1.1 Abstract
This paper defines software symmetry (symmetry in software architecture in particular), presents a taxonomy of symmetries in software architecture and describes a set of patterns that identify and preserve symmetry in software architecture. The running example will center around n-tier distributed software architectures that are typically observed within the context of e-business systems.
Each pattern’s solution section contains an element of what may be described as “symmetry”. The categories of software symmetry are: static; when the system architecture is being designed and built or is not running, thus being able to change configurations, nodes and make choices on deployment options and optimization points, warm-up; when the system is starting up, configuring itself or initializing, dynamic; when an application(s) is running, adaptive; when the system is being re-configured, or changed to accommodate new rules, workflow, requirements; maintained for new dynamic collaborations usually without having to bring the running system “down”).

We contend that proper symmetry in software architecture contributes to achieving excellent levels of non-functional and functional requirements that would be more difficult to attain without their use and provide case studies and rationale for this.

1.2 Background

To me, these seemed to be obvious examples of locating symmetry in software architecture and the answer to the hot topic of “okay, you can find symmetry in a Persian rug, or in repeated patterns of designs in building architecture, or even (“that may be stretching it” you may think) in spatial analysis of “lines of code and indentation”, but where is the “symmetry” that is a sign of QWAN (Quality Without a Name) in software architecture? And once found, will it help me solve my day-to-day problems as a software architect? Will it give guidance to less experienced architects and allow the creation of long-lasting, robust and scalable software architectures?”

Coplien points out that the architectural metaphor needs to be readdressed [9]. Shaw, Garlan, Clements and others have made progress in defining this often ethereal discipline [10] [11] [12] [13].

To that problem and that context, I have written up this solution; which resolves some of the forces such as “symmetry being immediately identifiable” or “pragmatism” in finding symmetry.

1.3 Introduction

“Lots of the discussion of symmetry in computer systems design has been very ungrounded, but these patterns are attempting to apply them to a very concrete and important domain. [Noble b]”

In this pattern language we start by defining types of symmetry then showing how they can be applied in systems.

1.4 The Definition of Symmetry in Software Architecture

Realizing that this is just the tip of an iceberg of research and practice, we need to lay down some foundations and define some terms based on a taxonomy. As a preliminary step, we attempt to categorize the types of symmetry in software architecture in terms of a taxonomy we can later reference.

Reevaluating The Architectural Metaphor: Toward Piecemeal Growth, IEEE Software Vol. 16, No. 5, September/October 1999. The essence of the definition relates symmetry with the ability of a software structure to maintain stability along an axis of symmetry subsequent to transformation. Thus, “invariance to change” is a concise definition of symmetry. What are the qualities or axes of symmetry in software architecture that we would like to keep ideally immutable or practically, stable?

This is the balance between the functional and non-functional requirements of a living operational software architecture that is withstanding change coming in the forms of functional changes to requirements and business rules as well as changes in non-functional aspects such as increases in usage volume (increase in “number of hits”), security threats, translations and mappings to new formats, the need to integrate and retrieve information from multiple disparate data sources and message sources.

Every change introduces a degree of entropy into the software architecture; until it breaks. The degree to which a software architecture has accounted for change points (along its axes of symmetry), based on variation-oriented design Arsanjani, A., “Dynamic Configuration and Collaboration of Components with Self-Description”, submission to OOPSLA 2001. and can remain stable in the face of changes to functional and non-functional aspects; the system is said to be stable and symmetric. The inability of a brittle software architecture to cope with change results in asymmetry and instability—the system “goes down”, and changes cannot be easily incorporated (such as addition or modification of new products or services, or customization of a new business line or product line.)

According to physics, balanced symmetry-breaking produces the material world. The Unified Field’s super-symmetry is broken and particles and fields start to appear. Much like nodes and processes running on them. The node is the particle and the process is the field in this analogy.

Although analogy is not a formalism, it helps us understand and gain insight into the activities which must be conducted to produce lasting software edifices.

### 1.5 The Patterns of Symmetry in Software Architecture

There are fourteen patterns in this set, of which we will cover seven in this paper: Fractal MVC, Two-way Mapping (Bi-directional Transformation), Mapping Layer, “Ten” Layers, Multiple Tier Map, Configurable Workflow, Configurable Profile, Cache and Hash, Wrap and Map, Rule/Exception Pair, Contract (pre- and post) and Pair Message Queues, Balance Functional and Non-functional.

The following pattern language map paints a picture of the relations between the patterns.

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry-preserving Transformation</td>
<td>How do we ensure that the mappings from design to implementation will not imbalance our software architecture?</td>
<td>Ensure the presence of a component within each tier (logical and physical) that provides a two-way symmetry-preserving transformation back and forth across the tier, thus preserving architectural symmetry.</td>
</tr>
<tr>
<td>Fractal MVC</td>
<td>How do you perform separation of concerns and partition, functionally separate aspects, of a subsystem running on a given tier?</td>
<td>Look inside each level granularity (e.g., entities cohesively clustered) and apply the separation of roles into MVC at three levels of granularity: large-grained, medium-grained and fine-grained: e.g., workflow across large enterprise components, routing, reified collaboration and transport across and between middle-level components and finally, business rules for each individual fine-grained business object.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Two-way Mapping; Bi-directional Transformation</td>
<td>How do you ensure that a transformation or mapping from one tier to the next leaves the overall system integrity intact and in a consistent state?</td>
<td>Ensure that each axis of symmetry (where a transformation can apply to) holds a two-way mapping capability (can serialize, de-serialize, can map strings to objects and back, can map object to tables and back, can map encrypted data to a decryption algorithm and vice versa, can translate into XML and back, etc.)</td>
</tr>
<tr>
<td>Capture the manners; Components have Manners</td>
<td>A tier has a 2-way mapping and can thus accept and transmit data between tiers. What about preserving symmetry within a tier? How should the tier behave under various contexts and events? How should the tier govern the interaction between disparate components?</td>
<td>Capture the Manners. Define the rules governing the behavior of a component within a given execution context that will supply it with meta-data, events and context. Send its rules of behavior (manners) along with a context to the component [6]. This can be as simple as sending an object, or as complex as using Grammar-oriented Object design [6] to define a new configuration and collaboration for the tier or layer or component.</td>
</tr>
<tr>
<td>Mapping Layer</td>
<td>You want to preserve the semantics of a compound data element as it is transformed across multiple tiers allowing each tier to perform its own processing of data elements and messages, often represented in a common enterprise format. How can you guarantee that each tier maintains the semantic integrity of the component as it is mutated from GUI to DB across all layers?</td>
<td>Therefore, include a Mapping Layer between major architectural tiers; acting as gatekeepers for transforming data and messages from one format or context to another. Take care to preserve symmetry and have a corresponding Mapping Layer on the receiving end to re-transform, decrypt, resolve, route, consolidate the returning data or messages coming back to the application tier initiating the transformation. Input objects and data from one source tier are mapped into an Enterprise format, or formats, which will allow the continuation of processing in the recipient tier.</td>
</tr>
<tr>
<td>Ten Layers; Mille Feulle</td>
<td>You are trying to determine what layers and functional responsibilities per layer to include in your enterprise architecture and how to ultimately structure the component architecture.</td>
<td>You see trivial solutions called three-tier architectures. But you end up running into problems arising primarily from the lack of organized mapping between layers: how information gets routed and</td>
</tr>
</tbody>
</table>

---

1[1] Akin to radial symmetry in geometry, where the component manners acts as the center of symmetry and a controller for preserving the message-passing symmetry between components.
transformed; mapped from one layer to another and processed, only to find that it has to retrace its steps back through the patches of the forest of tiers.

**Therefore**, use "MilleFeulle" or "Ten" Layers. Eliminate the layers that are not applicable to your solution. Be aware that you must have Mapping Layers and create and manage them accordingly.

<table>
<thead>
<tr>
<th>Business-driven breadth-first</th>
<th>How do you balance the need for functionality with the guarantee that the end product, once rolled out will actually perform and scale based on the non-functional requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Therefore</strong>, strike a balance between the depth of functionality (low-depth) and the extent of architecture exercised end-to-end. Define a thin-slice of functionality that will be used to exercise the entire operational architecture end-to-end. The functional requirements should be business driven and contribute to the fulfillment of Iteration One which is the first extensible prototype (not throw-away) that will be demonstrated to management.</td>
</tr>
</tbody>
</table>

Figure 1: Pattern Language Map

Three of the important ones are mentioned here: **Two-way Mapping**, Mapping Layer and Fractal MVC. Two-way mapping is a generalization of how we generally tackle the problem of impedance mismatches such as those object-relational mapping, GUI to XML and GUI to Object (mid-tier) mapping. Mapping layer is closely related to **Two-way Mapping** (Bi-directional Transformation). Fractal MVC arises from the observation that we can find the participants and roles [Riehle99] of Model, View and Controller not only in the user-interface (presentation) layer, but also at multiple levels of detail; within tiers and across tiers in a software architecture.

1.5.1 Context and Scope

Architectural Styles might seem to be a natural starting point for this discussion. Instead of trying to find the symmetry within each architectural style, the focus of this paper is the web-based n-tier architectural style as the context of the running example.

Software architecture concerns itself with taking a set of rigid and inanimate pieces of hardware along with a myriad of ethereal software artifacts and weaving them into a practically useful and “well-functioning” application: to paraphrase Barry Boehm’s definition of software engineering, software that is “of use and benefit to human life.”

The degree of success in achieving functional and non-functional requirements of a project is the determining factor in the degree of utility and wholeness it brings the

---

2[^2] *Mille* literally means “thousand” in French. Here the connotation is “many” and we have taken poetic license to limit this to the magic number Ten.
human end users. As such, good architects are hard to come by and are scarce and far between.

Good architects apply the techniques of caching, minimizing network chatter, pooling connections and frequently used and contended resources, placing rules in the appropriate tier, using a load-balancer and a slew of other magic tricks to achieve reliable fail-over, recovery, redundancy, security, scalability, overall system reliability and extensibility.

In observing their work and results they successfully and consistently produce, we can identify certain patterns of behavior across most projects. These are patterns that govern the “architecting” of well-performing and scalable systems that bring useful function to human life in a holistic sense (not just a toy or a gimmick for a special case but a tool of larger and deeper utility; for example the ability to purchase items reliably from the comfort of their homes).

1.6 Motivation for Identifying and Categorizing Symmetry

Symmetry in software architecture concerns the balancing of functional and non-functional forces to design a system that is useful to humans, increases their sense of wholeness and well-being and provides services that will aid them in conducting day-to-day activities with less stress and greater ease. Intuitively, asymmetrical systems tend to fail, because they lack the balance that is associated with stability.

1.6.1 Stability, Balance and Symmetry

Symmetry is centered on the notion of invariance with respect to change. One of the most important notions in software symmetry is that of an axis of symmetry. This is the answer to the question “symmetrical with respect to what?” When a point on a geometrical plane is translated (moved without rotation) it is done so with respect to a given coordinate system. Transformations are symmetrical when they preserve something. If you asked Scottie to beam you up to the Starship, your molecular structure would be partitioned, transported and hopefully reconstituted in an identical fashion. Thus, the transformation has conserved the mutual spatial alignments between your molecules. The process has left the mutual relationships between your molecules unchanged.

It can be argued that software architecture is about balancing conflicting constraints. These often arise in the context of functional versus non-functional requirements and carry on into each one of them: non-functional requirements are met by balancing asymmetries. For example, using Entity Beans to access back-end enterprise databases can be a good choice if the application is not producing undue distributed calls to fetch and re-fetch (often inadvertently)

The balance of conflicting functional requirements, conflicting non-functional requirements and the resolution of the tension between functional and non-functional requirements is the essence of successful software architecture.

As patterns generate architectures [1], patterns resolve forces in the problem domain, potentially unbalancing others. Thus, additional patterns may need to be applied to resolve those newly created imbalanced forces. This process is one of introducing symmetry.
Example: Enterprise Java Beans and Symmetry
Using session beans to serve as a façade for entity beans and other session beans is a pattern because it preserves symmetry. If every class is defined as an entity bean, the hops to the back-end will imbalance the need for a timely and robust response. Thus it will be prohibitively expensive. In order to reduce network hops we do not use entity beans all the time. But the rationale can be tested if we test to see if symmetry is being preserved.

1.7 Categories of Symmetry
The categories help us understand what type of symmetry to look for at each stage in the creation and evolution of software architecture.

There are five basic categories: static, configuration, startup, dynamic and adaptive. When you are designing the conceptual, logical architecture, you need to test it for static symmetry. No part should violate static symmetry. Next, we have configuration symmetry when we are constructing the physical architecture and how the network pieces fit together. Once a reasonable set of transformations have been tested to conform to configuration symmetry, the system is executed in a system test. This system test includes testing for startup symmetry. System startup includes the determination of what to cache, what to persist, what Servlets to pre-load, etc. These architectural decisions should exhibit startup symmetry.

Once the software has started, now, it must scale and perform.

For a detailed description of each of these categories of symmetries in software architecture, please see Appendix A.

The general solutions outlined below relate to the practical applications of local and global symmetry.

1.8 Local and Global Symmetry
The distinction of local and global symmetry stems from physics where global symmetries and conservation laws are preserved while local symmetry can be broken, especially in the field of elementary particles and their properties.

The relevance to software architecture is that a local symmetry can be broken, but resolved or balanced by applied another remedying transformation that would restore global symmetry. This is the essence of the patterns outlined in this paper: in
many case a local symmetry is broken leading to the imbalance of forces while the pattern attempts to restore global symmetry by introducing its solution.

1.8.1 Pattern: Symmetry-preserving Transformation

We are trying to balance conflicting elements of the functional and non-functional requirements. We have modeled the system. We now need to map the design onto an implementation technology, say Enterprise Java Beans. But the selection of technology in itself is a mapping/transformation that has the potential to imbalance the symmetry in our software architecture. We need to ensure that introducing new technology solutions does not imbalance the architecture; e.g., compromise performance, scalability or security while adding reusability or distribution, for example.

How do we ensure that the mapping from design to implementation will not imbalance our software architecture?

Technology selections look harmless enough – and feasible. They all claim to perform and scale very well under all circumstances... until you try and integrate them and perform end-to-end testing. Then they usually fail. This is often not due to the inadequacy of the product; but rather our team’s lack of skill in configuring and optimizing the product; especially alongside other products that must, as a necessity be integrated with each other to provide a solution end-to-end. For example we may need a HTML page to interact with a Servlet who will invoke a JSP, Bean or other Servlet who in turn may invoke an EJB who in turn may invoke a JDBC call to get some preliminary data, perform computations and apply business rules at a Rule Layer and pass the intermediary results to a back-end transaction processing system running on a mainframe with CICS through some kind of message-oriented middleware such as MQ-series. Each piece may function well, but when put together, the entire system (greater and more difficult to configure and optimize than the sum of its products)

Therefore, use a transformation across tiers that preserves architectural symmetry. For example: If you decide to represent all business subjects as Entity Beans (EJBs), the resulting system will suffer in performance due to the overhead associated with network latency for connecting to the Entity Home, accessing the database and the Container. In this way, performance is imbalanced: an asymmetry has been introduced as a result of a design decision: the symmetry between uniformity in modeling (same type of beans) and the system performance would be compromised. Instead, to balance the architecture at the axis of symmetry—in this case being the network overhead and database access --, which in this case is “persistence latency”, using Entity beans would incur a huge overhead. Instead, if frequently used reference data are “Cache and Hashed” using a Session bean, which used some form of JDBC to access the data, then the symmetry is restored.

Now if the Session bean is used to access data using straight JDBC, the symmetry point of “Resource Contention” would necessitate using a Connection “Resource Pool”

---

3[^3] A System is something that is harder to configure and optimize than each of its individual components.
to gain access to a connection into the database as there will be multiple requests coming in for gaining access to the shared contended resource, the database.

Thus, in this case, two successive symmetry-preserving transformations where used to balance a) the need for persistence with performance and subsequently, b) the need for a connection resource pool with direct database access.

1.8.2 Pattern: Fractal MVC
Model-View-Controller applied recursively within successive layers of an architectural layer.

Problem
How do you perform separation of concerns and partition functionally separate aspects of a subsystem running on a given tier?

Context
Does MVC apply to the First and Second tiers only? I.e., is model-view-controller for the GUI to the middle tier only? Not necessarily. Each tier has its own set of classes that can be mapped into the MVC architecture. Wasn’t MVC used for GUIs? But we encounter it across tiers: GUI/presentation layer is the view, the model is the back-end database and the application layer is more of a controller. In each of the tiers and layers themselves, we find an element of MVC, very distinct and unmistakable. This is not a coincidence. MVC is a fundamental notion in software engineering that can be found to exist at various levels of abstraction and detail within a software architecture. This fractal nature of MVC is describe in this pattern.

Forces
We want to preserve the separation of concerns within each layer /tier. Layers often may to a tier. Sometimes a tier contains multiple layers: a middle tier may have the web server and application server as well.

Solution
Therefore, for each tier in the architecture, check to see if the three distinct roles of model-View-Controller are present and accounted for. Indeed many variations can be found: there are Document-Model variations, delegates (as in Swing) where Model is separated from a View-Controller, and of course Presentation-Abstraction-Control is another variation. But in all these variations, it is how the MVC are bundled that varies, not the existence eof the MVC roles.

Thus, distinguish the model, the view(s) and the controller(s). Each tier can be seen to have a model, a view and a controller. Even if a given tier is seen to be acting primarily as a controller (for example servlets on a middle tier), then, that tier will display stability only if its mode, view and controller sub-roles can be identified and are operating well in concert. Imbalance in one results in imbalancing and introducing asymmetry into the architecture.

1.8.3 Pattern: Two-way Mapping\(^4[4]\)

Aka

\(^4[4]\) Part of the CBDI Pattern Language.
Bi-directional Transformation

**Motivation**

Object relational mapping is an old problem that has been solved many times over. This is a special case of the Two-way Mapping pattern.

**Example**

Consider a web-based application context where a customer is making an online purchase. They either select or enter their customer information, billing information, product information (usually by Browse and Select\(^5\))[5], shipping information. They then submit this information. But how does the information get processed at each tier? For every tier, there is typically a transformation or mapping done that maps one type or representation of (for example, String) data onto a mid-tier representation, perhaps a graph of business objects (e.g., Java Bean or Session Bean).

![Diagram](image)

We represent this by a function \( M \), which maps one tier \( T[i] \) to tier \( T[j] \).

In the simplified view of the world, there are three tiers, presentation, application logic and persistence. Another function, \( D \), maps the data from the application logic tier which has just processed the data in its objects, to a persistent store, after having checked and run business rules in its tier. Each of those business rules are a self-mapping, \( S \), from \( T[I] \) onto itself \( T[I] \). This is called idempotence with respect to software architecture tiers.

Symmetry with respect to a software architectural tier means there is a commutative mapping across that tier.

When a transaction is initiated in a business system that spans multiple tiers, then there must be transitive transformation between successive tiers.


\[ A=B; B=C; \Rightarrow A = C \]

Transaction processing across multiple tiers and databases is a symmetric operation.

**Problem**

How do you ensure that a transformation or mapping from one tier to the next leaves the overall system integrity intact and in a consistent state?

\(^5\)[5] See author’s patterns for eCommerce: the user experience.

\(^6\)[6] For those conversant with the notion of onto mappings in mathematics, see [5].
Solution
You supply a commutative mapping function from one tier to the next, so that the flow and data going forth to that tier will return in a non-loss transformation. Therefore, provide a mapTo() and a mapFrom() method on the interface of each layer controller. The architecture should also have a Mapping Layer, one per tier, that maps its layer’s contents and format to a target layer’s anticipated format.

Alternatively, a standard format may be used in which case the role of the Mapping Layer in each tier would be to ensure mapping the local dialect to the globally understood language (for example, XML DTD or XML Schema).

Example
For example, map a GUI string set to a mid-tier business object graph representation, take the object graph and after performing symmetric transformations on it, map it to a persistent tier (e.g., database).

1.9 Pattern: Capture the Manners
Aka Components have Manners

Data is often mapped from tier to tier using a Two-way Mapping. But this often is a static representation. What about the behavior of tiers? Each tier may behave in its own unique way, but has to send data to another tier in an acceptable format. This is taken care of using Two-way Mapping.

But what about the ways in which a tier behaves? A tier can send data through its Mapping Layer to another layer, view the Two-way Mapping. But now, along with that data, it may need to send some instruction on how to use the data or send some information that is a behavioral specification which is not pure data, as such. Rules governing the behavior of the tier may need to be sent to the succeeding tier, so it knows how to manipulate the data that has been sent.

Manners are about capturing and ensuring that the elements of the software architecture behave correctly in various business contexts and most importantly, meet the constraints of non-functional requirements. Thus, a well-mannered software architecture will preserve symmetry.

Therefore, Capture the Manners. Define the rules governing the behavior of a component within a given execution context that will supply it with meta-data, events and context. Send its rules of behavior (manners) along with a context to the component [6]. This can be as simple as sending an object, or as complex as using Grammar-oriented Object design [6] to define a new configuration and collaboration for the tier or layer or component.

1.10 Pattern: Mapping Layer

Context
The transformation of one format of data into another across multiple architectural layers must preserve the semantic symmetry which is associated with preserving internal relationships amongst a group a atomic data elements.

7 Part of the CBDI Pattern Language
In order to ensure this, we create code to handle Two-way Mapping or a Bi-directional Transformation. Often creating code within a class or two to handle mapping of an object’s format is not enough: an object’s semantics need to mapped in a non-loss fashion across multiple enterprise architectural tiers. One of the biggest headaches in such projects arises from tracking this mapping of attributes on one tier to another set of corresponding attributes on a subsequent tier; often with a different name possibly a different implementation or representation.

Not only are strings transformed into object graphs, or object graphs mapped onto relational database structures, but the values of attributes are mapped at different tiers into different representations in order to allow the recipient tier to receive the data elements in a format and representation that will allow continued processing.

**Problem**

*You want to* preserve the semantics of a compound data element as it is transformed across multiple tiers allowing each tier to perform its own processing of data elements and messages, often represented in a common enterprise format.

**Forces**

Each tier assumes a unique form that their inputs data will have in order for it to be valid for further processing. Assumptions about format and semantics are often distinct yet complementary: format assumptions may relate to the type of data coming in; whereas the semantics may relate to the rules governing the associations between complex and composite data elements. Although these data elements are often encapsulated within the objects, in many cases where interaction with Legacy systems are done, an intermediate structure may be created to serve as a temporary input mechanism to a legacy system.

For example, various branches of the company that using the same base software may have different codes for the same thing. This often happens after a merger or acquisition. Thus “123” may mean “Three year loan” in one “locale” or user-group while in another, the same type of loan is coded as “3yr”. The Mapping Layer maps these values to an internally uniform value.

**Solution**

Therefore, include a *Mapping Layer* between major architectural tiers; acting as gatekeepers for transforming data and messages from one format or context to another. Take care to preserve symmetry and have a corresponding *Mapping Layer* on the receiving end to re-transform, decrypt, resolve, route, consolidate the returning data or messages coming back to the application tier initiating the transformation.

Input objects and data from one source tier are mapped into an Enterprise format, or formats, which will allow the continuation of processing in the recipient tier.
Consequences
Mapping layers always come in pairs: often one is out-bound or sending data and objects to a target system several tiers removed. And another is the in-bound mapping layer which receives the information results from the back-end tiers and re-maps them into intelligible data elements, formats or content (semantics) that tier level tier must ultimately require and eventually display to the user. The out-bound mapping layer transforms the data or objects into an intermediate or standard enterprise format and submits it to the next tier.

At some point processing is completed and data is returned to a previous tier such that it eventually tends to trickle back up to the presentation layer. As soon as it hits the in-bound mapping layer it is transformed into its corresponding format for the target format/ architecture.

1.11 Pattern: “Ten” Layers

Context
Consider a scenario in the world of e-business; with the advent of the Internet there are typically four major layers or tiers in a "layered" or n-tiered software architecture. These layers consist of the presentation or user-interface layer, the application layer, the business logic layer and the persistence layer. The application and business logic layer are sometimes mashed into one single layer; where the application specific code is mingled or merely kept together alongside the business rules and processing logic.

Forces
You want to minimize the number of layers to manage complexity; yet you need separation of concerns to put cohesive units of functionality each in their own conceptual layer.

These tiers must communicate with one another: a flow of data for an order may pass from GUI down to the persistence layer, going through all the intermediate layers. This communication is by no means trivial or even straightforward: each layer speaks a different language. The GUI talks in strings and sends its data to the middle tiers where they must be transformed and mapped into an alternative representation (become the attribute of objects, become serialized, parsed into XML, etc). The middle tier is often an object graph with cluster of collaborating subgraphs. Ultimately, these object graphs need to be mapped onto a persistence tier where they are either saved or participate in a back-end legacy transaction. Sometimes, the objects are serialized using some middleware protocol (RMI, CORBA, EJBs, COM+) to communicate with the next tier where more processing may be performed on them or the data they contain.

Each mapping is different and needs to be controlled and managed slightly differently. Mapping Layers from tier to tier provide traceability and tracking of the valid transformations and mapping that occur when data is transported from one
architectural layer to another; yet each layer added will have its “cost” in terms of complexity, performance, extensibility and security. Thus, adding new layers may be costly.

Note that layers are logical and tiers are physical representation of nodes. Layers will reside inside of tiers. The right mix of layers to tiers is tuned to preserve dynamic symmetry within the software architecture.

**Problem**

*You are trying to* figure out what layers and functional responsibilities per layer to include in your component enterprise architecture.

**Solution**

You see trivial solutions called three-tier architectures. But you end up running into problems arising primarily from the lack of organized mapping between layers: how information gets routed and transformed; mapped from one layer to another and processed, only to find that it has to retrace its steps back through the patches of the forest of tiers.

*Therefore*, use “MilleFeulle” or “Ten”\(^8\) Layers. Eliminate the layers that are not applicable to your solution. Be aware that you must have Mapping Layers and create and manage then accordingly.

---

**Discussion of the Solution**

There appear to be twelve layers above. The ValueObject is a transport mechanism. The client side and server side communication layers are mapping layers that the technology “automagically” maps for us: MQ-Series will do it for you, EJBs will do it for you and CORBA will do it for you, if you configure and build the right infrastructure.

1. An HTML Form is filled out to process an order (for example) The system converts the Strings in the form to an intermediate transport format called a Value Object or a Data Object or a Transport Object.
2. The Value Object is sent to the Web Server who delegates to the Application Server. The Application server handles the incoming request through a Servlets, which sends the information typically to a Java Bean or a Session Enterprise Java Bean for reification into or addition to an object graph.
3. The business layer then uses this newly augmented or constructed object graph to perform computations and apply business rules to the data entered and may decide to send the information to a back end legacy system or to a persistent data store sitting in the back-end.

---

\(^8\) *Mille* literally means “thousand” in French. Here the connotation is “many” and we have taken poetic license to limit this to the magic number Ten.
4. In order to do this, the business slayer often converts the object graph into an XML representation to be transported either via a Message-oriented Middleware protocol or SOAP or an EJB to the back end to be stored in a database.

5. Often in large organizations having multiple clients and regions and departments and having been joined by the shot gun marriage of a merger or acquisition, they often have different codes and abbreviations for the very same thing. A five year loan may be “5yrL” to one system and “515” to another. Thus, an enterprise data format may be employed to map each of the known formats into a uniform neutral format that will gradually drive the local non-standard versions into obsolescence. Thus, the Mapping Layer here maps a field (often generated at the business layer with the object graphs and not from the user-interface layer) into an acceptable standard enterprise-wide format. This often sent across the wire through some middleware.

6. The sending to back-end (or further tier) through some middleware is a Two-way Mapping of its own. There is typically a client-side communication layer and server-side communication layer. The client-side may reside in places such as the application server tier where the business logic and application logic may reside.

7. We need to ensure that the return values coming back from a back-end system, whether the result of processing or of database storage operation is propagated back up to the user in an appropriate, non-redundant, controlled and sensible, non-cascaded manner. Thus results need to be re-mapped (thus the name Two-way Mapping) back through a tier and onto another.

1.11.1 1.11.1 Consequences
You now know what a non-trivial scenario looks like; one that is not overly trivial like the vanilla three-tier architecture diagrams that are frequently seen with little or no true detail; the detail that often makes or breaks projects. Although each client does not have to use the “Ten” Layers, they can start with this vast expanse of functionality and systematically rule out what is not applicable or irrelevant to their solution.

1.11.2 1.11.2 Pattern: Business Driven Breadth-first
You have been asked to build an architecture. Time is of the essence. The business owners are skeptical of technology due to prior bitter experiences. You have been brought in to “save the day” or at least make something work in a short amount of time. Expectations are high, budgets are lower than anticipated and tempers are flaring.

***
Unit Testing must begin from a very early stage in order to ensure functional capabilities are being implemented correctly. This should not overshadow the responsibility of the Architect to exercise the operational architecture end-to-end in this breadth-first (Thin-slice of functionality) based on business priorities.

---

9[9] One in which the physical media, network, programs (middleware, database, language, etc.) and technologies have been selected but not tested in concert to see how they should be tweaked and optimized to work together, if at all.
How do you balance the need for functionality with the guarantee that the end product, once rolled out will actually perform and scale based on the non-functional requirements.

Often, team leads tend to focus on getting the functionality in place and demonstrating it in a controlled test environment which is far from the issues to be encountered in the field. This artificial test environment gives a false sense of comfort and appears to buy time and say “things are going fine, see, this works!”. But this is an illusion. As soon as you put the technologies together and spread them across n-tiers with real volumes, you start running into problems of scalability and performance; not to mention potential security issues as you must conform to corporate security standards, esp. in institutions where there is an issue of financial transactions and liability.

Therefore, strike a balance between the depth of functionality (low-depth) and the extent of architecture exercised end-to-end. Define a thin-slice of functionality that will be used to exercise the entire operational architecture end-to-end. The functional requirements should be business driven and contribute to the fulfillment of Iteration One which is the first extensible prototype (not throw-away) that will be demonstrated to management.

This business-driven breadth-first approach is one of our symmetries: resolving the constraints of development time, budget and business priorities with the technical feasibility of creating a quantum of functionality that also performs well. The essence of this quantum of functionality is to be as “thin” as possible (thus the term a thin slice of functionality restricted to one or two use-cases at most), thus beginning to meet the business driven priorities and yet making end-to-end testing of the architecture a feasible enterprise.

1.121.12 Conclusion

One of the main imports of this paper is that the discipline of software architecture can be viewed from a slightly different perspective, one of symmetry and the maintenance of symmetry as a means to maintain stability and balance in software architecture. This provides a powerful theoretical perspective on the discipline as well as providing it with a practical means of maintaining symmetry in software architecture.

1.131.13 References


[2] Software Architecture in Practice


[4] GoF

2 Appendix A

2.1 Categories (Types) of Symmetry

In mathematics (and we will not go into too much detail there), we have various notions of symmetry. In software architecture, we have identified at least two: fractal symmetry and transformational symmetry.

![Symmetry in Software Architecture](image)

**Figure** Error! Bookmark not defined.: Symmetry in Software Architecture

2.1.1 Fractal Symmetry

This refers to the predictably recurrent patterns of order seen at successive levels of a software architecture. Fractal MVC is an excellent example. Here the trait that remains intact even as we pass from large-scale to the small-scale form cross tier to
within-tier analysis of collaborations and of design, we witness recurrence of a pattern within a pattern; frequently at these levels:

2.1.2 Transformational Symmetry

This refers to the kind of symmetry that transforms one tier's output into another tier's input and vice versa in a non-destructive way. By that we mean a type of non-loss transformation where no item of information is "lost" due to errors not handled or exceptions not caught. This is best exemplified by Two-way Mapping.

2.1.3 Static or Design Symmetry

During design, software architects tend to define the application architecture based on functional requirements. It is often in a subsequent phase where the logical architecture (where good architecture displays static or design symmetry) is assigned to the operational architecture. Then it has to be fine-tuned to behave according to the non-functional requirements: manners are about rules (non-functional) governing how the functional architecture should behave under non-functional constraints.
A well-mannered architecture is one in which the functional requirements have been tuned to behave in accordance with non-functional requirements.

Subsystems turn into components when they are realized in code and assigned to processing nodes. The implementation is designed to be in accordance with functional requirements. But this is still in design. When the architecture is put into production, the static or design symmetry is a necessary but not a sufficient characteristic to realize a scalable, reliable, well-performing system.

In order to have a secure, scalable, robust and extensible application and operational software architecture, static symmetry must be augmented by adaptive and dynamic symmetry.

2.1.3.1 Logical Symmetry
When the system is being conceptualized and designed. The Component architecture is being created: “what are the pieces, the parts? How will the pieces work together?” This frequently involves the balancing of functional and non-functional requirements.

2.1.4 Physical Symmetry
This is the design of the network topology, the operational architecture: load-balancing displays symmetry; workload management displays symmetry, cloning, etc.

Load testing is a tool to test your symmetry: if the system does not display symmetry, load testing will uncover the weak parts that are not symmetrical.

2.1.5 Configuration Symmetry
This is when we are in the process of balancing conflicting (usually) non-functional requirements of whether to get values from a data store or to cache the values; we are setting up the connection pooling, the caches, the design level objects that will serve to Hash and Cache, for example.

2.1.6 Start-up Symmetry
This kind of symmetry is needed and exhibited when the application is getting ready to perform its main functional role: it is starting up. It is loading configuration information that was creating (hopefully, with the Configuration Symmetry in mind).

2.2 Dynamic Symmetry
This type of symmetry occurs when an application is running; this is distinguished from its design, configuration, warm-up and subsequent change through adaptive symmetry.

2.3 Adaptive Symmetry
Once you have run the application and new customers requests have come in and have been prioritized and planned, you need to incorporate changes to the application and the architecture (the context the application is running within) with minimal disruption to production operation. Adaptive symmetry are the set of...
transformations that leave the running system intact and in a consistent state, which adding or modifying functionality or through-put.

3 3 Appendix B: Definitions

3.1 3.1 Webster

Etymology: Latin *symmetria*, from Greek, from *symmetros* symmetrical, from *syn- + metron* measure -- more at MEASURE

Date: 1541

1: balanced proportions; also: beauty of form arising from balanced proportions

2: the property of being symmetrical; especially: correspondence in size, shape, and relative position of parts on opposite sides of a dividing line or median plane or about a center or axis -- compare BILATERAL SYMMETRY, RADIAL SYMMETRY

3: a rigid motion of a geometric figure that determines a one-to-one mapping onto itself

4: the property of remaining invariant under certain changes (as of orientation in space, of the sign of the electric charge, of parity, or of the direction of time flow) -- used of physical phenomena and of equations describing them