Constructing the Domain:

A Pattern Language for Object Oriented Software Design

(Draft, July 15, 1999)

Eric Evans, evans@acm.org

Table of Contents

Introduction

It is not so obvious how to build object-oriented software that works. The domain layer is particularly vital and deceptively simple-looking. Representing the domain with objects can be so intuitive that it lulls people into not noticing the design challenges that remain. Developers often start naively implementing an analysis model, which leads to unmanageable webs of objects that don't do much work for the application. In the end they may take shortcuts to produce some of the functionality they need, losing in the process any benefits an object model might have provided.

This is an attempt to use pattern language to bridge that gap, to show how to construct practical object software faithful to a conceptual domain model.

The basic demands on the design of the domain layer are to express the model, to provide access to that expression, and to encapsulate parts of the domain design that are not being fully exposed for various reasons. In order to pull this off, the domain must be isolated from the tug of responsibilities that belong to other parts of the system. Also, the model must be of a form that accommodates practical design considerations.

These patterns show how to meet those demands. The result is a design for software components that are good vehicles for a powerful and elegant domain model and that perform well in the deployment environment.

Modeling the Domain

What is the "domain", anyway? At its base, the domain is the realm of knowledge or activity of interest to the users of the system we are designing. It may be a part of the real world, or it may be a more abstract subject. A shipping software system is in some way related to the real world of shipping. An accounting program is in some way related to the not-so-real world of money. These problem domains usually have little to do with software. (An exception would be a CASE tool, whose problem domain is software design itself.)

Models capture abstractions. An abstract model is a simplification of the reality it represents. It should capture the aspects of the domain that are relevant to solving the problem at hand and ignore extraneous detail. When we abstract the problem domain into a model, we call that the "domain model" or sometimes the "analysis model".

The 18th century Chinese map on the previous page represents the whole world. In the center and taking up most of the space is China, surrounded by perfunctory representations of other countries. This was a model of the world appropriate to a society turned inward. Every model represents some aspect of reality or of an idea that is of use or interest to the user of the model.

MODEL DRIVEN DESIGN

Models are the crystallization of domain knowledge. How can we transmit more of that knowledge into our software?

If the design, or some central part of it, does not map to the conceptual domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between conceptual models and design functions are difficult to understand, and, in practice, impossible to maintain as the design changes.

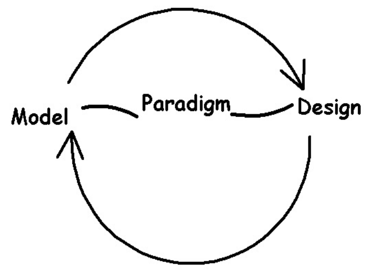
The design has to specify a set of components that can be constructed in the programming language we use, that will perform efficiently in our deployment environment, that will correctly solve the problems posed by the application coordinators. It should also be easily understood and maintained.

Design a portion of the software system to reflect the conceptual model in a very literal way, so that mapping is obvious. Revisit the model and modify it to more naturally be implemented in software.

The map is not the territory. The map is a model of the territory. There can be many models of the same territory and, in a few cases, the same model can represent more than one territory. The domain model is neither the business domain itself nor the software that solves problems in that domain. But models can represent either of those things. The ideal of model driven design is that the same model can be used to represent the problem domain and the software design.

There is always more than one way of abstracting a domain. There is always more than one design that could solve a stated problem. The desire to closely relate the analysis model to the design suggests one more criterion for choosing the more useful models out of the universe of possible models. This statement cuts two ways. The design model should retain as many of the conceptual qualities of the analysis model as possible, while the analysis model should be chosen such that a practical design model can be created that corresponds to it. When the same model reflects both analysis and design, I refer to it as the conceptual model or simply the domain model. (Actually, "conceptual model" is redundant. A model is, by definition, conceptual. But it emphasizes the elimination of mechanistic detail from the design.)

To make such a close correspondence of model and design possible, they both have be developed in a modeling paradigm that has software tools that directly support. (Objects are the paradigm used in the rest of this paper.)



The conceptual model provides the language of the domain and the basic assignment of responsibilities. The design should not invent new language. The objects in the design (aside from purely technical issues) should be described in terms of the conceptual model and should fulfill some role necessitated by it.

It is not always possible to make the design a mirror of the conceptual model, but those cases provide the opportunity to create the most lucid and powerful software.

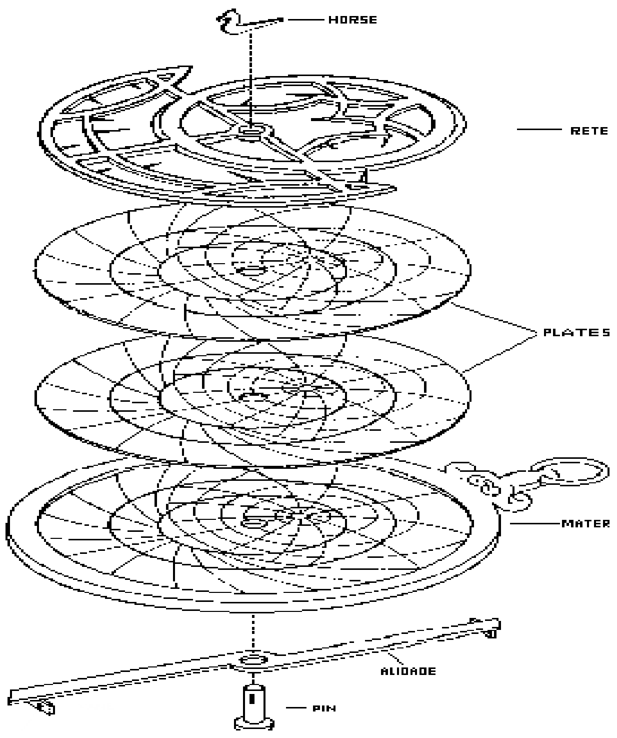
Really, the only way to arrive at this happy confluence of concept and design is through an iterative process running all the way from analysis through design and back again to analysis and back again to design, reabstracting the models on each pass until names and relationships are found that give both a powerful way of looking at the problem domain and a powerful way of solving problems in that domain. Then you have a conceptual domain model that can be the basis of good software.

Development process is beyond the scope of this article, but a compatible discussion of process applied to this problem can be found in [Cockburn1998], in the section on Domain Modeling and Reuse. It is also beyond the scope of this article to explain how to model well. The focus here

is on how to design software that holds onto the power inherent in a good model. Even so, I will briefly summarize some hallmarks of such models.

The integrity of the domain must be protected using ISOLATED DOMAIN LAYER. The model and design must be in the context of a matching paradigm. The one treated here is the DOMAIN OBJECT MODEL…

DOMAIN OBJECT MODEL



…In the context of MODEL DRIVEN DESIGN, this pattern introduces the paradigm of object-oriented design.

It is difficult to obtain the benefits of a domain model if the design and implementation do not closely correspond to that model, but traditional software design methods did not support model driven design. Even when object languages are used for implementation, the domain model is not usually effectively reflected in the implementation, and much of the value of the object paradigm is lost.

The versatility of domain design in responding to changing requirements is directly related to the degree to which the design expresses the domain model, but conventional procedural programming does not support this very well. Just about the only kind of model that can be represented in a procedural language is a function, which is why FORTRAN has been so successful in computational problem domains.

But most software is not based primarily on computational. Procedural languages often support complex data types that begin to correspond to more natural conceptions of the domain, but they are only organized data, and don't capture any active aspects of the domain.

The result is that software is written as complicated functions linked together based on anticipated paths of execution, rather than by conceptual connections in the model.

Model the concepts in the domain as a collection of interacting objects. Then, using an object language, design and implement software objects that express the model very literally.

Medieval astrologers devised an instrument called an astrolabe. It was an intricate arrangement of rotating rings and disks that could be used to compute and predict the position in the sky of the sun and planets. A plumb line represented the trajectory of a celestial body against the sky, which was represented by one or more disks. Their mechanical model was later supplanted by more sophisticated physical, mathematical models by Copernicus and Newton. But they served well in their time and for the purposes of their designers, and it is an early example of an object model.

An object model captures a concept in the form of a set of elements, called objects, and relationships between those objects. Activity is modeled as operations directly attached to and encapsulated by the involved objects. It turns out that a lot of common and important domains can be modeled naturally as interactions among such semi-autonomous objects. The language of a domain may generate and object model where nouns become objects and verbs become operations and interactions of those objects. (This is much too superficial a view of object modeling, but that is a huge topic, way beyond the scope of this paper.)

Much of the power of object-oriented software comes from the ability it gives us to have a particularly close relationship between the domain model, domain design and domain implementation. If a domain is modeled as objects, then object languages allow abstractions to be expressed explicitly, so that code can be written that corresponds very closely to the model. This is why modeling is so important in object-oriented development – more important, perhaps, than in conventional programming. And those models must be of a particular nature to support design.

Unfortunately, the use of an object-oriented language does not, inherently, make this problem go away.

A bunch of ad hoc objects sending messages back and forth is not a model, although the diagrams can look a lot alike. A useful object model must provide strong abstractions with clear names. The conceptual model provides the language of the domain and the basic assignment of responsibilities. The design should not invent new language. The objects in the design (aside from purely technical issues) should be described in terms of the conceptual model and should fulfill some role necessitated by it.

Domain Object Model addresses a large number of practical software problems. There are, however, domains that are not natural to model as discrete elements. Intensely mathematical domains or domains where global logical reasoning dominates are examples that do not fit well into the object oriented paradigm.

The conceptual domain model is the heart of successful object software development, but a heart is useless without arteries. Given that the domain

model is in this form, the model must be expressed, using ENTITIES and VALUE OBJECTS, access must be provided using REPOSITORIES, and new instances created using FACTORIES.

Before we turn to the means of expressing an object model, we'll take a necessary detour to look at ways of keeping the domain intact in a system with many needs tugging in different directions – THE ISOLATED DOMAIN LAYER and related patterns…

Isolating the Domain

The domain, that part of the software that solves business related problems, problems from the domain, usually constitutes only a small part of the software of a system, though its importance is disproportionate to its size.

Experience of the first two decades of object-design has shown that, in order to implement the domain effectively, it must be decoupled from other functions of the system. Sophisticated techniques for this isolation have developed. This is well trodden ground, but it is so critical to the successful application of the principles of this article that it must be reviewed. I will summarize very briefly.

Defining what is not in the domain also helps to clarify what is in the domain.

ISOLATED DOMAIN LAYER

Developing software involves much analysis and design that does not closely relate to the domain. The design of database architectures and pull-down menu widgets is distinguishable from the design of a shipping logistics package, even though all might be part of the same shipping system. These are not found in any analysis of the problem domain, of course, being purely an artifact of a software program meant to solve some problems in that domain.

So in order to keep that close correspondence between model and design, one of the first steps is to distinguish between the domain and other parts of the system, and find ways of loosely coupling the two so that we can design and implement the domain portion in isolation.

The design pattern generally applied to this problem of system architecture is layering. This is used in some degree in almost all modern software. Many good discussions of layering are available in the literature, sometimes in the format of a pattern as in [BMRSS96], pp. 31-51.

SEPARATING APPLICATION FROM INFRASTRUCTURE

Here is a pattern that is almost universally applied today. Functions such as persistence, user interface presentation and communication (along with many

others) are separated into reusable modules that serve applications. For example, if a domain object needs to send an e-mail, do not build an e-mail interface as part of the domain or an application. Such a server for external resources is an infrastructural responsibility and it should be kept in those other layers. This constitutes a LAYER because the infrastructural modules are not in any way dependent on the application. Some of these modules are included in modern operating systems. Others are included in powerful development environments. Some can be bought off the shelf. A few still have to be developed in-house. They tend to require high technical expertise, but little knowledge of the domain, although the application determines the requirements for the infrastructure, of course.
Examples would be print servers, e-mail servers, equipment automation controllers, or database transaction servers with frameworks for defining units of work.

SEPARATING USER INTERFACE FROM DOMAIN

This separation is often attempted, but, unlike the separation of application and infrastructure, projects frequently fail to apply it properly. It was pioneered in the Smalltalk world with the Model-View-Controller pattern. It is nicely explained by [Larman98] as the MODEL-VIEW SEPARATION PATTERN. It separates the user interface into a layer that is invisible to the domain. To communicate changes from the domain to the UI, a dependency mechanism such as the one in Smalltalk is needed. The OBSERVER pattern of [GHJV95] is a good example.

SEPARATING APPLICATION FROM DOMAIN

If the problem domain is the realm in which the system is expected to solve problems, the applications pose specific problems in that domain and orchestrate their solutions. This subtle distinction was also pioneered in the Smalltalk world.

The solution was to construct an additional component, the APPLICATION COORDINATOR [Larman98] (a.k.a. the Application Model, in the Smalltalk world), which defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems

Sun has not provided such a clear pattern to follow in Java, but the language has all the mechanisms needed to build a framework to do this. Some vendors may provide a form of it and many projects will roll their own.

The details of the framework are not interesting here so much as determining what is and is not in the domain. When this layering is applied, the specific tasks and views the user needs are part of the application while the conceptual model of the business and universal business rules constitute the domain model.

SMART UI ANTI-PATTERN

…That sums up the widely accepted layering patterns for object applications. But this separation of UI, application, and domain is so often attempted and so seldom accomplished that its negation deserves a half-serious discussion in its own right.

A project needs to deliver simple functionality, dominated by data-entry and display with few business rules. Staff is not composed of advanced object modelers.

This pattern says, “Put all the business logic into the user interface”.

Heresy! The gospel (as advocated everywhere, including elsewhere in this paper) is that domain and UI should be separate. In fact, it is difficult to apply any of the methods discussed later in this article without that separation. Therefore, this might be considered an “anti-pattern”, but it isn’t always, and it is important to understand why we want to separate application from domain, even including when we might not want to. In truth, there are advantages to this pattern and situations where it works best, which may partially account for why it is so common.

Advantages

- Productivity is high and immediate for simple applications.
- Less capable developers can work this way with little training.
- Even deficiencies in requirements-analysis can be overcome by releasing a prototype to users and then quickly changing the product to fit their requests.
- Applications are decoupled from each other so that delivery schedules of small modules can be planned relatively accurately, and expansion of system into additional, simple behavior is easy.
- Relational database works well and provides integration at data level.
- 4-GL tools work well.
- When applications are handed off, maintenance programmers will be able to quickly redo portions they can’t figure out since the effects should be localized to the UI being worked on.

Disadvantages

- Integration of applications is impractical except through the database.

- Complexity buries you quickly, so growth path is strictly toward additional simple applications. There is no graceful path to richer behavior.
- There is no reuse of behavior and no abstraction of the business problem.

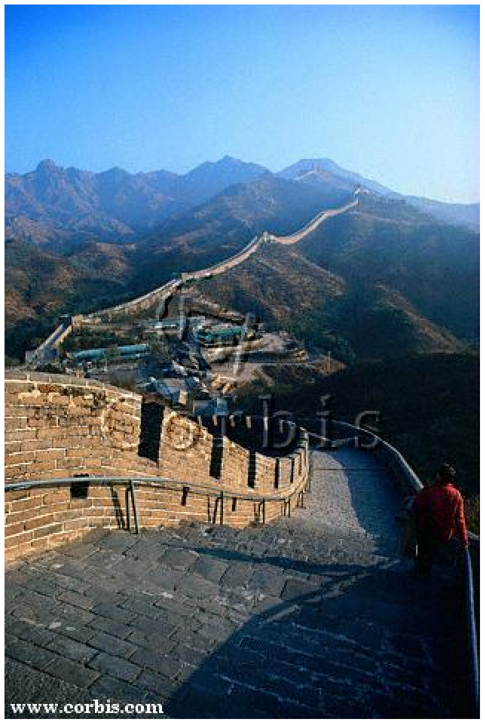
Put all the business logic into the user interface. Chop the application into small functions and implement them as separate user interfaces, embedding the business rules into them. Use a relational database as a shared repository of the data. Use the most automated UI building and visual programming tools available.

If this pattern is applied consciously, a great deal of overhead can be avoided that has to be taken on to attempt other approaches. The biggest mistake is to build an infrastructure and use tools much heavier weight than are needed.

Most true OO systems (such as Java) are overkill for these applications and will cost you dearly. A 4-GL style tool is the way to go. Remember, one of the consequences of this pattern is that you can't migrate to another design approach except by replacement of entire applications, and that integration is only through the database. Therefore, a later attempt to use Java will not be helped very much by the use of Java in the initial development.

This pattern is here to clarify why and when domain partitioning is needed. The lack of partitioning can really be a disaster if applied in an inappropriate setting, but if the project does not have the necessary expertise for the more sophisticated approaches, and if it can meet the other requirements of this pattern, it provides an option. If not, bite the bullet, get the necessary experts and avoid this pattern…

ANTICORRUPTION LAYERS



…Unfortunately, infrastructure, user interface and application are not the only things you need to protect your delicate model from. You must also control the interfaces to other domain components that are not fully integrated into your model.

We face a tension between two forces. In order to be powerful, or even reliable, a model based program must be based on a unified model. Yet, in real large systems it is impractical for all software to be designed (or potentially rewritten) in accordance with a single model. In fact, integrating with existing systems is a valuable form of reuse. When systems based on different models are combined, the need for the new system to adapt to the semantics of the other system can lead to a corruption of the new system's own model. Even when the other system is well designed, it is not based on the <u>same</u> model as the client. And often the other system is not well designed

There are many hurdles in interfacing with an external system. For example, the infrastructure layer must provide the means to communicate with another system that might be on a different platform or use different protocols. The data types of the other system must be translated into those of your system. But often overlooked is the certainty that the other system does not use the same conceptual domain model. This is profoundly important because a model is only useful if it is consistent, and it is unlikely that the other systems model can be unified with the one you are working with. The consequences of trying to combine two models that are not unified seem fairly clear, but this problem tends to sneak up on us because we think that what we are transporting between systems is primitive data whose meaning is unambiguous and must be the same on both sides. This is usually wrong, and even if some of the primitive data elements do mean the same, it is a mistake to make the interface to the other system operate at such a low level.

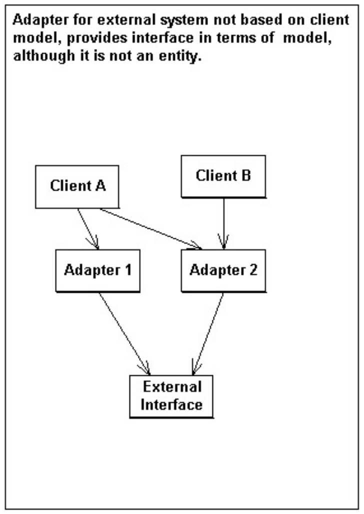
The impossibility of unifying all models is particularly clear when our system uses domain behavior that is provided by an external system, but the issues are the same with a legacy system, or any separate subsystem not based on the same domain model. A means is needed to provide a translation between the parts that adhere to different models, so that the models are not corrupted with undigested elements of foreign models.

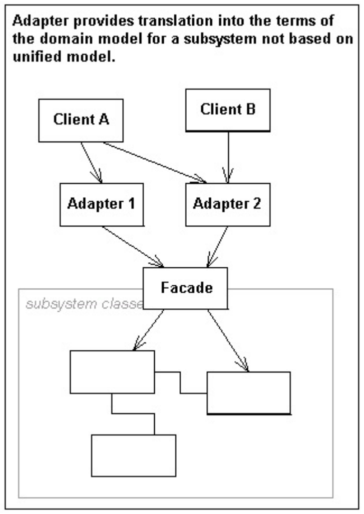
To protect their frontiers from raids by neighboring nomadic warrior tribes, the early Chinese built the great wall. It was not an impenetrable barrier, but it allowed a regulated commerce with neighbors while providing an impediment to invasion and other unwanted influence. For two thousand years it defined a boundary that helped the Chinese agricultural civilization to define itself with less disruption from the chaos outside.

A cautionary note: Although China might not have become so distinct a culture without the Great Wall, the Wall's construction was immensely expensive and bankrupted at least one dynasty, probably contributing to its fall. The benefit of isolation strategies must be balanced against their cost. There is a time to be pragmatic and make measured revisions to the model to make a smoother fit to the foreign ones.

Layers built of Adapters and Facades

Building a whole new layer responsible for the translation between the semantics of the two systems gives us an opportunity to reabstract the other systems behavior and offer its services to system consistently with our conceptual model. The "anti-corruption layer" presents the services and information of the other system through FAÇADES [GHJV95]. (This is a variation on FAÇADE, actually, since this interface is meant to be the only way of reaching the other system.) The conceptual model of the external system may not be easily compatible with our own, but it is important that the interfaces of the facades be in terms of our conceptual model. Therefore, the anti-corruption layer will be more than just a mechanism for sending messages out to the other system, it will be the interface of an ADAPTER that translates conceptual objects and actions from one model to the other. It may not even make sense, in our model, to represent the external system as a single component. It may be best to use multiple FAÇADES, each of which has a coherent responsibility in terms of our model.


Adapter for external system not based on client model, provides interface in terms of model, although it is not an entity.
Client A
Client B
Adapter 1
Adapter 2
External Interface


Adapter provides translation into the terms of the domain model for a subsystem not based on unified model.
Client A
Client B
Adapter 1
Adapter 2
Facade
subsystem classe

There are even situations in which it makes sense to create an independent subsystem and hide it behind an anticorruption layer. You can take a whole functioning chunk of software, complete with its own entities, repositories, and even external interfaces and completely encapsulate it behind a façade. Communication through the façade is in terms of abstractions shared with the outer system, but the internals of the subsystem can use an entire conceptual model of its own that does not have to be reconciled with that of the outer system. Even if the model of the subsystem is completely unified with that of the main system, explicitly reducing the possible collaborations down to the few legitimate ones reduces the overall complexity of the system and makes it easier to understand both pieces and their relationship.

Create an isolating layer using a combination of FAÇADES [GHJV95] and ADAPTERS [GHJV95] to provide clients with functionality in terms of their own domain model.

The interface and access to an ANTICORRUPTION LAYER can be in the form of a SERVICE (presented in a later section)…

SUMMING UP ISOLATION

Separating these layers allows a much cleaner design of each layer, including the domain. It also helps with deployment in a distributed system, by allowing different layers to be placed in different servers or clients, in order to minimize communication overhead and improve performance [Fowler96].

Best of all, now that we have that other stuff out of the way we can really focus on the domain…

Expressing the Model

The DOMAIN OBJECT MODEL pattern states that a very direct correspondence exists between elements of the model and objects in the implementation, so that there are a set of objects that express the model.

The most natural part of object oriented design is to create objects that correspond to the elements of the analysis model. If you have succeeded in bringing your analysis model and design model together, as called for in MODEL DRIVEN DESIGN, you will have identified most of these elements of the design.

The next patterns will delve into the important distinctions between these expressive objects and the problems encountered in designing them.

ENTITIES

…OBJECT MODEL calls for the domain to be conceptualized as interrelated objects. In this pattern we begin to differentiate and design those objects that represent entities with identity.

Some objects have an identity that must be tracked and maintained. These must be distinguished from other objects even if they have exactly the same attributes.

Usually the most prominent objects in the domain turn out to have identity. This is not necessarily the strong sort of identity that we associate with people, cities, cars or computer processes, which involves identifiers of interest to the users. It simply means that we care which object is which, even if all the attributes turn out to be equal. For example, in a banking application, two deposits of the same amount to the same account on the same day are still distinct transactions, so they have identity and are entities. On the other hand, the amount attribute of those two transactions are probably instances of some money object, has no identity, since there is no meaning to distinguishing them. In fact, some objects can have the same identity without having the same attributes, as, for example, an update of a customer record for the same customer.

While object languages have "identity" operations that can determine if two references are to the same object, this is much too fragile. If persistent storage of the objects is not in an object database, when an object is retrieved and a new instance is created, this identity is lost. If objects are transmitted across networks, new instances are created on the destination, and this identity is lost. Even with powerful frameworks that simplify these problems, the fundamental issue exists: How do you know that two objects represent the same conceptual entity?

Actually, the problem can be even worse when multiple versions of the same object can exist in the system, as when updates propagate through a distributed database.

Explicitly flag the objects with identity and operationally define that identity.

Each entity must have an operational way of establishing its identity with another object.

The most common solution is to have a unique id of some kind. Sometimes this is a value meaningful to the user, such as a Social Security number or a confirmation number for an airline reservation. Sometimes it is internal, when the identity has to be tracked, but the user isn't interested that means of lookup, such as an address-book where entries are listed or searched by name or other attributes. The id can be generated by the system, as for the internal id's or the confirmation number, or can be a preexisting identifier, such as Social Security number.

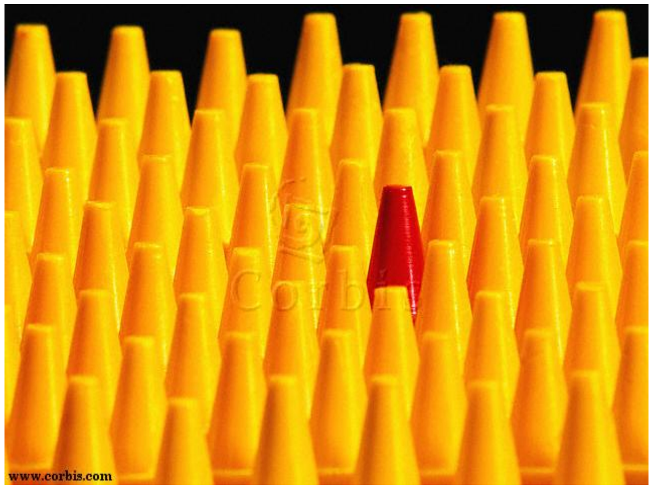
Whatever the visibility or assignment mechanism, the number must be guaranteed to be unique within the system. This can be a challenging even for automatically generated, when distribution or concurrency are involved. It places sever constraints on the kinds of externally assigned identifiers that can be used. The name of a person, for example, is not guaranteed to be unique. Social Security number works pretty well until a child or non-resident of the United States is involved.

Designing associations that involve ENTITIES

Designing entities comes pretty naturally, since it is the most closely related part of the design to the analysis model. However, designing the associations between entities is more involved. Remember that model associations represent different things in the analysis model and in the design model, although the models look the same or very similar because we've worked hard to bring them together. While associations in the analysis model represent conceptual relationships between elements of the problem domain, associations in the design model represent software constructs that can give components access to other components. For example, a simple one-to-one association would probably be designed as a direct reference in an instance variable, while a simple one to many might represent a Set in that instance variable. When associations have multiplicity, collections are called for. Qualified associations translate to a key look-up, such as a hash map.

In the design model, managing references becomes really critical. Many bi-directional associations in the analysis model should be reduced to unidirectional references in the design model. Whereas in the analysis model an association is an association, in the design model we care how that association will be traversed. So we may decide that a certain association can be traversed in one direction directly, but in the other direction will be traversed by a database search (which we'll handle in the section on repositories), or even say that it cannot be traversed (because application requirements don't demand it.) In general, references in the design model are more restrictive and we begin to choose mechanisms for those that are not simple traversal.

VALUE OBJECTS



…DOMAIN OBJECT MODEL calls for the domain to be conceptualized as interrelated objects. In this pattern we continue the design of objects that express the model now considering the lighter weight values.

It is an awful lot of work to manage an object in a model with interdependent objects. You have to analyze the effects of changes or deletions to the object, resolve references when the objects are moved in and out of storage or across a network. In a domain with many objects, this can be overwhelming.

<< lots of overhead tracking identity of objects or the consequences of changing them... Overhead both analytical and performance… Spell it all out.>>

Flag any objects without identity and treat them in the design in ways that free you from the analytical overhead of entities.

Value objects are instantiated to represent aspects of the domain that do not have identity. They represent elements of the design that we care about only for what they are, not who they are. There are many design options to reduce analytical overhead or to improve performance, which cannot be used for entities. By restricting the overhead of entities only to those objects that really need it, we free ourselves to handle the potentially huge number of value objects in simpler or more efficient ways.

A single object is the simplest possible value, and this is a common case (e.g. primitive types), but values are frequently made up of other value objects. They can even reference entities, but they must only do so in a way that does not confer identity on them. For example, an object representing a route from San Francisco to Los Angeles via the Pacific Coast Highway could be a value, even though the three objects (two cities and a highway) it references are all entities.

A "Person" object obviously has identity and is an entity. That person's name is a value. If two people have the same name, that does not make them the same or interchangeable. But the name could be copied from the first person to the second since only the properties of the name matter – it has no identity. Primitive objects, such as strings and numbers, are values.

Conceptually, values can be treated as immutable, meaning they cannot change after creation, since their purpose is to express a particular value.

Value objects are usually used as properties of entities or parameters in messages between them. They are frequently transient, created for an operation or to act as arguments in a message and then discarded. As long as value objects are immutable, change management is relatively simple, since there isn't any except full replacement. Immutable objects can be freely shared, and, if garbage collection is reliable, deletion is simply a matter of dropping all references and ignoring internal structure. Unfortunately, things may not be that simple in practice.

For one thing, sharing may not be a good idea if clustering in the database is a concern. By making a copy, rather than sharing a reference to the same instance, a value that is acting as an attribute of an entity can be stored on the same page as the entity. In a relational database, you may want to put the value in its owning entity's table, rather than creating an association to a table of value objects. In a distributed system, holding a reference to a value on another server is probably inefficient, rather a copy of the whole object should have been passed to the other server.

When an object is asked for one of its attributes, the safest thing to do is to pass a copy. Sharing is best restricted to cases where it is most valuable and least troublesome.

- When saving space or object count in the database is critical.
- When communication overhead is low (e.g. centralized server).
- When the shared object is strictly immutable.

More troublesome, there are even cases when it is best to allow the implementation value to be mutable (able to be modified after its creation), even though it is conceptually a replacement with another value. Forces that would lead to that decision are:

- If the value is frequently changed
- If object creation or deletion is expensive
- If replacement (rather than modification) will disturb clustering
- If there is not much sharing of values, or if such sharing is forgone to improve clustering or for some other reason

Note: If a value's implementation is to be mutable, then it **must not** be shared.

Implement value objects as immutable when you can -- that is unless some of the reasons above are strong. In Java, you can partially enforce immutability of an object by declaring no operations that can change the state of the object, and setting them only from the constructors.

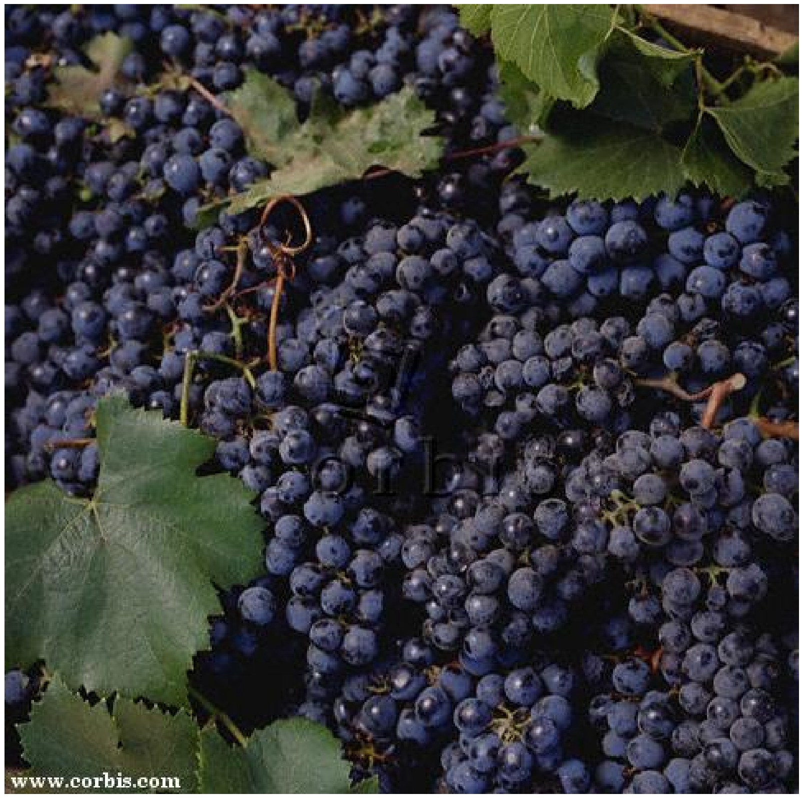
Note that in making these distinctions are being made based on each object's significance in the domain model. This might not reflect the underlying implementation of primitives in your language system. For example, some language systems make only one instance of each integer and essentially identify the integers to maintain this. This is irrelevant here. Integers are values in the domain model.

Designing associations that involve VALUE OBJECTS

The discussion above of associations in the design model apply to value objects, too. But between values bi-directional associations make no sense at all, since, without identity, it is meaningless to say that an object points back to the object that points to it. The most you could say is that it points to an object that is equal to the one pointing to it, but what object is responsible for knowing that? Putting in two associations, one pointing each way, would be possible for value objects, though it is hard to think of examples where it is useful. If something like that seems necessary, you may want to rethink the decision to declare the object a value in the first place.

The mapping of the objects in the conceptual model to entity and value objects in the design should be fairly straight-forward. If it is not, go back and refactor the model. It is important to resist the temptation to add anything to these objects that does not relate closely to the conceptual entity they represent, or relationships that are not part of the model, even if the system will not work without your additions. If you need to revise the model, do it. If you are not expressing the model there are other places for the behavior you need.

AGGREGATES



…Although we now have designed the associations, and have simplified the traversal paths somewhat, we could still trace long, deep paths down through object references.

How do we know where an object made up of other objects begins and ends? It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects.

In any system with persistence in which there is change to the state of expressive domain objects, we need a way of knowing the scope of a transaction. The need is acute in a system with concurrent access. Here I'll adopt an approach developed by David Siegel called an "aggregate". It provides a strong abstraction for the encapsulation of references to entities.

An aggregate is a cluster of associated objects which are intended to be treated as a unit in transactions. Each aggregate has a root and a boundary. The boundary defines what is inside the aggregate. The root is a single specific entity contained in the aggregate, which is the only part of the aggregate outside objects are allowed to hold references to, although objects within the boundary can hold references to each other. Entities other than the root have identity, but it only needs to be relative to the root, since no outside object can ever see it out of the context of the root entity.

An example will clarify. Take a model of a car with four tires. The car has global identity – we want to distinguish that car from all other cars in the world, even very similar ones. The wheels have identity also. There are four wheel positions on the car and we might want to know the rotation history of the tires through those positions. We might want to know mileage and tread wear of each tire. But it is very unlikely that we care about the identity of those tires outside of the context of that particular car. If we replace the tires and send the old ones to a recycling plant, either our software will no longer track them at all, or they will become anonymous members of a heap of tires. No one will care about their rotation history. Likewise, even while they are attached to the car, no one will try to query the system to find a particular tire and then see which car it is on. They will query the database to find a car and then ask it for a transient reference to the tires. Therefore, the car is the root entity of the aggregate whose boundary encloses the tires also. On the other

hand, engine blocks have serial numbers engraved on them and are sometimes tracked independently of the car. In some applications, the engine might be the root of its own aggregate.

Once the aggregate is defined, by identifying the root entity and drawing the boundary, these rules apply:

- Root entities have global identity. Entities inside the boundary have local identity, relevant only within the aggregate.
- Nothing outside the aggregate boundary can hold a reference to anything inside, except to the root entity. The root entity can hand references to the internal entities to other objects, but those objects can only use them transiently, and may not hold onto the reference. The root may hand a copy of a value to another object, and it doesn't matter what happens to it, since it's just a value and no longer will have any association with the aggregate.
- As a corollary to the above rule, only aggregate roots can be obtained directly with database queries. All other objects must be found by traversal of associations.
- Objects within the aggregate can hold references to other aggregate roots.
- A delete operation must remove everything within the aggregate boundary at once. (With garbage collection, this is easy. Since there are no outside references to anything but the root, delete the root and everything else will be collected.)
- When a change to any object within the aggregate boundary is committed, all invariants of the whole aggregate must be satisfied.

Cluster the expressive objects of the domain into “aggregates” and define boundaries around each. Choose one entity to be the “root” of each aggregate, and control all access to the objects inside the boundary through the root. Only allow references to be held to the root. In any state-change, enforce all invariants for objects in the aggregate and for the aggregate as a whole.

Access to Expressive Objects

Newcomers to object oriented software often ask, “where does it start?” They look at the expressive objects and they may see that their structure or behavior has meaning in the domain, but it is hard to see how to build a program. Part of the answer lies in the APPLICATION COORDINATOR, discussed in an earlier section, which defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. But there is still one piece missing: How did the application coordinator get hold of the right expressive components? Those objects must be created and, in systems with persistence, found in and retrieved from the database.

There are patterns for doing both of these things. Objects who’s responsibility is the creation of other objects are called “factories”. There are many kinds of factories and many ways of using them, but collectively they solve the problem of making new objects. Patterns of providing access to already existing objects are less well documented. The one I present here is effective and pretty common and it is called a “Repository”. A repository is a globally accessible object responsible for finding and retrieving an object that has previously been created and stored.

Together, factories and repositories make it possible for the application to get hold of the expressive components, entities and value objects, it needs to solve the business problems posed to it.

REPOSITORIES (AKA REGISTRIES)

…Associations in the OBJECT MODEL allow us to find an object based on its relationship to another. But we have no means of finding the starting point for such a traversal. This pattern addresses access to an ENTITY in the middle of its life-cycle.

The OBJECT MODEL alone provides no means to find an ENTITY based on its identity or its attributes. Without this, there is no starting point for traversal of associations, and, consequently, no preexisting objects can be accessed.

Every object has a life-cycle. It is born, it may go through various states, it eventually dies and is either archived or deleted. The creator of an object presumably has a handle on the newly created object, and we don't need a reference to the deleted object. But in the middle of the life-cycle we need a way of finding a reference to the objects in our domain.

More specifically, we need a way of finding ENTITIES by their identity or attributes. VALUE OBJECTS do not present this problem. Preexisting value objects should be found by traversal from the root ENTITY whose AGGREGATE boundary encapsulates them. A global access to a value is meaningless, as finding a value by its properties would be equivalent to creating a new instance with those properties. (If you find you need to search for a preexisting value, then reconsider if it is really a value.)

Still, there are a few cases where global access to value objects may be needed. One is in implementing the FLYWEIGHT pattern [GHJV95]. Another is to represent an enumeration of all possible instances of a type, through a modified singleton pattern. This would be used typically when an attribute can have one of several values of a particular type, but the value must come from a predetermined set (an enumeration of values). Generally, though, only entities need to be found by their properties in mid-life.

Some *ENTITY* objects do not present this problem because they are always referenced by other entities and the application will not need to find them globally. This is a firm rule for entities that are encapsulated in an aggregate boundary, but is sometimes true of roots as well if there is no application need.

Create a SINGLETON [GHJV95] for each ENTITY type that is the root of an AGGREGATE. The singleton supports queries that find entities by their attributes (including unique keys that define the entity's identity). It provides this through methods that take as arguments the specified values of attributes and return references to instances or iterators (or their equivalent) over collections of instances that have those attributes.

Repositories can provide a variety of queries that return instances or collections of instances according to their properties in any useful combination. They can also return summary information, such as how many instances meet some criteria or even the sum of some numerical attribute of all the instances that meet some criteria.

These queries may take some care in implementation, but it can be essential to performance, since instantiating all the instances and localizing them just to add up one number could be a tremendous performance hit. Summing on an attribute of instances selected in a query is straight-forward and efficient in a relational database. It might be more expensive in some object databases, but still would need to be done on the database server machine, rather than bringing a large number of objects over the network. This is one of those cases where the underlying technology must be taken into account in the design, even though the interface is completely abstracted. This can actually be a risk to encapsulation – far outweighed by its benefits…

Repository (a.k.a. Registry)

Context	A client (which could be another domain object) that manipulates expressive objects.
Problem	Need to obtain a reference to a preexisting persistent entity object when traversal is not possible or is inappropriate. These forces come into play: • Isolation of the domain demands that database access be encapsulated. • It is best for clients to deal with abstract type of objects, rather than their concrete implementations.
Solution	Create a SINGLETON [GHJV95] for each entity type that is not internal to an aggregate boundary. The singleton has methods to support queries by value of attributes (including unique keys) that return references to instances or iterators (or their equivalent) over collections of instances.
Consequences	This pattern favors de-coupling of design from database over optimization of queries. This may force special-case optimizations when performance problems arise. However, caching, and other generalized optimizations, can be handled completely transparently to the application and domain.
Related Patterns	AGGREGATE BOUNDARIES: Make repositories only for the root entity of an aggregate.

Implementation of Repositories

This may vary greatly depending on the technology being used for persistence. The two broad categories of common options are relational database and object database. Ideally, the repository hides this distinction, but there will be performance implications that cannot be hidden or ignored, and the relational option may place some practical limits on deep compositional object structures, forcing you to restrict your model.

The ideal is to base repositories on "type" of the object retrieved from them, meaning that all objects must implement the same interface, but they might not have the same internal structure. This is sometimes difficult to implement, though, so a common compromise is to base a repository on class of the objects retrieved from them, trading off some flexibility in the client for easier implementation of the infrastructure. This difference is only felt in a highly abstracted domain model, but if the intention is to create powerful abstractions in the domain, this is a constraint.

Another simplifying assumption sometimes made is that a repository represents the "extent" of the class being retrieved from it, meaning that all instances of the class can be found through the repository. This is usually easier to implement than the more flexible collection based repository, in which the repository can represent some arbitrary subset of the objects of the type or class retrieved.

In any case, it is best to build a framework in the infrastructure layer from which repositories can be created for the domain. If, for some reason, this is not done, be sure to encapsulate the implementation of the repository so that the client can't tell the difference.

FACTORIES

…As mentioned above, every object in the DOMAIN OBJECT MODEL has a life-cycle. The REPOSITORIES provide access to the expressive objects in mid-life, and AGGREGATES encapsulate them in that phase. Now, let's take up their birth, when a client (which could be another domain object or something else, like the user interface) indicates that a new object is needed.

Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the expressive objects. Combining these responsibilities can produce ungainly designs that are hard to understand. Shifting the responsibility to the client breaches encapsulation if parts internal to an aggregate are assembled. This, along with directly referencing the concrete class by calling a constructor, overly couples the client to the implementation of the expressive object, and complicates the client.

Every object language provides a mechanism for creating objects (constructors in Java and C++, instance creation class methods in Smalltalk), but there is often a need for a more complex or more abstract construction mechanism.

Complexity of the object creation process is the easier case to see intuitively and to explain. Think of a car engine (a real one, not a computer model). It is an intricate piece of machinery with dozens of parts collaborating to perform the engine's responsibility: to turn a shaft. Shouldn't such a marvelous machine be able to assemble itself? One could imagine trying to design an engine block that could grab onto a set of pistons and insert them into its cylinders, spark plugs that would find their sockets and screw themselves in, and so on, but it seems unlikely that such a complicated machine would be as reliable or as efficient as our typical engines are. Instead, we accept that something else will assemble the pieces. Perhaps it will be a human mechanic or perhaps it will be an industrial robot. In fact, either of these is actually more complex than the engine they are assembling, yet their job is completely unrelated (assembling parts versus spinning a shaft). The assemblers are only used during creation of the car -- you don't need a robot or mechanic with you when you are driving. Since cars are never assembled and driven at the same time, there is no value to combining both of these complicated functions into the same mechanism. Likewise, assembling a

complex compound object is a job that is best separated from whatever job that object will have to do when it is finished.

But neither should the complexities of instance creation be turned over to some client in the application. The client must know and enforce all the invariants that apply to the relationship of parts in the domain object. In addition to complicating the client and blurring its responsibility, it breaches the encapsulation of the domain objects being created. Since the constructors of the concrete classes are being called directly by the client, and the assembly of those parts into complex wholes is being carried out by the client, no change to the implementation of the domain objects can be made without changing the client. This tight coupling of the application to the specifics of the implementation, strips away most of the benefits of the abstraction of the domain layer, and makes continuing changes ever more expensive.

Shift the responsibility for creating instances of specific expressive objects to a separate object, which may itself have no responsibility in the domain model, but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the objects being instantiated.

An object whose responsibility is the creation of other objects is called a "factory".

Just as the interface of an object should encapsulate its implementation so that users of that object can use its behavior without knowing how it works, so good constructors and factories separate the interface for requesting the creation of an object from the actual process of creation of that object. They provide interfaces that reflect the goals of the requestor, rather than reflecting the internal structure of the object.

There are many ways to design factories. Sometimes another object in the domain model can be found whose responsibility can reasonably accommodate the creation of another object type, but often a new, artificial object is needed, that does not have any responsibility for expressing the model.

Several special purpose creation patterns were thoroughly treated in [GHJV95]: Factory Method, Abstract Factory, Builder and Prototype (wherein the factory is actually another instance of the same type). The point here is that this is one of the structural components of a domain design.

In Smalltalk, where instances are created by the class which is first class object, a kind of factory is always used. This allows some additional abstraction of object creation but still exposes the name of the class being created. Therefore other factories are still used. The super-classes of hierarchies are used as abstract factories, other domain objects implement factory methods, and artificial factory classes are even used (though these are less often needed than in Java). The use of patterns in Smalltalk is explored in depth in [ABW98].

The two basic requirements for any good factory are:

- It should only be able to produce an object in a consistent state. For an entity this may mean the creation of the entire aggregate, with all invariants satisfied. For an immutable value this means all attributes are initialized to their correct final state. If the interface makes it possible to request an object that can't be created correctly, then an exception should be raised or some other mechanism should be invoked that will ensure that no improper return is possible.
- The factory should be abstracted to the type desired, rather than the concrete class(es) involved. (I'm afraid this is an ideal that is not always attainable in current environments, but it should be the goal. Many of the sophisticated factory patterns help with this.)

Constructing Value Objects

This is simpler than creating entities. Remember, values have no identity and are conceptually immutable. (The special cases discussed above in which value objects have mutable implementations are not distinguished at this point in the life-cycle.) These are usually lighter-weight objects than entities and generally have lighter-weight factories, but some value objects do present interfaces that encapsulate significant complexity. When value objects are attributes of other domain objects, it often works well to use that object as a factory for the value object, probably using a factory method.

Constructing Entities

Three factors make constructing entities trickier than constructing value objects. One is that entities usually have more complex internal structure. Another is that identity must be managed. Finally, since entities are generally have complex life-cycles in which their state is changed and added, and a factory must sometimes construct something to be added to the aggregate while keeping the aggregate consistent.

To sum up, the access points for creation of instances must be identified and their scope defined explicitly. They may simply be constructors, but often there is a need for a more abstract or powerful instance creation mechanism and this introduces new constructs into the design -- factories.

This pattern emphasizes low coupling and division of responsibility at the expense of multiplying the classes in the design. It could be obscuring with objects that are simple to create and have no abstract level.

When a constructor is all you need

In most designs I've seen, all instances are created by directly calling class constructors. I'll start by saying that you have to have these, since they are the languages way of creating new instances. The problem arises when very primitive constructors are made public. This violates both criteria above. You need those primitive constructors, but they should often be declared private, hidden behind more abstracted constructors or factories.

Even so, a public constructor may be sometimes be appropriate in a certain design style where the following are true.

- The class is the type. It is not part of any interesting hierarchy, and you don't intend to make it polymorphic with any other classes using an Interface.
- All of the attributes of the object are primitives types that can safely be constructed in the constructor, or they can be passed in as arguments without essentially exposing the internal structure of the object, or they

are entities that are enclosed by the aggregate boundary whose root is the object being constructed.

- The construction is not too complicated.

Examples can be found in the class libraries of Java. Instances of collections implement interfaces that allow the client to ignore concrete implementation, but instance creation is handled directly with constructors. Actually, a factory might have been beneficial, allowing a collection to be requested based on its characteristics. But the direct approach can also be justified on the grounds that, first, the collections are often not used in the same place they are created, so the ultimate consumers of the collections can still operate on the interfaces, and, second, that the requestor of a new collection may actually care about implementation, since this will be a performance sensitive area in many designs.

A primitive constructor should take information sufficient to create an object that satisfies its invariants. If that object is a value, then it must be complete. Entities can have other attributes added later so long as they are not integral to its identity or required by an invariant.

Be careful about calling constructors within constructors of other classes, unless maybe you have a very clear ownership relationship (like that between a Cargo and its Delivery History). Complex assemblies call for separate factories. Write the primitive constructors and call them from the factory.

PAUSE TO ITERATE

In the process of design work, understanding is always deepened, and new problems come to light (or seemingly small problems become more annoying). There is no need to wait until all the pieces are in place before going back to change early decisions, but now that they are, it is definitely time to look back and, with the value of hindsight, stack the design deck in our favor. (See "An Alternative Design" in Example.)

When it just isn't a thing: Beyond Classical Objects

SERVICES

… We've reached the end of the story for a classical object-oriented domain. But practical design considerations and the realities of modern computing environments lead us to one more construct.

Sometimes what we want just isn't a thing. Requirements call for domain functionality that cannot be assigned as a responsibility of an entity or value with an intentional definition that would be easily used or understood.

It could be a process that simply doesn't fit the object paradigm. Perhaps the behavior is the provided by a remote source, and a light-weight interface has to be provided to the local process. Or maybe it is a very complex collaboration of many entities, whose complexity we want to encapsulate.

Services fill this need. The name emphasizes its relationship with other objects. Unlike entities and value objects, it is defined purely in terms of what it can do for a client. It can still have clear responsibilities, but a service tends to be named for an activity, rather than an entity, a "verb" rather than a "noun". Hence it is a departure from the object modeling paradigm and should be used carefully and not allowed to strip the entities of all their behavior. But, used judiciously, it has advantages.

It turns out that fine-grained objects are not usually an effective unit of reuse. Medium grain, stateless objects can be easier to reuse in large systems because they encapsulate significant functionality behind a simple interface, They are also more manageable for distributed systems. Common practice in popular distributed architectures such as CORBA and DCOM lean heavily on this pattern, and they add distribution and access capabilities.

A service can't usually have the kind of abstract, intentional definition that an entity or value object would have, but it still must have a clear definition and responsibility describing its role.

The other unique consideration for these objects is statelessness, in the sense that any client can use any instance of the service without regard to its individual history. This is a tricky term, because it does not mean that the

object is independent of the state of the system. It may use information that accessible globally, and may even change information (have side-effects).

This pattern favors interface simplicity over client control and versatility, and it is a departure from classical object design ideals. But it provides a medium grain of functionality very useful in packaging in large or distributing systems. And sometimes it is the most natural way to express a domain concept.

Create an object whose definition is based on an activity or group of related activities. Implement methods named for the service provided with arguments and return values strictly in terms of entities and value objects from the domain (or standard class library). Make the service stateless.

SERVICES are a useful form for presenting ANTICORRUPTION LAYERS.

Access patterns for services are quite different than those for the expressive objects. The simplest approach is to provide SINGLETON access [GHJV95]. Distributed system architectures provide special publishing mechanisms for services, with conventions for their use...

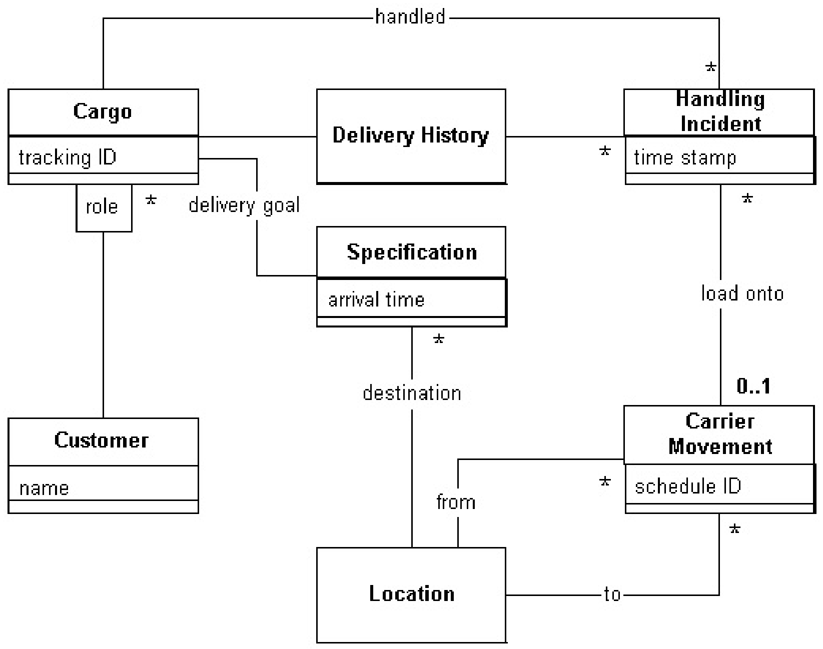
Using the Language in an Example: A Cargo Shipping System

This will be a drastically simplified model of shipping, but will attempt to show the various forces that apply and how these patterns help resolve them.

Initial requirements are three basic functions:

1. Track key handling of customer cargo
2. Book cargo in advance
3. Send invoices to customers automatically when the cargo reaches some point in its handling.

We'll start from a model that seems to abstract the relevant part of the problem domain.



I'll briefly overview the model and a few of the considerations behind it. I hope that the parts of the model that are not now clear will be explained as the example runs along.

Handling Incident is a particularly conspicuous over-simplification. It would probably be a hierarchy of different kinds of incidents, such as loading, unloading, and also including other handling other than loading. Instead, I've made "load onto" an optional association and ignored unloading altogether. Some compromises are necessary in an example.

Specification is a use of the SPECIFICATIONS pattern [EF97]. It is an abstraction of the need to describe something that is not present. In this case, we have a delivery goal which at least will be a destination location and an arrival date, but could be more complex. This could have been modeled as attributes of Cargo, but that would have at least three disadvantages. First, the Cargo object would be responsible for the detailed meaning of all those attributes which will clutter it up and make it harder to change. Second, it would have forced me to expose more detail on the diagram. As it is, I'm saying there is a Specification of delivery, but that could include a great deal of detail that doesn't need to be shown here and in fact could be easily changed later. Finally, this model is more expressive. By adding the concept of specification, I can say explicitly that I don't know how the Cargo will be delivered, but that this is our goal, and I can make that object responsible.

Role might just be a word, but it distinguishes the different parts played by customers in a shipment. One is the "shipper", one the "receiver", one the "payer", and so on. Since only one customer can play a given role in a particular Cargo, the association becomes a qualified many-to-one instead of many-to-many.

Carrier Movement represents one particular trip by a particular carrier (such as a truck) from one Location to another. Cargoes can ride from place to place by being loaded onto carriers for the duration of one or more Carrier Movements.

All the concepts needed to work through the requirements described above are present in this model (except unloading), assuming we have mechanisms to persist the objects, find the relevant objects, etc. These issues are not dealt with in the conceptual model, but must be in the design.

I will show, as we go along, that this model can be adapted to a clean design. How did I find a model that would serve this way? Well, basic object modeling principles are important to understand, and years of experience help. But guess what – **I cheated. Yes, and so should you!** This was not my initial model. I started with something that seemed reasonable, progressed through the steps toward design, found aspects of that model that made design difficult, and then I went back and changed it to make it a better basis for design. This basic iteration is the only way I know to produce useful models beyond the simplest, most obvious level.

Isolating the Domain: Simplified Shipping Applications

To prevent domain responsibilities from being mixed with those of other parts of the system, apply ISOLATED DOMAIN LAYER, and, specifically, SEPARATING APPLICATION FROM DOMAIN.

Without any deep analysis, there seem to be three application coordinators:

1. A Tracking Query that can access past and present handling of a particular cargo.
2. An Incident Logging application that can record each handling of the cargo (providing the information that is found by the Tracking Query).
3. A Booking application that allows a new cargo to be registered and prepares the system for it.

Remember, these are coordinators. They should not work out the answers to the questions they ask. That is the domain's job. That is the main focus of this article, and we'll get to that part shortly.

Expressing the Domain: Finding the ENTITIES and VALUE OBJECTS

I'll consider each object in turn and look for identity that must be tracked. By process of elimination, if it is in the domain model and has no identity (not an entity) then it is a value.

Entities and their Identities:

Customer: Let's start with an easy one. Since a customer is a person or a company, it clearly has identity. How to track it? Tax ID might be appropriate in some cases, but an international company could not use it. We'll use an automatically generated identification code that probably is visible to the user.

Cargo: Two identical boxes must be distinguishable, and in practice all shipping companies assign tracking ID's to each. This is another

automatically generated code that is visible to the user (and, in this case, probably conveyed to the customer).

Handling Incident and **Carrier Movement**: We care about such individual incidents because they allow us to keep track of what is going on. These are slightly more ambiguous than the others, but keep in mind that they reflect real-world events, and those are not usually interchangeable.

Location: Two places with the same name are not the same. Latitude and longitude could provide a unique key, but probably not a very practical one, since it is not of interest to most purposes of this system, and would be fairly complicated. More likely, the **Location** will be part of a geographical model of some kind that will relate places, and another arbitrary, internal, automatically generated identifier will suffice.

Value Objects:

Delivery History: This is a tricky one, and could, perhaps, be designed either way. But generally it is best not to carry identity unless we really have to, and, for Delivery History, all that really matters is the ability to retrieve the right **Handling Incidents**.

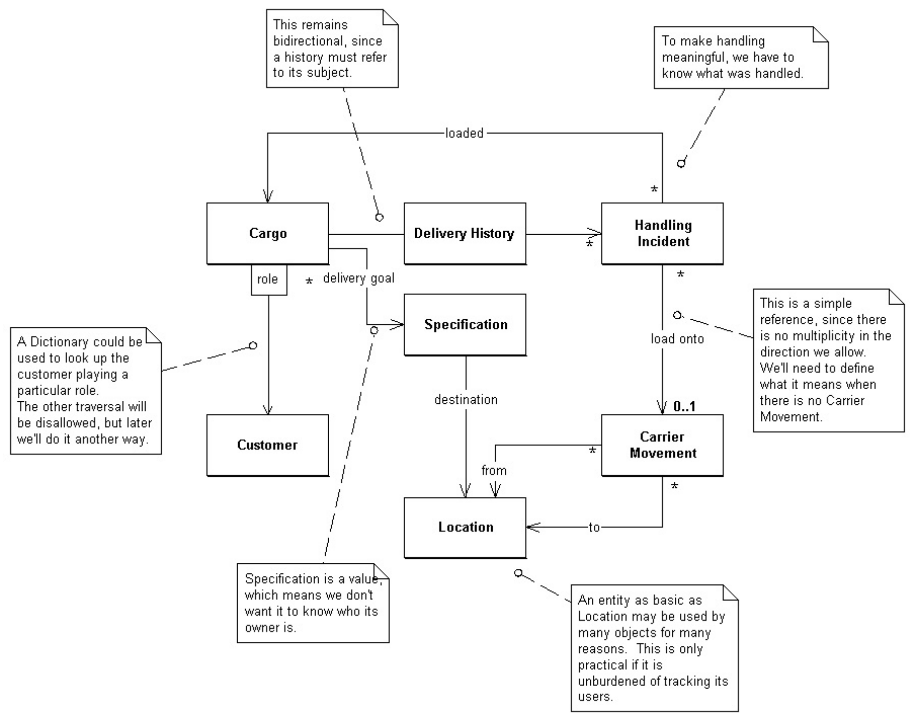
Specification: This it is the delivery goal of a **Cargo**, but the abstraction does not depend on **Cargo**. It really expresses a hypothetical state of some **Delivery History**. We hope that the one attached to our **Cargo** will eventually satisfy the **spec**. But if we had two cargoes going to the same place they could share the same Specification, while they could not share the same History, even though the histories start out the same (empty).

The **role** a customer plays in a Cargo

all the attributes (such as time stamps or names) shown in the previous diagram

Designing Associations in the Shipping Domain

Now as we move into design, we'll be adding things that are not related to the analysis but primarily to the concerns of the computer system. At the same time, we will be careful not to disturb the expression of the conceptual model.


This remains bidirectional, since a history must refer to its subject.
To make handling meaningful, we have to know what was handled.
loaded
Cargo
Delivery History
Handling Incident
role
delivery goal
Specification
A Dictionary could be used to look up the customer playing a particular role. The other traversal will be disallowed, but later we'll do it another way.
load onto
This is a simple reference, since there is no multiplicity in the direction we allow. We'll need to define what it means when there is no Carrier Movement.
destination
0..1
Customer
Carrier Movement
from
Location
to
Specification is a value, which means we don't want it to know who its owner is.
An entity as basic as Location may be used by many objects for many reasons. This is only practical if it is unburdened of tracking its users.

If the customer has a direct reference to every cargo he or she has shipped it will become cumbersome for long-term repeat customers. Also, in a large system, the customer may have roles to play with many objects. Best to keep it free of such specific responsibilities. If we need the ability to find cargoes by customer, this can be done through a database query. We'll return to this in the section on Repositories.

There is one circular reference: Cargo knows its Delivery History which holds a series of Handling Log Entries, which in turn point back to the Cargo. Circular references logically exist and are sometimes necessary in design as well, but they are tricky to maintain, since the same information is being held in two places and must be kept synchronized. In this case, we'll retain this for

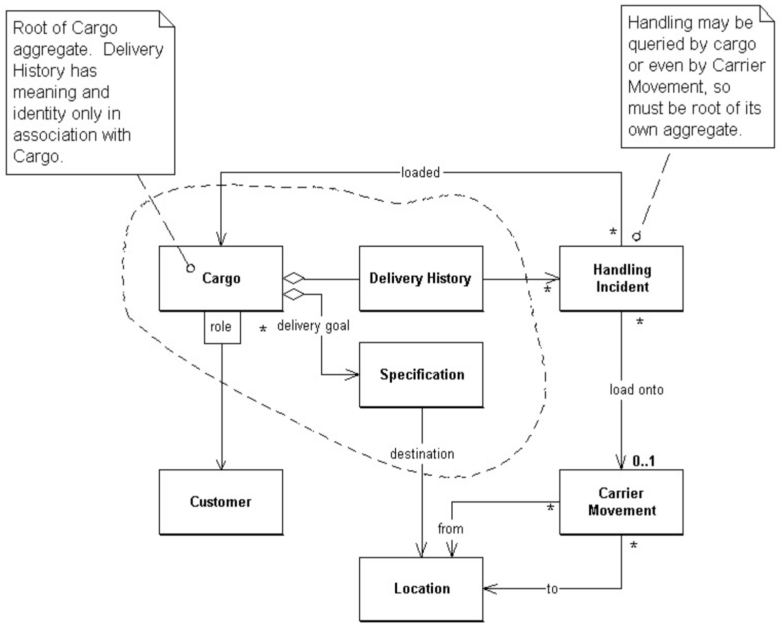
our prototype, with an Array List (Java) or Ordered Collection (Smalltalk) in Delivery History, but in the final design we'll probably drop the collection in favor of a database lookup with Cargo as the key. If the query to see the history is relatively infrequent, this should give good performance and simplify maintenance and reduce the overhead of adding Handling Log Entries. If this query is very frequent, then it is better to go ahead and maintain the direct pointer. These design tradeoffs are based on simplicity of implementation and on performance. The conceptual model is the same.

AGGREGATE BOUNDARIES in the Shipping Model

Customer, Location and Carrier Movement have their own identity and are shared by many Cargoes, so they must be the roots of their own aggregates. Cargo is also an obvious root, but where to draw the aggregate boundary takes some thought.

We could sweep in everything that only exists because of this Cargo, which would include the Delivery History, the Specification of the delivery goal, and the Handling Incident. The delivery goal is a value, so that's easy. The Delivery History seems to be something no one would look up directly without the Cargo itself. With no need for direct global access, and with an identity that is really just derived from the Cargo, the History fits nicely inside of Cargo's boundary, and does not need to be a root.

The Handling Incident is another matter. Previously we have considered two possible database queries that would search for these: one to find what was loaded onto a particular carrier movement; another as a possible alternative to finding the Handling Log Entries for a Delivery History. It seems that the activity of handling the Cargo has some meaning even when considered apart from the Cargo itself. It should be the root of its own aggregate.

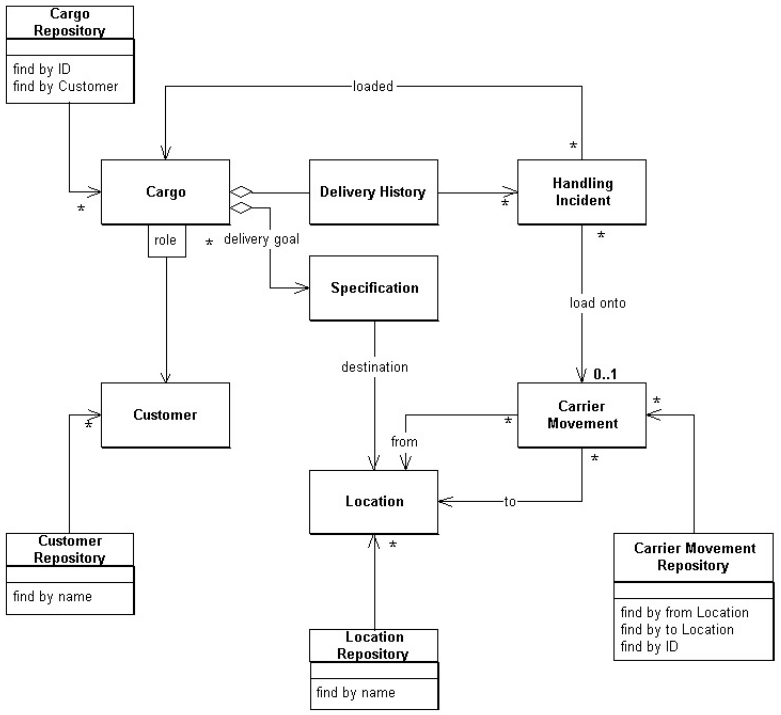

Root of Cargo aggregate. Delivery History has meaning and identity only in association with Cargo.
Handling may be queried by cargo or even by Carrier Movement, so must be root of its own aggregate.
loaded
Cargo
Delivery History
Handling Incident
role
delivery goal
Specification
load onto
0..1
destination
Customer
Carrier Movement
from
Location
to

REPOSITORIES for selected ENTITIES

There are five entities in the design that are roots of AGGREGATES, so we can limit our consideration to these, since none of the other objects are allowed to have repositories.

Now we must consider the application requirements. In order to take a booking through the Booking Application Coordinator, the user to select the Customer(s) playing the various roles (shipper, receiver, etc), so we have a Customer Repository. We also need to find a Location to specify as the destination for the Cargo, so we have a Location repository.

The Activity Logging Application Coordinator needs to allow the user to look up the Carrier Movement that a Cargo is being loaded onto, so we need a Carrier Movement Repository. This user must also tell the system which Cargo has been loaded, so we need a Cargo Repository.



For now there is no Handling Incident Repository because we decided to implement the association with Delivery History as a collection in the first iteration, and we have no application requirement to find out what has been loaded onto a Carrier Movement. Either of these reasons could change, and then we would add a repository.

Some Sample Instance Creation

Changing the Destination of a Cargo

Occasionally a customer calls up and says, "Oh, no. We said to send our Cargo to Hackensack, but we really need it in Hoboken" We are here to serve, so the system is required to provide for this change.

Specification is a value object, so it would be simplest to just to throw it away and get a new one.

<<Use Specification's abstract factory to create new one and then use setter method on Cargo to replace old one with new one.>>

Repeat Business

The users say that repeated bookings from the same customers tend to be similar, so they want to use old cargoes as prototypes for new ones. The application will allow them to find a cargo in the repository and then select a command to create a new cargo based on the selected one. We'll design this using the "Prototype" pattern [GHJV95].

Cargo is an ENTITY and is the root of an AGGREGATE. Therefore, it must be copied carefully, deciding what should happen to each object or attribute enclosed by its aggregate boundary. Let's go over each one:

Delivery History: We should create a new, empty one, since the history of the old one doesn't apply. This is the usual case with entities inside the aggregate boundary.

Customer Roles: We should copy the HashTable (or other collection) that holds the keyed references to Customers, including the keys, since they are likely to play the same roles in the new shipment, but we have to be careful not to copy the Customers. We must end up with references to the same Customers as the old Cargo referenced, since they are entities outside the aggregate boundary.

Tracking ID: We must provide a new tracking ID from the same source as we would when creating a new Cargo from scratch.

Notice that we have copied everything inside the Cargo aggregate boundary, made some modifications to the copy, and have affected <u>nothing</u> outside the aggregate boundary at all.

Primitive Constructors for Cargo

Even if we have a fancy FACTORY for Cargo, or use another Cargo as the factory, as in Repeat Business, above, we still have to have a primitive constructor. We would like the constructor to produce an object that fulfills its invariants or at least, in the case of an ENTITY, has its identity intact.

Given these decisions, we might create a method on Cargo such as:

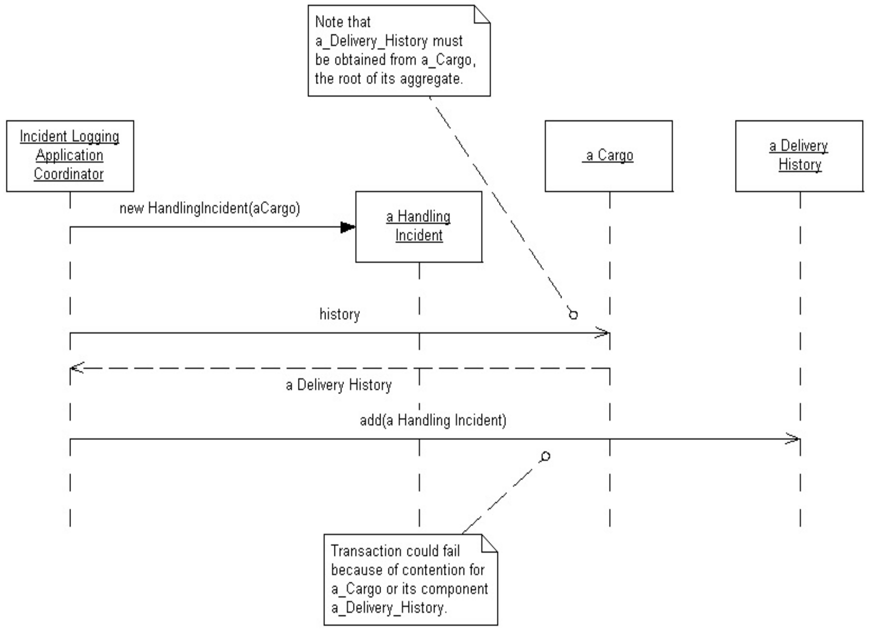
```
public Cargo copyPrototype(String newTrackingID)
```

Or we might make a new constructor such as:

```
public Cargo(Cargo prototype, String trackingID)
```

Adding a Handling Incident

Each time the cargo is handled in the real world, some user will enter a "Handling Incident" using the Incident Logging Application Coordinator. The basic constructor for Handling Incident was presented above, but the story isn't quite that simple. The Delivery History holds a collection of Handling Incidents relevant to its Cargo, and the new object must be added to this collection as part of the transaction.


Note that a_Delivery_History must be obtained from a_Cargo, the root of its aggregate.
Incident Logging Application Coordinator
a Cargo
a Delivery History
new HandlingIncident(aCargo)
a Handling Incident
history
a Delivery History
add(a Handling Incident)
Transaction could fail because of contention for a_Cargo or its component a_Delivery_History.

As I mentioned at the outset, the Handling Incident model is an over-simplification. In a real analysis of the shipping domain, there would doubtless be a variety of specialized Handling Incidents, ranging from loading and unloading to sealing, storing, and other not related to carriers. They might be implemented as multiple subclasses and/or have complicated initialization. In this case, a single FACTORY, used by all Handling Incidents, abstracts instance creation, freeing the client from knowledge of the implementation. Methods on the factory would be provided to request instances to suit specific needs, and the factory would be responsible for knowing what class was to be instantiated.

Primitive Constructors for the Handling Incident

Every class must have primitive constructors that are passed, in some form, the desired attributes and return the new instance. The Handling Incident constructor must take a Cargo and a Carrier Movement as arguments.

```
public HandlingIncident(Cargo c, CarrierMovement m, Time
t)
{  loaded = c;
   loadedOnto = m;
   timeStamp = t;
}
```

Another constructor could be used for handling other than loading.

```
public HandlingIncident(Cargo c, Time t)
```

The two way pointer between Cargo and Delivery History present a more interesting case. Neither Cargo nor Delivery History is complete without pointing to its counterpart, so they must be created together. Remember that Cargo is the root of the AGGREGATE that includes Delivery History. Therefore, we can allow Cargo's constructor to create a Delivery History. The Delivery History constructor will take a Cargo as an argument. Something like this:

```
public Cargo(String id)
{  trackingID = id;
    deliveryHistory = new DeliveryHistory(this);
   customerRoles = new HashTable();
}
```

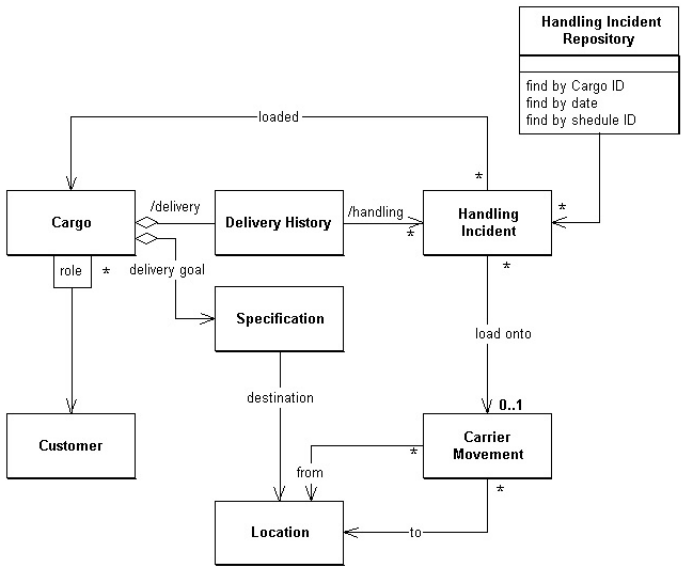
The result is a new Cargo with a new Delivery History that points back to the Cargo. The Delivery History constructor is used exclusively by its aggregate root, namely Cargo, so that the composition of Cargo is encapsulated.

Pause for Iteration: An Alternative Design of the Cargo Aggregate

Modeling and design is not a constant forward process. It will grind to a halt unless there is a frequent return to the beginning to take advantage of new insights to improve the model and the design.

By now, there are a couple of cumbersome aspects to this design, although it does work and reflect the model. There are two circular references and they complicate instance creation and maintenance of consistency. Let's pause to consider a slightly different design that would fit the same model and would provide some advantages in some environments.

Rather than holding a collection of Handling Incidents in the Delivery History, let's just derive it whenever we need it. That means creating a repository for the Handling Incidents, which we can do since Handling Incident is an entity and an aggregate root.



This can simplify maintenance because we have eliminated the double update that had to be made consistent whenever an Incident was added. It gives some capabilities we might want if, for example, someone wants to ask, "What is on this Carrier Movement?". It could also improve performance under some circumstances. The Handling Incident entry transaction will probably be faster, and will have fewer contention problems (which, in the other design, could arise if another transaction involving the Cargo happened at the same time). The query could be optimized to answer specific questions efficiently. For

example, if the typical access is just to find the last reported incident to infer the current status of the Cargo a query could just return the one relevant Incident. If there are a lot of Handling Incidents being entered and relatively few queries, this design is more efficient. In fact, if a relational database is the underlying technology, a query was probably being used anyway to simulate the collection.

If we go a step further, we could derive Delivery History itself whenever it is needed to answer some question. We are allowed to do this since Delivery History is a value object and all that counts is that we get the right attributes into it. This simplifies the Cargo constructor – no tricky circular reference to be created and maintained. It reduces database space slightly, and might reduce the actual number of persistent objects considerably, which is a limited resource in some object databases. If the common usage pattern is that the user seldom queries for the status of a Cargo until it arrives, then a lot of unneeded work will be avoided altogether.

On the other hand, if we are using an object database, traversing an association or an explicit collection is probably much faster than a repository query. If the access pattern includes frequent listing of the full history, rather than the occasional targeted query of last position, the performance tradeoff might favor the explicit collection. And remember that the nice added feature ("What is on this Carrier Movement?") hasn't been requested yet, and may never be, so we don't want to pay much for that option.

These kinds of alternatives and design tradeoffs are everywhere, and I could come up with lots of examples just in this little simplified system. But the important point is that these are degrees of freedom within the same model. By modeling values, entities, and their aggregate boundaries as we have, we have reduced the impact such design changes have. For example, in this case all changes are encapsulated within the Cargo's aggregate boundary. It is dependent on the existence of a Handling Incident Repository, but that is a separate design element that could have been added for other reasons. The addition of the repository does not call for any redesign of the Handling Incident itself (although some implementation changes might be involved, depending on details of the repository framework).

New Feature: Cargo Loading

We have a new requirement. Instead of just tracking cargo, our system must now decide which cargoes to load onto which Carrier Movements. This could be done in many ways, but one attractive possibility is to create a subsystem responsible for making this decision and present its services through SERVICE, in order to minimize the complexity exposed to the rest of the system.

<<Put diagram here showing façade, and maybe another showing object interaction>>

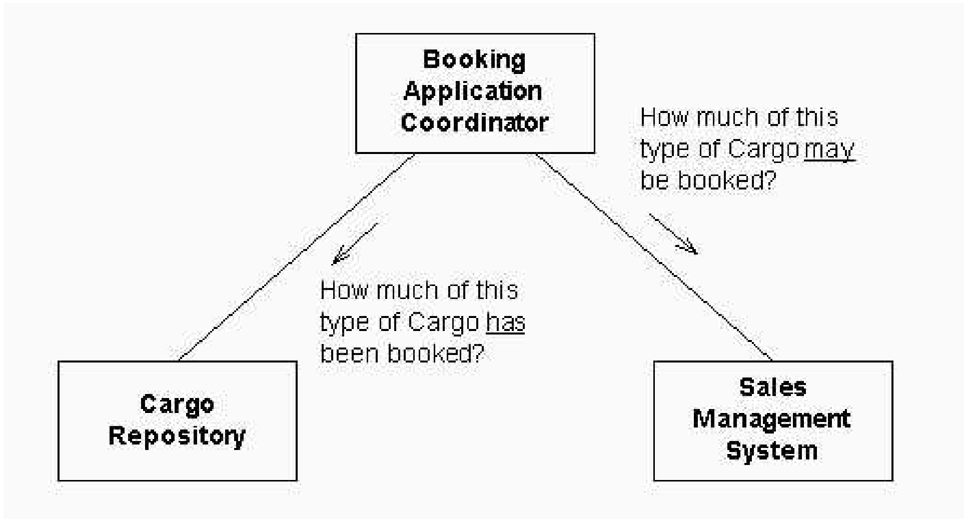
New Feature: Allocation Checking

Now the first major new functions are going to be added. In this design we will reabstract the domain of an external system and hide it behind an ANTI-CORRUPTION LAYER..

The sales division of our imaginary shipping company uses other software to manage their client relationships, sales projections, and so forth. One feature they use supports yield management by allowing them to allocate how much cargo of specific types they will attempt to book based on the type of goods, the origin and destination, or any other factor they may choose that can be entered as a category name. These constitute goals of how much will be sold of each type, so that more profitable types of business will not be crowded out by less profitable cargoes, while at the same time avoiding under-booking, (not fully utilizing their shipping capacity) or excessive overbooking (so that so much cargo gets bumped that it affects customer relationships).

Now they want this to be integrated with the booking system. When a booking comes in, they want it checked against these allocations to see if it should be accepted.

The information needed resides in two places, which will have to be queried by the booking application coordinator so that it can either accept or reject the requested booking. A first stab might look like this.

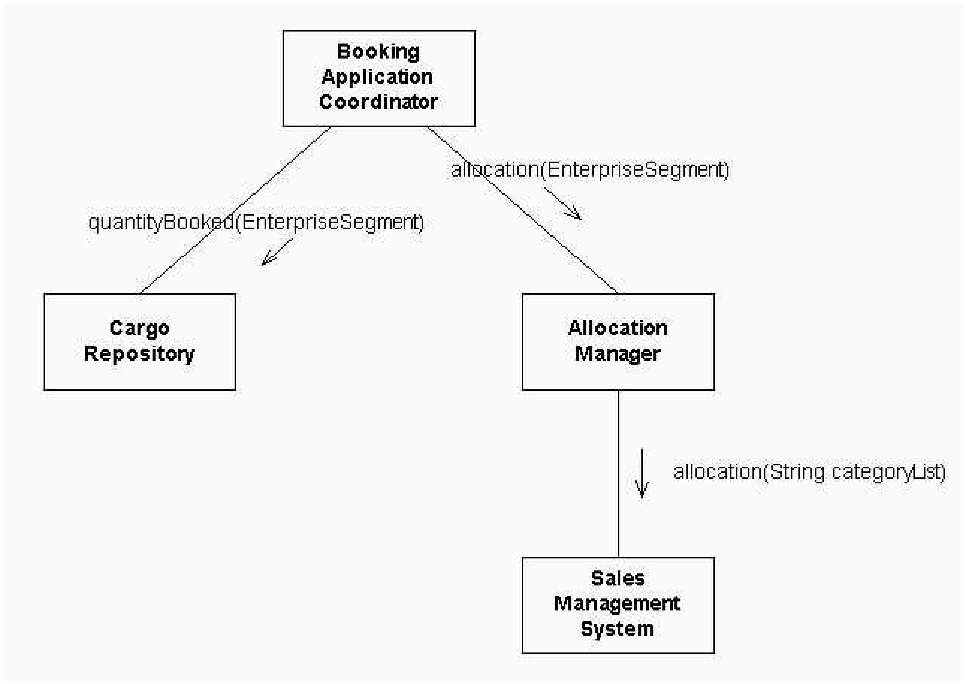

Booking Application Coordinator
How much of this type of Cargo may be booked?
How much of this type of Cargo has been booked?
Cargo Repository
Sales Management System

The anti-corruption layer pattern should be applied, giving us a façade and adapter to control the interface to the sales management system. We could call it something like "Sales Management Interface". This would allow us to hide all the mechanics of talking to the other program and give us a global access point. But we would be missing an opportunity to recast the problem along lines more useful to us. Instead, lets give it a name related to its responsibility in our system, and call it "Allocation Manager", and present it as a SERVICE. All services related to allocation will be channeled through here.

If some other integration is needed (for example, using the sales management system's customer database instead of our own customer repository), another façade can be created with services fulfilling that responsibility. The lower level "Sales Management Interface" might be useful for shared machinery of talking to the other program, but it would be hidden behind the other façade(s), and wouldn't show up in the domain design.

Now, what kind of interface are we going to supply that can answer the question, "How much of this type of Cargo may be booked?" The tricky issue is to define what "type" it is, since our domain model does not categorize cargoes yet. In the sales management system, this is just a set of category key words, and we could conform to this. We could pass a collection of strings in as an argument. But we would be passing up another opportunity – this time, to reabstract the domain of the other system. Let's do a little domain modeling.

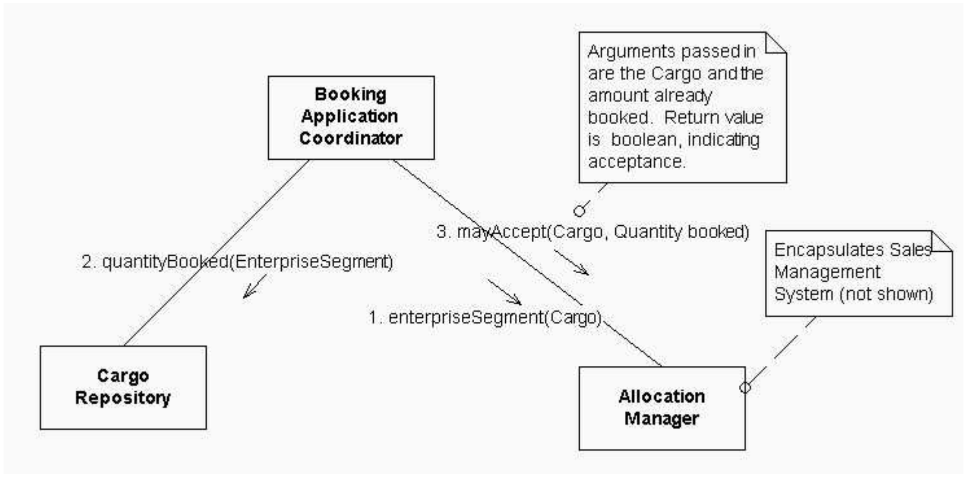
A more powerful concept to use for this kind of problem is the ENTERPRISE SEGMENT pattern described by [Fowler96], p64, which is a set of dimensions that define a way of breaking down a business. These dimensions could include all those mentioned above and also time dimensions, such as month-to-date. Using this concept in our model of allocation makes the model more expressive and simplifies the interfaces. The Enterprise Segment will appear in our domain model and design as an additional value object which will have to be derived for each Cargo.


Booking Application Coordinator
allocation(EnterpriseSegment)
quantityBooked(EnterpriseSegment)
Cargo Repository
Allocation Manager
allocation(String categoryList)
Sales Management System

It will be necessary for the Allocation Manager to translate between Enterprise Segments and the category names of the external system. The Cargo Repository must also provide a query based on enterprise segment. In both cases, collaboration with the Enterprise Segment object can be used to

perform the operations without breaching the Segment's encapsulation and complicating their own implementations. (Notice that the Cargo Repository is answering a query with a count, rather than a collection of instances

There are still a few problems with this design. We have given the Booking Application Coordinator the job of applying a business rule, which is a domain responsibility and shouldn't be performed by an application coordinator. Also, it isn't clear who is responsible for deriving the Enterprise Segment. Both of these seem to belong to the Allocation Manager.


Arguments passed in are the Cargo and the amount already booked. Return value is boolean, indicating acceptance.
Booking Application Coordinator
3. mayAccept(Cargo, Quantity booked)
Encapsulates Sales Management System (not shown)
2. quantityBooked(EnterpriseSegment)
1. enterpriseSegment(Cargo)
Cargo Repository
Allocation Manager

The interface of the Allocation Manager is the only part that needs to be considered in the rest of the domain design. It does have its own internal design, which can present opportunities. One worth contemplating is making the objects responsible for deriving enterprise segment relocatable. If there is communications overhead from going to the sales management system (which may be on another server) it would be useful to cache on the client the data and behavior needed to derive this value, which should be relatively static compared to the allocations themselves. Flexible deployment is an important design goal in distributed systems.

The only serious constraint imposed by this integration will be that the sales system mustn't use dimensions that the Allocation Manager can't turn into Enterprise Segments. (Without Enterprise Segment, the same constraint would take the form that the sales system mustn't use dimensions that the

Cargo Repository can't use in a query. This is feasible, but the sales system spills into other parts of our domain, whereas in this design, the Cargo Repository need only be designed to handle Enterprise Segment, and changes in the sales system ripple only as far as the Allocation Manager, which was conceived as a façade in the first place.)

That's it. This integration could have turned our simple, conceptually consistent design into a tangled mess, but now, using ANTI-CORRUPTION LAYER, SERVICE and ENTERPRISE SEGMENT, we have integrated the functionality of the Sales Management System into our booking system while cleanly enriching the domain.

A final design question: Many may wonder about giving the Cargo the responsibility of deriving the Enterprise Segment. It certainly seems elegant, if all the data the derivation is based on is in the Cargo, to make it a derived attribute of Cargo. Unfortunately, it is not that simple. Enterprise Segments are defined arbitrarily to divide along lines useful for business strategy. The same entities could be segmented differently for different purposes. We are deriving the segment for a particular cargo for booking allocation purposes, but it could have a completely different Enterprise Segment for tax accounting purposes. Even the allocation enterprise segment could change if the sales management system is reconfigured because of a new sales strategy. So the Cargo would have to know about the Allocation Manager, which is well outside its conceptual responsibility, and it would be laden with methods for deriving specific types of enterprise segment. Therefore, the responsibility for deriving this value lies properly with the object that knows the rules for segmentation, rather than the object that has the data that those rules are applied to. Those rules could be split out into a separate "strategy" object, which could be passed to a Cargo to allow it to derive an Enterprise Segment. That seems to go beyond the requirements we have here, but it would be an option for a later design and shouldn't be a very disruptive change.

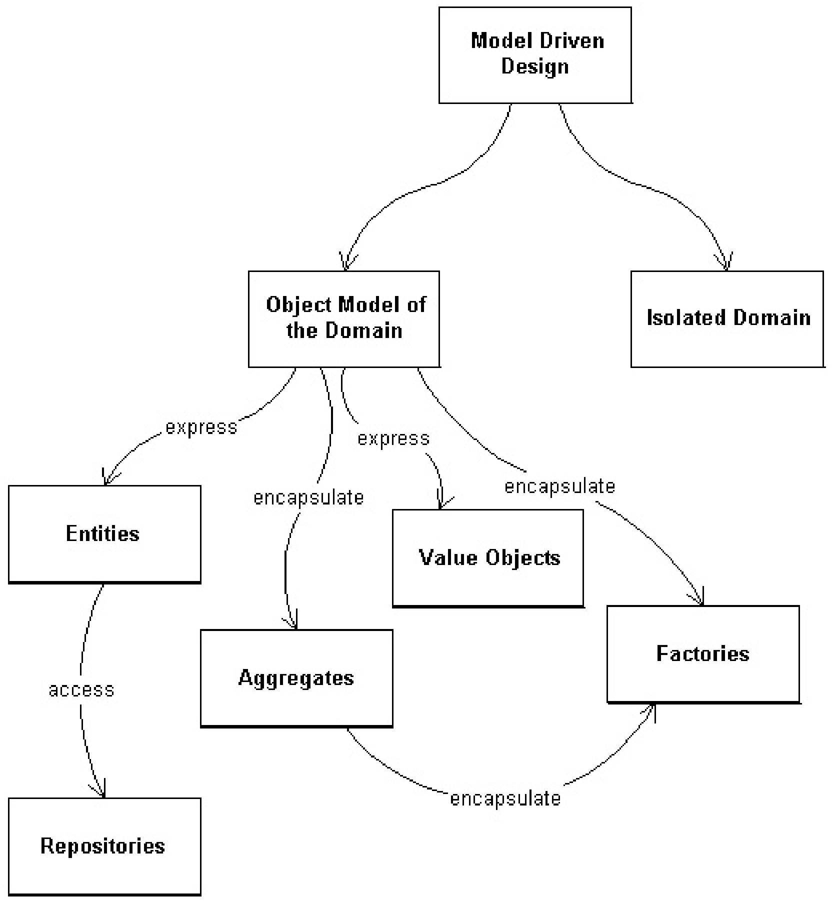
Conclusion

Accomplished object designers do not start from scratch with each new object, but follow a relatively few patterns that, taken together, allow them to build various components that embody the model and that interact to vitalize the model. Knowing these design patterns, they create a conceptual model that can support design. In the absence of these guiding patterns, the novice suffers from an excess of freedom and wonders into a bog, with an insufficiently defined conceptual model and a shapeless design.

Developing a good domain model is an art. But practical design and implementation of a model can be relatively systematic, and can even improve the modeling process by more clearly expressing the model in the design and by isolating the domain from other influences.

This pattern language lays out the tools to build practical object-oriented software with a model driven design.

Pattern Map



References

[ABW98]	Alpert, S., Brown, K., Woolf, B. 1998. *The Design Patterns Smalltalk Companion*, Addison-Wesley.
[BMRSS96]	Bushman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. P. *A System of Patterns.* Wiley.
[Cockburn98]	Cockburn, A., 1998. *Surviving Object-Oriented Projects.* Addison-Wesley.
[EF97]	Evans, E., Fowler, M. 1997, “Specifications”, PLoP 97.
[Fowler96]	Fowler, M., 1996. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.
[GHJV95]	Gamma, E. Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns*. Addison-Wesley.
[Larman98]	Larman, C. 1998. *Applying UML and Patterns*. Prentice Hall.

Acknowledgements

Several people have helped to read and critique this paper. I particularly want to thank Robert Mee for reading early drafts and for contributing the term “corruption layer”, and David Siegel for putting forward the concept of “aggregates”. Thanks also to Joshua Kerievsky for all his help while acting as shepherd for the paper in preparation for PLoP 1999.