Deconstructing the Domain:

A Pattern Language for Handling Large Object Models

(Draft July 14, 1999)

Eric Evans

Table of Contents

Introduction

As systems grow too complex to know completely at the level of individual objects, we need new techniques for manipulating and comprehending large models. This paper presents principles that enable the modeling process to scale up to very complicated domains.

The goal of the most ambition enterprise systems is a tightly integrated system spanning the entire business. Yet the entire business model for almost any such organization is too large and complex to manage or even understand as a single unit. The system must be broken into smaller parts, in both concept and implementation. The challenge is to accomplish this modularity **without losing the benefits of integration,** allowing different parts of the system to inter-operate to support the coordination of various business operations. It is important to strike a balance, avoiding pitfalls such as an unworkable, single, spaghetti-like, all-encompassing domain model with no barriers, or a far less powerful set of distinct subsystems, stove-piped together.

A good domain model captures an abstraction of the business in a form defined enough to be coded into software. Domain architecture must provide a guide to design decisions for the business model that reduce interdependence of parts and improve clarity and ease of understanding and analysis without reducing their interoperability and synergy. It must also capture the conceptual core of the system, the "vision" of the system. The four broad themes explored in this article can help accomplish these goals: *unification context, distillation, modularity*, and *large-scale structure*.

Unification context, the least obvious of the four principles, is actually the most fundamental. A successful model, large or small, has to be logically consistent throughout, without any contradictory or overlapping definitions. Enterprise systems sometimes integrate subsystems with varying origins or have applications so distinct that very little in the domain is viewed in the same light. It may be asking too much to unify the conceptual models implicit in these disparate parts. By explicitly defining a *unification context* within which a model applies, and then, when necessary, defining its relationship with other contexts, the modeler can avoid bastardizing the model.

Distillation is not so much a design technique as a way of focusing the attention appropriately. Often a great deal of effort is spent on peripheral issues in the domain. The overall domain model needs to make prominent the most value-adding and special aspects of your system and be structured to give that part as much power as possible. While some supporting components can be critical, they must be put into their proper perspective. This not only helps to direct efforts toward vital parts of the system, but it keeps the vision of the system from being lost. *Distillation* can bring incredible clarity to an overall model.

Modularity is certainly not a new concept. If the full domain is modeled as a single web of objects it is almost impossible to understand and will surely be impossible to implement. The domain must be broken into many smaller models, each of which has a clear focus and can be easily described.

Large-scale structure completes the picture. In a very complex model, even though modular, you may not see the forest for the trees. Distillation helps, by focusing the attention on the core and presenting the other elements in their supporting roles, but the relationships can still be too confusing without some *large-scale structure* that allows system-wide design elements and patterns to be applied. The principle of large-scale structure is explored here by examining one such pattern, *layers of responsibility*, in which a small but powerful set of fundamental responsibilities are identified that can be organized into layers with defined relationships between layers, such as modes of communication and allowed references. Such structure can bring a uniformity to the design that can accelerate the design process and improve integration.

These principles, useful separately but particularly powerful taken together, help us to produce good designs even when systems become too big to understand as a whole while thinking about individual objects. With unification contexts and large-scale structure, we can bring consistency to the disparate parts that will help those parts mesh together. Structure and distillation let us make the complex relationships between the parts comprehensible while keeping the big picture in view.

Unification Context

It was six men of Indostan
To learning much inclined,
Who went to see the Elephant
(Though all of them were blind),
That each by observation
Might satisfy his mind.

The *First* approached the Elephant,
And happening to fall
Against his broad and sturdy side,
At once began to bawl:
`God bless me! but the Elephant
Is very like a wall!'

…

The *Third* approached the animal,
And happening to take
The squirming trunk within his hands,
Thus boldly up and spake:
`I see,' quoth he, `the Elephant
Is very like a snake.'

The *Fourth* reached out his eager hand,
And felt about the knee.
`What most this wondrous beast is like
Is mighty plain,' quoth he;
`'Tis clear enough the Elephant
Is very like a tree!'

…

The *Sixth* no sooner had begun
About the beast to grope,
Than, seizing on the swinging tail
That fell within his scope,
`I see,' quoth he, `the Elephant
Is very like a rope!'

And so these men of Indostan
Disputed loud and long,
Each in his own opinion
Exceeding stiff and strong,
Though each was partly in the right,
And all were in the wrong!

…

John Godfrey Saxe (1816-1887), based on a story in the Udana, a Hindu text.

Although we seldom think about it explicitly, the unification context is the most fundamental of the principles presented here, **indispensable even for small models**, because a model is meaningless unless it is logically consistent. In the ideal world, we would have a single conceptual model spanning the whole domain of the enterprise. This model would be unified, without any contradictory or overlapping definitions. Every logical statement in the domain would be consistent. But the world of large systems development is not the ideal world.

Total unification may not be cost effective for various reasons. Our system may have to be integrated with external systems that were developed based on a different conceptual model and that cannot be changed. Older parts of our own system that are too expensive to change in the short term may have used earlier conceptual models that are subtly different from our current model. Different teams may have been working on different parts of the problem and come up with divergent models that suit their part of the business problem well. Yet, for an object model to be powerful, it must be logically unified.

This belongs to the class of problems for which a major part of the solution lies in simply seeing the problem. Explicitly define one or more "unification contexts" in each of which the model will be strictly consistent, but between which there can be inconsistencies in representation. By drawing an explicit boundary, you can keep each part pure, and therefore potentially powerful, and avoid confusion when we shift our attention to other contexts.

This introduces a new problem: How can you integrate such distinct models? This is the tradeoff. Deep integration between different unification contexts is impractical. The

integration is limited to those parts of one model that can be rigorously stated in terms of the other model, and even this level of integration may take a considerable effort. This makes sense when there will be a small interface between two systems. It may be adequate for some needs to view an elephant as a wall, held up by tree trunks, with a rope at one end and a snake at the other.

Example: Contextual confusion on a shipping project

There are multiple teams on the project to develop the shipping system introduced in the previous examples. A few of the teams have been working closely together to produce the model presented above. Another team has been working on the related problem of sales allocation.

This model supports yield management by allowing the sales management 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). Their model looks something like this.

<<put allocation model here>>

In one scenario, they attempt, with the best of intentions, to integrate with the other teams at the design level without realizing their conceptual models were not unified. In the name of reuse, they take from the other team implementations of those objects that are named the same in both models. That would be <<…>>. Since some of the objects don't do all that theirs do, they add those operations, along with the attributes and associations needed to support them. The result is something like this (in the design, though perhaps they never realize its impact on the model):

<<show overloaded model>>

The consequence of this sort of weird, overloaded objects is unpredictable software that no one can understand or make reliable. What can be done?

There are two choices:

1. Refactor the models and fully unify them. Full integration will then be yours.
2. Declare two separate contexts. Decide what must pass between them and define translations for those parts of the model.

Our teams by now acknowledge they are not good at coordinating. Furthermore, they realize that the essential points of contact between their parts of the system are going to be small. They decide on the less ideal but more practical, for them, option to declare two contexts.

<<show their separate models side-by-side>>

Now both teams can continue their independent development, coordinating carefully on a small and clearly delineated part of their overall work.

The decision to separate contexts is not irrevocable. Sometimes two contexts can be unified into one by refactoring one or both models. Finding a deeper level of abstraction may make it

easier to express both in the same language. Or it may just require the hard work of eliminating synonyms and choosing one factoring over another.

The decision is usually a cost benefit trade-off. If tight integration is required, there is a high benefit to unifying the contexts and a high cost to interfacing two separate contexts. If your project is poorly coordinated or you do not have very advanced modelers, the risk of attempting more integration is high, and you may be better off with separate contexts.

Sometimes you may have subsystems with varying origins or have applications so distinct that very little in the domain is viewed in the same light. It may be asking too much to unify the conceptual models implicit in these disparate parts. And if you must integrate with an entirely separate system, you clearly cannot refactor both to unify them. All you can do is define the translation between the two models at the essential contact points.

Unifying an Elephant

The various blind men can still do useful work, even if they don't fully agree on the nature of the elephant. If no integration is required, then it doesn't matter that they the models are not unified. If they require some integration, they may not actually have to agree on what an elephant is, but they will get a lot of value from merely recognizing that they don't agree. This way, at least they don't unknowingly talk at cross-purposes. They can learn to communicate with each other about the few aspects they care about in common – the location of the elephant, perhaps.

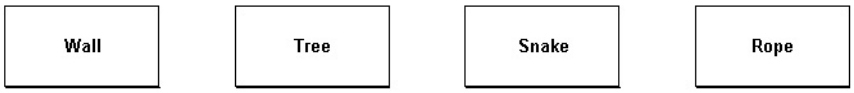
As more integration is required, the value of unifying the models goes up, but unification probably does not mean that all but one person gives up his model and adopts the remaining one. After all, the man who touched the trunk knows the elephant is not like a tree, and that model would be useless to him. Unification almost always means creating a new model.

In this particular case, unification of the various elephant models is easier than most because we can eventually recognize that blind men have been describing and modeling different parts of the whole. For many purposes, a part-whole unification may not require much additional work. At least the first stage of integration only requires figuring out how the parts are related. It may be adequate for some needs to view an elephant as a wall, held up by tree trunks, with a rope at one end and a snake at the other.

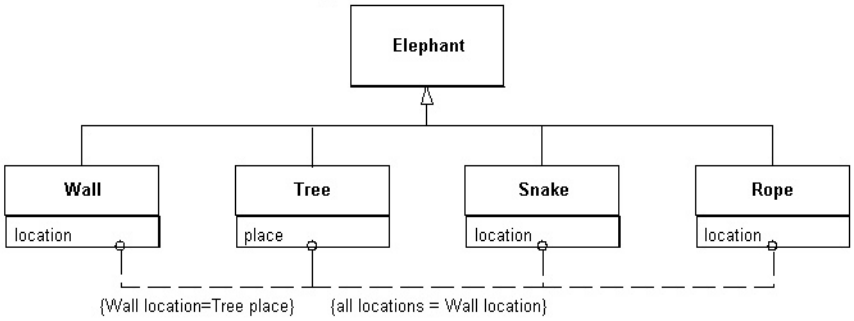
Unfortunately, it is the exception when two models purely describe different parts of the whole. Matters are more difficult when two models are looking at the same part in a different way. If two men had touched the trunk and one described it as a snake and the other described it as a fire-hose, they would have more difficulty. If one tried to accept the other's model, he could get into trouble. In fact, they need a new abstraction that incorporates the "aliveness" of a snake with the water shooting functionality of a fire-hose, but leaves out inappropriate implications of the first models, such as the expectation of possibly venomous fangs, or the ability to detach it and roll it up in a compartment in a truck.

This second pass of integration tends to slough off incidental or incorrect aspects of the individual models and creates new concepts – in this case, "animal" with parts "trunk", "leg", "body", and "tail" that each have their properties and are connected in a particular relationship. Successful model unification to a large extent hinges on minimalism. A trunk is both more and less than a snake, but the "less" is probably more important than the "more".

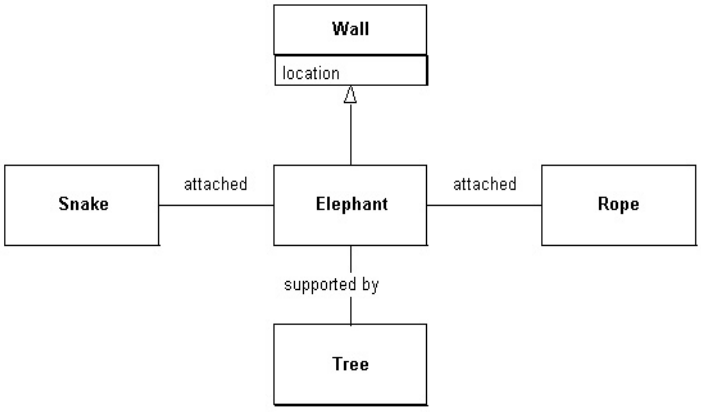
Four Separate Contexts: No integration



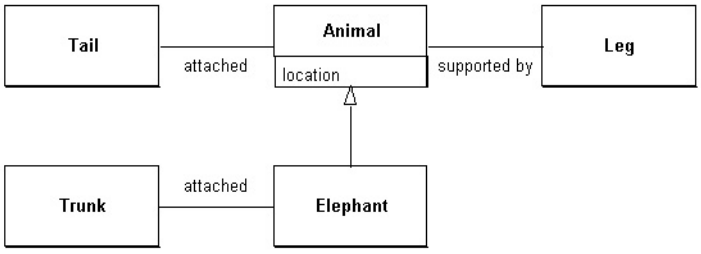
Four Contexts: Minimal Integration



One Context: Crude Integration



One Context: Deeper Model



The recognition of multiple clashing domain models is really just accepting reality. By explicitly defining a context within which each model applies, you can maintain the integrity of each, and clearly see the implications of any particular interface you want to create between the two. There is no way for the blind men to see the whole elephant, but their problem would be made manageable once they recognize the fact that they cannot.

Patterns for Applying Unification Context

Since unification contexts must always exist, and should always be declared explicitly, the question is where to draw them. There are an unlimited variety of situations that would lead to different decisions. The following set of patterns cover some of the most common and important cases, which should give a good idea of how to approach other cases. They are particular resolutions of a set of forces common to the basic problem of where to draw the boundary of a unification context.

Forces Favoring Larger Unification Contexts

- Integration improves flow between tasks
- It is easier to understand one model than two
- Translation between two models can be difficult (sometimes impossible)
- Shared terminology fosters clear communication

Favoring Smaller Unification Contexts

- Developing unified models requires a large overhead of communication between developers
- Developing unified models requires a high skill level of modelers to maintain the absolute consistency required
- Specialized terminology and models of subdomains can be optimized for particular needs

Fully Integrate Interrelated Processes

Context

Problem A system is being built which must manage a range of business processes that are interrelated. Although some interfaces to other systems will be needed, but the dominant concern is the tightly integrated heart. How can the new model maximize integration?

Forces Additional Forces Particular to Full Integration

- Unifying a complex model requires high modeling skill

Solution Define a single **unification context** for any part of the total system for which tight integration is the dominant concern and develop a model that is self-consistent and can be used for all needs within that context.

The project must be organized to provide very good communication between the people working within the context, and it must be recognized that additional skill and effort will be required to fully unify the model. Unified models are often found in a deeper level of abstraction, which requires a high skill level on the team.

This choice favors smoothness of integration over independence of parallel development efforts.

Related Sometimes, in spite of best intentions, subdomains within the boundary

Patterns of what was intended at a unification context may diverge. When this is recognized, it can be rectified by refactoring the models, but it is also valid to decide to split the context into two, allowing the two models to evolve in their own directions. (See *Parallel Development in Separate Contexts* and *Cater to Special Needs in Separate Contexts*)

Parallel Development in Separate Contexts

Context **Multiple Design Teams**

Problem If a large modeling project is divided between teams, subtle discrepancies in the semantics of related model elements creep in. These differences may not be obvious, but introduce incorrectness when the output of these different groups are brought together in attempts to integrate or to reuse elements. How can these errors be prevented while maximizing the speed of parallel development?

Solution Declare a separate unification context for each team and allow them to develop in parallel with minimum coordination overhead.

The integration needs must be identified between each of the resulting models, and a translation layer must be defined. No reuse of model elements should be attempted between unification contexts.

Where integration is not needed, or is relatively limited, this allows maximum parallel development and avoids corruption of the models with a minimum of cost.

Where integration requirements are extensive, the cost of translation goes way up. It may be necessary to increase coordination of the teams to make choices in the modeling to make translation easier, while still not requiring full unification.

Related Patterns Like any other aspect of modeling and design, decisions are not irrevocable. When integration needs or difficulties turn out to be more than expected, two contexts may be merged into a single unified model, at the cost of the additional modeling work and migration effort. (See *Fully Integrate Interrelated Processes*.)

Cater to Special Needs in Separate Contexts

Context **Serving Many Masters**

Problem How much do you need to tailor individual parts of the system to meet specialized needs? Different groups within the same business have often developed their own specialized terminologies, which may have diverged from one another. These local jargons may be very precise and tailored to their needs. Changing them (for example, by imposing a standardized, enterprise-wide terminology) requires extensive training and extensive analysis to resolve the differences. Even then, the new terminology may not serve as well as the finely tuned version they already had. How can modeling proceed, using the original local jargons, without introducing errors?

Forces Additional Forces Particular to Catering to Special Needs

- Jargon that is tuned to a particular need can be powerful
- Shared language improves communication and integration

Solution Declare a separate **unification context** for each part of the system that is to be based on a distinct terminology.

The integration needs must be identified between each of the resulting models, and a translation layer must be defined. No reuse of model elements should be attempted between unification contexts.

Where integration is not needed, or is relatively limited, this allows continued use of customary terminology and avoids corruption of the models with a minimum of cost.

Where integration requirements are extensive, the cost of translation goes way up. It may be necessary to increase coordination of the teams to make choices in the modeling to make translation easier, while still not requiring full unification.

Related Patterns Like any other aspect of modeling and design, decisions are not irrevocable. When integration needs or difficulties turn out to be more than expected, two contexts may be merged into a single unified model, at the cost of the additional modeling work and migration effort. (See *Fully Integrate Interrelated Processes*.)

Segregate Alien Components

Context **Alien Components**

Problem When distinct systems are being meshed, the problem appears as developers try to match related model terminology between systems. When a new system is being built that must have a large interface with a legacy system, the difficulty of relating the two models can eventually overwhelm the intent of the of the new model altogether, causing it to be modified to resemble the other system's model, in an ad hoc fashion. How can the new model be preserved without giving up necessary integration?

Forces Additional Forces Particular to Alien Components

- New functionality often requires modifying preexisting parts of the model
- The models of many systems is weak

Solution You have only two choices: you either adhere completely to the other system's model or you go your separate way.

It is seldom practical to adhere to model of a legacy or external system (after all, why are you building a new system?) but in the case that you do, you must do it wholeheartedly. You restrict yourself to extension only, with no modification. This might be appropriate in the case of peripheral extensions to a large system that will continue to be the dominant system. Examples of this choice include the light-weight decision-support tools that are often written in Excel or other simple

tools. This is not the problem context we are dealing with, though.

Typically, you will define a **unification context** for **each alien component** of the overall system. The system under design will be made up of one or more unification contexts. The integration needs must be identified between each of these and each alien component. A translation layer must be defined between each pair of models that have integration needs.

Where integration is not needed, or is relatively limited, this avoids corruption of the models with a minimum of cost.

Where integration requirements are extensive, the cost of translation goes way up. It may be necessary to make choices in the model of the system under design to make translation easier, without compromising the integrity of the model.

Resulting Context of Unification Context

Resulting Context

Unification context comes first and foremost because it is necessary to provide for a correct model. Logical consistency is now theoretically protected, and its related communication issues can now be managed. Powerful and correct models are possible. But if any large *unification contexts* are defined, they may still be too complex to fully comprehend as a whole, or to analyze completely. If many small unification contexts arise, the overall understanding may be difficult. This pattern should NOT be used to break down a model simply because it is too big to manage. This pattern is strictly for solving the logical consistency problem.

Related Patterns

Managing complexity is the focus of the remaining patterns in this paper. *Distillation* and *Large-scale Structure* address the problems of understanding the big picture, while *Modularity* allows analysis of isolated parts without considering the whole. (Other patterns are needed for translation between contexts, which is not covered in depth in this paper.)

Distillation

$$\mathbf{div}\,\mathbf{D} = \rho$$

$$\mathbf{div}\,\mathbf{B} = 0$$

$$\mathbf{curl}\,\mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\mathbf{curl}\,\mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

--- James Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 1873.

These four equations, along with the definitions of their terms and the body of mathematics they rest on, express the entirety of classical, nineteenth century, electromagnetism.

How do you focus on your central problem and keep from being drowned in a sea of side issues? Cutting across all *unification contexts* is the "vision" of the system – that which makes it valuable.

In designing a large system, there are so many contributing components, all complicated and all absolutely necessary to success, that the real essence of the system can be obscured and neglected.

How can we keep the focus on the aspects of the total system that are most valuable and grasp the vision of the system? Many of the model elements needed to support a large system are not central to the vision, and the primary value-added, of the system, yet these components can divert resources from the conceptual core of the system. Worse, it may become difficult to see the forest for the trees, and the conceptual core can be lost entirely.

Often a great deal of effort is spent on peripheral issues in the domain. I personally have witnessed two projects who have employed their best developers for weeks in redesigning dates and times with time-zones. While such components must work, they are not the conceptual core of the system. Even if some such generic model element is deemed critical, the overall domain model needs to make prominent the most value-adding and special aspects of your system and be structured to give that part as much power as possible. This distillation depends on the point of view of the designers, but can bring incredible clarity to an overall model.

The whole process of modeling is an exercise in distillation. 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. This is of a piece with that, but applied on a larger scale to the whole domain model, to bring into sharp relief the most valuable and least general features.

Boil the model down to the core domain, retaining only those aspects of the model that are distinctive and central to the purpose(s) of the intended applications. Be sure this core is particularly clear, and as powerful as is needed to accomplish the vision of the system. Keep other parts of the model as generic as practical, ideally using standard components. Justify and judge these other parts by how they support the distilled core.

This can be done in a variety of ways, that may be called for under different circumstances. These range from a simple *vision statement* to a *segregated core* that requires refactoring the model itself…

Domain Vision Statement

At the beginning of a project, the model doesn't exist, yet the need to focus its development is already there. Even in later stages of development there is a need for an explanation of the value of the system that does not require a significant study of the model.

Most projects write "vision statements" for management. The best of these documents clearly lay out the specific value the application will bring to the organization. Some of these describe the creation of the domain model as a strategic asset. Usually the vision statement document is abandoned after the project gets funding, and is never used in the actual development process or even read by the technical staff. These documents, or closely related ones that emphasize the nature of the domain model, can be used directly by the management and technical staff during all phases of development to guide resource allocation, to guide modeling choices, and to educate team members. If the domain model serves many masters, you can use this document to show how their interests are balanced.

Write a short (~1 page) description of the core of your domain and the value it will bring. Ignore those aspects that do not distinguish this domain model from others. Show how the domain model serves and balances diverse interests. Keep it narrow. Write this statement early and revise it as you gain new insight.

This still leaves the work of relating the vision to the specifics of the model to the interpretation of each individual…

Highlighted Core

It would be better not to leave the identification of the core elements up the vagaries of individual interpretation. Yet a very intrusive method that reorganizes the model is not feasible on all projects. Adding a UML stereotype or a special comment might be more suited to the situation.

Flag each element in the model that contributes to the essence, without particularly trying to elucidate that contribution.

The core of the domain is now clearly visible to those working with the model, with a fairly small effort and low maintenance, to the extent that the model is factored fine enough to distinguish the contributions of parts.

Segregated Core

Although the *vision statement* and *highlighted core* inform and guide, they do not actually modify the model itself. Now we come to the most aggressive form of distillation.

Elements in the model may partially serve the core and partially play supporting roles. Core elements may be tightly coupled to generic ones. The conceptual cohesion of the core may not be strong or visible.

This next step is not for everyone, but when appropriate, can be very powerful.

The model itself can be refactored in such a way as to separate the core from supporting players and strengthen the cohesion of the core.

To see how this can be accomplished, let's first examine the fractions we'll drive off on our way to distilling the essence of the domain model.

Generic Models

There is a part of your model that you would like to take for granted. It is undeniably of the domain model, but it abstracts concepts that would probably be needed for a great many businesses. For example, a corporate organization chart is needed in some form by businesses as diverse as shipping, banking, or manufacturing. For another example, many applications track receivables, expense ledgers, and other matters that could all be handled using a generic accounting model. Distillation calls for you to identify all generic aspects of your model and factor them out, with the result being a general model of the generic subdomain that has no trace of your specialties.

There are practical considerations when devising these generic models. While the concept of such a model may be applicable to many situations, you do not have to develop the model in its full generality. You can model only the part you need for your business, but you must be strict about keeping within the generic concept. Introducing industry-specific model elements will have two costs. First, you may need to expand the model later. Although you need only a small part of it now, your needs will grow. By introducing anything to the design that is not part of the concept, you make it much more difficult to expand the system cleanly without completely rebuilding the older part and redesigning the other modules that use it.

The second, and more important, reason is that those industry-specific concepts belong in their own models, and those models are even more valuable than these are.

There are some incidental advantages to this factoring. Sometimes it may be possible to find generic models in literature or even buy implementations in the form of frameworks, which implement a very abstract model which can be integrated with and specialized for your application.

Another advantage is that sometimes you can hire out-side design expertise to help in creating them, since it does not require deep understanding of your specialized domain. And confidentiality is less of a concern, since little proprietary information or business practice will be involved in these modules.

Over time, I believe our ideas of what constitutes the core model will narrow, and more and more generic models will be available as implemented frameworks, or at least as published analysis models as in [Fowler96]. For now, we still have to develop most of these ourselves, but there is great value in not confusing them with the core models.

Mechanism

No doubt about it, the mechanism by which the operations modeled in the domain are accomplished present some of the difficult problems in software development. But in most cases, these simply do not belong in the domain model. Mechanism should be factored out of the domain objects and then, if at all possible, should be encapsulated behind an abstracted interface that speaks in terms of the intention, rather than the means.

Take, for example, a system that must respond to real-world events. The domain model must define the nature of those events, and the manner of response, but the mechanism by which

those events are registered and propagated to the interested respondents belongs in a separate infrastructural framework.

A different sort of example is presented by a model of an organization chart. This model will represent that one person works for another, and will declare an interface by which such questions may be asked and answered. But the means by which this answer is obtained is of no concern to the domain model. Perhaps a generalized directed graph traversal mechanism is being used. The organization model could then simply state, using standard graph terminology, that each person is a node, and each relationship between people is an edge (arc) connecting those nodes. After that, presumably, mechanisms within the graph module can find the relationship between any two people. If this mechanism is incorporated into the domain model, you pay in two ways. The model is now coupled to a particular method of solving the problem, limiting future options, and, more importantly, the model of organization is greatly complicated and muddied. Keeping mechanism and model separate should result in more reuse of the intricate code for graph manipulation and a much clearer model of organizations.

Another way of saying this is that the domain model should be "declarative". It should state relationships and rules, but not how they are enforced; describe events and actions, but not how they are carried out; formulate the equation but not the numerical method to solve it. It should pose the question but not present the means by which the answer shall be found.

The one exception is when a mechanism is itself proprietary and key part of the value of the software. This is sometimes the case with highly specialized algorithms. For example if one of the distinguishing features of a shipping logistics application was a particularly effective algorithm for working out schedules, that mechanism stays.

Core Domain (Essence)

Factoring out the general mechanisms and generic business models leaves behind a focused model of your business, without distractions that add no particular value to the way you do your business. Finally we are looking at those parts of the model particular to representing your business domain and solving your business problems. **This is where the most value should be added.**

In a shipping application, these would be the model of how cargoes are consolidated for shipping, how liability is transferred when containers change hands, or how a particular container is routed on various transports to reach its destination. In investment banking, these would include the models of syndication of assets among assignees and participants.

Which aspects of the model constitute the essence is circumstantial. One application's generic model is another application's core. Many applications need a generic model of money that could represent various currencies and their exchange rates and conversions. But an application to support currency trading might need a more elaborate model of money, and would consider it part of the core. Even in such a case, there may be a part of the money mode that is very generic. That part would ideally be separated from the specialized part, continuing the distillation.

It is often not practical to hire out-side design expertise to help in creating the core domain, although there may be value in hiring domain experts, who know your business deeply and are experienced in analysis in that area. And if some aspect of your system is to be kept secret as a competitive advantage, this is it.

It is unlikely that the core domain can be purchased, although in the future, such industry-specific models may become available as frameworks. Some efforts in that direction have been made by specific industry groups, as with Sematech's CIM framework for semiconductor wafer fabs, and by technology companies, as with IBM's "San Francisco" project, which is an attempt to develop frameworks for a wide range of businesses. But the greatest contribution of such off-the-shelf software will be to supply you with the generic

models you need. Even if a framework is available for your core domain, it should be used with caution, and probably serve as a guideline. At best, a well-designed framework may be able to provide high-level abstractions that you can specialize for your use. If it does more than that, you should question whether this aspect of the model belongs in the core, or, if it does, why you are developing custom software in the first place.

Choosing your weapon

When to use this most aggressive distillation?

The greatest value-added of enterprise software comes from the enterprise specific aspects of the model. The segregated core will let you enhance the cohesion of the core. On the other hand, some elements of the conceptual core can have high cohesion and coupling with more generic elements, and these relationships will be made harder to understand. And it is more work.

The time to use this is when you have a large unification context that is critical to the system, but where the essential part of the model is being obscured by a great deal of supporting capability.

Therefore, when you decide to go down this path,

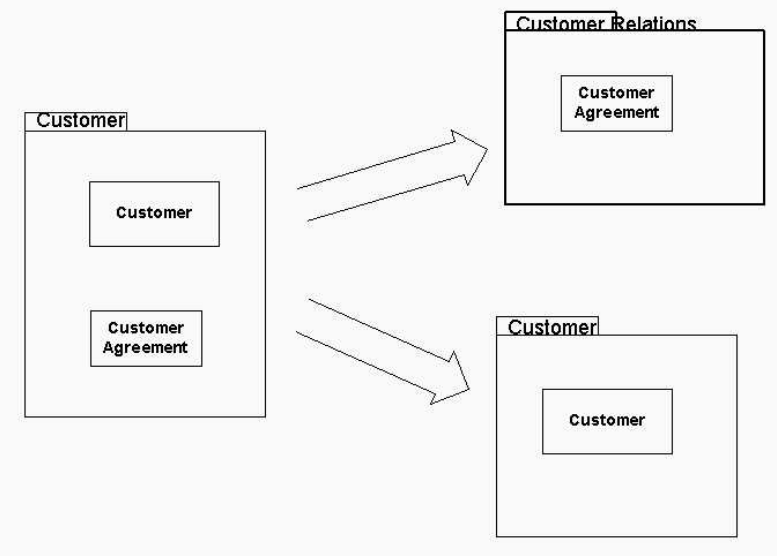
Separate the core concepts into separate objects in the model and place them in separate modules (applying the "*Modularity*" pattern) identified as the "core domain". Factor all generic or supporting elements into other objects and place them into other modules, even if this means refactoring the model in ways that separate highly coupled elements.

This can not usually be done completely, in any case, since distillation spans *Unification Contexts*. This pattern adds a consideration to the *modularization* decision process.

The core domain is the part of your system most focused on your particular business problem. By cleanly separating it from the other, more generic, business issues, you can express it powerfully and clearly. Other models can be clearly shown in their supporting role. I believe that tools will continue to evolve that will automate or prefabricate more and more of the other layers of an application so that increasingly the effort of modeling will be spent on the core business problems, greatly improving productivity and quality.

Example: Distilling the Shipping Model

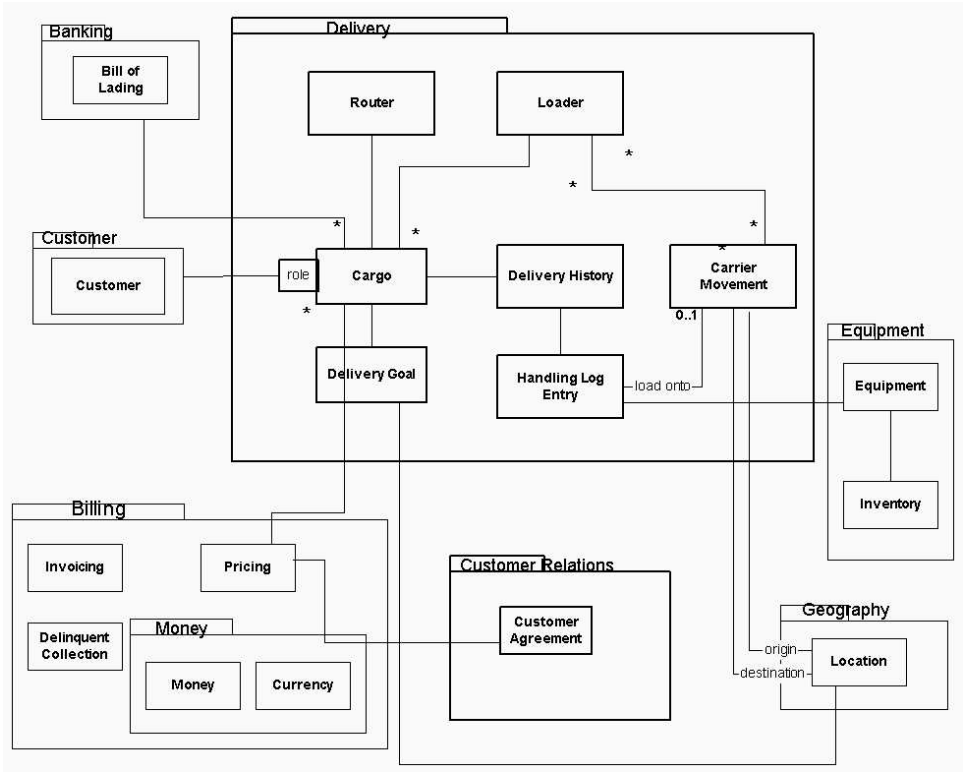
What is the essence of the shipping model? Usually good place to start looking is the "bottom line", certainly central to any business. But this application is not being designed for the CFO. It is going to be used by the front-line operators of the company. So let's relegate all money related issues to (admittedly important) supporting roles. What about customer? Certainly a customer focus is important to such an operation. Yet the basic model of a customer is pretty generic. We wish we didn't have to do this one yet again. What about those agreements, though? Shipping according to customer agreement is at the heart of the operation and of what this system is for. If we accept customer agreement as core, then we would like to separate it from the generic Customer. This leads us to split the Customer package into two:



I hate introducing new notation, but I know of no way of laying emphasis on some packages over others in UML, so I have drawn the core domain packages in a bolder line. In laying out diagrams of many packages, I attempt to convey the importance of those packages subjectively by placing them centrally, but this is only an aide and some notational distinction is needed to be unambiguous.

As an aside, we are going to end up with packages that show just one class. One reason for this is that the true complexity of those packages is much greater than can be shown in this example. Another is that I favor a sharp definition over an arbitrary size rule.

So, what are we left with? The obvious core domain of a shipping operations system is the routing and handling of cargo. But many systems might manage inventories of equipment, and geographic models are certainly potentially generic. Within the Billing module I've further separated the Money model, which is a generic model that there may be a lot of value in designing cleanly. This initial distillation gives us a diagram something like this.



In this case, the classes are the same as before distillation, although sometimes distillation leads to refactoring to separate the generic and domain specific responsibilities of the classes themselves. But although the classes haven't changed, it is now easier to explain clearly what this model is about, and think about allocating our efforts to directly support the core domain.

Modularity

The entirety of the system was broken down into *unification contexts*, that insure that a single consistent model is in force within a given scope. But if the model within a single unification context is still large and complex, the problem needs to be brought down to size. *Modularity* applies within a single *unification context*.

There is a limit to how many things a person can think about at once. If the full domain is modeled as a single web of objects it is almost impossible to understand and will surely be impossible to implement. How can a very large model be made understandable in detail?

This is well trodden ground, and does not need a great deal of explanation. Something so large must be partitioned into smaller parts, but not arbitrarily.

Two factors are usually considered in trying to draw lines: low coupling between modules and high cohesion within a module. These terms have been around for a long time. One patterns-style explanation can be found in [Larman98].

Whenever two model elements are separated into different modules, the relationships between them become less direct than they were, which makes them harder to understand and increases overhead of managing the design (and potentially of run-time performance). At the same time, the complexity of the relationship between the modules is increased. Enter low coupling. By keeping strongly interrelated elements in the same module and moving unrelated elements to other models, the cost of these inter-module relationships is minimized. Low coupling between modules makes them easier to manage in isolation, without having many complex interactions to consider. It also makes them easier to unit-test.

The elements of a good model should have a synergy that makes them more powerful and expressive together than they are separately. This tends to happen when there is a clear conceptual connection between them. The principle of high cohesion states that elements should be grouped with other elements that, together, form a richly expressive, cohesive whole. In such a case, powerful capabilities can be designed at the micro-level of a module, a scale of complexity a human mind can handle. Usually, high cohesion coincides with low coupling, so that there is not much tension between these two goals.

A third factor that comes into play is *distillation*, especially if a *segregated core* is being used. In this case, elements deemed to be at the distinctive conceptual core of the model are separated from more generic elements. Ideally, this leads to a model with high cohesion within the core, maximizing the power and clarity of that critical part. On the other hand, unfortunately, the core modules often end up fairly tightly coupled to generic elements.

Modularity is not an afterthought, as if a fully formed model were taken and breaking it up as-is. The modules usually co-evolve with the rest of the model. This means that the responsibilities of model elements will tend to be factored and refactored with low coupling, high cohesion, and distillation in mind.

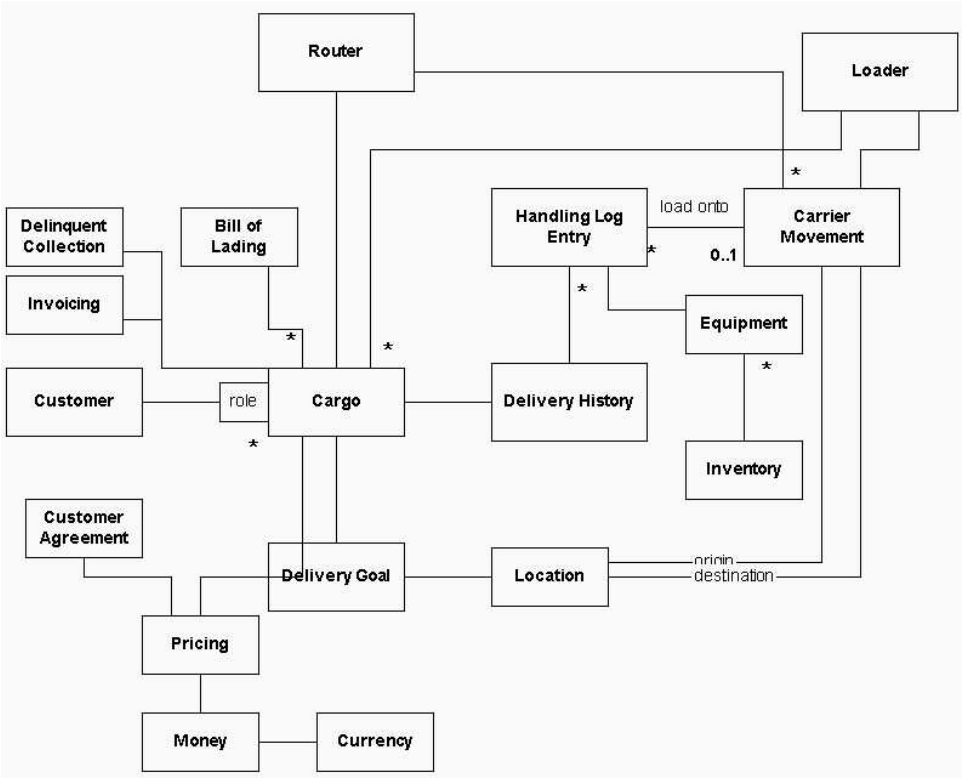
Rather than developing a single large model, develop many smaller, interrelated ones, each of which has a conceptual focus. Show relationships between modules in views that exclude interior detail of the modules. Choose the boundaries so that there is high cohesion within each module and low coupling between modules.

Modularity resolves most problems of understanding and analyzing the details of a model. It may also make analyzing the system as a whole more manageable, since the relationships between modules can be studied rather than the much more numerous relationships between individual objects. If the model has been *distilled*, there will also be focal points to guide understanding.

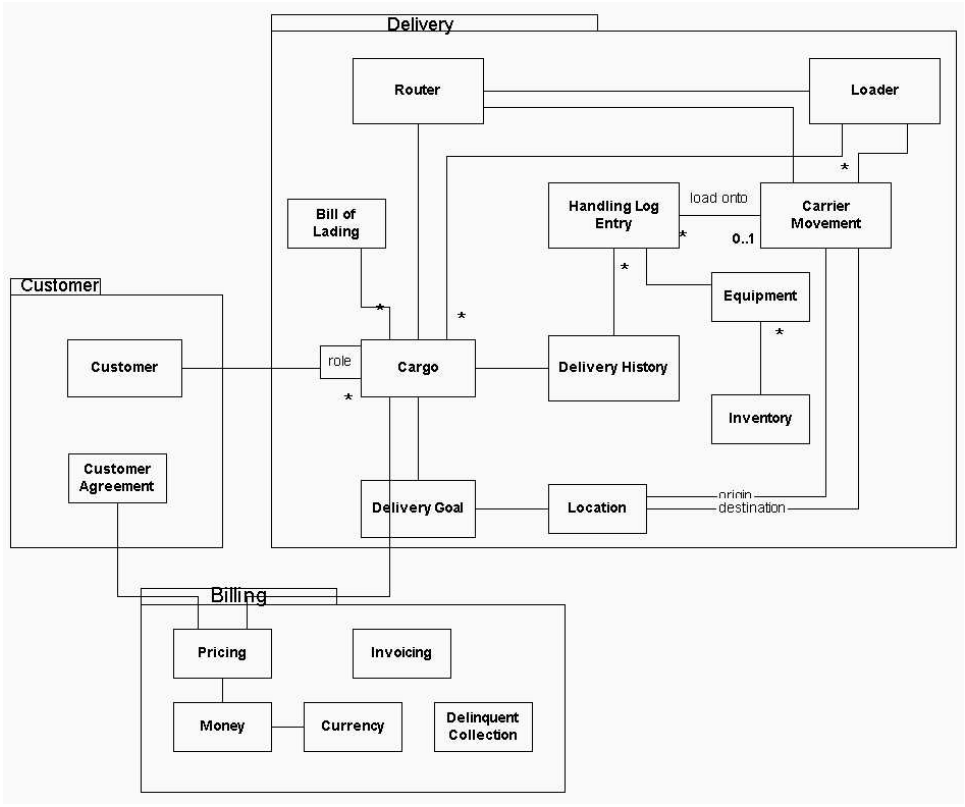
But beyond a certain size, a model may still be unmanageable. Ensuring logical consistency within a module may be easy, but throughout a large *unification context* it may be difficult when many modules are interacting. For the same reason, it becomes difficult to understand the role of one small part in the whole, and to make design decisions about small parts that fit into a "big picture". It may be necessary to impose a *large-scale structure*.

Example: Modularizing a Shipping Model

Let's Consider a system for the management of cargo shipping. It will track cargoes, take bookings, invoice customers, route cargoes onto appropriate carriers to send them toward their destination, and no doubt many other useful things. A simplified domain model to serve this purpose might look like this.



Even neglecting many associations that would exist but have been left out, this is pretty confusing. I'm not even going to try to explain the parts until I've broken it into modules. Now, I do not mean to imply that these principles, modularity and the others to follow, should be applied after the fact – far from it. This is presented more to show the various end-points that might be reached by applying these principles, which are likely to be used together and iteratively. Few would present a large model in one piece like this. Here is an off-the-cuff modularized view of the same model.



Modularity is used by almost everyone, and is well explained in many places. Curiously, though, the guidelines are quite abstract. Cohesion and coupling are the basic principles applied, but it really comes down to an intuitive division into related objects. The intuition of some is refined by talent and experience, and iteration also refines the partition of a particular design. This is one intuitive breakdown. It could be refined, certainly, and the sections to come will lead to refinement following some additional guidelines.

This is certainly easier to manage and explain. Some objects are fairly self-explanatory, such as cargo, customer, and location. Carrier movement represents a ship, train or other carrier moving from one location to another, potentially carrying cargo. Router and load lister work together to get the right cargoes onto the right carriers. The router figures out where the cargo should go, the load lister figures out which cargoes should be loaded onto a carrier (and which get bumped). The details are not important, since this is not about shipping but about breaking down a model. It will also become easier to explain as we apply the remaining principles.

Large-scale Structure

Even with a *modular* breakdown, a large model can be too complicated to grasp. The strict segregation imposed by *unification contexts* prevents confusion, but can actually make it harder to see the system as a whole. *Distillation* helps, by focusing the attention on the core and presenting the other models in their supporting roles, but the relationships can still be too confusing without some overarching principle that allows elements to be interpreted in terms of their role in patterns that span the whole design.

Can't see the forest for the trees.

We need to be able to understand the role of an individual part in the whole without delving into details of the whole.

The forces applying to this pattern are related to those of the *unification context*, but here the emphasis is on comprehensibility, whereas there it was on correctness.

- Applying any principle to all parts of a system makes it easier to understand and manage as a whole.
- Individual parts have natural or useful ways of being organized and expressed that may not apply to the whole, so imposing global rules makes these parts less ideal.
- Designers may have no control over the model of some parts of the system, in the case of external or legacy systems.

Structure may be confined to one *unification context* but will usually span multiple *contexts*, providing the conceptual organization to hold them together. A good structure gives insight into the model and complements distillation.

This pattern favors manageability of the model as a whole over optimal structuring of the individual parts. Therefore, there will be some compromise between powerful structure and freedom to express individual components in the most natural way. This can be mitigated by careful selection of the structure and by avoiding over-constrictive structures. When the unification contexts were kept separate to allow for the tailoring of models to the needs of specific applications, a weak structure (if any) is appropriate to avoid forcing a concept upon the subdomain that is not optimal for it.

Large-scale structure is not needed for systems that are simple enough to be understood when broken into modules. It should not be applied unless a structure can be found that greatly clarifies the system without forcing an unnatural constraint into model development.

Strong vs. weak structures

Unlike *unification context*, a *large-scale structure* is an optional pattern to be applied when costs and benefits favor it. There is also a lot of room to trade-off between tight or "strong" structures and relatively loose or "weak" ones. Because diverse parts of the model have to be handled, in most cases, such structures need to be loose and conceptual, rather than tight frameworks, but there are exceptions.

A strong structure affects individual modeling decisions throughout the system. When dealing with a large, complex *unification context*, a strong structure can be helpful in resolving the problem of finding and maintaining a rigorously consistent model of such complexity. The strong structure can also be helpful when *unification contexts* were chosen to facilitate parallel development of new software. It can ease integration and may even leave the door open to merging contexts in later refactoring. Easing off to a medium strength in these cases can lower overhead without loosing much of the interpretability that is the main purpose of the pattern. (This would probably forgo the easy merger of contexts, but that concern is properly addressed through the *unification context* pattern, anyway.)

When some unification contexts involve external systems with completely distinct models, a strong structure is completely unworkable. A weak structure may be able to provide some guide to understanding the system as a whole without demanding the revision of models to fit its concepts. Even this may not be practical when there is no control of the external model (the usual case). In such a case, the structure may at least be able to communicate the role of the interface to the external system in the model being created. Such an "integration model" can be especially useful in systems that are mostly composed out of preexisting subsystems.

Many structures can be tuned in strength for different parts of the system, so that it could be applied in a strong way in the main system under design and then weakly to help communicate the role of external systems that are being integrated.

Devise a pattern of roles and relationships that will span the entire system and which allows some understanding of a part's place in the whole even without detailed knowledge of the part's responsibility. Then make all affected modules follow a consistent pattern that can be learned once and used to interpret all.

This is really a pattern for a pattern, so specific patterns for imposing *large-scale structure* are needed.

Although a lot of work has been done on large-scale architectural structure for software systems as a whole, little has been published on the structuring of the domain model. Some approaches that have been tried weaken the object oriented paradigm, such as those which break down the domain by an application task being performed.

An approach more harmonious with objects is the *dominant metaphor* [Kent Beck, ~1999?]. This is favored by the Extreme Programming community. A model can be organized around a dominant metaphor that need not be related to the domain itself (such as structuring a payroll system based on a manufacturing metaphor). It can provide a weaker structure that may be easier to apply across contexts.

Abstract Domain Framework is an example of a very strong structure, high-level abstractions are identified and shared across a breadth of the system while specialization occurs in modules. This would generally be applied within a unification context (and, indeed, can help to manage the unification) but it can also apply across contexts when specialization within the modules diverges or the framework is very conceptual.

Here I'll explore in depth one such structuring pattern, *layers of responsibility,* that I have used to good effect on two large projects. It can range from a strong structure to medium strength.

Layers of Responsibility

This pattern for *structuring* a model is built upon *layering*, one of the most successful architectural design patterns, and *responsibility driven design*, one of the most fundamental and successful object design philosophies.

Some domains have a natural stratification. Some concepts and activities take place against a background of other model elements that change independently and at a different rate for different reasons.

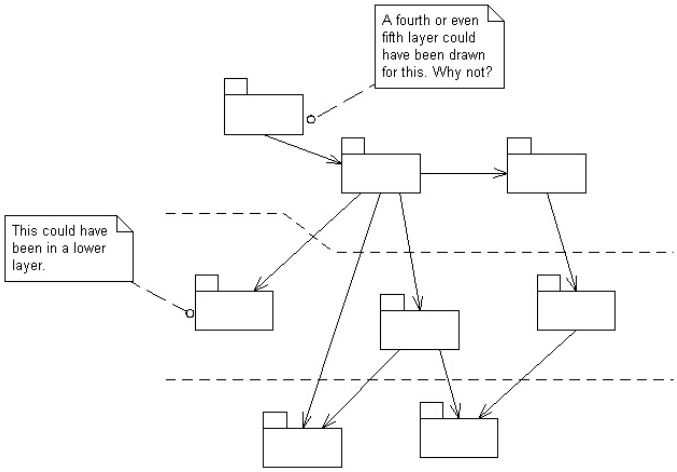
How can we take advantage of this natural structure, to make it more visible and useful? It immediately suggests layering. Layers are partitions of a system in which the members of each partition are aware of and able to use the services of the layers "below", but unaware and independent of the layers "above". (Or vice-versa, but I'll use the convention that "higher" layers depend on "lower" layers. Physical arrangement does not have any meaning in UML,

only topology does. But the physical layout of the diagram can be suggestive and can help make the model easier to interpret.)

When the dependencies of modules in the model are drawn, they are often laid out so that a module with many dependents appears below them. In this way, layers sometimes sort themselves out in which none of the objects in the lower levels are conceptually dependent on those in higher layers. The layering pattern that best fits is the variant called *Relaxed Layered System,* [BMRSS96], p. 45, which allows components of a layer access to any lower layer, not just the one immediately below.

But this ad hoc layering, while it can make tracing dependencies easier, and sometimes makes some intuitive sense, doesn't give much insight into the model or guide modeling decisions. We need something more intentional.

Figure 1: Ad hoc layering: What are these packages about?



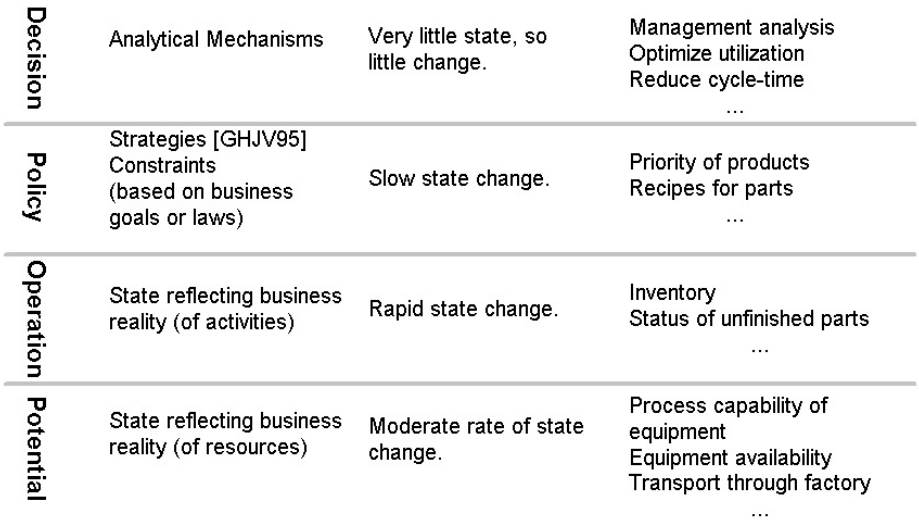
Fundamentally, most object design principles rest on *responsibility driven design,* which entails assigning a narrow set of closely related responsibilities to each object. In conventional responsibility driven design, each individual object has hand-crafted responsibilities. There are no guidelines, no uniformity, and no ability to handle large swaths of the domain together. To give coherence to a large model, it is useful to give some structure to the assignment of these responsibilities. In a model with a natural stratification, conceptual layers can be defined around major responsibilities, uniting these two powerful principles.

These responsibilities must be considerably broader than those typically assigned to individual objects, as shall be seen below. As individual modules and objects are designed, they are factored to keep them within the bounds of one of these major responsibilities. This named grouping of responsibilities by itself could enhance the comprehensibility of a modularized system, since the responsibilities of modules could be more readily understood, but a jump in power comes when it is combined with the layering principle.

Some widely useful layers

It isn't necessary to start from scratch in defining layers for each new model. Certain layers show up in whole families of related domains. At this point it will help to discuss a particular layered responsibility scheme that has been of value to me in at least three domains.

Figure 2: Layering fixed capital operation (e.g. a factory)



In a surprising range of typical business enterprise systems, almost all responsibilities can be placed into the four layers in the figure. The layers have different responsibilities and different visibility to other layers. They also differ in the their rate of change and the amount of state they encapsulate. These are guidelines only, though. For most purposes you still need flexibility in those areas.

Now we need a definition for each layer:

- *Potential*: What can be done? Never mind what we are planning to do. What could we do? The resources of the organization, including its people, and the way those resources are organized are the core of potential. Also contracts with vendors define potentials. This layer exists in all business domains, but is most prominent in those, like transportation and manufacturing, that have a relatively large fixed capital investments that enable the business.
- *Operation*: What is being done? This is what we really have done with those potentials. Like Potential, this should reflect the reality of the situation, rather than what we want it to be, but in this layer we are trying to see our own efforts and activities. It is very typical of operational objects to reference or even be composed of potential objects, while a potential object shouldn't reference the operations layer.

In many, or even most, existing systems, these two layers cover everything. One of the many advantages of layers is that the lower layers can exist without the higher ones. The higher layers can later be added, with some change to the lower ones. This is one of the ways projects phase delivery.

But in more ambitious systems, this simple tracking is not enough. The upper two layers add the intelligence.

- *Policy*: What are the rules and goals? Rules and goals are mostly passive, but constrain applications and decision making components when performing transactions on the operational level. Goals can often follow the "Strategy" pattern [GHJV95].
- *Decision*: What action should be taken or what policy should be set? The final layer is for analysis and decision making. It can use the state of the Potential and Operations layers, including historical state, to find opportunities for current operations. The Decision layer provides the means to seek the goals set by Policy, constrained by the rules set by Policy.

Choosing appropriate layers

While these four layers are applicable to a range of enterprise systems, they do not capture the salient responsibilities of all domains. In other cases, it would be counterproductive to try to force the design into this shape, but there may be a natural set of responsibility layers that do work. For a domain completely unrelated to those we've discussed, these might have to be completely original. But when working with a new domain that has some similarity to those we've worked with before, we can often use responsibility layers we've used before, perhaps with a substitution or modification.

For example, as was mentioned above, the Potential layer is most prominent in businesses where capability is determined by capital investment, whereas, in financial services or insurance, to name two, the potential is to a large extent determined by current operations. An insurance company's ability to take on a new risk by underwriting a new policy agreement is based on the diversification of its current business.

In these cases, what often comes to the fore is commitments made to customers. This suggests the merger of the Potential layer into the Operations layer and the insertion of another layer, "Commitment", between the Operational and the Policy layers.

- *Commitment*: What have we promised? : This layer has the nature of policy, in that it states goals that direct future operations, but it has the nature of operations in that commitments come about and are changed as a part of normal business activity.

Figure 3: Layers when operations determine resources (e.g. commercial lending)

Decision	Analytical Mechanisms	Very little state, so little change.	Risk analysis Portfolio analysis Negociation tools ...
Policy	Strategies [GHJV95] Constraints (based on business goals or laws)	Slow state change.	Reserve limits Asset allocation goals ...
Commitment	State reflecting business reality (of activities)	Moderate rate of state change.	Customer agreements Syndication agreements ...
Operation	State reflecting business reality	Rapid state change.	Status of outstanding loans Accruals Payments and distributions ...

The rule of four

The Potential and Commitment layers are not mutually exclusive. A domain in which both were prominent, say a transportation company with a lot of custom shipping services, might use both. There is a value to keeping the layering system simple, though, and going beyond four or possibly five becomes unwieldy.

"Strong" structure, constraining relationships between layers

These responsibility categories can help factor the model in a consistent way, but in some circumstances it is useful to go a step further. In addition to partitioning the responsibilities of

the model, we can describe the relationships and possible communication paths between members of the different layers.

I've stated in a general way that lower layers should not know about higher ones, but until this point the decision as to how this should be done has been left to be made case-by-case during the modeling process. This is generally appropriate, but sometimes you may want to constrain the relationships between the layers more explicitly.

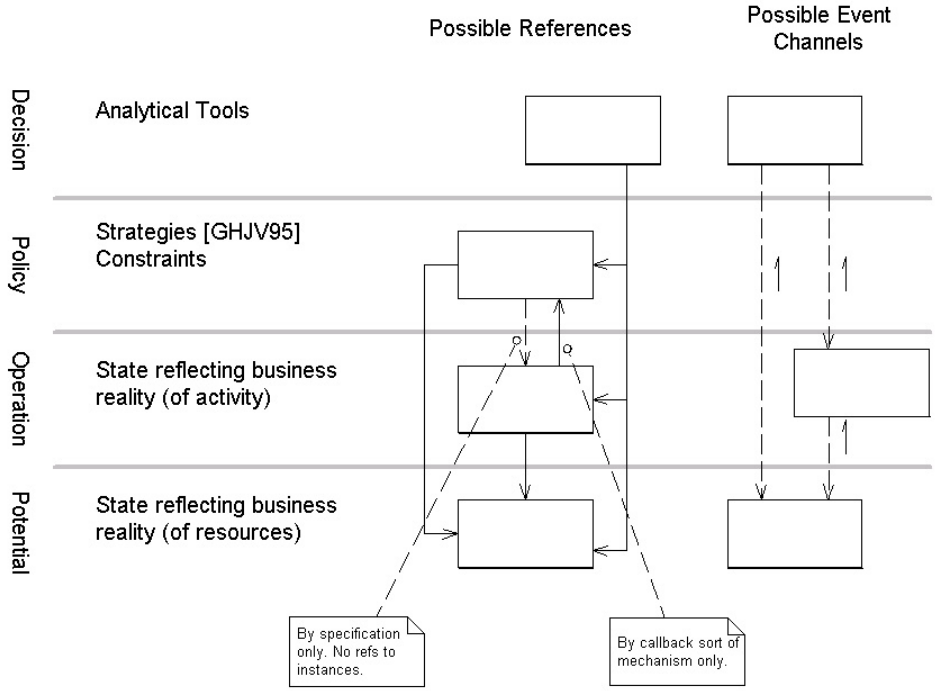
For example, intentional state-changes in the *operations* layer are made in accordance with the *policy* constraints. In a factory, a part will be routed to a machine that has the process capability called for in its recipe. But operations must reflect the real world, which means that sometimes the part will be put in the wrong machine. The real-world information must be accepted unconditionally. At this point we would like the system to strive to realign itself with policy, perhaps by rerouting the part to a repair process or scrap. But operations does not know anything about policy. What a spot.

One solution is to use an observer. Following this pattern, the operations objects broadcast events whenever their state changes. Policy layer objects listen for events from the lower layers. When an event occurs that violates a rule, the rule executes an action (which is part of the rule's definition) that makes the appropriate response.

In the banking example above, the value of assets change (operations), shifting the value of segments of our portfolio. When these exceed portfolio allocation limits (policy), perhaps a trader is alerted, who can buy or sell assets to redress the balance.

Now, we could figure this out for each relationship, or we could decide once and for all that this is how policy and operations objects should interact. If we make a set of such constraints that govern the interactions, we'll have what I call a "strong" structure. A strong structure is not practical in many situations, but it does provide the maximum affect of uniformity.

Figure 4: A structure that strongly constrains the model



Summing up

Layers of responsibility can be applied in a variety of ways and at different strengths. The overriding issues are that it should give insight into the domain and that it should make development easier. Design structures make the complex relationships between the parts which helps deal with large unification contexts. It can also give shortcuts to design decisions that could, in principle, be found by working on the individual object level, but which would, in practice, take too long and have inconsistent results. These layers help to satisfy the various ends to which the system is being put without randomly pulling the models in different directions until they fragment into redundant expressions of the same fundamental domain or become bloated with poorly factored behavior. Of course, continuous refactoring is still necessary, but this will make it a more manageable process, and can help make different people come up with consistent solutions.

Identify the natural strata in the domain and assign them broad abstract responsibilities. Factor the responsibilities of objects and modules to fit within the responsibility of a specific layer. Define the relationships between layers to a strength appropriate to your circumstances and then highlight exceptions to these relationships when they occur in the model.

<<<Put this near beginning of examples>>>

Granted that this seems like overkill at this point. There is an inherent difficulty of demonstrating by example a technique whose primary usefulness is in the handling of complexity, since any example that can be clear enough to illustrate the principle without exhausting the reader with detail will necessarily be too simple to really benefit from the additional layering. But imagine its application to a system of hundreds of domain objects.

Applying *Large-scale Structure* to the Shipping Model

Layering seems like a natural fit for our shipping model. It is quite reasonable to discuss shipping schedules without referring to the cargoes aboard the ship. It is harder to talk about tracking cargo without referring to a shipping schedule. The conceptual dependencies tend to be pretty clear. I'll try applying the first specific layering scheme presented in the text – potential, operation, policy, and decision.

Identifying "Potential" in the Shipping Model

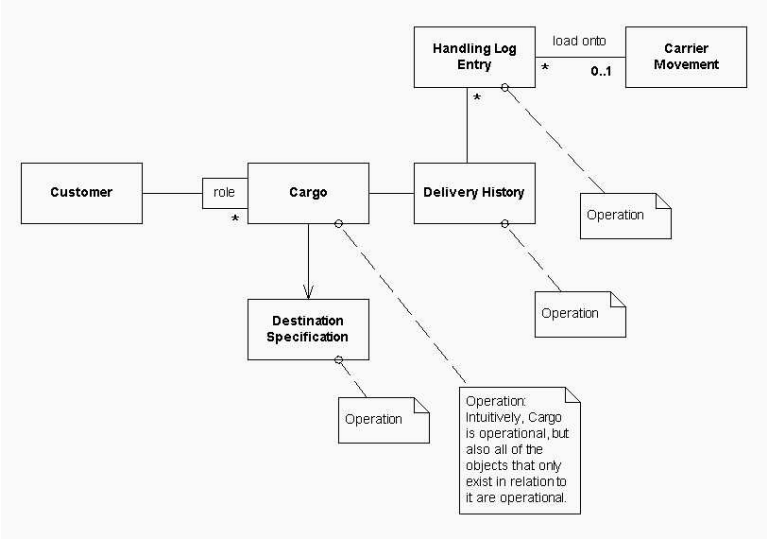
Scheduled carrier movement is potential, because this is a resource to be used to accomplish the goal of moving **cargo**. This is true whether the vehicles are owned by the company or space is leased on the carriers of other companies.

Customer is also potential, since the hope is to cultivate ongoing relationships so that they reflect our ability to make money in the business. (It could be argued that customer is operation. This would fit better in a business where repeat business did not dominate and customers were chiefly of interest in a specific shipment.)

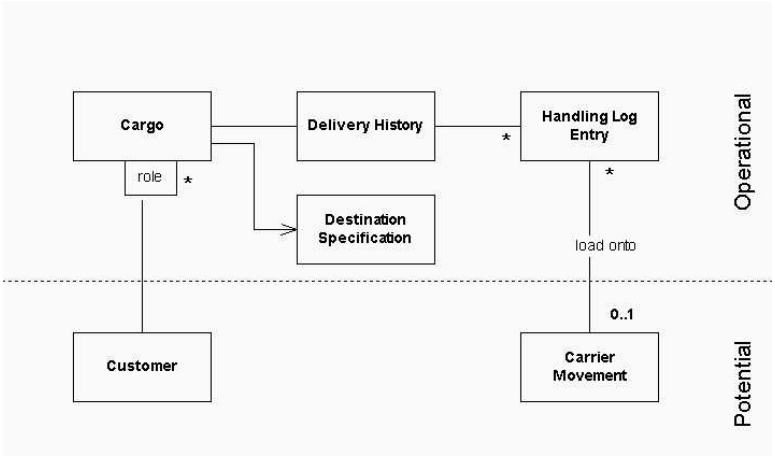
"*Operational*" Responsibilities

In our shipping application, the **cargoes** that are actually booked or in transit are the main operational objects. Of course, that includes the **delivery history**, even though that object is an aggregation of **carrier movements**, of

the potential layer. It is very typical of operational objects to reference or even be composed of potential objects.



So, separating the bottom two layers yields this view.



Policy in the Shipping Model

Pricing and **bill collecting policy** are classic policies. Since we don't want to add a commitments layer, **Customer Agreements** also constitute.

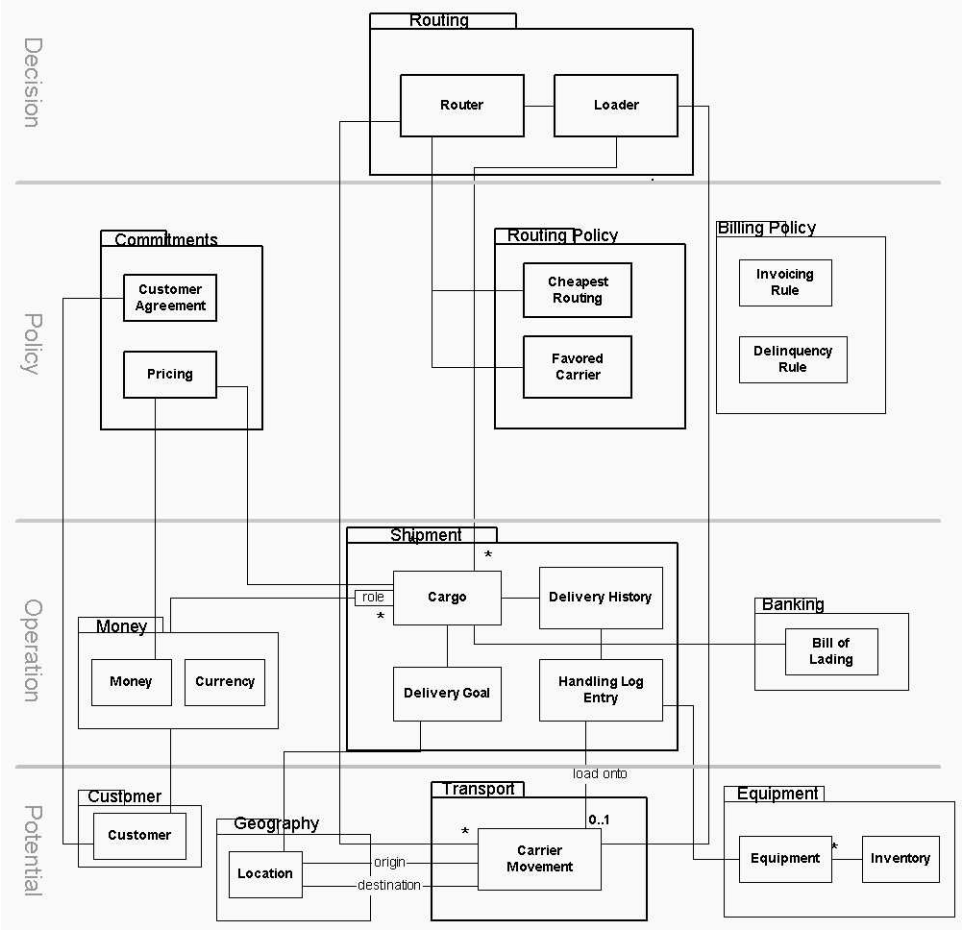
<< The following refers to an example in "Constructing the Domain": Incidentally, using the structure gives us a shortcut to the earlier discussed decision not to have **cargo** derive **enterprise zones**. Enterprise zones, which represent corporate strategy, are part of the policy layer. Cargo (operations) is not supposed to be dependent on higher layers. The original justification is the fundamental one, but shortcuts, or at least hints, to the right decision are important when hundreds of decisions like this one have to be made. It is also an important validation of the choice of structure when you find that the modeling decisions it pushes you toward make sense in their own right. >>

Decision Mechanisms in the Shipping Model

The original requirements listed didn't have any obvious decision making. In this kind of system, likely uses of decision making would be automatic routing of cargoes (**Router**) and listing the order of loading of cargoes (**Loader**).

A Final Look at the Shipping Model

We've modularized, distilled, and layered the shipping model, and come up with something like this.



On a real project, there would be additional iterations that would embellish and refine it, of course, but here we've almost come to the end of the line.

An Unexpected New Feature

Ah, but we simply wouldn't have covered the territory without dealing with an unexpected change. Enterprise systems are never conceived initially in their entirety. They grow. Our example shipment system started with just these requirements:

1. Track customer cargo
2. Book cargo in advance
3. Send invoices to customers

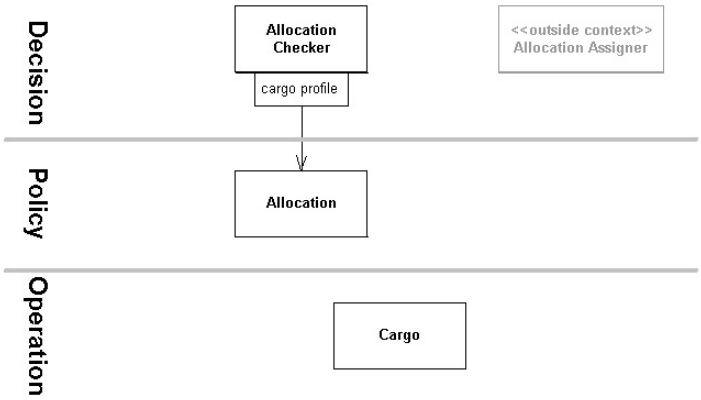
A simple object model emerged. Now the first major new functions are going to be added: Allocation Checking.

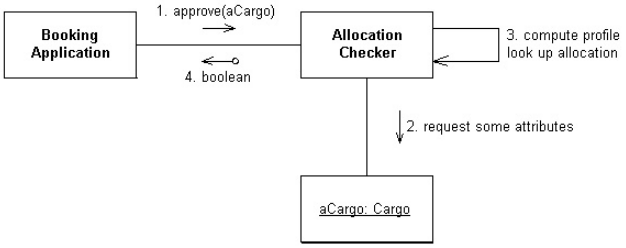
The sales division of our imaginary shipping company uses other software to manage their client relationships, sales projections, and so forth. One feature they use supports yield management, which is a process of choosing which cargoes to accept to maximize profit. Shipping companies don't want to fill a ship with discount cargo, but they don't want to sail with empty space, either.

They want to integrate with our system in a limited way. They will use their current software to work out the allocations, which will be exported into our system. The allocations will regulate which bookings the company accepts. When each new **cargo** will be profiled and categorized. If the allocation for its category has been used up, the booking will be rejected. Otherwise, it is accepted and the allocation remaining is reduced.

The first issue is unification context. The "alien" software will be taken as a separate unification context. The model needed for allocation checking, we'll incorporate into our main unification context.

Keeping in mind our layering scheme, the team decides to model it something like this:





The only remaining issue is the translation between the two contexts. In this case, the only information that is being exchanged is (as modeled in our context) the **Allocation** and the **Cargo Profile**. Translations must be defined between these and the equivalent representations in the alien tool.

Conclusion

Using these principles provides many advantages. It gives guidance that may lead more quickly to good factoring, and it gives consistency that can help coordinate the work of different people over long spans of time. It help keep the focus on the main goals of the system.

It is not meant to be a rigid methodology, but a box of tools, to be used when needed, individually and in combination.

And, as a pattern language, it also adds to the vocabulary of the project team, allowing better analysis and discussion of the state of the model regardless of whether any of the patterns are actually applied.

Vocabulary of the Pattern Language

The Context

Some combination of the following contextual elements appear in most enterprise modeling situations. Enterprise system modeling shares much with other software design, and these patterns should be applicable in other situations where these contextual elements dominate

1. **Large Model:** Specifically, when a model becomes too complex for its own designers to understand in detail and as a whole at the same time.
2. **Multiple Design Teams:** When more than one team is working in parallel on different parts of the system.
3. **Alien Components:** When the new system must integrate with a system over whose design the designers of the current system have little control. Typically, these would be legacy systems or external systems.
4. **Serving Many Masters:** Different users of the system will have different demands that are not just different steps in the same business process but actually distinct business goals.

The Problems

Problems of modeling usually revolve around trying to meet one or more of these goals.

1. **Correct**
2. **Integrated**
3. **Understandable**
4. **Flexible**

The Tools

The patterns in this paper present solutions base on these four tools, or principles, each of which is defined in its own section.

1. **Explicit Unification Context**
2. **Distillation**
3. **Modularity**
4. **Large-scale Structure**

References

[BMRSS96] Bushman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. P. *A System of Patterns.* Wiley.

[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 Rob Mee and Alistair Cockburn for reading early drafts and giving valuable feedback. Thanks also to Gerard Meszaros for acting as shepherd for the paper in preparation for PLoP 1999.