Thomson Financial (before it merged to become Thomson Reuters Markets):

- Gabo Gilabert
- Gene Osgood
- Joseph Ierullo (at that time working for Cyberplex, Inc)

[4] This is a bit like saying that as the Architect it is your job to take big problems and carve them into a finite number of discrete smaller problems that are small enough to be solved – "small enough to fit in one brain"

1.6 Diagram Guidelines and Drawing Conventions

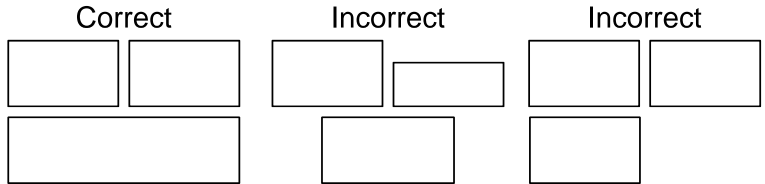
The sections that follow provide general guidance for creating the diagrams. By establishing a common language and notation, the complex architecture concepts may be communicated between teams and for the purpose of governance.

The symbols used are generally from the UML 2.0 notation. Where we have extended the base UML 2.0 notation, we will point it out. Using Visio describes the use of Microsoft Visio with some purpose-built shapes to create the diagrams and symbols used in the guidelines. This is not meant to be a recommendation of that tool or in any way imply it is best suited for the task. It's just one of many possible choices.

When creating a diagram the following standards should be applied:

- Try to use only the symbols and stereotypes that have already been defined (such as the set in this document). If concepts need to be communicated that are inconvenient or impossible with the defined shapes, then the standard should be amended first. Doing otherwise is like inventing new words in your "language" without first defining them. Since clear communication is the goal, this should be avoided. That being said, if in your architecture a new symbol can be defined that will express a complex problem in a concise way, then by all means, define it. That's the whole point of using diagrams rather than textual prose in the first place.
- Label each module on the diagram appropriately to convey its purpose and meaning. UML defines a *callout (note)* that can be attached to any object to convey additional meaning.
- As Modules, sub-modules and interfaces are carried between views, it is essential that exactly the same name is used – It is confusing to the reader, when the modules change names between the diagrams, just as it would be confusing to the reader if the main characters in a story changed names as the plot develops.
- Before a module appears in a white box view (as a container of other modules), it should first appear in black box form. It is confusing to see a module appear on a white-box view without context. It is similarly confusing to see a module referenced in a sequence diagram (showing data flow), which is not also represented in a system view (showing structure / relationship). In other words, we need the structural context to understand the flow context. That is how multiple views of the same system work together to tell a full story.
- Think spatially when laying out boxes in a system view. We try to think top-to-bottom, left-to-right. This is not meant to imply data flow, but is meant to imply that spatial relationships on the page matter. For example: Two boxes of the same size next to each other on the page tend to mean "peers". Bigger generally means more important (or more complex). Higher on the page, generally means "first". Size, symmetry and arrangement do matter when you lay out a page. For example: if you have two modules that are effectively peers that "go through" a third module to connect to the rest of the system, it is important to lay out the boxes to convey that meaning:

Copyright © iankoenig.com



- This top-to-bottom, left-to right thinking should be carried through the spatial orientation of modules as far as possible through each level of diagram. The more consistent the diagramming metaphors through the various views (i.e. chapters of the story), the clearer the story will be to the reader.
- Color is in general not used in the base guidelines[5].
- In a system view, while we specifically do not try to show data flow between modules, we recognize that data flow influences the structural relationship that we are trying to show. On a vertically oriented page, we tend to show modules that are first in the flow near either the top or the bottom of the page (depending on which way you like to draw). Modules that are later in the flow work progressively to the other edge of the sheet. Since everybody has their directional preference and we need to understand this bias to communicate effectively, we choose a method other than orientation on the page. Since we can generally classify modules based on their proximity to the user, we arbitrarily denote modules that are closer to the user as *south* and modules that are further from the user are *north*. In this way, a content oriented feed processing system (such as the one we will use in our examples, show data flowing north to south and from top to bottom on the page). For the sake of consistency, we strongly recommend that data always be thought of as flowing north-to-south and that a compass icon be placed on the page to reference which direction is north. See Figure 4 - Compass Direction, inset to the right.

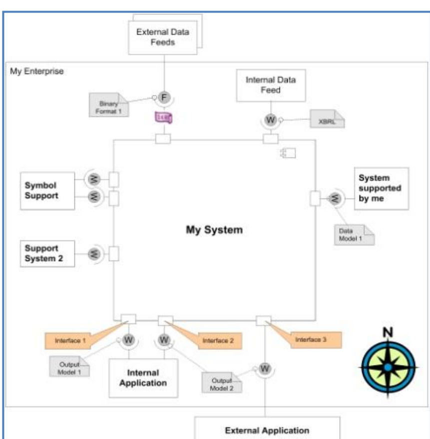


Figure 4 - Compass Direction

- As we progressively focus inward on more detail of a particular module, it is useful to de-scope other parts of the architecture that are outside the box, generally by eliminating them from the diagram as long as there is another view that brings these back into scope. That being said, it is always useful to maintain encapsulation context in the diagram and never lose that. To represent this diagrammatically, modules that are further "outside", but still in scope due to encapsulation context may be grayed out to denote this. For an example, refer to Figure 7 - System View n
- To increase readability, elements should be arranged on the page in such a way as to avoid the crossing of connectors between objects wherever possible.

  In fact, one of the metrics that can be used as a measure of system complexity is the number of overlapping connectors *that cannot be avoided* when diagramming the architecture.

[5] Color is often used to give meaning and it is entirely acceptable to extend these guidelines to define the meaning of color in a diagramming style. This basic model, does not define meaning for color (except certain stereotypes, which we will get to later)

2 Telling the (Functional) Architecture "Story"

Comparing an Architecture definition to "telling a story" as in a novel is a useful metaphor. A novel has a setting, a plot; it has characters, dialogue, a theme, a style, and ultimately a conclusion. The architecture *story* has all the same elements.

As the Architect, you play the role of narrator, using the tools and metaphors of your chosen language, the language of diagrams and symbols to develop the plot and tell the story of your system. In the process, you highlight the key decision points and flex-points in the architecture like plot-twists in the story line.

When a novelist tells a story, she uses words. When the architect tells the story, pictures and diagrams are used. Complex technology problems are represented through a series of individually easy-to-understand views that work together and integrate to tell the full story.

> To accelerate understanding and communication in the strange world of Quantum mechanics, Richard Feynman, the great 20th century physicist introduced a set of simplifying diagrams (now called Feynman diagrams) as a graphical computational aid, allowing ordinary graduate students to keep track of the complex mathematics(http://en.wikipedia.org/wiki/Feynman_diagram).

We start with a few key definitions (terms we have been using in the description so far).

- **Module** – A package of business logic and data that performs a function. Modules can (and do) contain other modules. We call a module *fully encapsulated*, when it is entirely contained within another. Modules interact with one another via *interfaces*. We do not want to get too physical here, but for a package of logic to be considered a module in the architectural sense it is useful to think of it as being either a process or a thread, as definitely being a module. For a callable sub-routine within a larger module, the decision has to be based on "architectural significance". On the other side of that arbitrary line we have the elements of design.
- **System** – An arrangement of *Modules* and interfaces performing a well defined function in the overall technology landscape. *Systems* are coarse-grained. Types of *systems* include: Services (as in an SOA), Solutions, et al.
- **Interface** – A set of methods and events (i.e. a protocol), plus the associated data models (e.g. data elements and structure) defined and offered by one *module* and consumed by another. *Modules* may support multiple interfaces, not all of which may be consumed. Interfaces may be private or public[6].

With these three building blocks we can begin to represent the key aspects of Architecture: The **system** we are defining, its decomposition into **modules** and the **interfaces** between the modules (and of course between the system and the outside world).

Modules are further decomposed into sub-modules and the interfaces between them. The decomposition and elaboration process continues as far as necessary to tell the story without "losing the plot".

---

[6] Whether an interface is private or public is really just a specification of scope from a point of view. In other words, an interface may be public from the point of view of a module, but private from the point of view of the encapsulating system.

Copyright © iankoenig.com

Before we describe the tools at our disposal for telling the story (continuing the metaphor of communicating architecture like telling a story), lets first correlate some of the aspects of telling a story (e.g. writing a novel) with their counterparts in describing architecture:

- **The Setting:** The *Perspective View* sets the technological context for our system in the overall enterprise landscape.
- **The Plot:** The *System Views* develop the plot to tell the story.
- **The Characters:** The *Modules* that perform the business logic of the system represent the characters in the story. The *Glossary* section holds for each module (character) the one paragraph "bio" describing its raison d'être.
- **Dialogue:** The *Sequence Diagrams* organize the gripping Dialogue of the story referencing the *Interfaces* between the *Modules*.
- **Conclusion:** The story concludes with the *Deployment View.* This is where the *Modules (i.e. characters)* all find homes in physical systems (and live happily ever after). When we write the Physical Architecture (i.e. the Sequel), the *Deployment View* from the Functional Architecture becomes the setting for telling that story.
- **Theme:** Systems often follow common architecture patterns. When describing one of these, the pattern can be said to set the theme for the architecture story we are telling. In other words, Architecture Patterns define common ways for laying out systems performing similar functions. One of the Patterns we define is the Content Distribution Pattern, which lays out a common *recipe* for architecting content delivery systems.
- **Style:** These guidelines either in their raw form or appropriately extended define a set of metaphors and constructs we use to convey meaning and tell our story.

So you can see, without having to squint too much, that many of the elements of writing do apply to conveying the architecture to an audience. And just as there are good stories and bad stories, there are good architectures and bad architectures. Just as there are good ways to tell a story, there are good ways to convey architecture through effective diagramming.

In the end, to win over your audience, you must both have a good story and you must tell it well.

2.1 The Setting – The Perspective View

The first view of the architecture is what we call the Perspective View. This View places the system we are describing in the context of the overall technology landscape, effectively setting the stage for the story we are about to tell. In other words, it gives us perspective as to how our problem domain fits into the rest of the technology world in which we live.

A UML 2.0 Component Diagram is used to represent the perspective view. In it, we treat our system as a black-box, focusing entirely on all the other systems and interfaces we require from / supply to the outside world. By outside world, we mean the world outside the system we are defining, including but not necessarily limited to the world outside the enterprise.

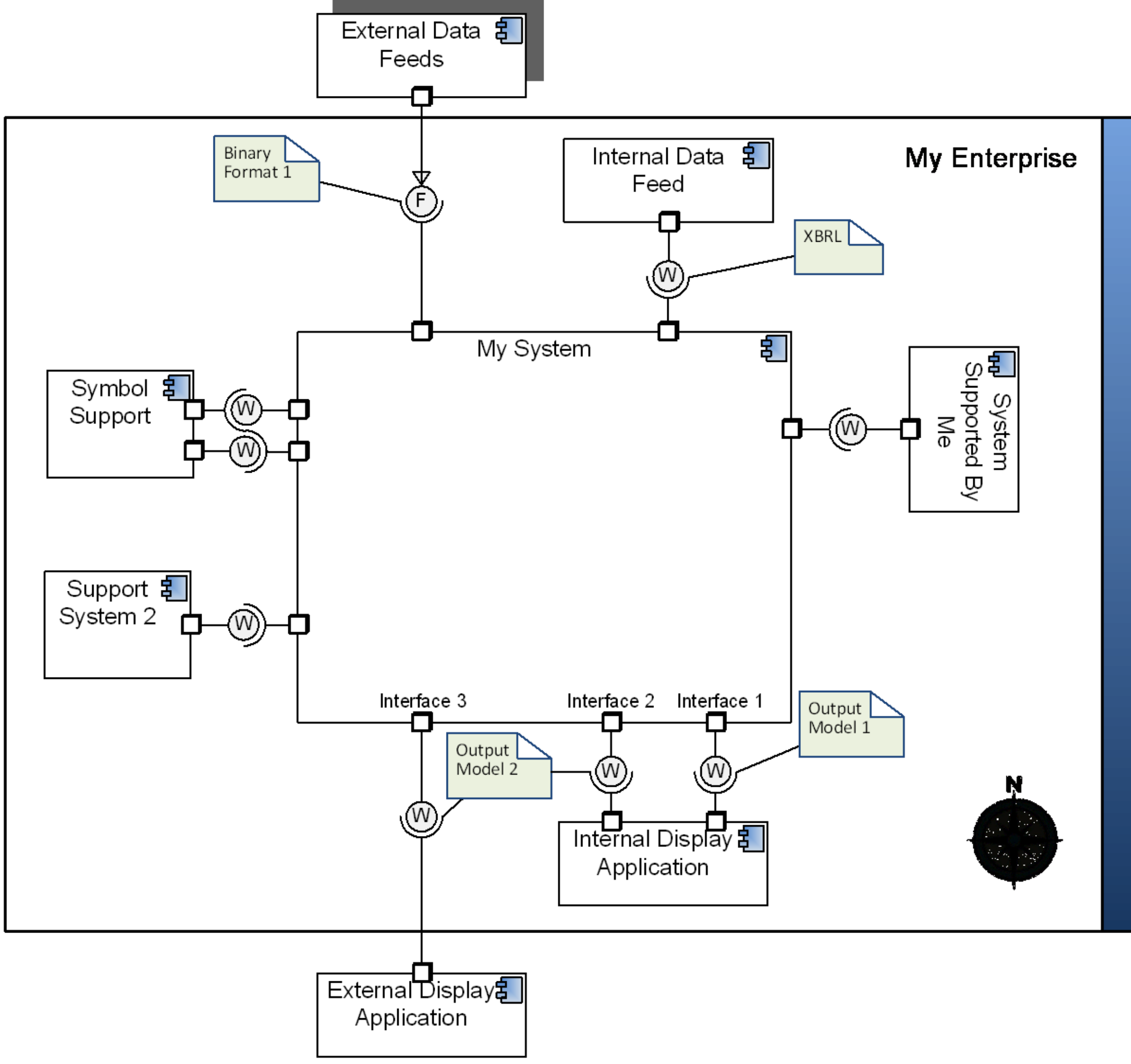


Figure 5 - Perspective View

In Figure 5 we show an example of a Perspective view. Our system ("My System") is treated as a black box (it's white, but we don't show the insides – so it's a black box), and we focus on the connections between "My System" and other systems both inside

and outside the Enterprise. We lay these out spatially on the page to show meaning. 'My System' is a content processing system, so in this example we chose to show the data sources on top and the application consumers on the bottom. For diagrammatic emphasis, we use the long side of the paper for the main data flow and the short side to show supporting infrastructure. We have placed the compass icon in the lower right hand corner of the page with North facing up, to show this. It is not vital that this model be followed prescriptively. But it is vital that whatever model is chosen is used consistently. For the architecture diagrams to communicate ideas clearly and effectively, the metaphors used to convey meaning cannot change during the course of the story. When your reader has to spin the page around (like a piece of modern art), to figure out which end is up, it just isn't very effective[7].

In addition to orientation, we are also communicating encapsulation (meaning boundaries and containership) in these diagrams. We show which systems are internal to our enterprise and which are external. In essence, the enterprise is our primary boundary, which likely translates to the physical boundaries of a data center and its infrastructure.

We have spoken about spatial positioning on the page as a communication vehicle. Another communication tool we can use is the size and positioning of various boxes with relation to each other. Boxes should be positioned so that they "fit" correctly, when we expose the white box versions in later diagrams. The size of a box (particularly in relation to other boxes) denotes *importance.* Making one box bigger in relation to another generally means it is more important to the part of the story we are telling..

You will also note that all external interfaces emanate from *ports*. These are the small boxes on the border of *My System* where the Interfaces connect. Use of ports is a UML 2.0 diagramming convention that we follow to convey the importance of encapsulation. *Ports* are the only way in or out of a properly encapsulated module.

> Mapping this metaphor to the real world is potentially interesting. Real ports (like in harbors) are where ships from far-off lands dock and off-load their cargo. Ports are also the location where the cargo is checked for safety, legality, etc. Extending this metaphor to the architecture realm, Interfaces (carrying data in payloads) are akin to ships (carrying cargo). Ports are where we check (or should check – hint, hint) that the interface has brought nothing illegal or harmful into the module (like a Trojan horse or a virus).

When we get to the white box diagrams we will elaborate the implementation of the exposed interfaces by carefully tracking the connection of ports through various encapsulation layers to the underlying modules providing the interface[8].

For the Interfaces at the bottom of the diagram, we have used a few extra symbols. We add additional elaboration for these, because these are the Interfaces that *My System* exposes to external systems. In other words, these are the *Interfaces* that allow other systems to use our implementation so effectively represent our raison d'être.

[7] Using mnemonics like the compass to show direction can be effective mechanisms in those cases where it is better to change how the direction of flow is oriented on the page.

[8] It is sometimes interesting to note the number of layers of encapsulation that interface implementations go through before they are exposed as a metric of complexity, but that issue is left for a discussion on proper architecture rather than diagramming style, which is what we focus on here

Copyright © iankoenig.com

The symbols we use to elaborate the Interfaces in the example above are:

W	This "ball and socket' ideogram, represents an interface connection between two modules. The *ball* represents the provider side and the *socket* the consumer. The "W" in the ball says that this interface is a SOAP Web Service (which gives it other well-defined characteristics) [9]
Interface 1	The name for the *Port* defines the Interface Name. For a Web Service, it should also refer to a WSDL.(Web Service Description Language) definition, which describe the Interface
Output Model 1	The data model *callout* for the *Interface* defines the Data Model of the returned *document.* For a Web Service, it should reference an XSD (XML schema definition)

There is further description on symbols and stereotypes in Section: 3 - Symbols and Stereotypes on page: 26.

[9] Use of the single letter mnemonic to denote the type of Interface is an extension to the base UML 2.0 specification. Additionally, UML allows for the ball-side and socket side to be used independently, which is useful in a design context. However in an architecture diagram we are only interested in the interaction of modules / systems with each other so we always represent both sides of the interface as a contract.

2.2 Outlining the Plot – System View 1

The next view of the architecture (after the perspective view in Figure 5 is the first System View. We give this the highly descriptive name "*System View 1". This view* takes the system presented as a black-box in the perspective view, and presents it as a white-box, exposing its implementation as a set of interconnecting modules.

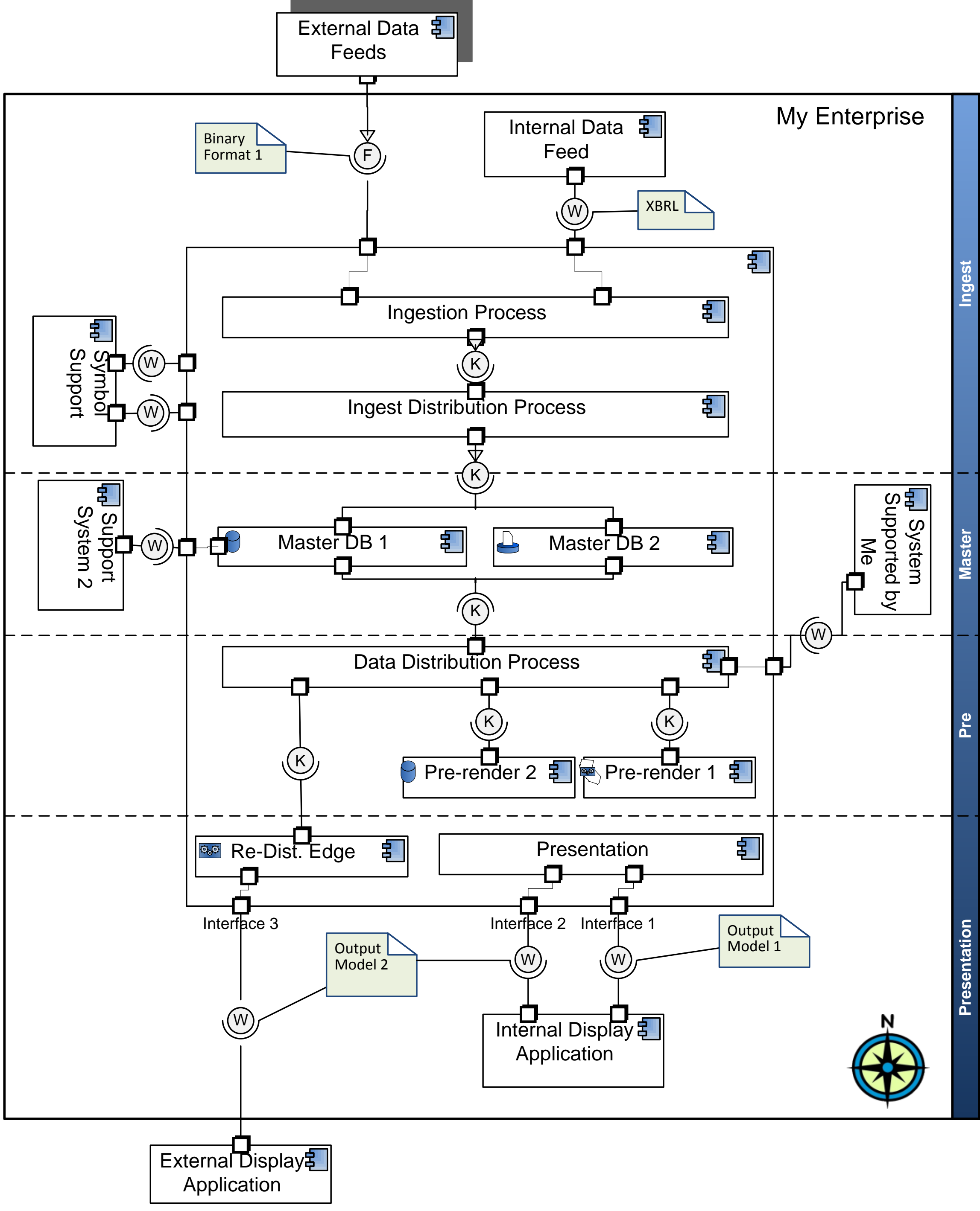


Figure 6 - System View 1

As we open up the implementation of "My System" essentially focusing in on its internals, we are also defocusing on the externals. In the first level decomposition are the large functional blocks of "My System" and the interfaces between them. These can be thought of as the main characters in our story. Their presence and interaction through interfaces both within the context of the system and through the boundary to the outside world is the main plot line of the architecture.

The modules are laid out spatially on the page according to tiers and layers. What this means is that they obey internal boundaries and rules associated with their roles in the system. It's nice to be able to section off areas (groups of modules) with straight dotted lines (zone demarcations) and state clearly that 'this zone is for this purpose[10]. For, example if we know a module exist to guard the perimeter of the system, then it is likely going to live in a DMZ in the physical architecture. Without getting too physical, it is useful to denote all the perimeter defenses in a zone. We do not mean the physical infrastructure like a firewall which we know exists, but which is not an element in this functional architecture. We do mean the modules in this functional architecture that perform this function. Another example would be presentation tier modules that take raw data (e.g. encoded in XML) and create human readable display (e.g. HTML).

Where clear boundaries can be drawn using zone demarcations, they should. Care must be taken when these are physical boundaries. We have to remember that we are drawing out the architecture logically and functionally, not physically. When architects go too fast into physical architecture (i.e. what the answer should be before truly understanding the problem), they generally get bad architectures. That being said, certain physical realities must be architected in from day one, for example the storage of customer private data in a secure zone of the network. Ensuring that logical modules do not cross what will ultimately be impenetrable physical deployment barriers is part of laying out a good logical architecture and knowing which physical barriers are architectural in nature and which are design in nature is part of being a good architect.

Each Module is depicted using a UML component, which provides for a stereotype (the icon in the upper right hand corner) to denote type. At the level of granularity we are working, this is generally going to be the 'generic component' stereotype ( ), but UML defines others that we find more useful during the design process.

We also add a behavior stereotype[11] to the upper left hand corner to denote what the module is (e.g. Stateless or Stateful business logic, a database, et al). These are further described in Section: 3.4 - Stereotypes on page: 30

[10] Anybody who has seen a computer processor chip under high magnification will clearly see different zones of functionality clearly separated from each other

[11] Use of behavior stereotypes is an extension to the UML 2.0 standard.

2.3 Developing the Characters – System Views 2-n

The next *n* views of the architecture continue to develop the main characters (the Modules introduced in System View 1, by taking each Module presented as a black box and then (as necessary), exposing it as a white box, thus exposing its implementation in terms of sub-modules.

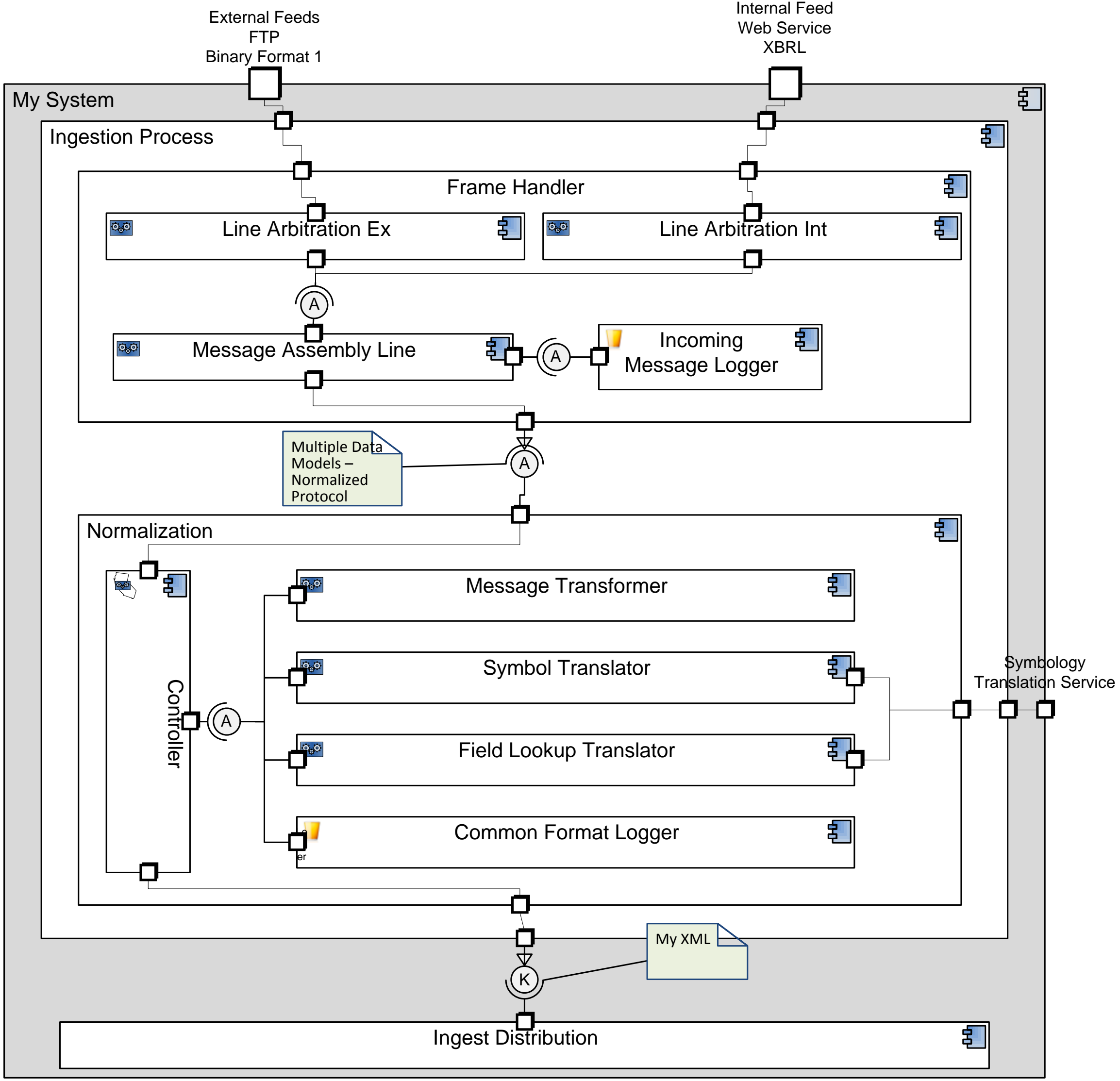


Figure 7 - System View n

As shown in Figure 7 as we progressively focus in further, we color the outer layers of encapsulation with shades of grey to show that they are out-of-scope. The further out-of-scope, the deeper the shape of gray should be. For the first time, in Figure 7, we can see the module that actually implements the handler for the external and internal data feeds that we introduced in the Perspective View (Figure 5). We can also clearly see that in order to handle external feeds, this implementation crosses three encapsulation barriers, which probably tells us a bit about the kind of physical security, ports and protocols that will be enforced on such an interface. Knowing this kind of information

before key design decisions are made can be the difference between a successful implementation and one that goes back to the drawing board.

It is important that all the main modules be decomposed to the point of clarity; at a minimum, this means understanding the key building blocks such that each one is defined well enough for a team to go off and implement it with a reasonable chance of getting it right (in the context of the overall system). Additionally, any module referenced in a sequence diagram must also be present in one or more system views (and the Glossary).

Copyright © iankoenig.com

2.4 Organizing the Dialogue - Sequence Diagrams

Data Flow is shown using UML 2.0 Sequence Diagrams. These diagrams take a key activity / function that the system performs, represent the modules of the system as participants in the flow and show the sequence of method calls / messages flowing through the participants, temporally ordered to accomplish the task.

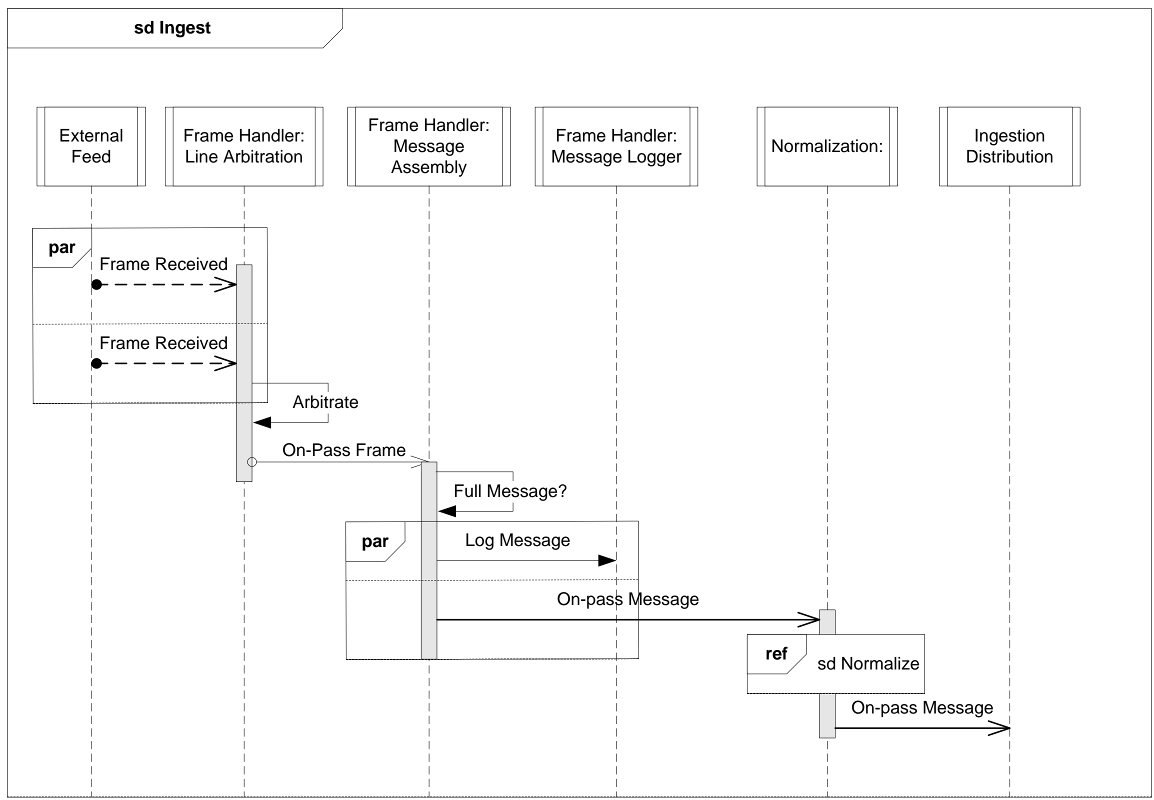


Figure 8 - Sequence Diagram

Since these method calls / messages are the Interfaces between the modules, these diagrams can be thought of as organizing the dialog in our story.

A sequence diagram should be included for every major data flow in the system. We define a Major data flow as one which is necessary to meet the business case for the system. Of course, the architect has significant leeway in deciding which are "architecturally significant" and which are not.

Care should be taken to ensure consistency across the architecture and to facilitate future diagram modifications: The Data Flow Views work in concert with the System Views (UML sequence diagrams and UML component diagrams) to tell the story. It's important that the different ways these two views tell the story be kept consistent and coherent.

In Figure 8, we show a few of the different constructs that help describe data flow. These symbols are further elaborated in Section 3: Symbols and Stereotypes. One of the more useful ones is the ref , which allows us to *black box a dialogue, so that it can be elaborated in a subsequent sequence diagram (See* Figure 9*)*

Here we can take the Normalize reference from the sequence in Figure 8 and describe with a fully elaborated sequence diagram what is really going on.

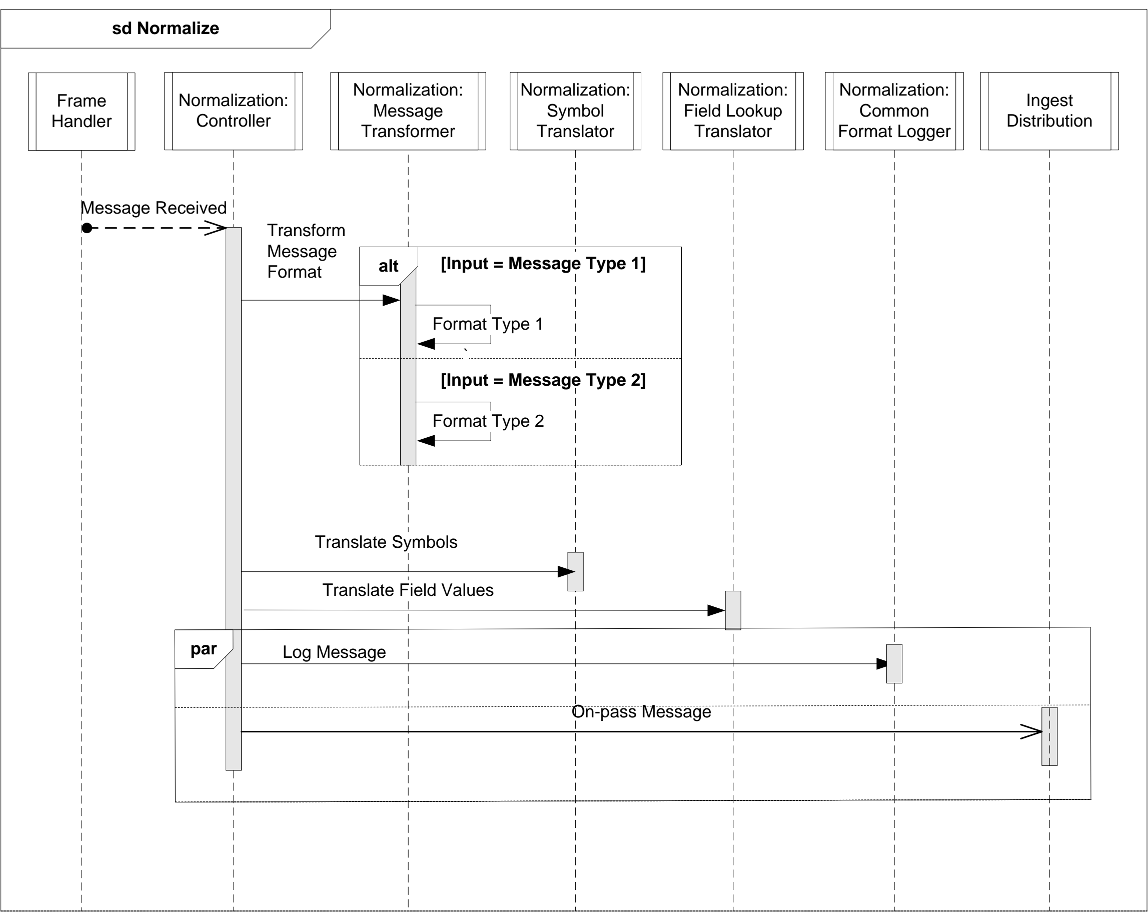


Figure 9 - Sequence Diagram Reference

2.5 The Conclusion – Deployment View

This view concludes the story by mapping the modules onto physical systems (called nodes) and defining the node name, platform, resilience model and scaling model. It should be recognized that the earlier the logical / functional architecture is done in the project process (a good thing), the more this part will be guesswork. Nevertheless it is extremely useful guesswork even if it has to be revised later on.

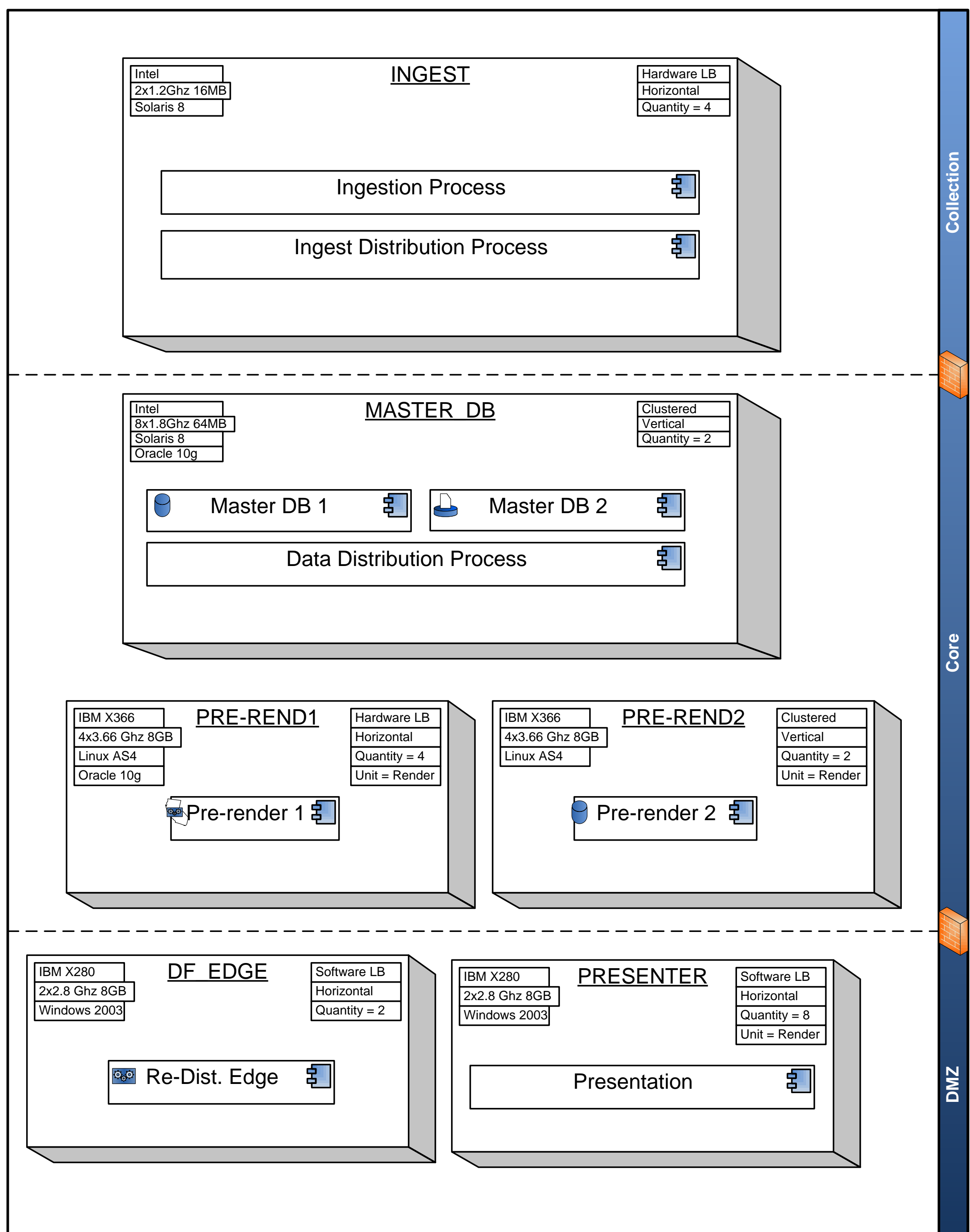


Figure 10 - Deployment View

By Platform, we mean:

- The physical hardware device - including Vendor, Model, CPU class and number, memory, disk, NICs, etc,
- The Operating system – Including Version
- Standard Vendor Software – For example Database software, ETL software, application server, middleware, etc

By Resilience model we mean, how a device failure is handled so that single device failure does not result in service failure. An initial list of models is:

- **Hardware LB**: A hardware load balancing appliance is used to distribute the load amongst multiple devices.
- **Software LB:** A software algorithm is used to distribute the load amongst multiple devices.
- **Clustered:** Multiple physical devices appear as a single device to customers of its functionality. They often use something like Virtual IP addresses and a heartbeat mechanism to detect whether a node in the cluster is healthy or not. .
- **Manual:** A manual operational procedure is required to fail over from one set of devices to another. Examples include changing configuration files or remapping DNS entries.
- **None:** there is no resilience model for the device – if it fails, there is a service outage.

By Scalability model we mean, how we adapt to increased load. It is good practice to specify the number of instances we intend to deploy as our initial scale and what the unit of scale is. An initial set of values for Scalability model is:

- **Horizontal**: To add capacity, we add more devices of the same type.
- **Vertical**: To add capacity, we increase the physical resources of the device. For example: adding CPUs, increasing disk space or I/O capacity, increasing network I/O capacity, etc.

Often devices do not act independently for resilience or scaling. Instead they operate as a *Unit*, where a group of devices fails over as one and to scale you add another *Unit,* consisting of multiple devices

It is interesting to note that the Resilience model and the scalability model are often highly correlated. For example, a web server farm may use a load balancer to share transaction load across multiple identical instances. In this case losing a single web server is handled by the load balancer reallocating transactions across the remaining devices. Handling additional load is achieved by adding another web server and having the load balancer allocate a portion of the transaction load to it.

But even though they may be solved in the same way in certain circumstances, resilience and scalability are different problems and it is important to not confuse the two.

Some of the deployment information may not be known at the time the logical architecture is produced. Placeholders should be inserted for such cases. All of the information should be known by the time the physical architecture is produced (duh!)

For the sake of clarity, following is a list of information we specifically DO NOT intend to capture in the logical / functional architecture:

- Low-level information regarding network segments, subnets etc.

- Low-level information regarding the number of interfaces on a machine
- Management software, anti-virus software etc.
- Detailed Business Continuity Strategies
- Monitoring and Backup strategies
- Detailed Security Analysis

These are reserved for views of the Physical Architecture.:

Through the Deployment View diagram we begin to understand how modules are distributed across physical machines, and how the logical message flow between modules crosses physical device boundaries and physical zone boundaries.

By placing the device nodes within network zones, we begin to infer network security, and the communication paths which must be facilitated between the nodes that communicate with each other. In this manner, we can easily identify any physical anti-patterns, and security issues, before they become issues.

We can also begin to infer initial device sizing and software licensing and so can approximate cost. Doing this early in the project cycle and doing it well is one of the major value propositions a good architecture process brings to an organization. Delivering stuff that actually works is another.

Copyright © iankoenig.com

2.6 Character "Bios" - The Glossary

The glossary provides a single, general definition of the overall system (service, solution, et al) being depicted. This definition serves as a "mission statement" for which all modules, data flows, and elements of contained diagrams should subscribe.

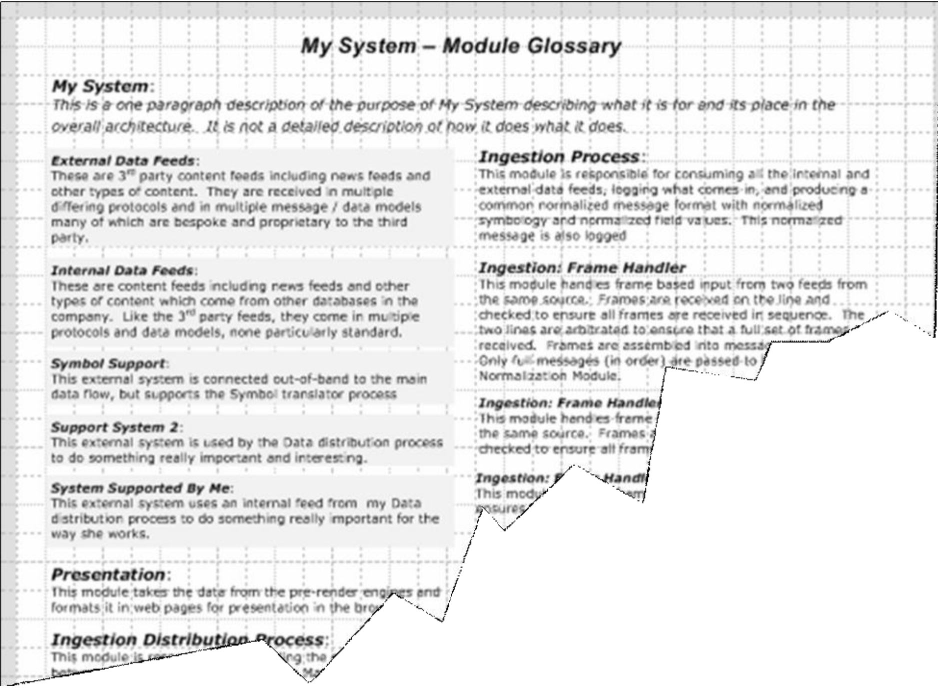
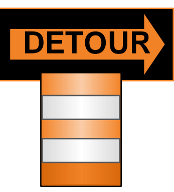


Figure 11 - Glossary

Additional terms and descriptions are provided for every module referenced in a system view. The description should be short, but descriptive enough to get across what the nodule is for.

In keeping with our metaphor of comparing production of an architecture diagram to writing a story, the Glossary represents the one paragraph "biography" for each character in the story (based on the correlation of a module to a character in the story).

2.7 The Afterward – Deviations from the Ideal



(Or what actually happened vs. what we wanted to do)

Few implementations (probably none) ever precisely match the architecture (at least not in the real world). So in order to make architecture relevant, we find ourselves with two additional challenges:

- How to preserve the architecture the way it was defined and not lose what is *the right thing to do;* noting that the best architecture is one that actually gets built. If you don't compromise that does not happen
- How to actually represent the reality of what got built and its relationship to the architecture, since we need the architecture to be real and not "ivory tower" for it to be useful and relevant.

To accomplish these seemingly contradictory goals, we define the concept of a *tactical deviation*. In other words, we try to keep the architecture as pure as possible, but note where the implementation has deviated from the architecture.

If the diagrams become so littered with deviation symbols that the underlying architecture cannot be communicated, then it is time to change the underlying architecture and compromise. Clarity is crucial in these cases.

What we are trying to avoid is the situation where those who come after us look at what we have produced and misinterpret some ugliness we were forced to do under duress release as something we intended to do. Just telling those who come after us that these things 'that make no sense' can and should be changed, is a huge help to them. We have this invented special symbols that essentially mean "We did it, but we did not like it. Not only can you change it, but you should".

There are two classes of these symbols (which are further described in Section 3.4):

- **Temporary Deviation** – Something you intend to fix, maybe in the next version
- **Permanent Deviation** – Something you have no intention of fixing but which is still architecturally wrong.

Done correctly, the architect will not have to significantly change the architecture documentation as the implementation incrementally evolves toward the architecture "vision". Only the deviations need be removed.

Some examples of deviations:

- The system was architected to use Web Services, but due to incompatibilities with client software the legacy APIs need to be maintained for a period of time.
- The strategic authorization system was not ready in time for implementation, so a tactical module based on an operationally maintained file was put in place instead.

Copyright © iankoenig.com

2.8 Additional Diagrams

Additional diagrams from the UML set may be added, where appropriate to augment the minimum set recommended. UML in general tends toward Model driven architecture (MDA) so has many other styles of diagrams available to represent more detailed design constructs. In an organization that uses UML for design, the continuum from architecture to design to implementation and testing can be quite powerful. These additional diagrams may or may not add value to the architecture process. It really depends on the culture of the organization and the process implemented. They are presented here for the sake of completeness.

2.8.1 Use Case Diagrams

Use cases are a way of specifying function al requirements in a manner that favors specifying the actual problem that is to be solved rather than a pre-supposed solution. A Use case diagram is a UML construct that defines a use case through persons or things responsible for invoking certain functionality via implementation elements (*actors* and *subjects)*

2.8.2 Activity Diagrams

UML Activity Diagrams focus on Process Flow rather than system construction or data flow. They are extremely useful where the business logic needs to be represented in what amounts to a flow chart.

2.8.3 Data Model Diagram

UML uses Classes and Relationships to represent data models, but it is common to use a more traditional Entity Relationship Diagram (ERD). In general we do not want to get too deep into internal data models in the system architecture, but the data models of exposed Interfaces are absolutely critical to understanding the architecture and must be defined. How they are defined is of less importance than the fact that it is defined in a way that is understandable by the reader.

Data Model diagrams are entity-relationship diagrams meant to represent the data entities, their identifying attributes, and the relationships they maintain within the information architecture.

An Entity relationship is a natural association that exists between one or more entities. It is important that cardinality be included in a relationship definition, detailing the number of occurrences of one entity for a single occurrence of the related entity. Typical relationship types include {“Has”, “Extracts”, “Is Assigned”, “Manages” etc.}. Relationships are represented by elbowed lines.

Each entity maintains data attributes which are characteristics of an instance of a particular entity. An attribute or combination of attributes that uniquely identifies one and only one instance of an entity is called a primary key. It is important to note the primary key (and composite elements that compose the primary key) in the data model diagram. An entity attribute that is a primary key of another entity is called a foreign key and is depicted within a data model diagram with bold text.

The following example outlines the major elements of a data model diagram:

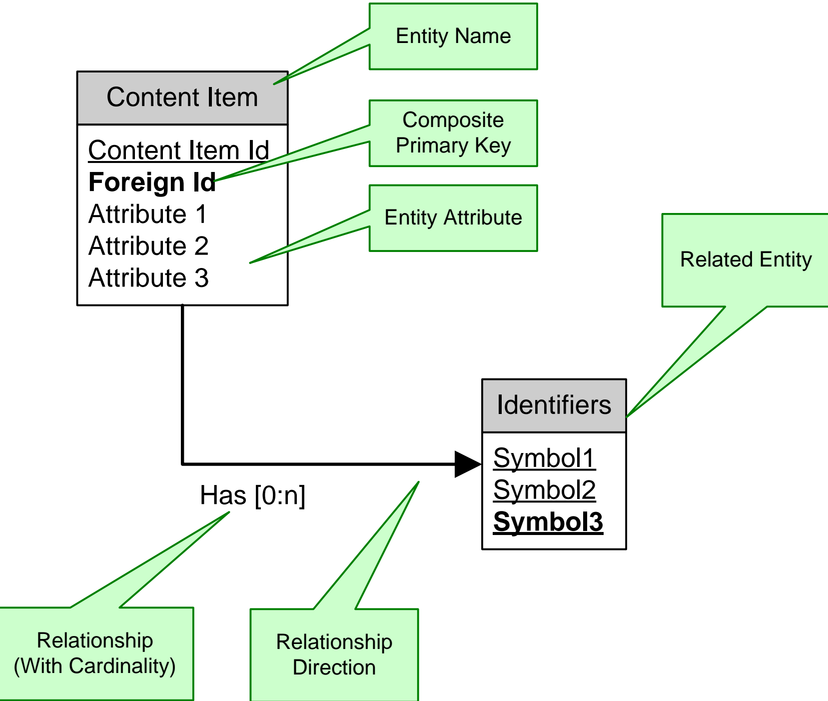


Figure 12 - Data Model Diagram

3 Symbols and Stereotypes

Symbols are used in diagrams to convey the existence of things or objects. Stereotypes are used to indicate class or behavior. The Symbols presented below are organized by the type of diagram they are intended to be used in.

3.1 System View Diagram Symbols

Symbol Name	Illustration	Description
Module		Uses a UML Component symbol to denote a module. A stereotype icon is placed in the upper right corner to denote type (or the generic component stereotype used where a more specific stereotype does not exist)..
Multi-Module		Denotes multiple modules of the same type.
Zone separator		Separates groups of modules into clear groupings (often based on an architecture pattern or physical isolation to support *separation of concerns*.
Zone Demarcation		As an alternative to individual zone separators, using the zone demarcation allows zones to be clearly named and related.
Interface symbols		
Interface		Interface (API) between two modules. Technically there are two sides, the provider side and the consumer side. By representing both together it is intended to define the whole contract. The letter inside the *ball* denotes the type of interface (See: Interface Descriptors)
Push Interface		This interface between two modules implies that once the connection is made, information is pushed from the provider to the consumer, without the consumer making specific individual requests. In general, the consumer is still required to initiate the connection.
Multi-Interface		Set of Interfaces that will be broken out and elaborated in subsequent decomposition.

Port (Interface) Name	Interface 1	The Interface Name (i.e. the Port name or name of the API).
Payload Description (Data Model)	Output Model 1	Provides additional information for an interface (See Service Interface component). Interface description contains the Protocol / Encoding / Data-model combination.

3.2 Data Flow Diagram Symbols

Symbol Name	Illustration	Description
Interaction	sd Interaction	The Interaction defines the communication between a set of modules (represented as participants). The Interaction is named and the participants are placed along the top with their lifelines extending down (potentially to the bottom. Messages flow between the participants in the interaction.
Participant		Denotes the module responsible for the actions performed on its lifeline (the dotted line hanging down). These are placed at the topmost portion of a sequence diagram.
Referenced Sequence	ref	Denotes a linked sequence diagram. Useful for indicating a very high-level view of a system, and then drilling down for more detail where space is limited.
Parallel, fragment	par Log Message On-pass Message	Used to frame a set of calls executed at the same time (in parallel). Horizontal swim-lanes are placed within the frame to denote the parallel paths.
Conditional	alt [Input = Message Type 1] Format Type 1 [Input = Message Type 2] Format Type 2	Used to describe when back-and-forth communication between two services is occurring. As an example, this may illustrate the general protocol for handshakes and protocol-level requests/acknowledgements. At the top of each swim lane is the guard condition specifying when that lane is invoked. For example, “X equals Y” is a

Symbol Name	Illustration	Description
		valid condition, but “X” is not.
Activation		Illustrates the duration of the participant’s activation / lifespan.
Found Message		Indicates presence of a new/dynamic item, and is most appropriate for usage in message queue scenarios.
Asynchronous call		Open Arrow denotes an asynchronous method call.
Method call		Closed Arrow denotes synchronous method call expecting control returns to the caller after completion of the task.
Message return	(Return object)	Optional notation, to be used when the method invoked returns an object or message that needs to be specified on the return (otherwise return is implied by the call)
Message to self		Denotes a call to the same object instance (generally used to show modularization in large processes).

Copyright © iankoenig.com

3.3 Deployment View Symbols

Symbol Name	Illustration	Description
Zone Demarcation		Provides the deployment skeleton for which to place nodes within zones. Each horizontal line denotes a different *mandatory* network zone, modeling your physical architecture.
Node		Illustrates a physical machine (or what you estimate will be a physical machine at the end of the logical architecture process). We then define the physical properties of the node and which logical modules will be deployed on the node.
Module		Illustrates a module deployed onto a node. These are the exact same modules we defined in the system view(s).

3.4 Stereotypes

Stereotypes should be placed in the top right-hand corner of the module to indicate its generic purpose. Stereotypes are relatively small in size and the list may be extended over time as there are clear behaviors that can be concisely represented through the use of an icon.

Symbol Name	Icon	Description
Component		This is the standard UML component stereotype indicating a module, but no other behavior is indicated.
Firewall		This stereotype (illustrated as a firewall) is applied in deployment views to indicate that a security checkpoint exists at the indicated zone / boundary.

3.5 Module Behavior Stereotypes

We use the UML component shape to indicate an encapsulation of business logic. To add some additional definition to the shape, the base UML shape is extended by adding a behavior stereotype. These are by no means intended to be a definitive closed list. Instead, the set should be extended as necessary.

Stateless Business Logic		A module that performs *business logic* in a Stateless way
Stateful Business Logic.		A module that performs *business logic* in a Stateful way
Database		A module that uses a Database (e.g. a SQL Database) to perform its task. .
File Server		A module that uses a file system to perform its task.
Directory		A module that uses a Directory (e.g. an LDAP store) to perform its task.
Logging Service	Logger	A module that performs the function of logging results and persisting them for a period of time
Shared Library		This stereotype is applied to a Module to indicate that it is a shared library. In other words, a container of multiple modules providing interfaces that are used across systems and / or sub-systems.
Plug-in		This stereotype indicates that the module is an extension to some "standard" framework that extends or specializes its logic. For example, an ISAPI Plug-in (extension)

Web Framework		This stereotype indicates that the module is the framework for a web server.
Cache		This stereotype indicates that the module implements a cache.

3.6 Architectural Deviations

The following symbols identify any areas of the architecture that tactically deviate from the desired architecture or from the organization's architectural standards. One of the following icons should be placed in a lower corner of each deviant module to illustrate a tactical deviation:

Deviation Icon	Description
	Indicates a deviation that is intended to be revised in order to adhere to architectural standards in the component's near-term future.
DETOUR	Indicates a component deviation that is expected to persist for the duration of that component's existence.

It is important to note that all modules annotated with either of the deviation stereotypes, should be further elaborated in specific views to clarify what was actually done as opposed to what was architected. The architecture should not be *polluted* by the compromises that had to be made during implementation, but neither can the architecture lose its grip on reality. The kind of *ivory tower* architecture that ignores what was actually built is of no real use to an organization.

3.7 Interface Descriptors

When the Ball and socket symbol (X) is used to denote an interface between two modules, we use a letter inside the ball to represent the type of interface. Where we are using standards, such as web services, other attributes of the interface are automatically implies, alleviating us of the responsibility of re-specifying the particulars over and over again. This can cut down on diagrammatic sprawl.

The following table is a pretty good start, but should be extended, particularly where an enterprise adopts particular industry standards are defines its own and uses them widely and consistently.

Layer	Mnemonic	Description	Definition
4	K	Raw Soc**k**et	A TCP socket on a specified TCP port.
5	F	**F**TP	File Transfer Protocol on TCP/IP
5	H	**H**TTP(s) post/get	Hyper Text Transfer Protocol on TCP/IP
5	L	**L**DAP	Lightweight Directory Access Protocol on TCP/IP
5	S	**S**MTP	Simple Mail Transfer Protocol on TCP/IP
5	N	**N**FS	Network file share protocol on TCP/IP
6	Q	S**Q**L query	Structured Query Language on Database specific Layer 5 protocol including access to stored procedures

6	W	**W**eb-service (RPC-style)	XML Web Service (RPC style) with SOAP on HTTP on TCP/IP unless otherwise specified
6	T	RES**T** style Interface	XML payloads transferred over HTTP without SOAP
7	R	**R**SS	Really Simple Syndication Protocol on HTTP with XML
7	E	**E**TL	Extract / Transform / Load – application / vendor specific protocols
0	A	Native **A**PI	Native API (method call / return) either in-process or out-of-process, but not crossing a device boundary
0	O	**O**ff-Line	A non connected interface (such as a manual / operational procedure that copies a file from one location to another) – used to indicate that while not connected by wires, systems are still electronically dependent on each other.
0	...	**V**arious	Indicates multiple Interfaces of different types (that will be broken out in later more detailed diagrams)

Copyright © iankoenig.com

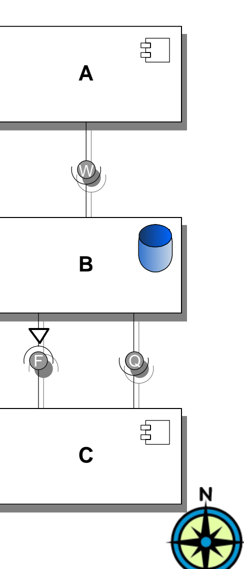
4 Effective Diagramming

In addition to the stylistic guidelines listed in section: 1.6 - Diagram Guidelines on page: 6, effective diagramming builds on these discrete concepts to include *diagramming patterns.* By *diagramming pattern* we mean the consistent use of combinations of elements to denote more complex constructs.

In the same way software elements can be aggregated into more complex units, so to can diagramming elements be aggregated in standard ways to represent more complex concepts. Following are some examples, which are by no means exhaustive.

4.1 Interface Pattern

The placement of the Interface symbol and Service Consumer symbol convey data direction and a push or pull as illustrated below:



- Module A exposes a single Interface (a Web Service) that Module B consumes. Since Module B is the consumer (denoted by it having the socket side of the interface) and noting there is no streaming stereotype (indicating push), Module B is responsible for pulling information from Module A. If the ball and socket were reversed, it would indicate that Module A was making the call, and handing the data to Module B. North-to-south orientation indicates data is flowing from A to B.
- Module B exposes a single SQL Query Interface consumed by Module C. This is the ‘Q’ interface at the right hand side). Module C is the consumer. Since data flows from North to South, Module C is pulling data.
- Module C exposes a single FTP interface consumed by Module B. Module B is using this FTP interface to push data to Module C based on presence of the push interface stereotype (the triangle pointing from B to C)
- Module A, Module B and Module C each expose a single Interface.

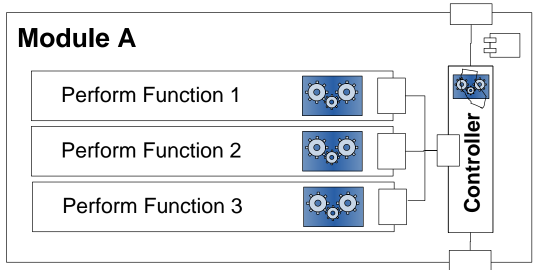
4.2 Shared Component Library

A shared component library is a collection of common functionality that is not exclusive to any one particular system, but is rather a modular piece of code that provides a level of reuse. It is therefore a useful pattern to represent consistently. Typically, these consist of .NET assemblies or Java archives and are employed by multiple systems or modules.

We depict a shared library using:

- A single module with a shared library stereotype
- A single Multi-Interface indicating that the shared library container is exposing the individual interfaces of the contained modules (Java Archives, .Net assemblies, etc)
- A set of contained Modules indicating the implementations of the exposed Interfaces.

4.3 Controller Pattern



A common processing pattern (especially when handling content) exists where one module acts to control the actions of other modules generally maintaining state, as a set of functions are performed. For example, in processing an incoming message, a common controller may decide based on the contents of the message, which sub-processes need to be performed in order to re-format / normalize the message. In the example to the right, Module A has a controller process that handles the input to module A, maintains state and decides (based on its internal business logic) which of Functions 1, 2 and 3 to perform and in which order. At some point, the controller decides when to on-pass data, which is indicated by its connection to Module A's south-end port.

.

5 Using Visio

Microsoft Visio, while possibly not the best tool to create these diagrams, is certainly a good tool, especially when paired with smart shapes and templates that are designed for the task.

5.1 Shapes and Stencils

There are several sources for these shapes:

- Microsoft includes a very robust set of shapes with Visio, including several UML stencils and templates. These are however primarily based on UML v1.0 and primarily targeted to the design process. They store data about the objects in the sheets and are really meant to be used as a whole. They do not seem to work all that well when you just want one or two shapes and don't start with the included templates.
- Pavel Hruby provides a free download of UML v2.0 stencils and templates for multiple versions of Visio at http://softwarestencils.com/uml/index.html(Hruby).
- Paul Herber provides UML / SDL shapes & stencils in both free and licensed form at: http://www.sdl.sandrila.co.uk/
- Another free source of Visio shapes can be found at http://pro.iankoenig.com/visio.php. These are the shapes used in all the diagrams in this guideline and the ones specifically tuned for the purpose of architecture rendering.

Once the appropriate stencils have been downloaded (usually into your "Documents\My Shapes" directory) and opened (within Visio – File/Shapes/Open Stencil): creating diagrams begin by dragging shapes off the stencil and dropping onto sheets.

The following assumes the use of the pro.iankoenig.com shapes.

Microsoft Visio uses a tab metaphor for navigating between pages. Each Page is given a tab at the bottom of the drawing window. We use one Visio page per architecture view, starting with the perspective view and working progressively toward the deployment view and glossary.

Each of the symbols illustrated originate from the pro.iankoenig.com Visio 2003/2007 Stencils for UML 2.0. Leverage these shapes to start. If you need to create new shapes, simple shapes can be created by grouping together other shapes and adding to a master stencil. To get more complex behavior, you have to go "inside" the shape and do some programming. While not to be taken lightly, Visio shapes are controlled by a shape sheet that defines their behavior. It's about a week or two of effort to understand how to do this and then time based on the complexity of the task to implement the shape. Some good sources are:

- Microsoft Visio 2003 Developer's Survival Pack by Graham Wideman (www.diagramantics.com)(Wideman, 2003)
- Visio Guy (www.visguy.com)(Visio Guy)

5.2 Connections

Modules are connected to each other via Interfaces. There are two styles of Interface shape provided, the straight interface connector and the dynamic connector. The straight connector is as the name implies: straight. It will angle itself to connect shapes at different points on the page. The dynamic connector, on the other hand uses three “leg” connectors on each side, so that it can more easily adapt to different orientations of shapes. Both are available depending on the circumstance.

When a single interface needs to be connected to multiple locations, we use the Visio provided Multi-tree square connector in concert with the interface connector. Connecting modules to ports and ports to port is done using Visio’s dynamic connector. . In general connectors should connect to connection points on the target shape. The included shapes try to predict the most likely place where connection points are needed, but it is unlikely this can be done properly.

To adjust, use the Visio Connection Point Tool , both to add connection points where needed (by selecting the target object and holding the Ctrl key) and to remove ones not needed (by selecting a connection point and pressing Delete).

5.3 Zooming

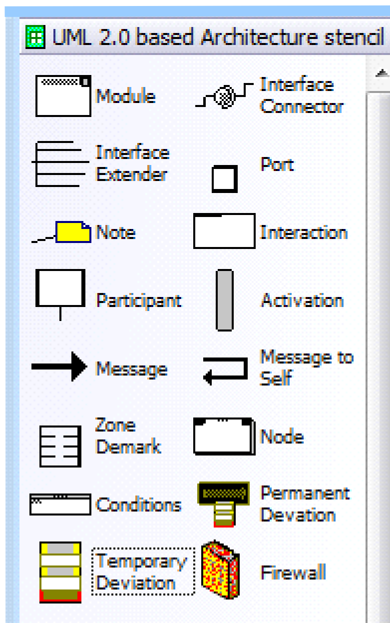
Visio provides a Zoom feature that is often useful in order to create or edit diagrams. Higher Zoom levels allow greater precision in manipulating objects.
Zoom is available from the View menu, on the toolbar or via the Pan & Zoom Window (View/Pan & Zoom Window)

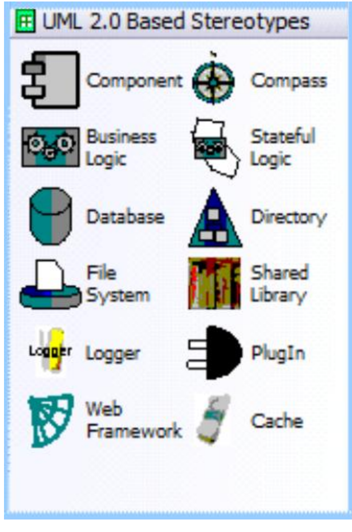
5.4 Aligning Objects

By selecting multiple shapes and selecting Shapes / Align Shapes on the menu, the shapes can be aligned based on an edge or based on the mid-point. Alternatively, using the Size & Position Widow, the X-position, Y-position, width and height of each shape can be set independently.

5.5 pro.Iankoenig.com stencils

On http://pro.iankoenig.com/visio.php there are two stencils, one for shapes and one that just has the stereotypes. In general, the stereotype stencil is not needed since these are all part-of the module shape already


UML 2.0 based Architecture stencil
Module
Interface Connector
Interface Extender
Port
Note
Interaction
Participant
Activation
Message
Message to Self
Zone Demark
Node
Conditions
Permanent Devation
Temporary Deviation
Firewall


UML 2.0 Based Stereotypes
Component
Compass
Business Logic
Stateful Logic
Database
Directory
File System
Shared Library
Logger
Logger
PlugIn
Web Framework
Cache

Following is description of the shapes provided:

Shapes for System Views		
Symbol Name	**Illustration**	**Description**
Module		Used for either a Module or Multi-Module. Right click on the shape to access the context menu to set: • Module Behavior (also set via smart tag) • Module Description • Single / Multi Module (single is default) Modules names are set by pressing F2 and entering text. In future versions, the Module Description will be 'scraped' to auto-create the glossary.
Port		Denotes the port via which Interface are exposed from modules. Ports may be named (press F2 to enter text). Text may be positioned via the control (yellow pin). There are two sizes (small, large) that have no prescribed meaning other than for aesthetics.
Interface - Connector	M	A connector to be used between ports (on modules) representing an Interface contract. Right click on the shape to access the context menu to set: • Interface Descriptor (See Interface Descriptors) <also set via smart tag> • Multi-Interface (single is the default) • Push Interface (Request/Reply is the default) Interface connectors need to be oriented using flip / rotate menus so that the provider side and consumer side of the interface are correct (provider side gets the ball; consumer side gets the socket). A dynamic connector that is functionally identical to the straight Interface, except that the legs of the interface have a right angle joint so as to keep everything at right angles. It is particularly useful to maintain aesthetics when the connected modules cannot be aligned.
Note	Output Model 1	Attach to Interfaces to add data model definition. Attach to anything else to add additional descriptive info. Visio comes with a ton of other notes / callouts that are just as good if not better than this one.

Interface Extender		This is one of the standard connectors provided in the Visio Connectors stencil (renamed). It is very useful when an Interface needs to connect to multiple ends. You attach one end of the tree-square to the Interface and the other ends (yellow control points) to the ports of the modules being connected. Visio has many such connectors, This is but one option.
Shapes for Sequence Diagrams		
Symbol Name	**Illustration**	**Description**
Interaction	sd Interaction	The Interaction shape represents a number of different elements in the Sequence diagram all with a single shape including: • The Interaction • Referenced Sequence (ref) • Parallel fragment (par) • Conditional (alt) The shape may be filled in or not (right-click menu). The text in the upper left corner denotes the type of Interaction A control on the bottom may be used to drag out guard lines (horizontal dashed lines) that separate parallel operations or conditionals
Participant		Denotes the module responsible for the actions performed on its lifeline (the dotted line hanging down). These are placed along the top of the Interaction shape in the sequence diagram.(usually in a single row).
Activation		Illustrates the duration of the participant's activation / lifespan along the lifeline as it reacts to messages.
Message		Multiple message types between activation areas of Participants. The right click menu allows selection between: • Found Message (•⟶) – This is a message that was unsolicited. • Asynchronous Call (⟶) – A one-way call with no expected return. • Method Call (⟶) - A synchronous call where the caller waits for the return.

		• Message Return (⟵) – An explicit return from a call
Message to self		Denotes a call to the same object instance (generally used to show modularization in large processes).
Shapes for Deployment Diagrams		
Symbol Name	**Illustration**	**Description**
Zone Demarcation		The Zone Demarcation shape is used as the Deployment skeleton or in system views where zone demarcation is necessary. There is a yellow control along the bottom edge allowing up to six zones to be defined. Click one of the zone separators until it alone is selected and press F2 to change the zone text (displayed along the right edge).
Node		Illustrates a physical machine. Press F2 to set the Node Name. Right Click to Configure the Node. This displays the shape data dialog allowing the following node attributes to be set: • Resilience Model • Scalability Model • Quantity of Nodes (Initial) • Unit (Node Group for resilience / scaling) • Hardware • CPU + Memory • Operating System • Platform Software
Firewall		Use this stereotype at the appropriate zone demarcations to show which logical boundaries are also physical boundaries.
Temporary Deviation		Indicates a deviation that is intended to be revised in order to adhere to architectural standards in the component's near-term future.
Permanent Deviation		Indicates a component deviation that is expected to persist for the duration of that component's existence.

The shapes in the Stereotype stencil are included for the sake of completeness. They are all members of the Module shape and accessed simply by setting the Module's behavior.

Table of Figures

Figure 1 - 3-D projection....2
Figure 2 - UML Diagram Model....3
Figure 3 - Matryoshka Doll....5
Figure 4 - Compass Direction....7
Figure 5 - Perspective View....10
Figure 6 - System View 1....13
Figure 7 - System View n....15
Figure 8 - Sequence Diagram....17
Figure 9 - Sequence Diagram Reference....18
Figure 10 - Deployment View....19
Figure 11 - Glossary....22
Figure 12 - Data Model Diagram....25

Bibliography

(n.d.). Retrieved from Visio Guy: www.visguy.com

Hruby, P. (n.d.). *Visio Stencil and Template for UMNL 2.0*. Retrieved from softwarestencils.com: http://softwarestencils.com/uml/index.html

http://en.wikipedia.org/wiki/Feynman_diagram. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Feynman_diagram

Pilone, D., & Pitman, N. (2005). *UML 2.0 in a nutshell.* O'Reilly Media, Inc.

Wideman, G. (2003). *Visio 2003 Developer's Survival Pack.* Trafford Publishing.

Index

Architecture Story
Characters, 9
Conclusion, 9
Dialogue, 9
Plot, 9
Theme, 9
Architecture Story
Style, 9
black-box view, 4
Business Process Architecture, 2
compass icon, 7
decomposition, 5
Functional Architecture, 1
Information Architecture, 1
Interface, 8
Module, 8, 14, 15, 26, 29, 30, 33, 34, 37
perspective view, 10
Physical Architecture, 2
***Port Name*, 12**
Russian Matryoshka Doll., 5
Setting, 9
symbols and stereotypes, 6, 12
System, 8
***System View 1*, 13**
***tactical deviation*, 23**
telling the story, 9
UML, 1, 2, 3, 4, 5, 6, 10, 11, 17, 24, 26, 30, 35
white-box view, 4, 6