Domain Model Normalization: Towards a Foundation for Managing Business Domain Architecture Evolution

Russell R. Hurlbut
Expertech, Ltd.
P.O. Box 4151 Wheaton, IL 60189
Document: XPT-TR-97-01
rhurlbut@charlie.iit.edu

ABSTRACT. *Domain engineering is concerned with the development of an architecture for implementing a family of related applications. We need to make certain that as we evolve and expand the domain model to accommodate new applications, that these future applications are not unnecessarily constrained or complex. This paper presents a foundation for managing the evolution of a business domain-specific architecture though the concept of 'domain model normalization.' Domain model normalization is presented as an extrapolation of data model normalization through the addition of process, policy, and context normalization concepts. Basic operations and properties that can be applied to a domain architecture are described.*

KEY WORDS: *Domain Architecture, Domain Normalization, Object Normalization, Use Case Formalisms, Object Oriented Analysis and Design*

1. Introduction

Domain engineering, which is concerned with integration of domain oriented components [AR94], is a field that has been emerging as the key to any domain-specific reuse strategy [Oops95b]. The generic elements identified during a requirements domain analysis are used during the domain engineering effort to construct design fragments [dF92b]. These designs can be implemented as domain-specific code or combined into larger constructs such as components and frameworks.

A framework is designed to cover a family of applications or subsystems in a given domain and is typically delivered as a collection of interdependent abstract classes, together with their concrete subclasses [GM95]. With the increasing interest in domain engineering and domain-specific application frameworks, software evolution and maintenance tools and techniques are essential to realize the benefits of these fields. Evolutionary changes to a system can occur for several reasons. User needs can change, the underlying real-world domain can change, additional functionality can be added, and the architecture can be improved. The changes may occur during any stage in the lifecycle of the system [Hurs95].

System administrators are often fearful of making changes to a system solely for improving the architecture. In order to ameliorate those fears, techniques such as such as refactoring [Opdy92], propagation patterns [Lieb96], and reflection [Hink94] have been developed to maintain structural and behavioral integrity of the application code when making modifications. Since a domain-specific framework is intended to be frequently reused to create new applications, these perfective improvements may be crucial for continued use of the framework.

By combining domain engineering and application framework development, a domain model is constructed with the intent that it is be used by a family of applications. In practice, it is not

uncommon for an initial application to be developed in conjunction with the domain model. Two primary concerns must be addressed in such a scenario:

1. How do we maintain non-interfering applications? In other words, as we add new applications, how do we assure that they are not destructive to each other? If we reduce this to a view problem, it becomes: what extra protocols are needed to safeguard non-interference over the domain of objects?
2. How do we minimize bias towards any individual application in developing the domain model? Building in flexibility that is not needed for an application under developed needs to be balanced against future needs. When those future needs dictate a reorganization of the domain model, existing application may need to be rebuilt to conform to the revised domain model. Therefore, the converse of the question is: how do we minimize the impact on existing applications when evolving the domain model?

Although the severity of these concerns may be exaggerated in this scenario of simultaneous development of the domain model and initial application, these concerns are still relevant for any domain engineering effort.

Domain Model Normalization attempts to address both of these concerns. It provides a smooth migration path for the domain architecture so that future applications contribute to a more stable domain model by leveraging existing artifacts. It also promotes a balance between building in future flexibility and reducing the impact on existing applications when changes are made by treating domain model properties of adaptability and usability with equal concern as structural and behavioral consistency.

2. Extending the concept Data Normalization to Domain Models

We begin our extension of data normalization with two expositions as to what normalization means to relations databases:

> Normalization is the process of putting things right, making them normal. The origin of the term is the Latin word *norma*, which was a carpenter's square that was used for assuring a right angle. In geometry, when a line is at a right angle to another line, it is said to be "normal" to it. In a relational database the term also have a specific mathematical meaning having to do with separating elements of data (such as names, addresses, or skills) into *affinity groups*, and defining the normal, or "right" relationships between them
>
> - Oracle - The Complete Reference [KL95]

> Normalization is the process of converting an arbitrary relational database design into one that will have "good" operational properties. In hierarchical and network models, it was not possible to investigate these problems in the systematic, rigorous way that the relational model allows because they were not solidly grounded in mathematical theory.
>
> The process of converting an arbitrary relational database design to one that avoids certain anomalies is referred to as normalization.
>
> The objective of normalization is to produce a database design that can be manipulated in a powerful way with a simple collection of operations while minimizing data anomalies and inconsistencies.
>
> The normalization process does not consider the numbers (absolute or relative) of particular entities or relationships, or what types or frequencies of queries to be processed. It is not biased towards a particular usage environment. (e.g., Network models are tuned for expected workloads in their designs)
>
> The importance of normalization lies not in following it slavishly, but in using it to make explicit trade-offs between efficiency and integrity.
>
> - Database - Technology and Management [Gold85]

Affinity groups

In order to extrapolate these normalization concepts, a few of the key aspects will be further analyzed in order to find an analogy. First, if data normalization separates elements (names,

addresses, or skills) into *affinity groups,* then behavior normalization should separate functions into similar types of affinity groups. The types of affinity groups that may be appropriate for functions are ones such as derivations, setting an attribute value, and propagating a message. These functional affinity groups are orthogonal to the data elements. If we create a matrix that places data affinity groups along one axis and the functional affinity groups along the other axis, it becomes possible to identify usage pattern between these two affinity groups. For example, all data elements would require some function to set its value. Furthermore, the particular method or style of setting the data element values would typically be consistent throughout the application, and possibly for the complete family of applications. As such, we can consider each application instance an affinity group along a third axis. Figure 1 illustrates the repetition of usage of the data attribute with the 'set' function affinity group across applications created from a shared domain architecture. Guidance as to how these intersecting affinity groups can be combined to create an application relies on composition rules. These composition rules are grounded in the basic principles of domain normalization, which will be presented in section four of this paper. For now, it will suffice to merely point out the distinction between composition rules which apply fundamental domain normalization principles and business rules which apply to the subject domain being modeled.

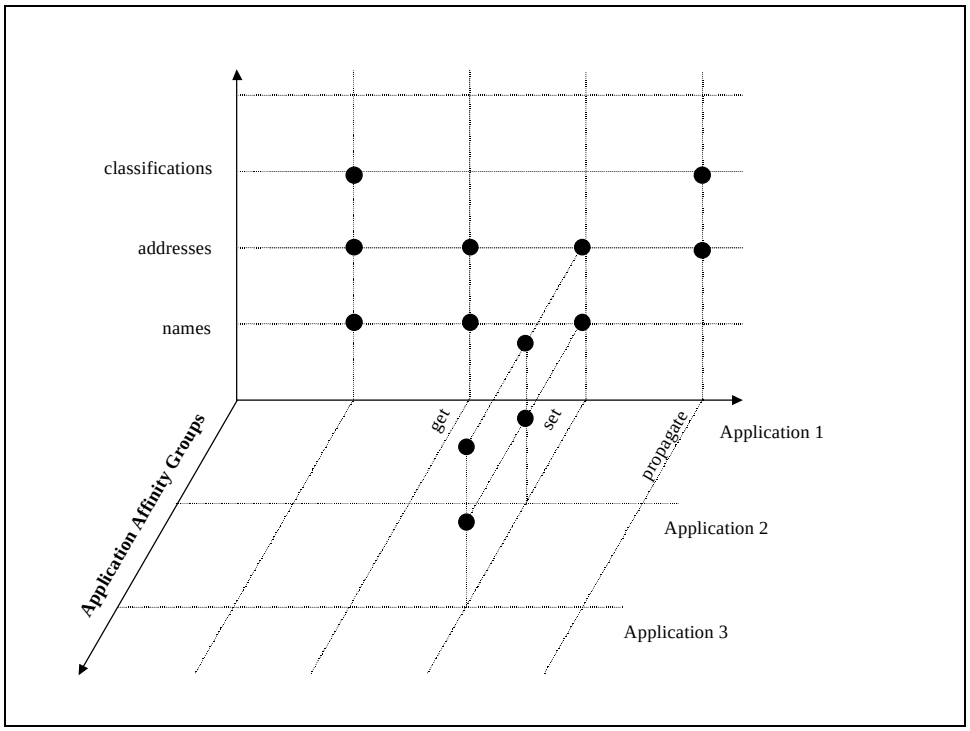

Figure 1 – Affinity Groups for Domain Normalization

Good Operation Properties

As the second citation suggests, domain normalization should be able to convert an arbitrary domain model into one that will have "good" operational properties. The relational database model stands in contrast to the hierarchical and network database models by its nature of being solidly grounded in mathematical theory. Because of the ability to couple procedures with data, the same capabilities that are provided in the relational model are available within a domain model.

Furthermore, addition semantic information is available that will allow relaxation of normalized data structures through the use of attached behavioral properties. Thus, database models that would have otherwise been considered denormalized, can still be represented as a normalized domain model. The basic premise behind this ability to relax certain data structures balances complete data model normalization at one extreme and complete database encapsulation at the other extreme. The process of enabling normalized behavior will be fully presented in section four.

Powerful Operations that avoid Anomalies

The primary objective of domain normalization is to produce a domain model that can be manipulated in powerful ways with a simple collection of operations while minimizing the behavioral (as well as data) anomalies and inconsistencies. Whereas the data model operations are intended to yield data from a database, the domain model operations are intended to yield executables from a domain architecture or framework. When considering operations on a domain architecture, the entire collection of objects is treated as data. This represents an intensional view of the domain model, since at this point the objects are treated as data only, not as elements capable of manipulating other elements (such as data).

A set of database tables, independent of any domain knowledge, is insufficient to complete normalization beyond second normal form. To proceed to third normal form, we need to understand the concept of functional dependencies. Specifically, knowing that there are no transitive dependencies carries a measure of semantic information. But this also implies some considerations as to whether we will be creating class hierarchies, composites, or some weaker relationship among object types. One concern is whether of not there is sufficient semantic information to provide appropriate relationships among classes. Many times the decision has seemed arbitrary. Domain normalization needs to be defined in such a way that certain semantic properties will be preserved, certain anomalies can be avoided, and certain class operations that are independent of any specific application instance of the domain model can be performed. In order to adhere to these constraints, three categories of operations are defined.

1. *Intensional operations.* Operations that are performed on portions of the domain architecture.
2. *Extensional meta-operations.* Operations that are performed by the entities within an application instance that either inquire about or manipulate the structures and protocols of collaborating entities.
3. *Extensional operations.* Operations that are performed by the entities within an application instance that fulfill the expected functionality of the application.

If, as in data normalization, one of the primary objectives of domain normalization is to avoid certain anomalies, then we need precisely define what anomalies can be present in the domain model. By first considering data modeling, the update, delete, and insert anomalies each refers to repeating data that is in either first normal form or second normal form. When certain fields are repeated, updating one row puts it out of synch with the others; similarly, the delete anomaly ignores the separation of the concept behind of the repeated field from the non-repeating attributes. For example, a library book (with attributes of title, author, and publisher) is separate from the collection of instances of that book (with attributes of catalog-number, copy-number, borrower-id, and due-date). The delete anomaly arises when deleting all instances would also destroy the book information. The insertion anomaly addresses the inverse of this situation, i.e. when information cannot be inserted, such as the book information, because no copies exist.

When relating these anomalies to domain models, updating an attribute or method typically relies of factoring of functionality to the highest possible level in a class inheritance hierarchy or

delegated to the same class as a member of an aggregation hierarchy. If an attribute is propagated through a complex class graph, then many classes may be affected by the deletion or addition of an attribute. Adaptive software deals with this problem through their propagation patterns. But propagation patterns don't address derivation functions that might not be needed. Subject oriented programming comes closer to pruning functionality, but assumes a more rigid data structure.

Delete and insert anomalies at the domain normalization level refer to the creation or removal of types or classes. If we remove all collaborations that refer to a role/class then the class disappears. Similarly, including a role/class in the domain model that does not yet have any collaborations essentially renders it meaningless. In order to avoid this, we must be able to maintain a class that captures information regarding what roles it can play. Just as we were able to preserve the book information by creating a new entity, we must preserve the entity information by creating a new meta-class to hold the class until needed. The join property is based on the collection of roles that it can play, and just like in relational algebra, the selection can be any one from the candidate set. Thus, one of the fundamental principles of domain normalization is that all classes/types must belong to a meta-class in order to be in normal form.

Unbiased Implementation

One outcome of domain model normalization is to provide constructs that are not biased towards any specific application instance. The echoes the second primary concern identified in the introduction of this paper. For example, starting with the assumption of a normalized data model, unforeseen queries, such as the ad hoc nature of decision support systems, can be handled in an unbiased way. The data normalization process does not consider the type or frequencies queries. It also does not take into consideration the numbers, either absolute or relative, of particular entities or relationships. Consequently, domain normalization should remain unbiased towards particular types of applications. In other words, the domain normalization process should not consider the numbers of classes or collaboration patterns, or what types of applications are generated. It is not tuned for a particular business domain framework.

This does not mean that performance strategies can be ignored. Consider relational data model designs that include disk strategies or needed indexes. These are not part of the logical design, but perform important needs with respect to deployment. In a data model, performance considerations may force changes to the logical design, such as restructuring of the data tables for warehousing or replication strategies. As previously introduced when discussing good operational properties, domain modeling can permit a data structure to become denormalized, while still preserving the desirable properties of normalization. By attaching behavior to data structures, domain model normalization allows alternative implementations that remain unbiased to a particular implementation by maintaining semantically equivalent structures of data and behavior as alternate representations. Since most database vendors provide triggers and stored procedures, domain normalization can be preserved by coupling certain triggers with denormalized tables to maintain semantic equivalency. Composition rules, as a form of meta-rule, can assist in determining when triggers coupled with denormalized tables are appropriate over normalized data structure.

As we can derive from the preceding illustration, normalization allows us to evaluate explicit trade-offs between efficiency and integrity. It should not be followed blindly. We will be exploring the implications and properties required for implementing such rules in section four.

3. Domain Model Operations and Properties

Data model normalization allows us to produce a database design that can be manipulated in a powerful way with a simple collection of operations. We know that the relational model provides

its own calculus of selection, projection, Cartesian product, and intersection. This section identifies a set of candidate operations and properties that we hope to distill into a small collection of operations that a domain model should support. In compiling this collection, we gave consideration to most of the fundamental concepts behind object-oriented analysis and design, with particular attention to collaborative design patterns. As stated in section two, three categories of operations are defined, intensional operations, extensional meta-operations, and extensional operations. We will be looking with particular interest at each of these three categories.

Pattern Operations

An analysis of the Gang of Four's design patterns [GHJV94] reveals several operations that should be considered in creating a set of operations for a domain model architecture. Beginning with the structural patterns, the creation patterns (Abstract Factory, Builder, Factory Method and Prototype) involve an instantiation operation. The Singleton pattern involves both an instantiation operation and a navigation operation. Most of the structural patterns all involve delegation and extension (Proxy, Facade, Decorator, Adapter, Bridge). The Composite utilizes, as its name suggests, the composition operation. The Flyweight pattern utilizes instantiation.

The behavioral patterns provide several more operations that are of interest. The Chain of Responsibility pattern utilizes the delegation operation. The Command pattern involves both instantiation and delegation. The Interpreter is a specialized pattern most closely related to the composition operation, but the authors' recommendation is to limit its application to grammars. The iterator pattern is obviously involved with the iteration operation. The Mediator and Observer patterns involve the notification operation. The Momento pattern is most closely associated with the externalization operation. The State pattern has more to do with role playing than any operation. The strategy pattern addresses the negotiation operation by specifying how the strategy object interacts with the context object (i.e. by the context object passing itself as an argument to provide needed parameters or through taking the data to the strategy). The Template Method pattern mostly involves extension -- the key here is know what methods are virtual and which are intended to be overridden. By its very nature, instantiation is important to this pattern.

Finally, the Visitor pattern is the most complex and most interesting for our purposes. It is involved with each of these types of operations: negotiation, delegation, and composition, iteration, and extension. Here a class is defined to extend the behavior of an object structure (through composition). Negotiation is usually accomplished by calling a unique operation that corresponds to its type as well as passing itself so that its state can be accessed, as needed. Delegation is involved since the added functionality is collected in a new class for the entire object structure. Iteration across the structure is then performed.

Upon examination of the underlying nature of the operations identified for the design patterns, it is evident that we are dealing exclusively in the domain of extensional operations. In other words, these design patterns represent collaborations among entities within an application instance that fulfill the expected functionality of the application. Furthermore, by their very nature of being collaboration patterns, they deal almost exclusively with weak coupling among the entities. There are two possible exceptions to the weak coupling nature of the design patterns. The first, the Composite pattern, demonstrates a strong coupling with its subordinate parts in an aggregation hierarchy. The second, the Iterator pattern, implies a more profound effect on the target of the collaboration than the other design patterns.

Our conclusion is that design patterns, although a significant contribution to object-oriented development community, are not a significant contributing factor in promoting a select set of operations on a domain model architecture. Recurring patterns of interaction may be desirable to

maintain, but they are not essential for domain model normalization. Consider, for example normalizing a database that contains address information for students and vendors. Normalizing collaboration patterns would equate to the collapsing of both groups of addresses into a single table. This is clearly not a desirable goal, since there is no redundancy of data. These are completely separate conceptual entities. Even if the physical deployment placed both sets in the same table, they logically remain separate. At best, we might impose some constraint on each of the addresses for consistency, such as each address type should have two fields for street address information, rules will stipulate how these two field should relate to each other, and each field should have a maximum length of thirty-five characters. Likewise, we do not believe that collaboration patterns should be ignored, since they can play an important role is establishing constraints that can promote other desirable properties of the domain model, particularly understandability.

Object-Oriented Analysis and Design Candidate Operations

In our efforts to derive a set of operations for domain model normalization, we considered the fundamental principles of object-oriented analysis and design. These operations were classified as members in one of four general categories. This classification is summarized in Table 1. The first two, weak and strong coupling operations, relate to extensional operations. The latter two, meta-operations relate to intensional (i.e. those operations that are performed on portions of the domain architecture) and extensional meta-operations. Although there is generally no direct correspondence to the final set of operations derived for domain model normalization, each of these four areas had significant influence on shaping the final list.

Weak Coupling (Association/Collaboration)	*Strong Coupling (Aggregation/Inheritance)*
• Delegation	• Generalization
• Instantiation	• Specialization
• Navigation	• Overriding
• Notification	• Extension
• Externalization	• Overloading
• Negotiation	• Restricting
Meta-operations (done-to)	*Meta-operations (done-by)*
• Abstract	• Iterate
• Migrate	• Meta-info query
• Refactor	• Order
• Encapsulate	• Mutate

Table 1 – Candidate Operations for Domain Normalization

Abstraction

An abstraction operation takes a concrete entity and distills out of it the essence that makes it of interest at a more simplified level. All details not of current interest are ignored. Abstraction, as applied to a domain model, labels a collection of classes, thus hiding the details of the component classes. These collections have been referred to by many different names, such as clusters, class categories, ensembles, subsystems, components, modules.

From an operation standpoint, the abstraction operation implies a layering construct on the domain model. Whether or not this operation constrains the architecture depends on the visibility of one layer from another not directly above it.

Abstraction tends to provide boundaries that become important in distributed computing environments, because the organization or selection of components may depend on these boundaries. Therefor this operation seems to bias a domain model towards a specific implementation, or application instance. This can be avoided by maintaining a focus on patterns of interactions and roles played by the interacting entities.

Aggregation

Aggregation contrasts with an abstraction operation in that there is no intent or desire to hide the details of the components. Usually an aggregation is made of similar entity types, such as a collection of employees. This does not have to be the case. Typically, the property of polymorphism will allow an aggregation to still be treated in a similar fashion. This relaxing of the degrees of freedom for an aggregation operation, thus usually must be preserved through a property or another operation.

Polymorphism may be implemented through interface inheritance or by simply reusing the same message passing protocol. An aggregation operation may also be combined with a meta-information query operation to obtain similar results. In this case, a further relaxation of the degrees of freedom of the aggregation imposes a constraint on the client entity to be able to reason about the services provided by each component member of the aggregation in order to negotiate a communication protocol. The levels of communication protocols among such objects may involve a new pattern of interaction that invokes a mediator, or translator to enable such communication.

Association

An association is a weak coupling of entities. It typically implies that the multiplicity property be relaxed to allow 1:n connections among entities. This relaxation of constraints for data imposes other constraints with respect to the behavior of entities involved in such an association. Rules that define an association limit the degrees of freedom of behavior for entities as long as they are involved in such an association.

Frequently, the navigation operation is combined with entities involved with an association. From an *extensional* perspective, navigation is something done *by* entities, whereas in an *intensional* context, association is something done *to* entities by the omniscient architect. If we only looked at navigation as something done to entities, this becomes a simple browser type operation. That's not of any particular interest since it has no affect on the domain model.

Composition

In order to semantically differentiate among composition, aggregation and abstraction, we make the following comparisons. Composition, like aggregation, differs from abstraction in that there is no attempt to hide details. However, composition differs from aggregation in the depth of the collection. Aggregation is generally a shallow, one dimensional collection. Composition implies unlimited depth to an arbitrary collection of entities.

Delegation

Delegation has overtones of workflow. It is an extensional operation that implies involvement in the basic speech act operations.

Extension

The extension operation adds new features in to subclass a subclass of the desired entity. This is clearly a meta operation.

Externalization

Externalization is similar to the Gang of Four's Memento pattern. The difference here is that rather than just capture and externalize the state of an object, a more CORBA-like implementation intended for larger grained entities is implied. Thus the externalization operation is best suited for higher level abstractions. This operation seems to have both extensional and intensional implications. Externalizing an entity's state is implementation oriented. Providing this capability to a given entity resides at the domain architecture level. However, since our focus is dealing with larger grained entities, it seem most appropriate to pair the externalization operation with a meta-object query operation. These two operations are essentially opposites sides of the same coin. The ability to externalize creates the availability of meta-object query.

Generalization

The generalization operation is basically a refactoring of behavior and/or data into a shared representation. The different implementations of this have long been a struggle for data modeling even without the additional burden of factoring behavior.

First, the issue of implementation versus interface inheritance must be considered. Implementation inheritance refactors behavior. Increasing behavior through extension is expected. On the other hand, increasing the interface raises serious concerns that involve contravariance and covariance concerns. The distinction between classes and types must be considered at this point.

Contravariance assures substitutability of instances with subclasses. A contravariant type policy requires that the method argument type for a class and subclass are the same (conservative contravariance) or a superclass of the type (regular contravariance). *Covariance c*an accept more specific information than the superclass at the expense of substitutability. A covariant type policy allows that the method argument type for a subclass to be a subtype of the method argument type for its superclass.

Although a covariant type policy seems to expand the degrees of freedom, it relies on additional knowledge to assure a safe downcasting of types. A meta-object query is one solution. The property of homomorphism, or the parallel mapping of two classes, may also be a practical solution to obviate this restriction.

Second, the issue of how a class hierarchy is best represented when only the data is considered tends to blur the conceptual design with the implementation. For example, a class hierarchy may define *legal entity* as the topmost entity. *Person, trust* and *corporation* may be subclasses. *Person* may be further subclassed into *employee* and *stockholder.* Where do we place the data members when modeling this is-a relationship? Several guidelines have been proposed. One such recommendation is to avoid data member attributes in the class that defined an interface hierarchy. Another is that only leaf nodes of a class hierarchy should be concrete entities. So how do we normalize the data members when a *legal entity* is both an *employee* and a *stockholder*? In a relational database, we could have separate tables for *person, employee,* and *stockholder* with the constraint that one could not exist without the other. However, this is the opposite direction of the referential constraint than what is usually defined. In our scenario, a *person* tuple could not exist without either a *employee* or *stockholder* tuple.

By looking at the class hierarchy conundrum from a different perspective, *employee* or *stockholder* can be considered roles played by person. Thus we can break the *is-a* relationship and replace it with a *plays-a* relationship. The relationship between *legal entity* and *person, trust* or *corporation* can rightfully be maintained as an *is-a* relationship. One constraint that we may want to maintain

for such a relationship is mutual exclusion. We may also impose an additional constraint that there can be no mutation, e.g. from a *person* to a *trust.*

Inference

Inference deals with the ability of an entity to reason about its environment. This appears to be only applicable as an entity operation.

Instantiation

Instantiation is an operation to create an entity of a certain type or class. Several design patterns deal with how this instantiation operation is carried out. As an intensional operation, this is generally a strategy that can build flexibility into a model, but does little for capturing the essence of a domain model.

Iteration

Iteration provides a mechanism for an entity to access the elements of an aggregate. This is described by the Gang of Four as a design pattern that provides sequential access without exposing the underlying implementation. This is an entity operation.

Meta-information query

A meta-information query is an entity operation that assumes that the entity has the capacity to reason about the services available to it from other entities. So this operation assumes certain constraints that the entity must possess.

Migration

Migration is a meta-operation that that allows evolution of the domain model from one version to another. Most object databases provide this capability through one of three strategies. At the two extremes, either the database is updated at the time of the change or it is never updated by favoring a transformation operation that is added to the new version of the entity. A hybrid approach, typically implementing a lazy transformation on a when requested basis is also a practical strategy. Configuration management will be of significant interest in normalizing a domain model over time.

Mutation

Mutation is an entity operation that allows an entity to change from one type to another. The importance here is that the identity of the entity remains the same in order to preserve associations that the entity may not be aware of.

Navigation

Navigation allows an entity to locate another entity by querying entities that it is aware of. As discussed under association, this is an entity operation. Certain assumptions must be made. We must be willing to relax the visibility of the set of entities available to the entity of interest. This violates the Law of Demeter, but when combined with other operations, such as composition and abstraction, may satisfy normalization properties.

Notification

Notification involves the broadcasting of the change in state of an entity. One pattern that takes advantage of this is the Gang of Four's Mediator pattern. A blackboard architecture may also take advantage of this operation. This is an entity operation.

Ordering

Ordering is a meta-operation that is built into aggregation. This is accomplished through defining the equality and less than comparisons. It also requires selection of a storage strategy, e.g. linked list, hash table, array, btree, unordered set, bag. A relation should also be considered as a storage strategy.

Overloading

Overloading allows for the same message to be sent to an entity that varies only in the types of arguments sent. As such it is part of the definition of an entity and thus is a meta-operation. The assumption here is that the entity will know how to properly handle whatever is sent to it.

Overriding

Overriding is the modification of the behavior of a method that has the same message and arguments. There are three basic purposes for overriding a method. Extension and restriction are described as their own categories of operation. The third purpose is to accomplish programming-by-difference. It is primarily a convenience operation that should be avoided. This will definitely be considered a violation of object normalization. If this operation is performed, deferred maintenance will usually follow. The automation of CASE tools with refactoring should be able relieve some of the tedium of the task to renormalize the domain model.

Refactoring

Refactoring typically is a compound meta-operation that involves generalization or aggregation along with migration and mutation. As such it may constitute a pattern for meta-operation. The relationship with refactoring and adaptive architectures will be a main focus of this paper.

Restriction

Restriction is achieved by constraining ancestor attributes of the subclass. It may also be achieved by utilizing a covariant typing policy that restricts the types of arguments of the subclass as described under generalization. One reason for restriction is to achieve optimization that would be unavailable to more general types.

Specialization

Specialization is the opposite of generalization. It is best represented through the restriction and extension operations.

Object-Oriented Analysis and Design Properties

Our primary goal for examining object-oriented analysis and design properties is to provide insight as to how the application of domain model operations can be performed in such a way as to preserve those properties considered desirable. Interaction patterns are an important consideration when considering what properties are desirable. Pree identified a set of seven meta-patterns to describe all collaborations [Pree95]. From these meta-patterns, we have distilled out their three distinct properties: recursion, connection (multiplicity), and unification. These and other properties considered are identified in table 2.

Property	Description
Abstraction	The level of abstraction of an entity is an attribute that indicates the level of clustering
Access rights	What entities are permitted to communicate with the subject entity
Attributes	The data aspect of the entity
Candidate keys	A set of data attributes that uniquely identify the entity when part of an aggregation
Concurrency	An entity and a relationship property. As a relationship attribute, it describes which entities must be active at the same time and which have activity that is mutually exclusive. If two entities can receive events at the same time without interacting then they are concurrent. The thread of control is a critical concept to concurrency in determining if can there be multiple threads.
Constraint	An entity attribute that manifests itself through assertions, such as preconditions or postconditions that must be true at certain points during execution, or through invariants that must always be true.
Contravariance	Describes the relationship of the entity with its super or sub class --indicates participation in an extension operation
Covariance	Describes the relationship of the entity with its super or sub class -- indicates participation in a restriction operation
Derivation	As a relationship attribute, it reflects the transitivity of other relationships (e.g. a grandparent is the result of two parent relationships). As an entity attribute it describes attributes (which can themselves be entities) that are redundant in that other attributes can used to generate it (such as age from birthdate and any other point in time). This is a form of redundancy that violates the most basic normalization principles.
Discrimination	As an entity attribute, it is the behavior of the object that operation on a candidate key. As a relationship attribute, a discriminator is an enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship -- i.e. what property varies among the subclasses.
Dynamic binding	An entity attribute that provide method resolution at run-time. Although this is typically a property of the implementation language, it can be controlled in those environments where it is available. In such cases it reflects the ability to perform overriding operations.
Encapsulation	An entity attribute that provides information hiding that allows the implementation to change without affecting is protocol.
Events	Both an entity and relationship property that potentially involves concurrency and/or transitivity. The causality of events determine behavior of entities. From a relationship perspective, an event is a one-way transmission of information from one entity to another. From an entity perspective, events have associated attributes that help set the state of the entity, that describe the normal behavior or exceptions.
Homomorphism	A relationship attribute that describes the parallel mapping of two (or four) classes.
Indirection	A relationship attribute that describes the result of a delegation operation.
Inheritance	A relationship attribute that describes the result of generalization, specialization operations -- specifically the extension and refinement operation. Multiple inheritance is also possible. The focus here is on implementation inheritance which should be a private inheritance rather than interface inheritance which is better suited for typing.
Link attribute	A relationship attribute that describes the relationship -- it is most useful in many-to-many relationships. This involves role playing.
Meta-data	An entity attribute that provides data about itself. It allows an application to reason about, and possibly even change, its own structure and capabilities with respect to operations, attributes and types.
Multiplicity	A relationship attribute that describes how many instances of one type that may relate to a given entity. Although 'one' or 'many' are most common, it is also possible to have specific and/or disjoint intervals.
Operations	An entity attribute that describes the behavior of an entity in response to a message in terms of side effects and internal state transformations.
Persistence	Maintain the state of the entity between executions of the application.
Polymorphism	A relationship attribute that describes the ability of the entities to respond to the same message, but unlike substitution, will behave in different ways.
Propagation	A relationship attribute that describes the automatic application of an operation to a network of entities upon the invocation of that operation on a given starting entity. Often referred to as triggering.
Protocol	An entity attribute that describes the set public messages and arguments along with their types that an entity will supports.
Qualification	A relationship attribute that reduces the effective multiplicity the association and separates the results of the qualification into disjoint sets (although disjoint sets of one each is most common). This property is important for navigation operations.
Recursion	As an entity attribute, it is the ability of an entity to invoke itself. As a relationship attribute, it describes the dependency between entities to accomplish an entity recursion that may not otherwise be possible.
Reflectivity	An entity attribute that describes the ability of the entity to modify its own behavior depending on its own state. Reflection involves meta-knowledge of the entity so that it can reason about itself. It is similar in purpose to changing the type of an entity.
Role	A relationship attribute that describes the assumptions of entities regarding each other
States	An entity attribute that captures the value of its attributes. It is possible that all attributes are not a part of its state if they are internal and we assume that the entity is not currently involved in an active thread of execution.
Substitution	A relationship attribute that allows one entity to stand in for another with no change in the resulting state.
Transitivity	A relationship attribute that provides that the ordering of the entities that perform services does not alter the ultimate end state. It also refers to class hierarchies in which all behavior and data is inherited (i.e. a subclass is automatically an instance of it ancestor class).
Type	An entity attribute Refers to adherence to a protocol rather than mere inheritance
Unification	An entity attribute that permits an entity to play collaborating roles within itself

Table 2 – Domain Model Properties

4. Domain Model Normalization Operations

Just as in data model normalization, there is a clear distinction between the steps needed to get the domain model into normalized form and those operations that can be performed on a normalized domain model. In deriving our essential set of operations for domain model normalization, we established four distinguishing aspects of a domain model. There are parameters, components, transformations, and use cases. These four aspects were selected because of how they interplay with the fundamental principles of domain architectures that form the basis for out work with domain normalization, as presented in Table 3.

1. Domain growth is chaotic.
2. Domain maturity is orderly.
3. Domain development, in absence of application development, tends to foster maturity, and thus tends to be orderly.
4. Application development, in absence of domain development, tends to foster growth, and thus tends to be chaotic.
5. Domain development, in conjunction with application development exhibits opposing tendencies towards chaos and order.
6. After the initial application is developed, domain development is primarily concerned with maintaining order.
7. Requirements for new application development are the driving force behind domain growth.
8. Domain growth is a direct cost to application development
9. Domain maturity reduces the marginal cost of application development
10. Absorbing new application development into the domain architecture is a reactive approach to domain growth

Table 3 - Fundamental Principles of Domain Architecture Normalization

The significance of these principles leads us to the following observations. A domain architecture that is used for application development will tend to grow and mature. Use cases will expand both in number and in elaboration of alternate path scenarios. Generation of applications from the domain architecture will increasingly rely on parameterization and domain specific languages to manage the increasing complexity of more components. Behavior preserving transformations will be relied upon to provide the flexibility necessary to respond to application development demands and domain architecture restructuring. Furthermore, domain development will be primarily driven by marketing and economic considerations regarding future application development.

Use Case Formalisms

Use case formalisms are the most versatile and essential aspect of domain normalization. The functional characteristics of use cases provide the closest analogy to data normalization and its rigorous relational algebra base. A use case represents a view, or projection of functionality of the domain architecture. A scenario represents an actual instance of that view. Thus, similar to the relational tuple, a scenario captures the persistent state of that functionality. However, it has the capability of extending far beyond capturing data structure by maintaining semantic information with respect to when, how, and why the scenario was invoked. It could be argued by data-centric

purists that this information is merely additional tuples that are captured in a database. However, it is the semantic coupling of process, policy, and data within the domain model that makes this more than the sum of its persistent parts. The ability to reason about this data through coupled behavior is the critical distinguishing feature that separates domain modeling from data modeling within the context of model normalization. We consider the explanation capabilities of the semantically rich domain model to be of the most value.

A simple query against a relational database table such as ‘select * from person where personId = 100’ can be contrasted against ‘select * from new-student-application-for-admittance where personId = 100’. In the former, data about the person is retrieved. In the latter, in addition to the same data, information with respect to the current state of the processing, what the actual process is for this individual, the policies affecting the processing, and the people or roles involved in the processing are available. In order for this type of query to be possible, it becomes necessary to define various levels of domain normalization. These are data normalization, process normalization, policy normalization, and context normalization. A domain model can be a data third normal form, yet not be normalized. There is also a difference between a normalized domain model and a normalized application created from the domain model. Just as reuse of code does not take into consideration multiple executions of the same

If we tried to make a similar comparison to the select, project, Cartesian product and intersection operations, we would fail. A projection deals with what attributes are of interest in a view. The closest analogy to a domain model would be to equate a data attribute to a use case. A selection operation then adds a where clause to the mix. This would be similar to identifying actual scenario invocations of a use case. Since use cases interact with other use cases in both inheritance and dependence hierarchies, there is some possibility to extend the analogy to Cartesian product and intersection by joining multiple use cases together (but this may be going to far). Nonetheless, a use case seems the best candidate for making an analogous comparison from a domain model to a data model with respect to normalization.

The difficulty here is that we are once again dealing on two levels – the intensional level and the extensional level. Extensionally speaking, the analogy may make some sense, but if we deal with the use case as merely data and not behavior, the analogy tends to break down. However a query language that deals with use case formalisms grounded in relational algebra might make some sense. The result of the query would be a piece of functionality of an application. The application would be assembled using a report generation metaphor, that is, as a series of query statements and formatting instructions.

Just as a database matures to contain more data, a domain architecture matures to contain more scenarios, in a sort of vertical expansion of the domain. Additional scenarios fill out a more complete set of alternative paths that may be taken. Similarly, just as additional tables or columns may be added to the database to include more information, additional use cases can be added that provide more features are added by horizontal growth of the domain model. For purposes of domain model operations the vertical growth is referred to as an expansion operation, while the horizontal growth is referred to as an integration operation.

Use case formalisms are primarily concerned with process normalization. At some point in the normalization process, all scenarios map to a set of components. This mapping is exemplified by weak coupling between components in the form of association and collaboration object-oriented constructs. Such constructs include delegation, instantiation, navigation, accessors, notification, externalization, and negotiation. Although each use case step is can be parameterized, it is generally more practical to identify use case dialog splice points where several use case alternate

paths originate or converge back into the main path. Normalization typically requires a use case to begin and end with the same step/state and each step must be an atomic indivisible function. Exception handling, by definition is not required to conform to these influences. Algorithms for minimization of event states and simplifying use case dialog regular expressions provide the mechanism for identifying this splice points.

The decomposition of processes into normal form deals with several interrelated properties. Each use case step represents an atomic action. This action can either be completely independent or it is dependent on another action. If it s dependent upon another action, the dependency can be either in the form of a controlling action or in the form of an influencing action. Likewise the action can exert a controlling or influencing effect on another action. Such influence is referred to as an interfering action. Such interference is typically expressed through temporal logic. For example, a blocking action would stipulate that if one action is currently in progress, then another action cannot occur. This is referred to as a 'prevention' in [DDB93]. Another interfering action is an 'obligation,' which stipulates that an action has to occur if the conditions are true. An obligation becomes an 'exclusive obligation' if the action can occur if and only if the conditions are true. The full range of interfering actions provide for constraints with respect to imposed serialization, parallel execution, and decision preservation. Decision preservation with respect to use cases involves scenario determination. A use case process must be known a priori, however the specific scenario typically is not known, but can be probabilistically determined. Decision preservation captures the identity of the actual scenario instance invoked as well as providing an explanation facility for the scenario selection.

Parameterization

As functionality increases (i.e. the number of use cases), the number of parameters needed to configure a system will also increase. In order to balance this trend, parameters must be generated that capture more information of choices. Focus must shift from similarity to differences. This is accomplished though the concept of refinement. As a domain architecture is utilized to implement many applications, patterns emerge as to similar choices that are made, these collections of choices need to be aggregated into refined architectures in order to simplify the choices available. Thus by automating the process of managing similarity, the user can focus only on differences in configuring a system from the domain architecture. High level choices must be capable of rapidly configuring large numbers of components without the burden of having to deal with minor variation.

Thus, new functionality increases parameterization, while new applications combined with domain architecture operations will reduce parameterization. The former is referred to as an extension operation, the latter as a refinement operation. This is a fundamental definition of normalization – i.e. the reduction of complexity, inconsistency, and complexity.

Parameterization is primarily concerned with policy normalization. Policies are ultimately represented formally through if/then/else constructs, derivations, structural assertions, and action assertions. The normalization process identifies and isolates them in such a manner that parameters can be attached to each rule at the component level and therefore influence behavior within a component. Similar to use cases, rules are structured in template form. Furthermore, hierarchies of rules in decision tree format can be created. Parameterization of rules does not preclude implementing policy as functional logic. Custom interpreted code that can parse and execute of rules dynamically is still possible, if necessary. However, this strategy represents a denormalized implementation of rules that is appropriate in limited contexts such as prototyping.

Externalization and meta-object queries can also be used to dynamically attach parameters to components in a manner that provides a slightly greater degree of normalization.

Transformations

Transformations are of two types –those that increase the domain of possible representations and those that reduce the number. The former is referred to as restructuring, the latter is referred to as evolving transformations. Evolutionary transformations are one measure of a domain model's maturity. Typically, this means progression from white-box, inheritance, and generative based approach to a black-box, aggregation, and compositional based approach.

Transformations are primarily concerned with actual deployment of applications from the domain architecture. More precisely, transformations can change deployment to different platforms Although restructuring and evolutionary transformations deal with design issues, such issues are only considered within the context of creating an executable. When methods and data are factored into parent classes through transformations, we focus on object-oriented concepts that exhibit strong coupling among classes usually found in inheritance and aggregation hierarchies. In particular, the use of generalization, specialization, overriding, extension, overloading, and restricting tends to create deployable modules that vary along a continuum of mutability that can range from very stable to highly volatile. As evolutionary transformations increasingly utilize delegation in lieu of inheritance, adaptable and adaptive software techniques allow closing off the component which can significantly alter the contents of deployable modules. Many ofthese change are in the realm of domain architecture meta-operations such as abstracting, migrating, refactoring, and encapsulating. Hence, transformations foster context normalization by emphasizing maturity and stability in the partitioning of deployable modules through discrimination of the stable and variant parts of the domain architecture. Context normalization of ultimately represents normalization of classes, processes, and policies across application instances. It touches upon the areas of version control, configuration management, and environment space. Through context normalization, we can create any version of application.

Componentization

Components play a crucial role in domain architecture since they form the encapsulated modules of reusability. Again there are two forces at work when dealing with components. The first is the trend towards framework completion. This means the filling in of variable behavior with pluggable modules. The second opposing trend is to segregate existing functionality into smaller modules of reusability to address finer grained specifications. This opposing trend recognizes the need to provide additional variable, or hot spots, where behavior was deemed to have been invariant or well defined. This type of domain architecture operation represents the 'semantic earthquake' that shakes up an existing perspective or visions of the domain that no longer adequately provides a rich enough description of the real world that it is modeling.

Componentization is primarily concerned with structural normalization. The simplest component consists of a single object. Taken to the extreme of decomposition, all attributes, functions, and rules can be defined as their own objects. However, for practical purposes, we consider such decomposition to constitute 'sub atomic particles' that should not exist in isolation within the domain architecture. We draw an analogy to protons, electrons, and neutrons of basic attribute objects. As part of our definition of basic component building blocks, we apply the Law of Demeter. This means that no object (which for our purposes extends to its class and/or type) has knowledge of any other objects that are not explicitly defined as member attributes or as arguments to its methods. Although a distinction among the concepts of composition, aggregation, and association

is not required, the general rule is that member attributes should be reserved for aggregation and composition, while method arguments should be reserved for collaborative associations.

Although it is conceivable to create basic objects that do not contain any data attributes, the most dominant and useful primitive atomic object is the data attribute. It encapsulates the storage of a value that is made available through accessor functions. Additionally rules stipulate valid values, presentation, and authorization. Thus, as elementary as this type of object appears to be, it still represents both of the basic structural properties. That is, encapsulation and separation of three basic responsibilities. Encapsulation requires the elimination of all global variables. Responsibility for knowing also requires that all data be encapsulated in objects that conform to a class/type definition. Responsibility for doing requires that accessor methods are utilized to set and retrieve values. Responsibility for enforcing requires that validation be performed by the attribute object.

In summary, domain normalization operations, meaning those operations that transform a domain model into normalized form, address four basic domain modeling aspects. Each of these aspects exhibits a tendency either towards chaos or towards order. Whenever then domain model grows, then tendency is towards chaos. The danger lies in too much growth to quickly. In such an environment, chaos can overwhelm to domain model to the point that it becomes unusable. At the other end of the spectrum, a maturing domain model tends towards order. Similar dangers lie in this extreme also. Too much order is symptomatic of a problem domain that is either too small or too unappealing as to warrant attention. The driving force here is to consider the fractal nature of domain modeling. Smaller domains should become integrated into larger domain through integration operations. The problem space of a mature domain should be subsumed by the problem space of progressively larger domains until the concerns of the entire enterprise are addressed. At this level, the domain model and the enterprise model are merely different view on the same problem space.

		Operation Bias	
Mechanism	**Aspect**	**Order**	**Chaos**
Use Case Formalisms	Process	Expansion	Integration
Componentization	Structure	Completion	Stratification
Parameterization	Rules	Refinement	Extension
Transformation	Context	Evolution	Reconstruction

Table 4 – Domain Normalization Operations

Operation	Effect	Descriptions
Expansion	Increase number of scenarios – (use case depth)	Adding new features to a domain model by providing distinctive new scenarios to existing use cases
Integration	Increases number of use cases – (use case breadth)	adding new features to a domain model by increasing the number of use cases that directly interact with actors at the boundary of the domain being modeled. These additional use cases are the result of either merging previously separate domains together, or by adding new use cases to the problem space
Refinement	Increase descriptive power or parameters	Creating a more specific framework by clustering sets of related components together that can be recalled by a single parameter or domain specific language construct. The intent is to reduce the quantity of decisions required to implement an application from the domain architecture.
Extension	Increases number of parameters	Creating a more generalized framework by introducing parameters or variability through domain specific language constructs. The intent is to increase the scope of the problem domain by leveraging existing components to perform more generic functionality at the expense of more effort to configure the components.
Evolution	Increases stability of components	Creating a more mature domain architecture by sealing off variability normally provided through white-box, inheritance based framework mechanisms, through transformations that replace these mechanisms with black-box, aggregation based framework mechanisms.
Reconstruction	Increases volatility of components	Creating a more volatile domain architecture by increasing the number of alternative representations or implementations that can be selected to implement a given application as the result of behavior preserving transformations.
Completion	Increases number of components - breadth	Adding concrete components or modules that provide the full range of functionality needed for a predefined hot spot in the domain model, thus potentially reducing the amount of new code that would have to be generated for any new application.
Stratification	Increases number of components – depth	Partitioning existing concrete components or modules by defining new hot spots where none previously existed, in effect adding additional layering of domain subsets that can be selected or customized to implement an application.
Absorption	n/a	A composite operation that result from reintegrating instantiated frameworks or applications with the domain model. This is typically the result of independent project efforts to deliver an application that are later identified as candidates for reuse by the domain model and architecture.

Table 5 – Domain Normalization Operations

Normal Forms

We begin our discussion of domain model normalization by examining what normal form means to a relational database and then extending this notion to an object-oriented model. From there, we extrapolate what constitutes domain normal form. Normalization for object-oriented class structures differs from relational database in several ways. However, corresponding concepts can be found that directly relate one to the other. Relational databases are concerned with relations, tables, tuples, primary keys, and foreign keys. Object-oriented domain models are correspondingly concerned with types, classes, objects, object identity, and associations between classes. Relational databases have several build-in facilities that provide default implementations that must be explicitly declared in a class model. Default sorting order rules for a table column are declared at the server level. Classes can duplicate this functionality by explicitly defining equality and less-than functions. Table 6 details the steps that would translate an object model into third normal form.

Form	Data	Object
1NF	Move data into separate tables Define a primary key	Move objects into classes Creating object instances that each have a unique object Id. Define equality and less-than test methods that conform to a candidate key
2NF	Separate data that's only dependent on a part of the key	Factor into a different class attributes that are only dependent on part of the candidate key; create an association
3NF	Separate date that doesn't depend solely on the primary key.	Factor into a different class attributes that exhibit transitive dependency; create an association

Table 6 – Applying Normal Form to an Object-oriented model

Our definition of domain normalization consists of four aspects –structure, process, policy, and context. In one respect, domain normalization measures the maturity of a business domain-specific architecture. In contrast to maturity models, such as the SEI Capability Maturity Model, domain normalization is only concerned with properties of the domain model itself, not with the process of using it. The significance of understanding the normalization level of a domain architecture comes into play when performing fit assessments and change costing analysis for implementing a new application from the domain architecture. Certain expectations can be made when operating with a model at various normalization levels with respect to estimating effort and expense.

A database model can be designed in third normal form, but implemented in a denormalized fashion to balance integrity and performance. The same holds true for the domain model and its implementation. We are ultimately concerned with what level of normalization the actual domain architecture is in, not what the design may have intended it to be.

As with a normalized database, a normalized domain architecture should be able to have simple, powerful operations applied to it to create result sets. In a relational database, these results will yield a data set. In the context of a domain architecture, the result yields an application set. Just as the data returned from a relational database query may have to be structured for screen display or report printout, the application set returned from a domain architecture query typically needs to be formatted to become an executable.

First Normal Form

As shown in Table 6, establishing first normal form involves two tasks. First, we need to separate elements into affinity groups. We generalize this task by describing it in this format:

```
move <affinity-group-elements> into separate <affinity-group-domains>
```

Table 7 summarizes the implications of this task for each of the four domain model aspects. The second task is to define a primary key for each affinity group element. We need to identify those characteristics that make each element within each affinity groups extent (i.e. domain) unique. For a table, this represents one or more columns. For a class, this represents one or more member attributes. Establishing uniqueness is a relatively straightforward mechanism, usually established with an identity value or scoping within unique namespaces. In a single execution space, object identity is relatively straightforward. However, this can become more complicated with implementation strategies such as the flyweight pattern, which manages the same object and references counts to it by other objects. Distribution also makes object identity more obscure, since parts of the instance may reside on different processors and some of those parts may be replicated, and thus be denormalized.

Aspect	Move…	Into separate…	That distinguish unique…	Represented as…
Relational	Attributes	Relations/tables	Entities	Tuples
OOAD	Attributes/methods	Classes/types	Entities	Objects
Process	Transactions	Use cases	Actors roles	Scenarios
Structure	Collaborations	Components	Operations	Functionality
Policy	Rules	Business Rule Statements	Policy Decision Parameters	Assertions & Derivations
Context	Deployments	Transformations	Applications	Modules

Table 7 – Affinity Groups for First Normal Forms

By their nature, use cases and policies are analysis constructs, components are design constructs, and transformations are implementation constructs. As just described, design and deployment issues presents special problems for comprehending what uniqueness as well as separation into separate affinity groups represents. By definition, a use case represents a series of transactions between an actor and the system. Thus, first normal domain process form represents delineation of the system boundary, naming the actors, and naming the sequences of transactions that represent a distinct use of the system. A use case model typically provides this information, but the presence of a use case model does not assure first normal form unless roles have been abstracted for the actors.

A business policy decision becomes manifest through business rule statements, which are then formalized through assertions and derivations. First normal domain policy form represents the clustering of such assertions and derivations into unique decision parameters. In this form, it is possible that an assertion or derivation can support multiple decisions. For example, a derivation may round all monetary calculations upward to the next penny. Such a derivation may represent a single policy or several policy decisions, such as for charging interest on outstanding balances or for calculating quantity discounts. In order to capture the correct intention, decision parameters are required to explicitly define the scope of each policy. Since such decision granularity is itself configurable for each application instance generated from the domain architecture, first normal domain policy form represents the clustering of such assertions and derivations into unique decision parameters at the highest possible level of commonality.

Instances of a component in first normal form represent alternative collaborative and algorithmic approaches to address the desired functionality. For example, the primary functionality may be to generate the grade point average for a particular student. One component may merely perform the

calculation and return the value, another may make available a semester by semester breakdown with an additional breakout for classes that apply to the major. A query for determining grade point average will return both components, but one will likely be a better match. A similar example can be presented for describing how first normal form affects the context aspect. When creating an application, a number of transformations may yield a functionally equivalent executables capable of being deployed. One transformation may be able to convert a module for deployment on several operating system platforms, such as an OLE object on a client machine, a C++ object module for linking into an executable residing on an application server, or as procedural SQL code residing in a database engine.

Second Normal Form

Second normal form requires that the separation of attributes into separate entities when those attributes are only dependent on a part of the key. This takes the form of:

```
separate an <affinity-group-element> from the <affinity-group-domain> if
it does not depend fully on the identity of the <affinity-group-domain>
```

Since process and policy constructs are the primary forces for an analysis effort needed to expand the domain model, they become more vulnerable to major restructuring than the structurally oriented components and the context oriented transformations. Specifically, actors can be subsumed by the system. Also, derivations, once considered unique, can be found to share discovered generalization. It is in second normal form that an interplay between process and policy first occurs.

Second normal form reduces a considerable amount of this potential volatility. For use cases, this means that redundancy caused by recurring transactions must be factored out into ‘uses’ and ‘extends’ use case stereotypes. To perform this separation, it is necessary to identify the key events that delimit each transaction. We thus separate a transaction from the use case if it can be triggered by an event that does not depend fully on the role that the actor is playing. If another actor can trigger the transaction, then the ‘uses’ and ‘extends’ use case stereotypes are required to convert the domain model into second normal process form.

Identification of these key events is represented in the policy context of the domain model as action controlling assertions that define the conditions for commencing a transaction. Furthermore, these assertions also dictate which of the scenarios for a given use case actually gets implemented. Consequently, second policy normal form requires a correspondence between use case transactions and action assertions though these key event definitions.

Second policy normal form also requires that we separate a rule from the business rule statement if it does not fully depend on the policy decision that forms the basis for the statement. In other words, if we parameterize the policy decision, the rule that gets invoked must be exclusively determined by that parameter. This implies that a rule can be specified independent of its actual use in any policy decision. This is particularly important for establishing key events. In general, policies map processes to components in order to establish a context for an application instance derived from the domain architecture. Therefore, policies represent the domain/extent/range of choices that can be made. As a analogy, consider each business rule as an automotive parts supplier, each derivation and assertion of a business rule as an individual part made by the supplier, and each application as a automobile. Consider each business policy decision as represented by an auto part list to build a component of the car, such as for the ignition system, cruise control, or headlight assembly. To implement this policy, the automobile manufacturer orders the specified parts from each supplier that has an item of the parts list. Similarly, we assemble our application from policies that call up the appropriate business rules resulting in a

collection of formal assertions and derivations. Just as an automobile manufacturer can assemble a seemingly identical car by purchasing parts from alternate vendors, applications can be composed and generated from alternate business rules that implement policy decisions. In each case, performance, reliability, and quality may vary.

The purpose of this example is to illustrate the importance of second policy normal form in providing the glue that binds the other three aspects together. It allows us to consider domain application decisions that have not yet been addressed through process or structural constructs. As a result, meeting client requirements through fit assessment, exception handling, and change costing activities are easier to recognize and accurately predict.

Second structural normal form requires that we separate any collaboration of objects from the component if it is not required to support the complete set of public methods of the component. This is a straightforward definition that suggests that collaborations should be nested inside sub-components. There are exceptions to this rule that deal with objects that play multiple roles, but this deals with context rather than structure. Second structural normal form permits redundant classes that separate out different roles. Second context normal form, on the other hand, must make the distinction of separating a deployment from the transformation if it does not depend fully on the identity of the application. We provided an example of how one transformation may be able to convert a module for deployment on several operating system platforms to illustrate first context normal form. To move that same example into second normal form, the deployable modules must separate out the language aspect (i.e. C++, SQL code) and the platform aspect (i.e. client, application server, database engine). Thus we are capable of capturing deployment options with respect to language and environment that can be made available in latent policy decisions.

Third Normal Form

Third normal form requires that the separation of attributes into separate entities that exhibit transitive dependencies. This takes the form of:

```
separate an <affinity-group-element> from the <affinity-group-domain> if
it is transitively dependent upon the <affinity-group-domain>
```

Through use case refinement, we identify nested transactions based on speech acts in which the performer becomes the customer in the decomposition. This represents a change in the system boundary. Such subsystems are intended to align with the structural aspects of the domain model manifested through components. Thus third normal form represents unification of the process and structural aspects of the domain model through representations such as sequence diagrams, where use case steps are represented as message invocations and components are represented as the message targets. Use case refinements demonstrate transitive dependency by exposing details previously unknown. This could equate to the sales invoice for an automobile which include the basic information of the sale, including total retail price, but refinement reveals itemization of the options and their cost. Collaboration refinement provides the same separation for the structural aspect of the domain model normalization process. Since third normal form requires a one-to-one correspondence between the use case scenarios and collaborations within and among components, it is likely that the selection of components will be accompanied by attached use cases. This presents its own unique problems that must be resolved by the policy aspect of the domain model. Policy may dictate that unneeded refinement is removed. It may, on the other hand stipulate that the additional functional capabilities be included as influencing conditions for application deployment, but not required. This gives rise to the notion of anti-scenarios, unwarranted scenarios, and under-specified scenarios. In the case of anti-scenarios, policy stipulates behavior that the system must not demonstrate. Unwarranted scenarios describes functionality that exists,

but is not required, therefore no assurance as to reliability, quality, or future availability are made. Under-specified scenarios occur when structural and process normalizations identify required policy decision parameters that have not been specified.

Third policy normal form carries the symmetry of a unified structural and process model to completion by associating a policy decision parameter at each component boundary. Since process and structural refinements are by definition transitively dependent, associated policy decision parameters which refine previous, larger grained parameters are likewise transitively dependent. Thus, in order to place the domain model in third normal form, the set of parameters must be separated along the same sub-system boundaries that are represented as actor/system boundaries in the process view and as inter-component communication in the structural view. Once such refinement has been made, pre-defined architectural templates can be specified by attaching default values to parameters. These mega-parameters may still exhibit transitive dependencies since the degrees of freedom for the parameter value may be restricted by other parameter elections made.

When considering the implications of third policy normal form, we must keep in mind that we are not assured a minimally configurable implementation. By this we mean that no assurances can be made that a fit assessment will reveal this most efficient implementation strategy. All that can be guaranteed is that the domain model will demonstrate certain properties that make it easier to identify a satisificing set of policy decision parameters coupled with domain model operations needed to grow the domain model to a level necessary to meet application implementation requirements.

Third context normal form requires that unless a deployable module is directly associated with another module, then any deployment specific code must be removed. For example, in a client-server architecture, a middle layer may be required to connect the client to a SQL database engine, such as through ODBC drivers, or a native database engine driver encapsulated in an OLE object. In third context normal form, the client cannot contain any engine specific syntax, such as pass through queries.

Additional Normal Forms

Fourth and Fifth Policy Normal form are patterned after the corresponding relational normal forms. Fourth relational normal form deals with elimination of redundancy that is caused by having multiple 1:N relationships in the same table. For example, a table consisting of {person, food, wine} should be split into two separate tables consisting of {person, food} and {person, wine}. This requires understanding the meaning of the relation. Fifth relational normal form is established when a relation cannot be reconstructed by a join operation of two or more smaller relations. This occurs only when there is a symmetric constraint on the data values. Both of these situations are most likely to occur when dealing with policy decision parameters. When the mega-parameter exhibit symmetric constraint imposed by sets of parameters, resolving to fifth normal form removes any remain redundancy imposed by such constraints. Policy is the only aspect that is of interest with these higher relational normal forms in our treatment of domain normaliztion. Therefore, we symmetric constraints are addresses in the fourth normal policy (rule) form. Table 8 identifies how each of the aspects relate to arrive at a particular domain normal form. This table does not preclude the relaxation of normal form in one aspect that is compensated for in another aspect.

Form	Structure	Process	Rules	Context
1DNF	1SNF	1PNF	1RNF	1CNF
2DNF	2SNF	2PNF	2RNF	2CNF

3DNF	3SNF	3PNF	3RNF	3CNF
4DNF	3SNF	3PNF	4RNF	3CNF

Table 8 – Domain Normal Forms

5. Conclusions and Future Work

This paper introduced the concept of domain normalization which integrates structural, process, policy, and contextual aspects of a business domain-specific architecture. The motivation behind this effort was to provide a foundation for managing the evolution for such a domain architecture. The author recognizes that considerable refinement in this research direction still needs to be done. However, the concepts presented have demonstrated usefulness on actual development projects in providing a context for evolving a domain architecture. In particular, the domain normalization operations have contributed to a more stable use case model by making a clearer distinction between elaboration of existing use cases and discovering new use cases. Also, guidance for strategic use of parameterization enabled project management to more easily quantify costs associated with the incorporation of anticipated flexibility.

Future research will focus on aligning the concept of domain normalization and domain operations with the semantics of the Unified Modeling Language version 1.1 [UML97]. In particular, a more formalized representation for Classifier compartments with respect to business rules, use case variations, and workflow oriented speech acts is planned. Additionally, plans include the incorporation of domain normalization into a larger conceptual framework for managing the evolution of a business domain-specific architecture. This large framework will include a fit assessment model, a change costing model, design patterns for evolving the domain architecture, transformations of use case formalisms, exception handling for workflow breakdowns, and visual programming constructs for assembling an application from a domain architecture. Is it anticipated that the principles promoted by the concept of domain normalization with provide the conceptual underpinning for that will guide and/or constrain the use of each of these conceptual components.

Russ Hurlbut is a principal of Expertech, Ltd., an information systems consulting organization specializing in the development of business domain-specific architectures. He has taught graduate courses in Object Oriented Development and Technology Transfer at the Illinois Institute of Technology. This paper represents one aspect of his research towards his Ph.D. Thesis "Managing Business Domain Architectures through Use Case Formalisms."

References

[AR94] A. Al-Yasiri, M. Ramachandran, "Developing software systems with domain oriented reuse," Proceedings of the 20th EUROMICRO Conference. EUROMICRO 94. System Architecture and Integration, 5-8 Sept. 1994, Liverpool, UK, p. 133-9, ISBN: 0 8186 6430 4

[DDB93] Eric Dubois, Phillippe Du Bois and Michael Petit, "O-O Requirements Analysis: an Agent Perspective", ECOOP '93 - Object-Oriented Programming 7th European Conference, Kaiserland, Germany, July 26-30, 1993, Proceedings, pp. 458-81

[dF92b] Dennis de Champeaux, Doug Lea, and Penelope Faure, "Object Oriented System Development," Addison-Wesley, Reading Mass, 1993

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Pattterns: Elements of Reusable Software", Addison-Wesley ISBN 0-201-63361-2, 1994

[GM95] Dipayan Gangopadhyay and Subrata Mitra, "Understanding Frameworks by Exploration of Exemplars," In Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95), IEEE Computer Society Press, ISBN 0-8186-7078-9, July 1995, pp. 90-99, ftp://www.pt.hk-r.se/~michaelm/Case95.Final.ps

[Gold85] Robert C. Goldstein, "Database - Technology and Management," John Wiley & Sons, New York, 1985, ISBN 0-471-88737-4

[Hink94] Bob Hinkle, "Reflective Programming in Smalltalk-80," Tutorial, OOPSLA '94

[Hurs95] Walter Hürsch. "Maintaining Behavior and Consistency of Object-Oriented Systems during Evolution." PhD thesis, College of Computer Science, Northeastern University, Boston, MA, August 1995. 331 pages.

[KL95] Geroge Koch and Kevin Loney, "Oracle – The Complete Reference," Third Edition, Osborne McGraw-Hill, Berkeley, California, 1995, ISBN 0-07-882097-9

[Lieb96] Karl Lieberherr, "Chapter 2, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns," PWS Publishing Company, 1996, ISBN 0-534-94602-X

[Oops95b] "PANEL: Objects and Domain Engineering," OOPSLA '95, Austin, Texas, 1995, p. 333-336

[Opdy92] William P. Opdyke, "Refactoring Object-Oriented Frameworks.", Ph.D. Thesis, University of Illinois at Urbana-Champaign 1992

[Pree94] Wolfgang Pree, Design Patterns for Object-Oriented Software Development, ACM Press, New York, 1994, ISBN 1-201-42294

[UML97] Unified Modeling Language, Version 1.1, http://www.rational.com/uml