Domain-Driven Design

Pattern Summaries

Excerpted from Domain-Driven Design: Tackling Complexity in the Heart of Software, by Eric Evans, Addison-Wesley 2004.

Part I

Putting the Model To Work

UBIQUITOUS LANGUAGE

To create a supple, knowledge-rich design calls for a versatile, shared team language, and a lively experimentation with language that seldom happens on software projects.

A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design. The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). And even the same person uses different language in speech and in writing, so that the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in writing.
Translation blunts communication and makes knowledge crunching anemic.
Yet none of these dialects can be a common language because none serves all needs.

Therefore:

Use the model as the backbone of a language. Commit the team to exercising that language relentlessly in all communication within the team and in the code. Use the same language in diagrams, writing, and especially speech.
Iron out difficulties by experimenting with alternative expressions, which reflect alternative models. Then refactor the code, renaming classes, methods, and modules to conform to the new model. Resolve confusion over terms in conversation, in just the way we come to agree on the meaning of ordinary words.
Recognize that a change in the language is a change to the model.
Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.
With a UBIQUITOUS LANGUAGE, the model is not just a design artifact. It becomes integral to everything the developers and domain experts do together.

Also,

Play with the model as you talk about the system. Describe scenarios out loud using the elements and interactions of the model, combining concepts in ways allowed by the model. Find easier ways to say what you need to say, and then take those new ideas back down to the diagrams and code.

MODEL-DRIVEN DESIGN

Tightly relating the code to an underlying model gives the code meaning and makes the model relevant.

If the design, or some central part of it, does not map to the domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between models and design functions are difficult to understand and, in practice, impossible to maintain as the design changes. A deadly divide opens between analysis and design so that insight gained in each of those activities does not feed into the other.

Therefore,

Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software, even as you seek to make it reflect deeper insight into the domain. Demand a single model that serves both purposes well, in addition to supporting a fluent UBIQUITOUS LANGUAGE. Draw from the model the terminology used in the design and the basic assignment of responsibilities. The code becomes an expression of the model, so a change to the code may be a change to the model. Its effect must ripple through the rest of the project's activities accordingly. To tie the implementation slavishly to a model usually requires software development tools and languages that support a modeling paradigm, such as object-oriented programming.

HANDS-ON MODELERS

If the people who write the code do not feel responsible for the model, or don't understand how to make the model work for an application, then the model has nothing to do with the software. If developers don't realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it. Meanwhile, when a modeler is separated from the implementation process, he or she never acquires, or quickly loses, a feel for the constraints of implementation. The basic constraint of MODEL-DRIVEN DESIGN—that the model supports an effective implementation and abstracts key insights into the domain—is half-gone, and the resulting models will be impractical. Finally, the knowledge and skills of experienced designers won't be transferred to other developers if the division of labor prevents the kind of collaboration that conveys the subtleties of coding a MODEL-DRIVEN DESIGN.

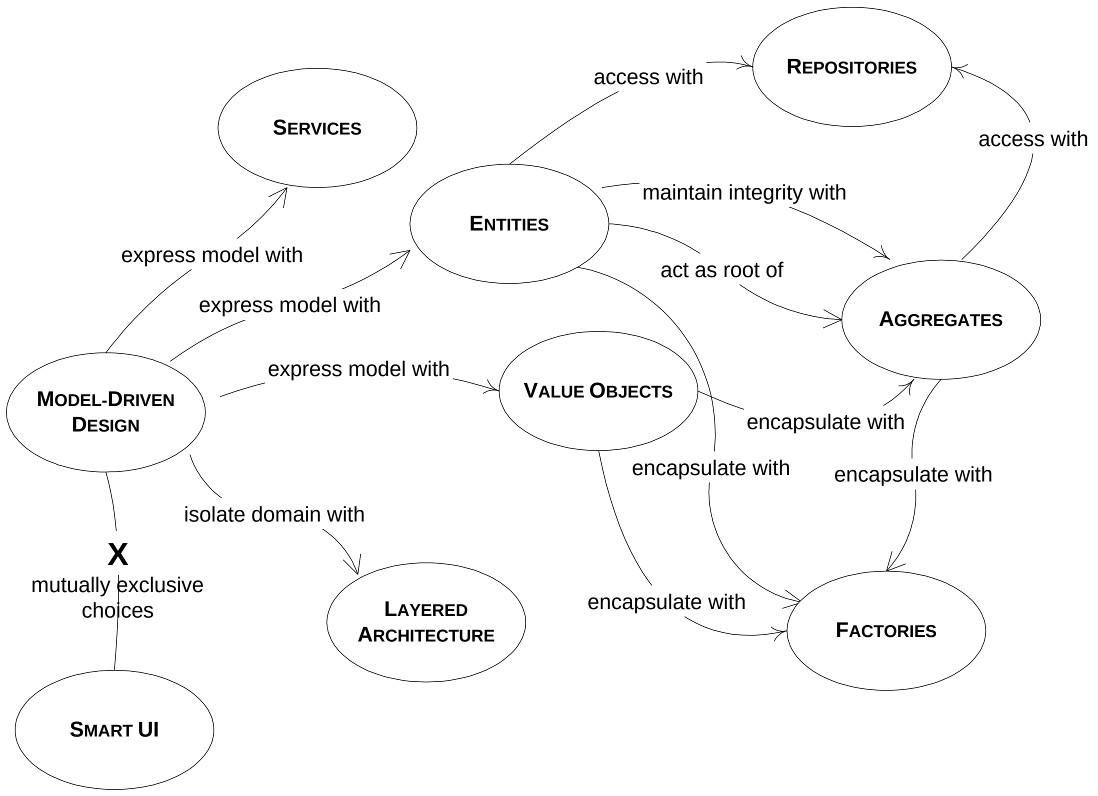
Therefore,

Any technical person contributing to the model must spend some time touching the code, whatever primary role he or she plays on the project. Anyone responsible for changing code must learn to express a model through the code. Every developer must be involved in some level of discussion about the model and have contact with domain experts. Those who contribute in different ways must consciously engage those who touch the code in a dynamic exchange of model ideas through the UBIQUITOUS LANGUAGE.

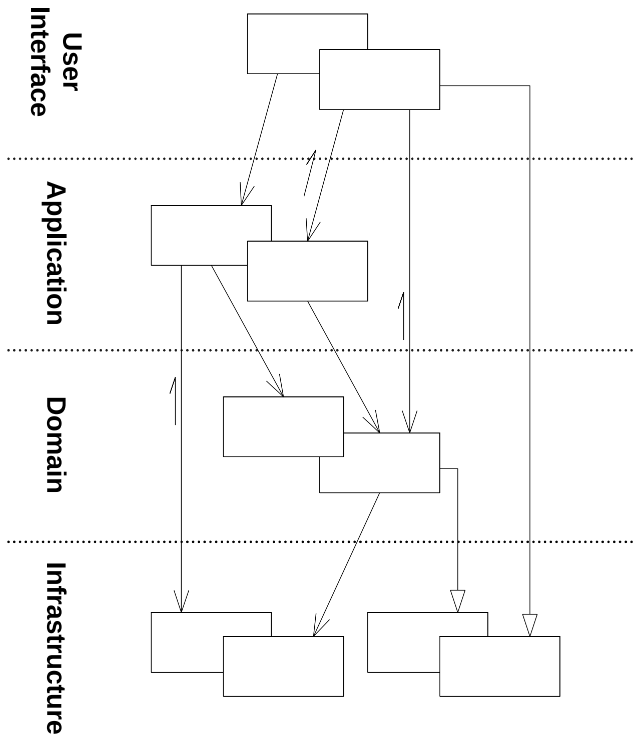
Part II

Building Blocks of a MODEL-DRIVEN DESIGN

These patterns cast widely held best practices of object-oriented design in the light of domain-driven design. They guide decisions to clarify the model and to keep the model and implementation aligned with each other, each reinforcing the other's effectiveness. Careful crafting the details of individual model elements gives developers a steady platform from which to apply the modeling approaches of Parts III and IV.



LAYERED ARCHITECTURE



In an object-oriented program, UI, database, and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and database scripts. This happens because it is the easiest way to make things work, in the short run.
When the domain-related code is diffused through such a large amount of other code, it becomes extremely difficult to see and to reason about. Superficial changes to the UI can actually change business logic. To change a business rule may require meticulous tracing of UI code, database code, or other program elements. Implementing coherent, model-driven objects becomes impractical. Automated testing is awkward. With all the technologies and logic involved in each activity, a program must be kept very simple or it becomes impossible to understand.

Therefore,

Partition a complex program into LAYERS. Develop a design within each LAYER that is cohesive and that depends only on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work.

ENTITIES (AKA REFERENCE OBJECTS)

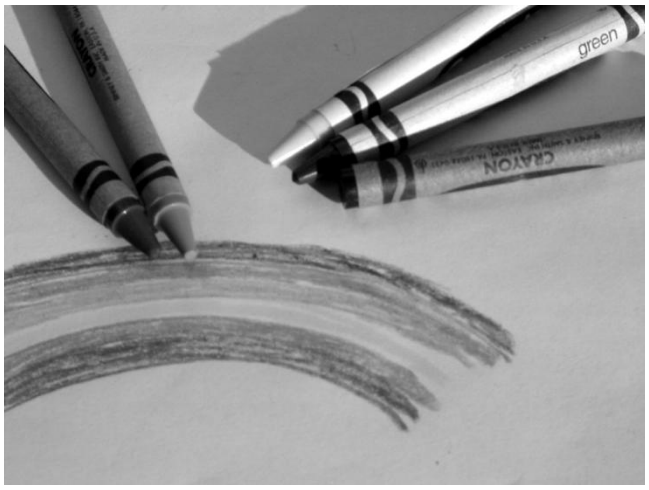
Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. An object must be distinguished from other objects even though they might have the same attributes. Mistaken identity can lead to data corruption.

Therefore,

When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity. Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes. Define an operation that is guaranteed to produce a unique result for each object, possibly by attaching a symbol that is guaranteed unique. This means of identification may come from the outside, or it may be an arbitrary identifier created by and for the system, but it must correspond to the identity distinctions in the model. The model must define what it *means* to be the same thing.

VALUE OBJECTS



Many objects have no conceptual identity. These objects describe some characteristic of a thing.

Tracking the identity of ENTITIES is essential, but attaching identity to other objects can hurt system performance, add analytical work, and muddle the model by making all objects look the same. Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary.
However, if we think of this category of object as just the absence of identity, we haven't added much to our toolbox or vocabulary. In fact, these objects have characteristics of their own, and their own significance to the model. *These are the objects that describe things.*

Therefore:

When you care only about the attributes of an element of the model, classify it as a VALUE OBJECT. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the VALUE OBJECT as immutable. Don't give it any identity and avoid the design complexities necessary to maintain ENTITIES.

SERVICES

Sometimes, it just isn't a thing.

Some concepts from the domain aren't natural to model as objects. Forcing the required domain functionality to be the responsibility of an ENTITY or VALUE either distorts the definition of a model-based object or adds meaningless artificial objects.

Therefore:

When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as a standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.

MODULES (AKA PACKAGES)

Everyone uses MODULES, but few treat them as a full-fledged part of the model. Code gets broken down into all sorts of categories, from aspects of the technical architecture to developers' work assignments. Even developers who refactor a lot tend to content themselves with MODULES conceived early in the project.

It is a truism that there should be low coupling between MODULES and high cohesion within them. Explanations of coupling and cohesion tend to make them sound like technical metrics, to be judged mechanically based on the distributions of associations and interactions. Yet it isn't just code being divided into MODULES, but concepts. There is a limit to how many things a person can think about at once (hence low coupling). Incoherent fragments of ideas are as hard to understand as an undifferentiated soup of ideas (hence high cohesion).

Therefore,

Choose MODULES that tell the story of the system and contain a cohesive set of concepts. This often yields low coupling between modules, but if it doesn't look for a way to change the model to disentangle the concepts, or an overlooked concept that might be the basis of a MODULE that would bring the elements together in a meaningful way. Seek low coupling in the sense of concepts that can be understood and reasoned about independently of each other. Refine the model until it partitions according to high-level domain concepts and the corresponding code is decoupled as well.

Give the MODULES names that become part of the UBIQUITOUS LANGUAGE. MODULES and their names should reflect insight into the domain.

AGGREGATES

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. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable.

Therefore:

Cluster the ENTITIES and VALUE OBJECTS 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. Allow external objects to hold references to the root only. Transient references to internal members can be passed out for use within a single operation only. Because the root controls access, it cannot be blindsided by changes to the internals. This arrangement makes it practical to enforce all invariants for objects in the AGGREGATE and for the AGGREGATE as a whole in any state change.

(Note: Many such schemes are possible. This section of the book describes one particular set of rules in detail.)

FACTORIES

When creation of an object, or an entire AGGREGATE, becomes complicated or reveals too much of the internal structure, FACTORIES provide encapsulation.

Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. Combining such responsibilities can produce ungainly designs that are hard to understand. Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or AGGREGATE, and overly couples the client to the implementation of the created object.

Therefore,

Shift the responsibility for creating instances of complex objects and AGGREGATES 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. Create entire AGGREGATES as a piece, enforcing their invariants.

REPOSITORIES

A client needs a practical means of acquiring references to preexisting domain objects. If the infrastructure makes it easy to do so, the developers of the client may add more traversable associations, muddling the model. On the other hand, they may use queries to pull the exact data they need from the database, or to pull a few specific objects rather than navigating from AGGREGATE roots. Domain logic moves into queries and client code, and the ENTITIES and VALUE OBJECTS become mere data containers. The sheer technical complexity of applying most database access infrastructure quickly swamps the client code, which leads developers to dumb-down the domain layer, which makes the model irrelevant.

Restating the problem:

A subset of persistent objects must be globally accessible through a search based on object attributes. Such access is needed for the roots of AGGREGATES that are not convenient to reach by traversal. They are usually ENTITIES, sometimes VALUE OBJECTS with complex internal structure, and sometimes enumerated VALUES. Providing access to other objects muddies important distinctions. Free database queries can actually breach the encapsulation of domain objects and AGGREGATES. Exposure of technical infrastructure and database access mechanisms complicates the client and obscures the MODEL-DRIVEN DESIGN.

Therefore:

For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type. Set up access through a well-known global interface. Provide methods to add and remove objects, which will encapsulate the actual insertion or removal of data in the data store. Provide methods that select objects based on some criteria and return fully instantiated objects or collections of objects whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology. Provide repositories only for AGGREGATE roots that actually need direct access. Keep the client focused on the model, delegating all object storage and access to the REPOSITORIES.

Part III

Refactoring Toward Deeper Insight

Using a proven set of basic building blocks along with consistent language brings some sanity to the development effort. This leaves the challenge of actually *finding* an incisive model, one that captures subtle concerns of the domain experts and can drive a practical design. A model that sloughs off the superficial and captures the essential is a *deep model*. This should make the software more in tune with the way the domain experts think and more responsive to the user's needs.

Traditionally, refactoring is described in terms of code transformations with technical motivations. Refactoring can also be motivated by an insight into the domain and a corresponding refinement of the model or its expression in code.

Sophisticated domain models are seldom developed except through an iterative process of refactoring, including close involvement of the domain experts with developers interested in learning about the domain.

Supple Design

To have a project accelerate as development proceeds—rather than get weighed down by its own legacy—demands a design that is a pleasure to work with, inviting to change. A supple design.

Supple design is the complement to deep modeling.

Developers play two roles, each of which must be served by the design. The same person might well play both roles—even switch back and forth in minutes—but the relationship to the code is different nonetheless. One role is the developer of a client, who weaves the domain objects into the application code or other domain layer code, utilizing capabilities of the design. A supple design reveals a deep underlying model that makes its potential clear. The client developer can flexibly use a minimal set of loosely coupled concepts to express a range of scenarios in the domain. Design elements fit together in a natural way with a result that is predictable, clearly characterized, and robust.

Equally important, the design must serve the developer working to change it. To be open to change, a design must be easy to understand, revealing that *same* underlying model that the client developer is drawing on. It must follow the contours of a deep model of the domain, so most changes bend the design at flexible points. The effects of its code must be transparently obvious, so the consequences of a change will be easy to anticipate.

Making Behavior Obvious

INTENTION-REVEALING INTERFACES

If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost. If someone other than the original developer must infer the purpose of an object or operation based on its implementation, that new developer may infer a purpose that the operation or class fulfills only by chance. If that was not the intent, the code may work for the moment, but the conceptual basis of the design will have been corrupted, and the two developers will be working at cross-purposes.

Therefore,

Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise. This relieves the client developer of the need to understand the internals. These names should conform to the UBIQUITOUS LANGUAGE so that team members can quickly infer their meaning. Write a test for a behavior before creating it, to force your thinking into client developer mode.

SIDE-EFFECT-FREE FUNCTIONS

Interactions of multiple rules or compositions of calculations become extremely difficult to predict. The developer calling an operation must understand its implementation and the implementation of all its delegations in order to anticipate the result. The usefulness of any abstraction of interfaces is limited if the developers are forced to pierce the veil. Without safely predictable abstractions, the developers must limit the combinatory explosion, placing a low ceiling on the richness of behavior that is feasible to build.

Therefore:

Place as much of the logic of the program as possible into functions, operations that return results with no observable side effects. Strictly segregate commands (methods which result in modifications to observable state) into very simple operations that do not return domain information. Further control side effects by moving complex logic into VALUE OBJECTS when a concept fitting the responsibility presents itself.

Assertions

When the side effects of operations are only defined implicitly by their implementation, designs with a lot of delegation become a tangle of cause and effect. The only way to understand a program is to trace execution through branching paths. The value of encapsulation is lost. The necessity of tracing concrete execution defeats abstraction.

Therefore,

State post-conditions of operations and invariants of classes and AGGREGATES. If ASSERTIONS cannot be coded directly in your programming language, write automated unit tests for them. Write them into documentation or diagrams where it fits the style of the project's development process.
Seek models with coherent sets of concepts, which lead a developer to infer the intended ASSERTIONS, accelerating the learning curve and reducing the risk of contradictory code.

Reducing Cost of Change

CONCEPTUAL CONTOURS

Sometimes people chop functionality fine to allow flexible combination. Sometimes they lump it large to encapsulate complexity. Sometimes they seek a consistent granularity, making all classes and operations to a similar scale. These are oversimplifications that don't work well as general rules. But they are motivated by a basic set of problems.

When elements of a model or design are embedded in a monolithic construct, their functionality gets duplicated. The external interface doesn't say everything a client might care about. Their meaning is hard to understand, because different concepts are mixed together.
On the other hand, breaking down classes and methods can pointlessly complicate the client, forcing client objects to understand how tiny pieces fit together. Worse, a concept can be lost completely. Half of a uranium atom is not uranium. And of course, it isn't just grain size that counts, but just where the grain runs.

Therefore:

Decompose design elements (operations, interfaces, classes, and AGGREGATES) into cohesive units, taking into consideration your intuition of the important divisions in the domain. Observe the axes of change and stability through successive refactorings and look for the underlying CONCEPTUAL CONTOURS that explain these shearing patterns. Align the model with the consistent aspects of the domain that make it a viable area of knowledge in the first place.

A supple design based on a deep model yields a simple set of interfaces that combine logically to make sensible statements in the UBIQUITOUS LANGUAGE, and without the distraction and maintenance burden of irrelevant options.

STANDALONE CLASSES

Even within a MODULE, the difficulty of interpreting a design increases wildly as dependencies are added. This adds to mental overload, limiting the design complexity a developer can handle. Implicit concepts contribute to this load even more than explicit references.

Low coupling is fundamental to object design. When you can, go all the way. Eliminate *all* other concepts from the picture. Then the class will be completely self-contained and can be studied and understood alone. Every such self-contained class significantly eases the burden of understanding a MODULE.

CLOSURE OF OPERATIONS

Most interesting objects end up doing things that can't be characterized by primitives alone.

Therefore:

Where it fits, define an operation whose return type is the same as the type of its argument(s). If the implementer has state that is used in the computation, then the implementer is effectively an argument of the operation, so the argument(s) and return value should be of the same type as the implementer. Such an operation is closed under the set of instances of that type. A closed operation provides a high-level interface without introducing any dependency on other concepts.

*This pattern is most often applied to the operations of a VALUE OBJECT. Because the life cycle of an ENTITY has significance in the domain, you can't just conjure up a new one to answer a question. There are operations that are closed under an ENTITY type. You could ask an **Employee** object for its supervisor and get back another **Employee**. But in general, ENTITIES are not the sort of concepts that are likely to be the result of a computation. So, for the most part, this is an opportunity to look for in the VALUE OBJECTS.*

You sometimes get halfway to this pattern. The argument matches the implementer, but the return type is different, or the return type matches the receiver and the argument is different. These operations are not closed, but they do give some of the advantages of CLOSURE. When the extra type is a primitive or basic library class, it frees the mind almost as much as CLOSURE.

Declarative Design

There can be no real guarantees in handwritten software. To name just one way of evading ASSERTIONS, code could have additional side effects that were not specifically excluded. No matter how MODEL-DRIVEN our design is, we still end up writing procedures to produce the effect of the conceptual interactions. And we spend so much of our time writing boilerplate code that doesn't really add any meaning *or* behavior. INTENTION-REVEALING INTERFACES and the other patterns in this chapter help, but they can never give conventional object-oriented programs formal rigor.

These are some of the motivations behind *declarative design*. This term means many things to many people, but usually it indicates a way to write a program, or some part of a program, as a kind of executable specification. A very precise description of properties actually controls the software. In its various forms, this could be done through a reflection mechanism or at compile time through code generation (producing conventional code automatically, based on the declaration). This approach allows another developer to take the declaration at face value. It is an absolute guarantee.

Many declarative approaches can be corrupted if the developers bypass them intentionally or unintentionally. This is likely when the system is difficult to use or overly restrictive. Everyone has to follow the rules of the framework in order to get the benefits of a declarative program.

.

A Declarative Style of Design

Once your design has INTENTION-REVEALING INTERFACES, SIDE-EFFECT-FREE FUNCTIONS, and ASSERTIONS, you are edging into declarative territory. Many of the benefits of declarative design are obtained once you have combinable elements that communicate their meaning, and have characterized or obvious effects, or no observable effects at all.

A supple design can make it possible for the client code to use a declarative *style* of design. To illustrate, the next section will bring together some of the patterns in this chapter to make the SPECIFICATION more supple and declarative.

Drawing on Established Formalisms

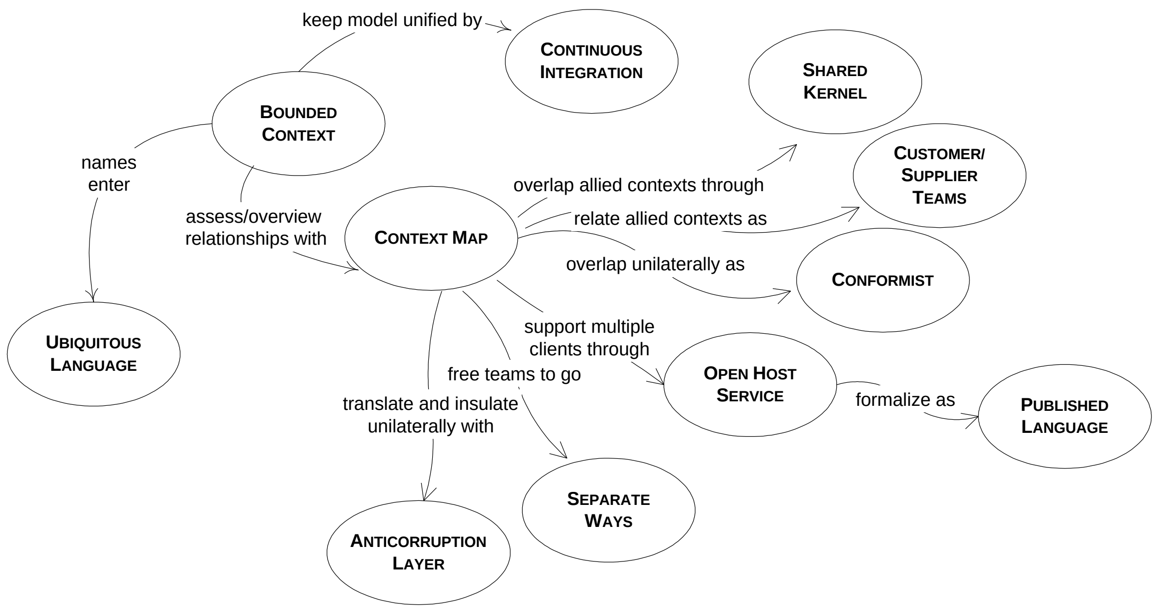
Creating a tight conceptual framework from scratch is something you can't do every day. Sometimes you discover and refine one of these over the course of the life of a project. But you can often use and adapt conceptual systems that are long established in your domain or others, some of which have been refined and distilled over centuries. Many business applications involve accounting, for example. Accounting defines a well-developed set of ENTITIES and rules that make for an easy adaptation to a deep model and a supple design.

There are many such formalized conceptual frameworks, but my personal favorite is math. It is surprising how useful it can be to pull out some twist on basic arithmetic. Many domains include math somewhere. Look for it. Dig it out. Specialized math is clean, combinable by clear rules, and people find it easy to understand. One example from my past is "Shares Math," which will end this chapter.

Part IV

Strategic Design

Maintaining Model Integrity



BOUNDED CONTEXT

Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should *not* be applied.

Therefore:

Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

CONTINUOUS INTEGRATION

Once a BOUNDED CONTEXT has been defined, we must keep it sound.

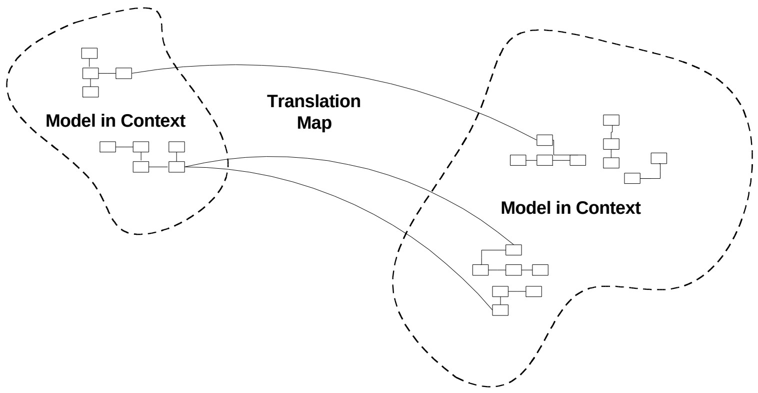
♦ ♦ ♦

When a number of people are working in the same BOUNDED CONTEXT, there is a strong tendency for the model to fragment. The bigger the team, the bigger the problem, but as few as three or four people can encounter serious problems. Yet breaking down the system into ever-smaller contexts eventually loses a valuable level of integration and coherency.

Therefore:

Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly. Relentlessly exercise the UBIQUITOUS LANGUAGE to hammer out a shared view of the model as the concepts evolve in different people's heads.

CONTEXT MAP



An individual BOUNDED CONTEXT *leaves some problems in the absence of a global view. The context of other models may still be vague and in flux.*

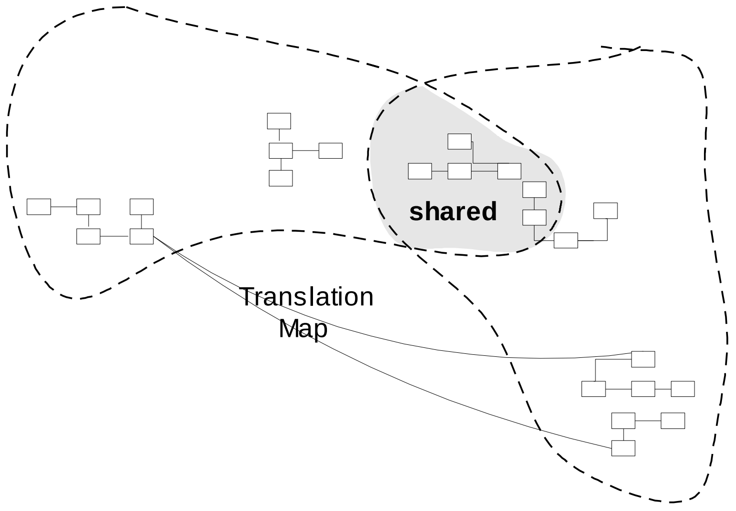
♦ ♦ ♦

People on other teams won't be very aware of the context bounds and will unknowingly make changes that blur the edges or complicate the interconnections. When connections must be made between different contexts, they tend to bleed into each other.

Therefore:

Identify each model in play on the project and define its BOUNDED CONTEXT. This includes the implicit models of non-object-oriented subsystems. Name each BOUNDED CONTEXT, and make the names part of the UBIQUITOUS LANGUAGE.
Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing.
Map the *existing* terrain. Take up transformations later.

SHARED KERNEL



When functional integration is limited, the overhead of CONTINUOUS INTEGRATION *may be deemed too high. This may especially be true when the teams do not have the skill or the political organization to maintain continuous integration, or when a single team is simply too big and unwieldy. So separate* BOUNDED CONTEXTS *might be defined and multiple teams formed.*

♦ ♦ ♦

Uncoordinated teams working on closely related applications can go racing forward for a while, but what they produce may not fit together. They can end up spending more on translation layers and retrofitting than they would have on CONTINUOUS INTEGRATION in the first place, meanwhile duplicating effort and losing the benefits of a common UBIQUITOUS LANGUAGE.

Therefore:

Designate some subset of the domain model that the two teams agree to share. Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team.
Integrate a functional system frequently, but somewhat less often than the pace of CONTINUOUS INTEGRATION within the teams. At these integrations, run the tests of both teams.

CUSTOMER/SUPPLIER DEVELOPMENT TEAMS

Functionality is often partitioned such that one subsystem essentially feeds another; the second performs analysis or other functions that feed back very little into the first. In such cases, the two *subsystems commonly serve very different user communities, with different jobs, where different models may be useful. The tool set may also be different, meaning that program code cannot be shared.*

♦ ♦ ♦

The freewheeling development of the upstream team can be cramped if the downstream team has veto power over changes, or if procedures for requesting changes are too cumbersome. The upstream team may even be inhibited, worried about breaking the downstream system. Meanwhile, the downstream team can be helpless, at the mercy of upstream priorities.

Therefore:

Establish a clear customer/supplier relationship between the two teams. In planning sessions, make the downstream team play the customer role to the upstream team. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule. Jointly develop automated acceptance tests that will validate the interface expected. Add these tests to the upstream team's test suite, to be run as part of its continuous integration. This testing will free the upstream team to make changes without fear of side effects downstream.

CONFORMIST

When two development teams have an upstream/downstream relationship in which the upstream has no motivation to provide for the downstream team's needs, the downstream team is helpless. Altruism may motivate upstream developers to make promises, but they are unlikely to be fulfilled. Belief in those good intentions leads the downstream team to make plans based on features that will never be available. The downstream project will be delayed until the team ultimately learns to live with what it is given. An interface tailored to the needs of the downstream team is not in the cards.

Therefore:

Eliminate the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team. Although this cramps the style of the downstream designers and probably does not yield the ideal model for the application, choosing CONFORMITY enormously simplifies integration. Also, you will share a UBIQUITOUS LANGUAGE with your supplier team. The supplier is in the driver's seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you.

ANTICORRUPTION LAYER

New systems almost always have to be integrated with legacy or other systems, which have their own models. When control or communication is not adequate to pull off a SHARED KERNEL or CUSTOMER/SUPPLIER DEVELOPMENT TEAMS, the interface can become more complex. Translation layers can be simple, even elegant, when bridging well-designed BOUNDED CONTEXTS with cooperative teams. But when the other side of the boundary starts to leak through, the translation layer may take on a more defensive tone.

♦ ♦ ♦

When a new system is being built that must have a large interface with another, the difficulty of relating the two models can eventually overwhelm the intent of the new model altogether, causing it to be modified to resemble the other system's model, in an ad hoc fashion. The models of legacy systems are usually weak, and even the exception that is well developed may not fit the needs of the current project. Yet there may be a lot of value in the integration, and sometimes it is an absolute requirement.

Therefore:

Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions as necessary between the two models.

SEPARATE WAYS

We must be ruthless when it comes to defining requirements. If two sets of functionality have no significant relationship, they can be completely cut loose from each other.

♦ ♦ ♦

Integration is always expensive, and sometimes the benefit is small.

Therefore:

Declare a BOUNDED CONTEXT to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope.

OPEN HOST SERVICE

Typically for each BOUNDED CONTEXT*, you will define a translation layer for each component with which you have to integrate that is outside the* CONTEXT*. Where integration is one-off, this approach of inserting a translation layer for each external system avoids corruption of the models with a minimum of cost. But when you find your subsystem in high demand, you may need a more flexible approach.*

♦ ♦ ♦

When a subsystem has to be integrated with many others, customizing a translator for each can bog down the team. There is more and more to maintain, and more and more to worry about when changes are made.

Therefore:

Define a protocol that gives access to your subsystem as a set of SERVICES. Open the protocol so that all who need to integrate with you can use it. Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent.

PUBLISHED LANGUAGE

The translation between the models of two BOUNDED CONTEXTS *requires a common language.*

♦ ♦ ♦

Direct translation to and from the existing domain models may not be a good solution. Those models may be overly complex or poorly factored. They are probably undocumented. If one is used as a data interchange language, it essentially becomes frozen and cannot respond to new development needs.

Therefore:

Use a well-documented shared language that can express the necessary domain information as a common medium of communication, translating as necessary into and out of that language.

Distillation

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

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

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

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

—*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 drowning in a sea of side issues? *Distillation* is the process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful. A model is a distillation of knowledge. With every refactoring to deeper insight, we abstract some crucial aspect of domain knowledge and priorities. Now, stepping back for a strategic view, this chapter looks at ways to distinguish broad swaths of the model and distill the domain model as a whole.

CORE DOMAIN

In designing a large system, there are so many contributing components, all complicated and all absolutely necessary to success, that the essence of the domain model, the real business asset, can be obscured and neglected.
The harsh reality is that not all parts of the design are going to be equally refined. Priorities must be set. To make the domain model an asset, the critical core of that model has to be sleek and fully leveraged to create application functionality. But scarce, highly skilled developers tend to gravitate to technical infrastructure or neatly definable domain problems that can be understood without specialized domain knowledge.

Therefore:

Boil the model down. Find the CORE DOMAIN and provide a means of easily distinguishing it from the mass of supporting model and code. Bring the most valuable and specialized concepts into sharp relief. Make the CORE small.
Apply top talent to the CORE DOMAIN, and recruit accordingly. Spend the effort in the CORE to find a deep model and develop a supple design—sufficient to fulfill the vision of the system. Justify investment in any other part by how it supports the distilled CORE.

GENERIC SUBDOMAINS

Some parts of the model add complexity without capturing or communicating specialized knowledge. Anything extraneous makes the CORE DOMAIN harder to discern and understand. The model clogs up with general principles everyone knows or details that belong to specialties which are not your primary focus but play a supporting role. Yet, however generic, these other elements are essential to the functioning of the system and the full expression of the model.

Therefore:

Identify cohesive subdomains that are not the motivation for your project. Factor out generic models of these subdomains and place them in separate MODULES. Leave no trace of your specialties in them.
Once they have been separated, give their continuing development lower priority than the CORE DOMAIN, and avoid assigning your core developers to the tasks (because they will gain little domain knowledge from them). Also consider off-the-shelf solutions or published models for these GENERIC SUBDOMAINS.

DOMAIN VISION STATEMENT

At the beginning of a project, the model usually doesn’t even exist, yet the need to focus its development is already there. In later stages of development, there is a need for an explanation of the value of the system that does not require an in-depth study of the model. Also, the critical aspects of the domain model may span multiple BOUNDED CONTEXTS, but by definition these distinct models can’t be structured to show their common focus.

Therefore:

Write a short description (about one page) of the CORE DOMAIN and the value it will bring, the “value proposition.” 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.

HIGHLIGHTED CORE

A DOMAIN VISION STATEMENT identifies the CORE DOMAIN in broad terms, but it leaves the identification of the specific CORE model elements up to the vagaries of individual interpretation. Unless there is an exceptionally high level of communication on the team, the VISION STATEMENT alone will have little impact.

♦ ♦ ♦

Even though team members may know broadly what constitutes the CORE DOMAIN, different people won't pick out quite the same elements, and even the same person won't be consistent from one day to the next. The mental labor of constantly filtering the model to identify the key parts absorbs concentration better spent on design thinking, and it requires comprehensive knowledge of the model. The CORE DOMAIN must be made easier to see.
Significant structural changes to the code are the ideal way of identifying the CORE DOMAIN, but they are not always practical in the short term. In fact, such major code changes are difficult to undertake without the very view the team is lacking.

Therefore (as one form of HIGHLIGHTED CORE):

Write a very brief document (three to seven sparse pages) that describes the CORE DOMAIN and the primary interactions among CORE elements.

Therefore (as another form of HIGHLIGHTED CORE):

Flag the elements of the CORE DOMAIN within the primary repository of the model, without particularly trying to elucidate its role. Make it effortless for a developer to know what is in or out of the CORE.

As a corollary and process opportunity:

If the distillation document outlines the essentials of the CORE DOMAIN, then it serves as a practical indicator of the significance of a model change. When a model or code change affects the distillation document, it requires consultation with other team members. When the change is made, it requires immediate notification of all team members, and the dissemination of a new version of the document. Changes outside the CORE or to details not included in the distillation document can be integrated without consultation or notification and will be encountered by other members in the course of their work. Then the developers have the full autonomy that XP suggests.

♦ ♦ ♦

Although the VISION STATEMENT and HIGHLIGHTED CORE inform and guide, they do not actually modify the model or the code itself. Partitioning GENERIC SUBDOMAINS physically removes some distracting elements. Next we'll look at other ways to structurally change the model and the design itself to make the CORE DOMAIN more visible and manageable. . . .

COHESIVE MECHANISMS

Computations sometimes reach a level of complexity that begins to bloat the design. The conceptual "what" is swamped by the mechanistic "how." A large number of methods that provide algorithms for resolving the problem obscure the methods that express the problem.

Therefore:

Partition a conceptually COHESIVE MECHANISM into a separate lightweight framework. Particularly watch for formalisms or well-documented categories of algorithms. Expose the capabilities of the framework with an INTENTION-REVEALING INTERFACE. Now the other elements of the domain can focus on expressing the problem ("what"), delegating the intricacies of the solution ("how") to the framework.

♦ ♦ ♦

Factoring out GENERIC SUBDOMAINS reduces clutter, and COHESIVE MECHANISMS serve to encapsulate complex operations. This leaves behind a more focused model, with fewer distractions that add no particular value to the way users conduct their activities. But you are unlikely ever to find good homes for everything *in the domain model that is not CORE. The SEGREGATED CORE takes a direct approach to structurally marking off the CORE DOMAIN. . . .*

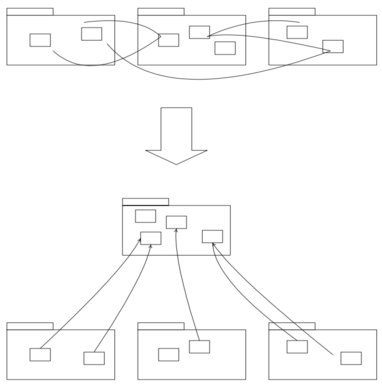
SEGREGATED CORE

Elements in the model may partially serve the CORE DOMAIN 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. All this clutter and entanglement chokes the CORE. Designers can't clearly see the most important relationships, leading to a weak design.

Therefore:

Refactor the model to separate the CORE concepts from supporting players (including ill-defined ones) and strengthen the cohesion of the CORE while reducing its coupling to other code. Factor all generic or supporting elements into other objects and place them into other packages, even if this means refactoring the model in ways that separate highly coupled elements.

ABSTRACT CORE



Even the CORE DOMAIN model usually has so much detail that communicating the big picture can be difficult.
When there is a lot of interaction between subdomains in separate MODULES, either many references will have to be created between MODULES, which defeats much of the value of the partitioning, or the interaction will have to be made indirect, which makes the model obscure.

Therefore:

Identify the most fundamental concepts in the model and factor them into distinct classes, abstract classes, or interfaces. Design this abstract model so that it expresses most of the interaction between significant components. Place this abstract overall model in its own MODULE, while the specialized, detailed implementation classes are left in their own MODULES defined by subdomain.

Large-Scale Structure

Thousands of people worked independently to create the AIDS Quilt.

In a large system without any overarching principle that allows elements to be interpreted in terms of their role in patterns that span the whole design, *developers cannot 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 the details of the whole.
A "large-scale structure" is a language that lets you discuss and understand the system in broad strokes. A set of high-level concepts or rules, or both, establishes a pattern of design for an entire system. This organizing principle can guide design as well as aid understanding. It helps coordinate independent work because there is a shared concept of the big picture: how the roles of various parts shape the whole.

Devise a pattern of rules or roles and relationships that will span the entire system and that allows some understanding of each part's place in the whole—even without detailed knowledge of the part's responsibility.

Evolving Order

Design free-for-alls produce systems no one can make sense of as a whole, and they are very difficult to maintain. But architectures can straitjacket a project with up-front design assumptions and take too much power away from the developers/designers of particular parts of the application. Soon, developers will dumb down the application to fit the structure, or they will subvert it and have no structure at all, bringing back the problems of uncoordinated development.

Therefore:

Let this conceptual large-scale structure evolve with the application, possibly changing to a completely different type of structure along the way. Don't over constrain the detailed design and model decisions that must be made with detailed knowledge.

Large-scale structure should be applied when a structure can be found that greatly clarifies the system without forcing unnatural constraints on model development. Because an ill-fitting structure is worse than none, it is best not to shoot for comprehensiveness, but rather to find a minimal set that solves the problems that have emerged. Less is more.

What follows is a set of four particular patterns of large-scale structure that emerge on some projects and are representative of this kind of pattern.

System Metaphor

Metaphorical thinking is pervasive in software development, especially with models. But the Extreme Programming practice of "metaphor" has come to mean a particular way of using a metaphor to bring order to the development of a whole system.
Software designs tend to be very abstract and hard to grasp. Developers and users alike need tangible ways to understand the system and share a view of the system as a whole.

Therefore:

When a concrete analogy to the system emerges that captures the imagination of team members and seems to lead thinking in a useful direction, adopt it as a large-scale structure. Organize the design around this metaphor and absorb it into the UBIQUITOUS LANGUAGE. The SYSTEM METAPHOR should both facilitate communication about the system and guide development of it. This increases consistency in different parts of the system, potentially even across different BOUNDED CONTEXTS. But because all metaphors are inexact, continually reexamine the metaphor for overextension or inaptness, and be ready to drop it if it gets in the way.

Responsibility Layers

Throughout this book, individual objects have been assigned narrow sets of related responsibilities. Responsibility-driven design also applies to larger scales.

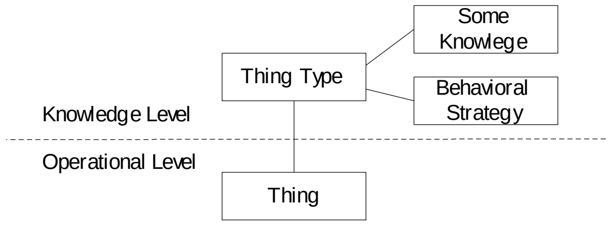
♦ ♦ ♦

When each individual object has handcrafted 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 impose some structure on the assignment of those responsibilities.

Therefore:

Look at the conceptual dependencies in your model and the varying rates and sources of change of different parts of your domain. If you identify natural strata in the domain, cast them as broad abstract responsibilities. These responsibilities should tell a story of the high-level purpose and design of your system. Refactor the model so that the responsibilities of each domain object, AGGREGATE, and MODULE fit neatly within the responsibility of one layer.

KNOWLEDGE LEVEL



A group of objects that describe how another group of objects should behave (from Martin Fowler's Web site, www.martinfowler.com)

♦ ♦ ♦

In an application in which the roles and relationships between ENTITIES vary in different situations, complexity can explode. Neither fully general models nor highly customized ones serve the users' needs. Objects end up with references to other types to cover a variety of cases, or with attributes that are used in different ways in different situations. Classes that have the same data and behavior may multiply just to accommodate different assembly rules.

Therefore:

Create a distinct set of objects that can be used to describe and constrain the structure and behavior of the basic model. Keep these concerns separate as two "levels," one very concrete, the other reflecting rules and knowledge that a user or super-user is able to customize.

PLUGGABLE COMPONENT FRAMEWORK

Opportunities arise in a very mature model that is deep and distilled. A PLUGGABLE COMPONENT FRAMEWORK usually only comes into play after a few applications have already been implemented in the same domain.

♦ ♦ ♦

When a variety of applications have to interoperate, all based on the same abstractions but designed independently, translations between multiple BOUNDED CONTEXTS limit integration. A SHARED KERNEL is not feasible for teams that do not work closely together. Duplication and fragmentation raise costs of development and installation, and interoperability becomes very difficult.

Therefore:

Distill an ABSTRACT CORE of interfaces and interactions and create a framework that allows diverse implementations of those interfaces to be freely substituted. Likewise, allow any application to use those components, so long as it operates strictly through the interfaces of the ABSTRACT CORE.