

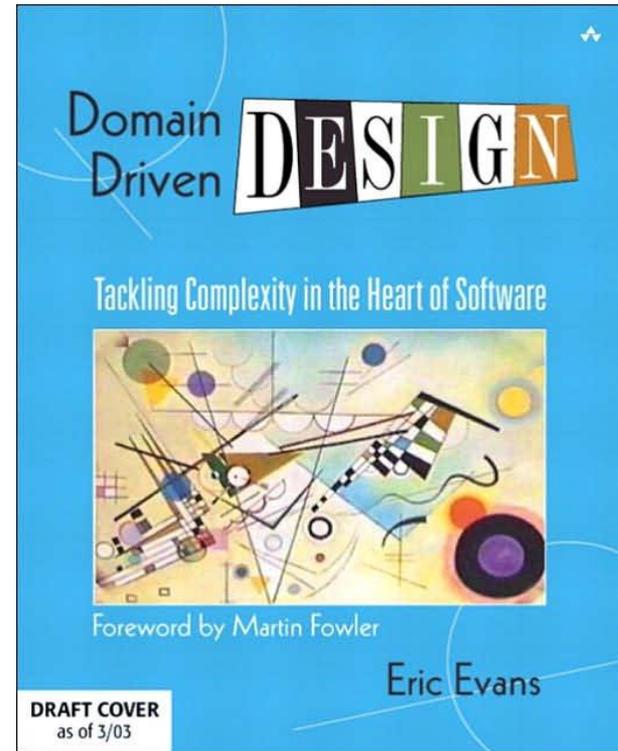
Domain-Driven Design

Brett D. Roads

Domain-Driven Design: Tackling Complexity in the Heart of Software

– By: Eric Evans

- This text address the analysis and design of software the relies on complex domain specific knowledge



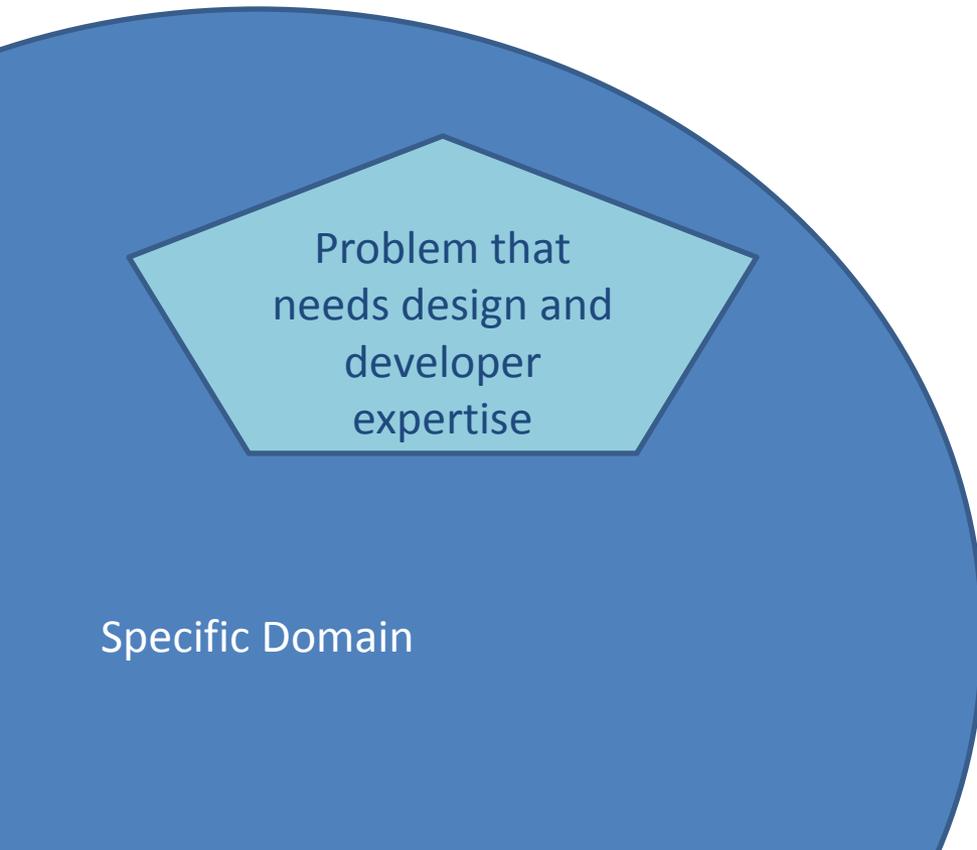
Motivation

- The Goal
 - A domain specific project that needs to leverage multiple realms of expertise
 - Design and Developer expertise
 - Domain specific expertise
- The Complication
 - Need to enable communication between the two groups.
 - Project organization can insulate the transmission of knowledge and retard the ideal evolution of a project
- The Solution
 - Strengthen the communication process and establish a methodology for making those communication more robust
 - This is primarily accomplished by developing a UBIQOUTOUS LANGUAGE and single model.

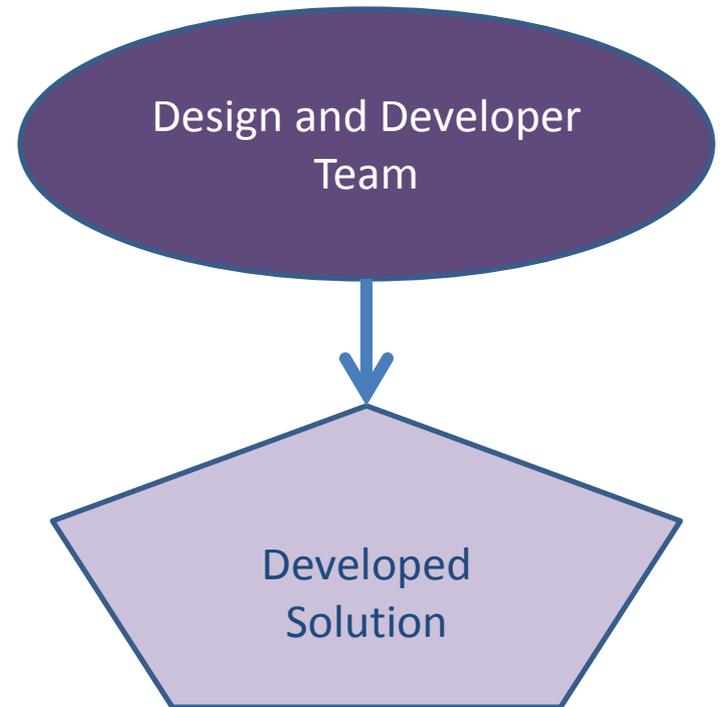
The Players

- Throughout the text, there are four main roles in the development process
 - Domain expert
 - Designer
 - Software developer
 - End user
- This approach seeks to leverage the skills of the designer, developer and the domain expert in order to create a scalable solution for a domain specific problem

The Goal



Specific Domain



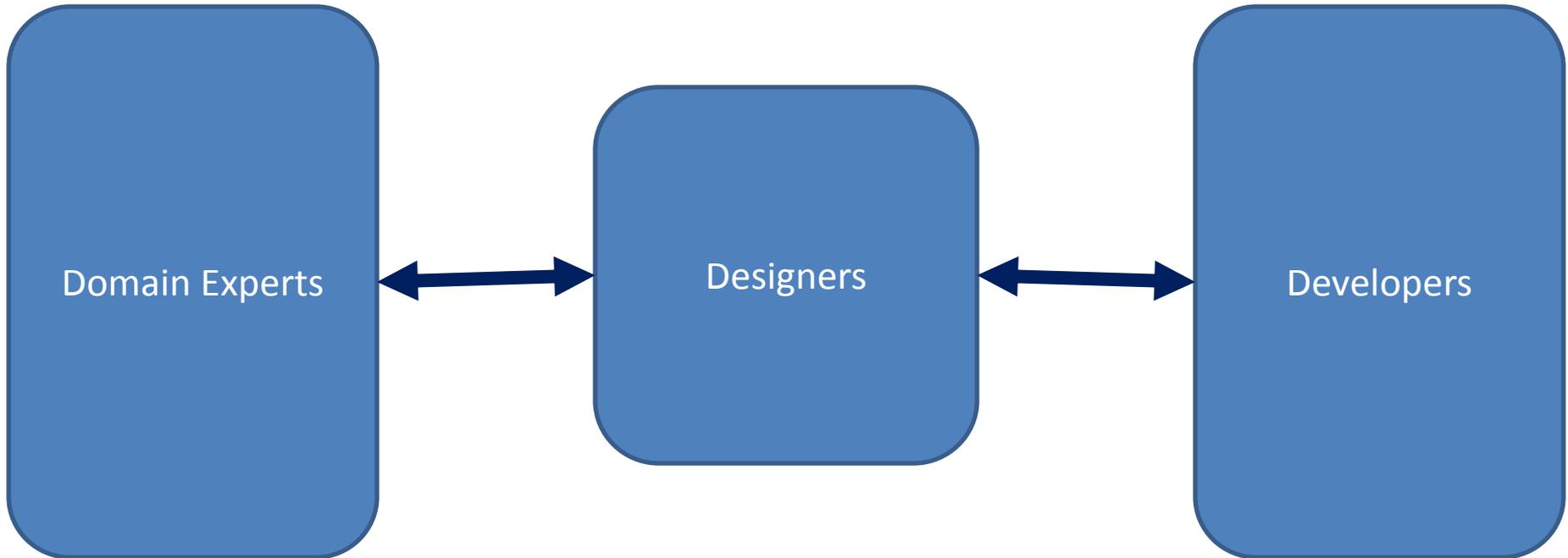
Design and Developer Team

Developed Solution

The Complication

- Initially, domain experts and developers likely do not share the same language for discussing the project.
- Concerns with scalability and quality mean that the solution must be especially careful to accurately reflect the domain.
- Therefore, domain experts and developers must be able to communicate with each other effectively.

Problematic Communication Structure

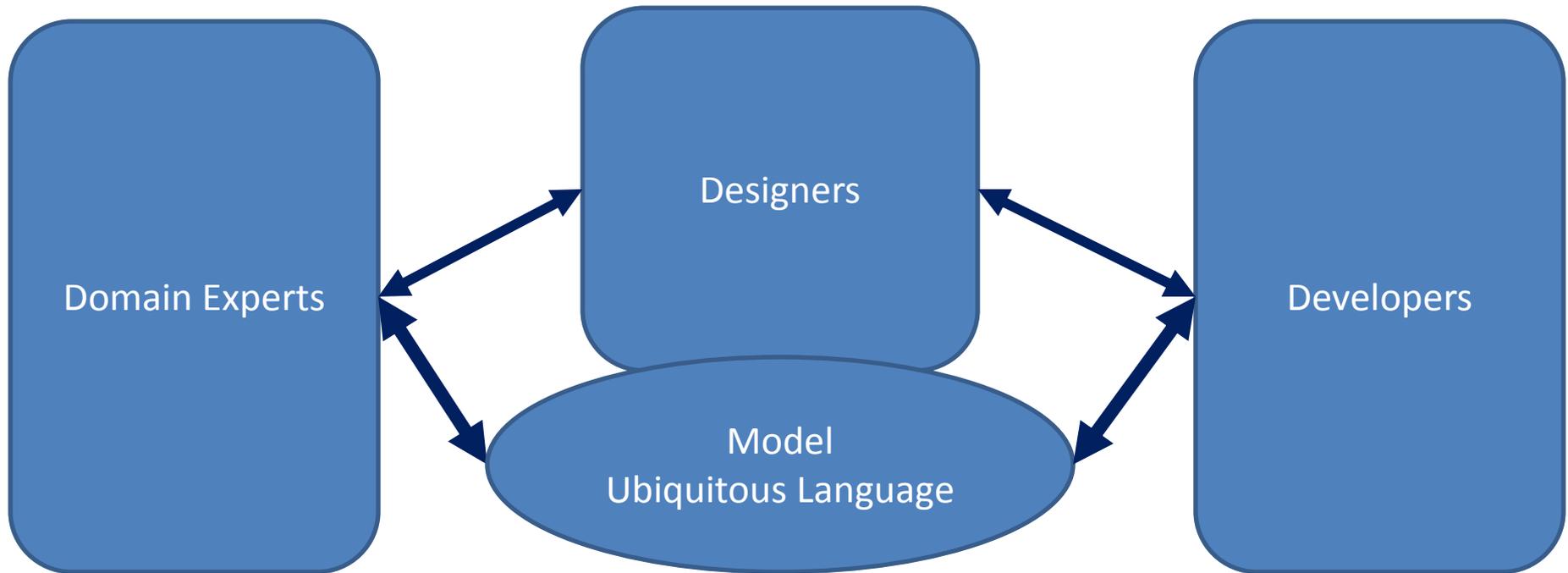


- Developers are insulated from the domain experts. If a developer does not understand a concept, it is likely the implementation will not accurately reflect the domain.

The Solution

- Facilitate communication between domain experts, designers and developers
- This is accomplished by ...
 - establishing a common language, i.e. a UBIQUITOUS LANGUAGE.
 - iterating a single model to reflect a shared understanding across domain experts, designers and developers.

Domain-Driven Design Communication Structure

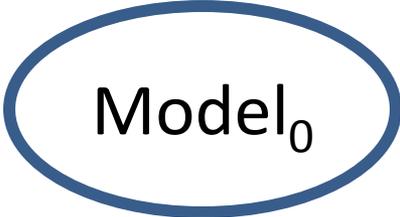


- Communication between developers and domain experts is facilitated by the development of a UBIQUITOUS LANGUAGE and a single model.

Models

- This text expresses a fundamental view of models that is perhaps at odds with other ways of thinking
 - Models live in people's heads
 - Diagrams, code, speech, etc. utilizes a model
 - Models are not a design artifact
- Models are the backbone of a project
- Consequently, Domain-Driven Design highly overlaps with Model-Driven Design

Knowledge Crunching

- Continuous learning that takes place between domain experts, designers and developers.
- “Knowledge crunching is an exploration, and you can’t know where you’ll end up (pg. 21)”
- Gives a starting model.
A blue oval containing the text "Model₀".
- Provides a mechanism for initiating model iterations.

UBIQUITOUS LANGUAGE

- The language that is used across aspects of the project.
- The model implies UBIQUITOUS LANGUAGE
- “The use of language on a project is subtle but all-important. (pg. 23)”
- “... the primary carrier of the aspects of design that don't appear in code...(pg. 27)”

UBIQUITOUS LANGUAGE

- “The vocabulary of that UBIQUITUOS LANGUAGE includes names of classes and prominent operations. The language includes terms to discuss rules that have been made explicit in the model. It is supplemented with terms from high-level organizing principles imposed on the model. Finally, this language is enriched with the names of patterns the team commonly applies to the domain model (pg. 25).”

UBIQUITOUS LANGUAGE

- “Persistent use of the UBIQUITOUS LANGUAGE will force the model’s weaknesses into the open (pg. 26)”

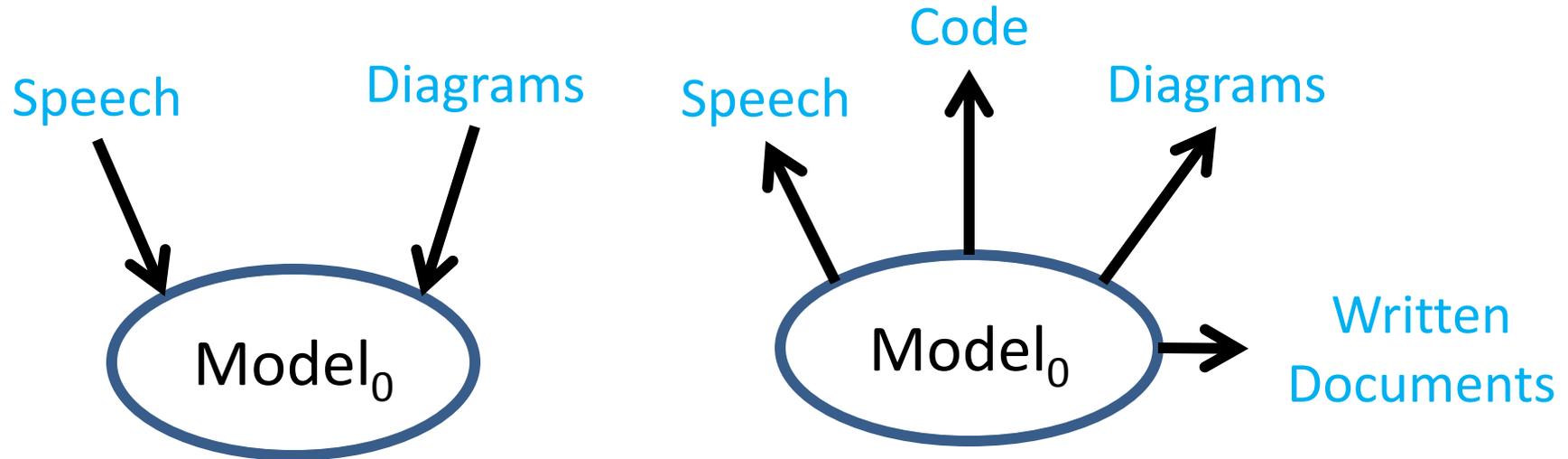
The One Model Solution

- Why?
 - The model is the source
 - Control the source, you control the consequences
- Complication
 - To work correctly...
 - “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 (pg. 62).”

Iterative Process

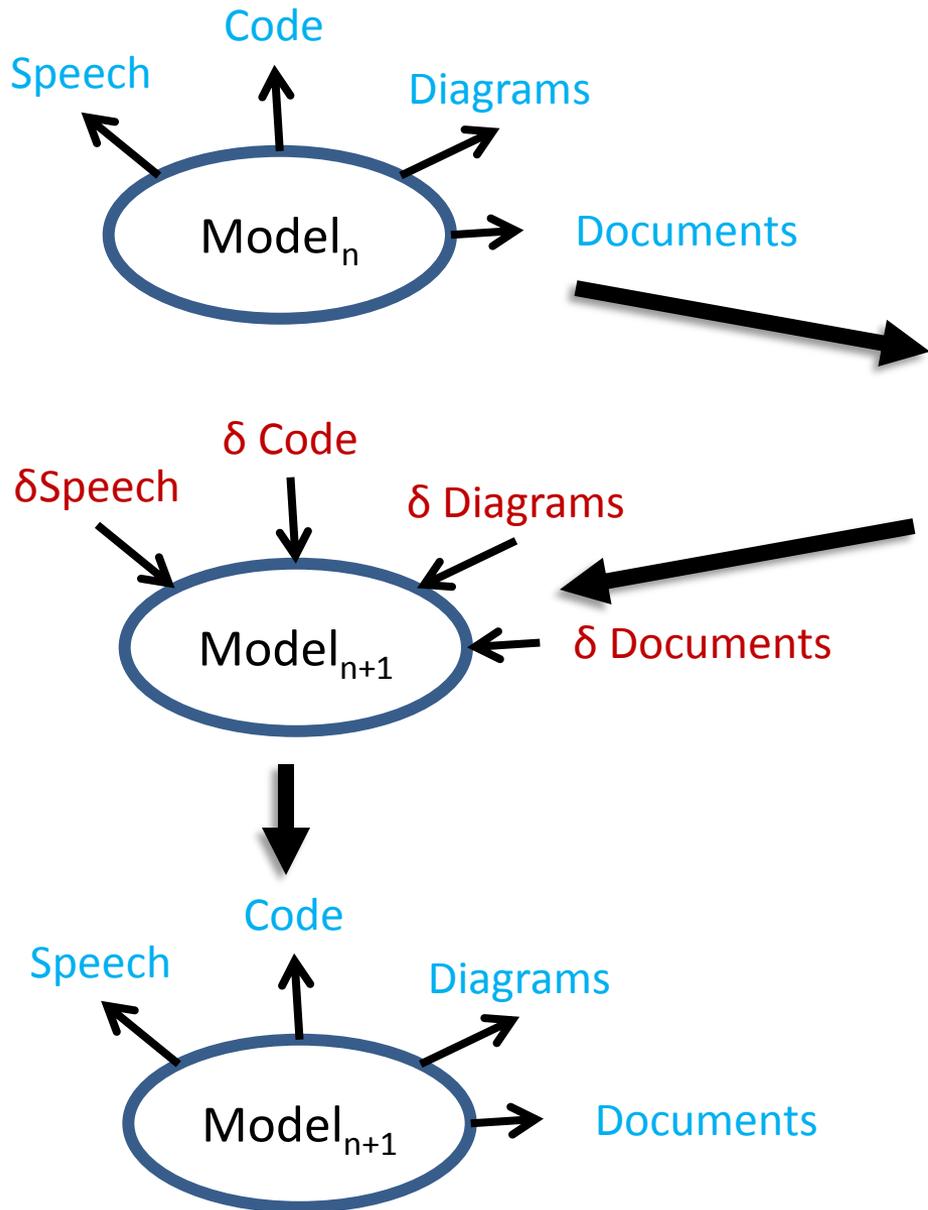
- A model is not a one-shot deal.
- Rather the model emerges out of multiple iterations of refactoring, discussion and knowledge evolution.
- Start with an initial model that is a best guess based on a discussion with domain experts.
- Evolve the model throughout the lifetime of the project.

Iterative Process



- A model implies
 - language to be used in speech
 - code implementation
 - diagrams
 - language in documents
- If the actual speech, code, diagrams and documents used are different, then the model needs to be revised

Iterative process



Knowledge Crunching and Refactoring
 $\delta \text{ Speech} = \text{Actual Speech} - \text{Speech}$
 $\delta \text{ Code} = \text{Actual Code} - \text{Code}$
 $\delta \text{ Diagrams} = \text{Actual Diagrams} - \text{Diagrams}$
 $\delta \text{ Documents} = \text{Actual Docs} - \text{Docs}$

Supported by the model
Actually used
Differences

A Single Model

- “MODEL-DRIVEN DESIGN discards the dichotomy of analysis model and design to search out a single model that serves both purposes. ... This requires us to be more demanding of the chosen model, since it must fulfill two quite different objectives (pg. 49).”

Developer Model

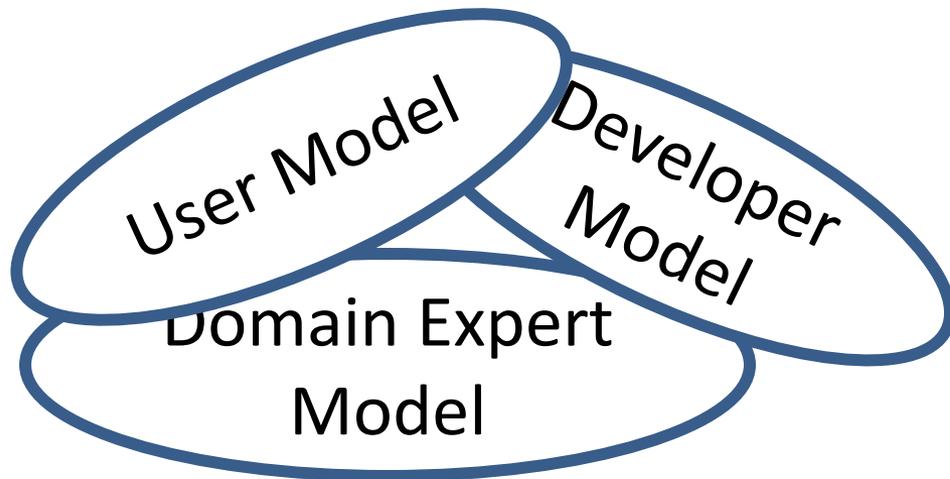
- Developers must buy-in and feel responsible for the model
 - “If developers don’t realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it (pg 61).”
 - The abstractions will not accurately reflect the domain knowledge
- “With a MODEL-DRIVEN DESIGN, a portion of the code is an expression of the model; changing the code changes the model. Programmers are modelers, whether anyone likes it or not. So it is better to set up the project so that the programmers do good modeling work (pg. 61).”

User Model

- “In theory, perhaps, you could present a user with any view of a system, regardless of what lies beneath. But in practice, a mismatch causes confusion at best – bugs at worst (pg. 57).”

A Single Model

- Although the initial model may not be identical in all cases, over iterations, the models should converge



Why One Model?

- “The single model reduces the chances of error, because the design is now a direct outgrowth of the carefully considered model. The design, and even the code itself, has the communicativeness of a model (pg. 50).”
- Note the one model view does not mean that different sub-systems cannot have their own model, but that all those involved in the sub-system need to use one model.

Hands-On Modelers

- “All teams have specialized roles for members, but over separation of responsibility for analysis, modeling, design, and programming interferes with MODEL-DRIVEN DESIGN (pg. 60).”
- Model’s intent can be lost in the handoff
 - “The overall effect of a model can be very sensitive to details, and those details don’t always come across in a UML diagram or a general discussion (pg. 60).”
- Indirectness of Feedback
 - Certain aspects of the model can be wildly inefficient, the project leader needs to know about this so the model can be reformulated. Otherwise the developers might abandon the model

The Building Blocks of a Model-Driven Design

- To maintain the correspondence between model and implementation there are specific techniques that Eric Evans suggests.
 - Isolate the domain using a layered architecture
 - Domain layer techniques
 - Use associations wisely
 - Use appropriate model elements
 - Utilize Modules

Isolating the Domain

- “To apply our best thinking, we need to be able to look at the elements of our model and see them as a system. We must not be forced to pick them out of a much larger mix of objects, like trying to identify constellations in the night sky. We need to decouple the domain objects from other functions of the system, so we can avoid confusing the domain concepts with other concepts related only to software technology or losing sight of the domain altogether in the mass of the system (pg. 67).”

Layered Architecture

- The architecture can be separated into layers with specific responsibilities,
 - User Interface
 - Application
 - Domain
 - Infrastructure

Layered Architecture

- The domain layer should be isolated
 - This allows domain objects to be designed without simultaneously thinking about the user interface
- “But the main benefit is simplifying the application layer, keeping it narrowly focused on its job: knowing *when* to send a message, but not burdened with *how* (pg 73).”
- Services should be loosely coupled to the rest of the system.

Domain Layer Building Blocks

- Associations
- Three patterns of model elements
 - Entities
 - An object that represents something with continuity and identity – something that is tracked through different states or even across different implementations
 - Value Objects
 - Attribute that describes the state of a particular object aspect
 - Services
 - Actions or operations
 - “Although it is a slight departure from object-oriented modeling tradition, it is often best to express theses as SERVICES, rather than forcing responsibility for an operation onto some ENTITY or VALU OBJECT (pg. 82).”
- Modules
 - “The ideas of hgh cohesion and low coupling, foten thought of as technical metirics, can be applied to the concepts themselves. In a MODEL-DRIVEN DESIGN, MODULES are part of the model, and they should reflect concepts in the domain (pg. 82).”

Associations

- A model typically has many associations which can make implementation and maintenance complicated (especially many-to-many associations)
- Making associations more tractable
 - Impose a traversal direction
 - Add a qualifier
 - Eliminate nonessential associations
- This makes associations more expressive of the model as well as more tractable

Entity Pattern

- “An object defined primarily by its identity is called an ENTITY (pg. 91).”
- “They have life cycles that can radically change their form and content, but a thread of continuity must be maintained (pg. 91).”
- “Their class definitions, responsibilities, attributes, and associations should revolve around who they are, rather than the particular attributes they carry (pg. 91).”
- Entity should be stripped down to characteristics that uniquely identify it and commonly used to find and match it

Value Objects

- Could make all objects entities...
 - “Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary (pg. 98).”
 - Only use entities where necessary
- An object that represents a descriptive aspect of the domain with no conceptual identity
- “[I]nstantiated to represent elements of the design that we care about only for *what* they are, not *who* or *which* they are (pg. 98).”

SERVICES

- “In some cases, the clearest and most pragmatic design includes operations that do not conceptually belong to an object. Rather than force the issue, we can follow the natural contours of the problem space and include SERVICES explicitly in the model (pg. 104).”
- Operation names should come from the UBIQUITOUS LANGUAGE
- Parameters and results should be domain objects
- Should be used judiciously
- Note: There is a distinction between services discussed here that are used in the domain layer and those of other layers. Technical SERVICES lack business meaning.

A Good SERVICE

1. The operation relates to a domain concept that is not a natural part of an ENTITY or VALUE OBJECT
2. The interface is defined in terms of other elements of the domain model
3. The operation does not maintain an internal state that affects its own behavior (stateless).

MODULES (a.k.a PACKAGES)

- “MODULES give people two views of the model: They can look at detail within a MODULE without being overwhelmed by the whole, or they can look at relationships between MODULES in views that exclude interior detail (pg. 109).”
- The MODULES in the domain layer should emerge as a meaningful part of the model, telling the story of the domain on a larger scale (pg. 109).”
- MODULES can be dangerous since the cost of refactoring MODULES can be prohibitive
- “If your model is telling a story, the MODULES are chapters (pg. 110).”
- “Give the MODULES names that become part of the UBIQUITOUS LANGUAGE (pg. 111).”

Refactoring Towards Deeper Insight

- The real challenge is to find an incisive model
- Success developing useful models comes down to three points
 1. Sophisticated domain models are achievable and worth the trouble
 2. They 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.
 3. They may call for sophisticated design skills to implement and to use effectively. (pg.188)

Types of Refactoring

- Micro-refactorings
- Refactoring to a design pattern
- Refactoring to a deeper model
 - Superimposed on micro-refactoring and refactoring to a design pattern
 - Occasionally characterized by a breakthrough

Breakthroughs

- Breakthroughs are brought about by increasing clarity of the domain and result in changes to the model that are a much better reflection of the domain
- Breakthroughs are often scary because they often require changing a lot of supporting code with few if any stable stopping points, all in the context of a looming deadline

Making Implicit Concepts Explicit

- Deep modeling often comes about by realizing that an important concept is present implicitly in the design and would be better expressed if present explicitly
- “Process starts with recognizing implied concepts in some form, however crude (pg. 205).”
- Identifying missing concepts is aided by...
 - Listening to the language of the domain experts
 - Scrutinizing awkwardness in the design
 - Listening for seeming contradictions in the statements of experts

Making Implicit Concepts Explicit

- Non-obvious implicit concepts
 - Explicit Constraints
 - Can factor out constraints into methods with intention revealing names or into a separate object entirely
 - Processes
 - Make explicit an important domain process that is otherwise obscured
 - SPECIFICATION
 - a separate VALUE OBJECT that contains business logic in the form of a method that resembles a predicate.
 - “A SPECIFICATION is a predicate that determines if an object does or does not satisfy some criteria (pg. 226).”

Supple Design

- Although the ultimate purpose of the software is to serve users, which are often the domain experts themselves, the software must first serve developers.
- A supple design is one that is a pleasure to work with and is inviting to change

Supple Design:

INTENTION-REVEALING INTERFACE

- “Type names, method names, and argument names all combine to form an INTENTION-REVEALING INTERFACE (pg. 247).”
- “Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise (pg. 247).”
- “Write a test for a behavior before creating it, to force your thinking into client developer mode (pg. 247).”

Supple Design:

SIDE-EFFECT-FREE FUNCTIONS

- “Interactions of multiple rules or compositions of calculations become extremely difficult to predict (pg. 250.)”
- To make code easier to use, separate calculations and state change into different operations.

Supple Design: ASSERTIONS

- “Assertions make side effects explicit and easier to deal with (pg. 255).”
- “State post-conditions of operations and invariants of classes and AGGREGATES. If ASSERTIONS cannot be coded directly in you programming language, write automated unit tests for them (pg. 256).”

Summary

- Communication is key
 - A UBIQUITOUS LANGUAGE facilitates the transfer of knowledge between domain experts, designers and developers.
 - A good design and a single model ameliorates cognitive overload.
 - Everything in domain-driven design is a communication mechanism
- A single model encourages a solution that accurately reflects the subtleties of the domain.
- An accurate model results in a quality product that is scalable.