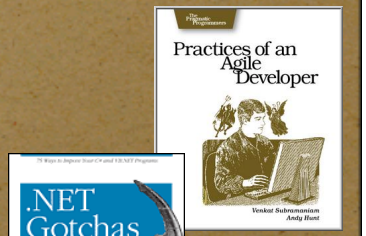
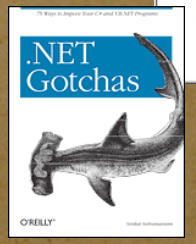
Domain Driven Design

spkr.name = 'Venkat Subramaniam'
spkr.company = 'Agile Developer, Inc.'
spkr.credentials = %w{Programmer Trainer Author}
spkr.blog = 'agiledeveloper.com/blog'
spkr.email = 'venkats@agiledeveloper.com'





Abstract

- Domain Driven Design (DDD) is an approach that places emphasis on the domain model and carrying it into implementation. DDD is mostly repackaging of fundamental OO Design. It brings new emphasis to what we should be already doing, but often find it hard and confusing given the realities and complexities of our real world. In this presentation we will take a close look at what DDD is and how to use it for agile development. We will discuss several design options, and also look at some examples of good modeling and layering.

- We'll delve into Domain Model, Model and the implementation, Domain objects and life cycle, Developing with domain model, Design strategies, Refactoring...

Who's this session for?

- If you're a master of OO Design, you don't need this
- If you've struggled with OO Design and need to
  - review some concepts
  - may be clear up some basics
  - rethink about modeling
  - this session will help you
- Domain Driven Design is mostly repackaging of fundamental OO Design.
- It brings new emphasis to what we should be doing, but often find it hard and confusing given the realities and complexities of our real world.

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

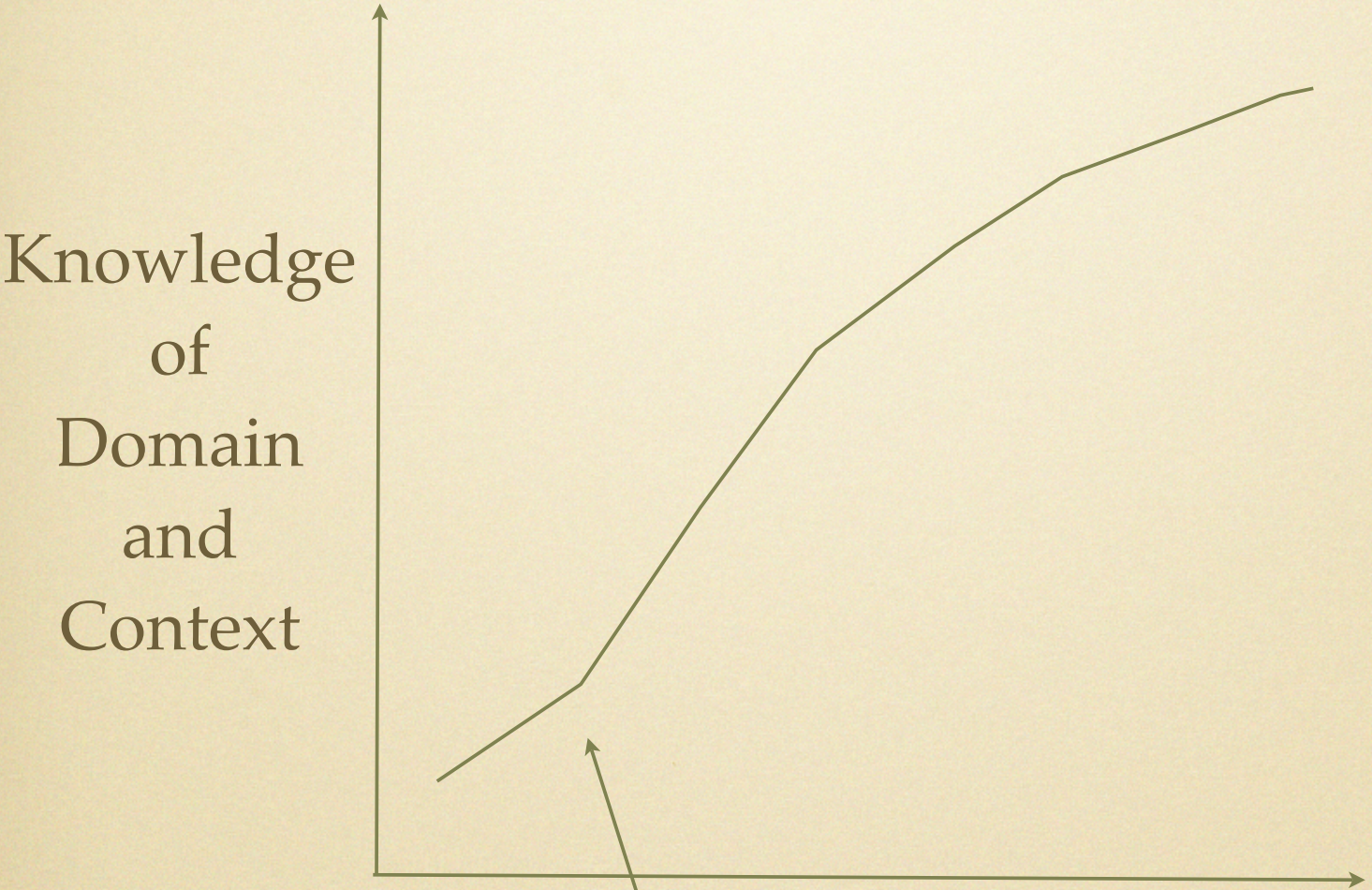
Successful/Famous System Represents Good Design?

Successful/Famous Systems Represent Great Process?

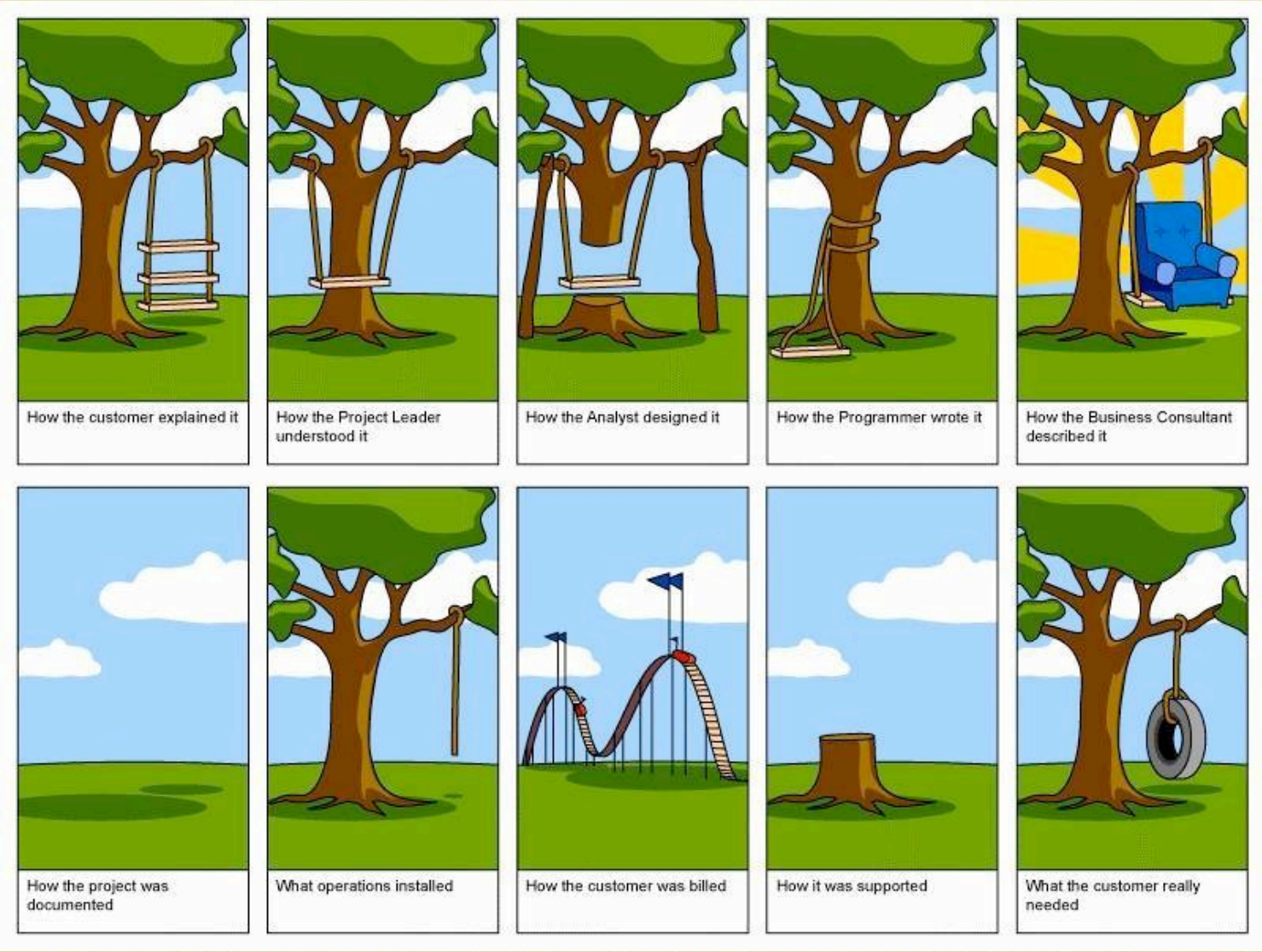
Development Challenges

- Is Software Development about
  - programming?
  - languages?
  - technology?
  - framework?
  - design?
  - application domain?

Development Challenges



From Requirements…



Understanding Domain is Essential

Source of picture unknown

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

What's Domain?

- It's www.?
- It's an area or sphere of knowledge, influence or activity

Domain and Developers

- Developers like coding, technical stuff
- Domain is
  - hard
  - unclear
  - unknown
  - uninteresting at times to technically focused
  - We didn't go to school for that...

Why's Domain important?

- What's the purpose of the software you're building?
- Does your software model the domain?
- Relevance of your effort to develop your application
- Domain discussions help foster communication with customers
- Provides an abstract to deal with complexities
- Design made by domain knowledge lacking is like promise made by a politician—it doesn't hold

One of the problems we face is understanding the problem itself

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

What's a model?

- Smaller scale representation of a person or structure?
- An Example?
- A person displaying a fashion?

ahem, What's it to the geeks?

A Model

- It's a concept or idea that is represented in some form or fashion
  - May be in a diagram
  - Written code
  - Textual description
- In OO we've called this abstraction
- Model is distilled knowledge

Purpose of Model

- Model is often tied to database making it hard to develop and test
- Modeling should help us focus on domain, not implementation
- Model serves a particular Use
- Model must be precise at modeling, design, and code level

What's it not?

- Focusing on model does not mean up-front design
  - Developing a good model is hard, takes time, iterations
  - Practice Evolutionary model development
- Model you've developed is not set on stone
  - You need to take time to evolve it
  - Technology you use must facilitate change
  - If hard, you will resist its evolution
  - Model does not mean to let UI access data directly
    - Good layering is critical to maintainability

Details in a Model

- Model should capture essential details
- What is a Car?
  - Depends on who you ask
    - Car manufacturer?
    - Driver?
    - Insurance sales man?
- Think about what it means—in your application—in the context of its domain

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

Challenges in Modeling

- Earlier we were told "Classes often are nouns in the problem statement"
- Often these tend to be mostly entity objects
- Control and Boundary often tend to get missed
- Model is skewed and ineffective

Lost Details

Often, details are lost

Manitou Springs Cliff Dwelling... About Cave Writing

"We can safely assume that the art created during prehistoric times had meaning to its creator. The symbols may have had spiritual significance; however, when the creator walked away the meaning was lost forever."

Lost Details

- We often hear something similar in our field
- We can safely assume that the design created during development times had meaning to its creator. The work may have had business significance; however, when the creator walked away the meaning was lost forever.

Capture that Model

- Model needs to be captured
- It can't live and die in our heads
- We need to express it, communicate it, share it, transfer it, in a precise, clean, unambiguous way

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD
- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

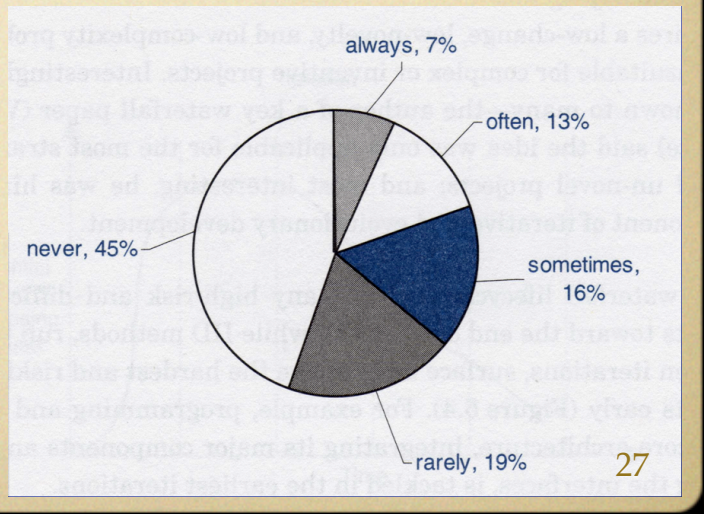
What's Domain Driven Design?

- Way of thinking with some reaffirmed set of priorities
- Emphasizes business domain more than technology
- Focus on domain logic
- Base complex designs on a model

Why DDD?

- We've got to Understand the Domain
- We need to put effort into areas that really matter
- Features we build must serve purpose
- Understand if a feature is relevant
  - Not everything customers say is relevant

Low actual usage of requested features



Source of Picture: Craig Larman's Agile and Iterative Development: A Manager's Guide

DDD & Agile Development

- Agile Development is about getting feedback to develop relevant working software
- Communication is critical
- DDD is intended to foster that communication

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

Design

- Strategic vs. Tactical
- There are two levels of design
- Strategic design is
  - big picture
  - high granular
  - implementation agnostic
- Tactical design is detailed, fine grain, and implementation specific
- DDD helps us get the most out of Strategic design activities

Developing a Domain Model

- Interact and Brainstorm
  - An analyst can't simply handout the requirements to you
  - Don't soldier alone
  - Collective effort of developers and domain experts
- Promote a language for communication
- Separate chaff from the wheat—winnow extraneous details
- Quickly prototype and get feedback
- Use a common language to communicate

What makes a Domain Model?

- Domain Data + Domain Logic = Domain Model
- Domain Logic includes
  - Validation logic
  - Calculations
  - Business Rules
- all pertinent to the abstraction
- Domain model is behaviorally rich

Ubiquitous Language

"A language structured around the domain model and used by all team members to connect all the activities of the team with the software"

- Has name of classes and key operations
- Discuss central rules
- Artifacts + Tasks + Functionality
- Used by developers to communicate with domain experts
- Domain experts can use it to communicate among themselves

Is Ubiquitous Language UML?

- It does not have to be
- UML is powerful, yet, not as expressive in cases
- The goal is not to use a diagram
- It is to communicate
- Find notation/language that helps your developers and domain experts communicate

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

From Model to Implementation

- Modeling is necessary but not sufficient
- Translating the model into appropriate code is critical
- Not effective if you've middleman in this
- Domain experts understand what they want
- Developers understand technology and how to implement
- Model has to be refined based on feasibility and capability
- Developer—Domain Expert direct interaction is critical
  - Developing the model is an iterative activity
- Discuss-prototype-feedback cycle

Signs of Ineffective Modeling

- A single change to a business rule results in cascade of change
- Often because code is not cohesive
- Code was hacked up along the way
  - Some mistake this to be agile!
- There must be a direct correlation between model and code

Beyond Class Level Cohesion

- Layering application is critical
- Layer is not simply piling up packages
- Layers must be cohesive
  - Each must focus on dealing with one central concept
- Layer must be loosely coupled
- It must depend on layer below
  - May be relaxed layering (may bypass a layer)

Which of these two conveys good design?

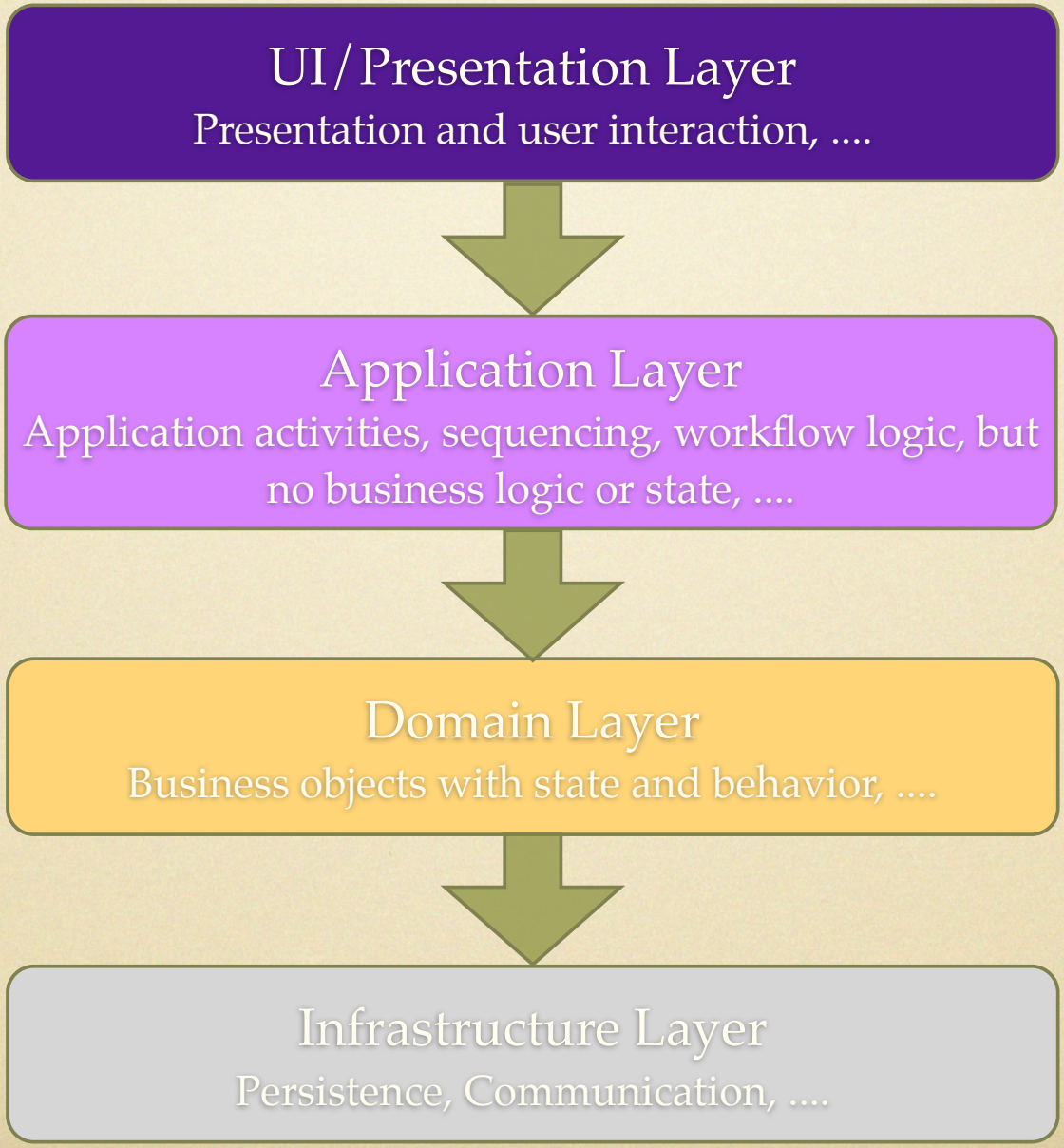
Why?

Layering

Examples of Bad Design

- UI does business validation
  - In the name of responsiveness
    - But, what if you need Browser side validation?
      - Can appropriate tier generate your client side script?
- Database Stored procedures do business validation
  - In the name of performance
  - Hard to modify
- Too closely tied to technology
- UI directly talks to database, in the name of RAD

Layers


UI/Presentation Layer
Presentation and user interaction, ...
Application Layer
Application activities, sequencing, workflow logic, but no business logic or state, ...
Domain Layer
Business objects with state and behavior, ...
Infrastructure Layer
Persistence, Communication, ...

Domain Layer

- Represents concepts of the business
- Information
- Business rules
- State of business
- Does not deal with storing these, however (leaves that to infrastructure)

Application/Service Layer

- Objects in this layer are responsible for application functionality
- These direct Domain objects
- Keep this thin
- No business rules or domain knowledge
- Has no business state, but may have state of task progress for user interaction

Anemic Domain Model

- Anemic Domain Models tend to look real
- They've rich relationships and structure
- Tend to lack behavior
- Service objects often capture all the domain logic and sit on top of these anemic data only objects
- Kind of like the unfortunate pure value objects in some Applications
- Look for signs that your domain model may be anemic—they look like structs

http://www.martinfowler.com/bliki/AnemicDomainModel.html

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

Creating a Model

- Manifests in the forms of
  - Entities
  - Value Objects
  - Service Objects
  - Modules

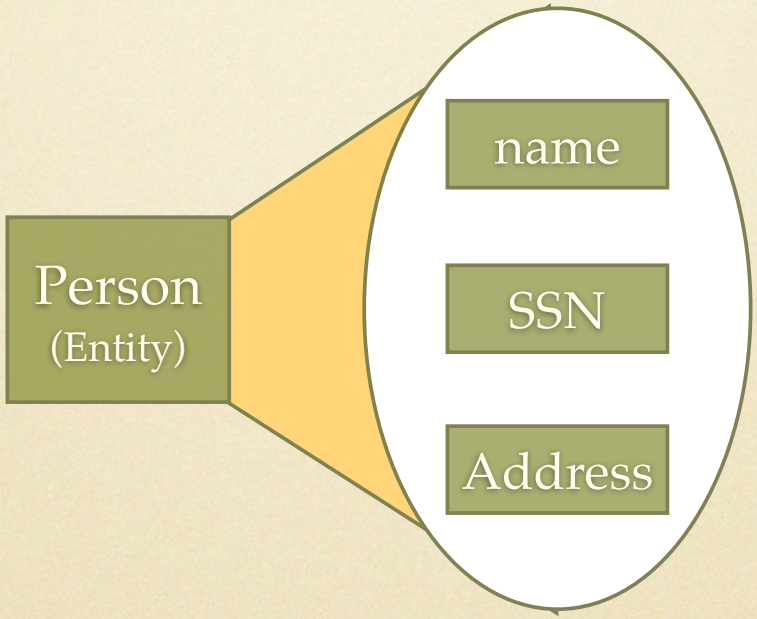
Entities

- These represent information
- Each entity has identity
  - Their life may span the life of the system
- Identity is unique
- Use caution in modeling and implementing Identity
- What about Identity of a person
- Well, could we use SSN as ID, after all that's unique?
  - Not prudent as some people don't have it, and laws around its use changes
- Use internal unique keys instead

Value Objects

- Has aspects or attributes of a domain
- Has no identity
- May be immutable
- May be shared
- Light-weight, easily copied if necessary

Entities and Value Objects



Service Objects

- You may have use the term "Control" objects
- These are behavior that does not belong to entity or value objects
- Does not in itself have state
- Often deal with interaction between multiple objects
- These objects may exist in different layers depending on the objects they "serve"

Module

- Large applications benefit from a higher level of description
- It's a grouping of classes
- System is viewed as organized by intercommunicating modules
- Modules must be cohesive—classes work towards common functionality
- Modules must be loosely coupled

Domain Objects Life Cycle

- We need to deal with
  - States of Domain objects
  - Creation, Use, Persistence, Destruction
- Aggregates
- Factories
- Repositories

Relationships

- Association represents relationship with objects of equal status
- Aggregation is a stronger form with objects taking ownership
- Need to deal with different relationships
  - one-to-one
  - one-to-many

Relationships Information

- Where do we put information that related to the relationships?
- Sometimes easier to conceptualize, but hard to implement
- Try to eliminate unnecessary relationships
- Simplify and abstract

Relationships and Navigation

- Which one of these is better?

Bi-directional Navigation: Engine has link to Car

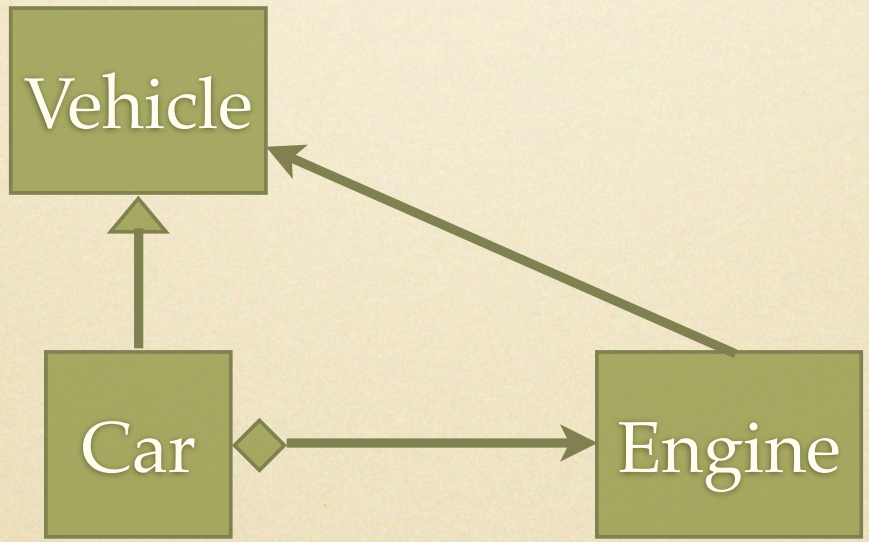
Uni-directional Navigation: Engine has no link back to Car

Design of Relationships

- Bi-directional Navigation
  - Makes navigation faster
  - Leads to more coupling
  - Hard to keep change consistent
- Single-directional navigation helps
- Lower Coupling
- Makes code more Reusable
- But, makes navigation slower
- How to have the best of both?

Design of Relationships...

- Dependency Inversion Principle
- Inversion of Control (IOC)



Aggregates

- Defines object ownership and boundaries
- Group of objects considered one unit with regard to a change
- Composite Design pattern emphasizes this
- Root is an entity and acts as entry point to the aggregate
  - Queries lead to these and other internal objects are traversed from here
- You may send out references to internal objects
  - User can't hold reference to it for later processing
  - You may get by sending copy of value objects just in case

Factories

- Abstract and encapsulate the creation of objects
- Help separate code from intricate dependencies that may otherwise exist
- May help separate creation of a flavor of an object
- Set of Creational Design Patterns emphasize this
- Almost trivial to do this in most modern languages

Repositories

- The layer that deals with persistence
- If client directly interacts with this, leads to stronger coupling
- Some technologies (that will remain unnamed ;)) promote, in the name of
- RAD, lead to coupling UI tightly with database
  - Makes it hard to make changes later on
- Frameworks and approaches must favor rapid development without compromising extensibility and maintainability
- Implementation of Repository is in the infrastructure
  - May be one per Aggregate
  - May be Generic
- Its interface is in the domain model

Refactoring

- It's an act of improving the design of code without changing its external behavior
- Developing a domain model is an iterative process
- It needs to be achieved through a series of refactoring
- Business rules and coarse grain domain objects are often recognized later
  - Handling these rules in different objects
    - Makes it messier
    - Hard to maintain
    - Hard to understand
- Good separation of concern can help make code expressive and more explicit
- This has to be recognized well at the domain model
- Must be expressed well in code and in Ubiquitous Language

Model Integrity

- Model should handle constraints and invariants well
- Model has to be maintained as our understanding of domain evolves

Continuous Integration

- Reality check
- Asserts consistency between models
- Especially critical when different teams involved in creating application
- Each team with different context need to come together

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

Context

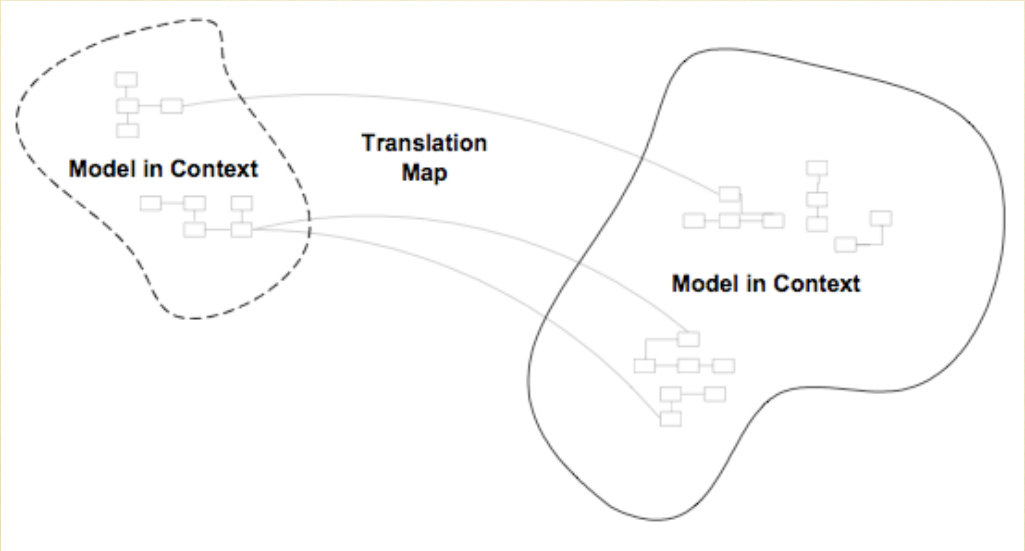
- Each model has a context
  - A set of conditions, constraints, and terms used in defining the model
  - Defines a logical frame within which your model evolves
  - "The settings in which a word or statement appears that determines its meaning"
- Multiple context may play a role in large applications
- Mixing context is challenging, error prone, confusing

Bounded Context

- Define the boundaries of the context
- Unify and keep your model consistent within those boundaries
- Not clearly knowing the model you are bound to leads to inappropriate modeling
- Over complication
- Unsatisfactory application
- Be keen on separating contexts to keep things sane

Context Map

- Helps us outline different Bounded Contexts and relationships between them



Patterns for Creating Context Maps

- Shared Kernel
- Customer-Supplier

High degree of interaction between contexts

- Anticorruption Layers
- Open Host Services

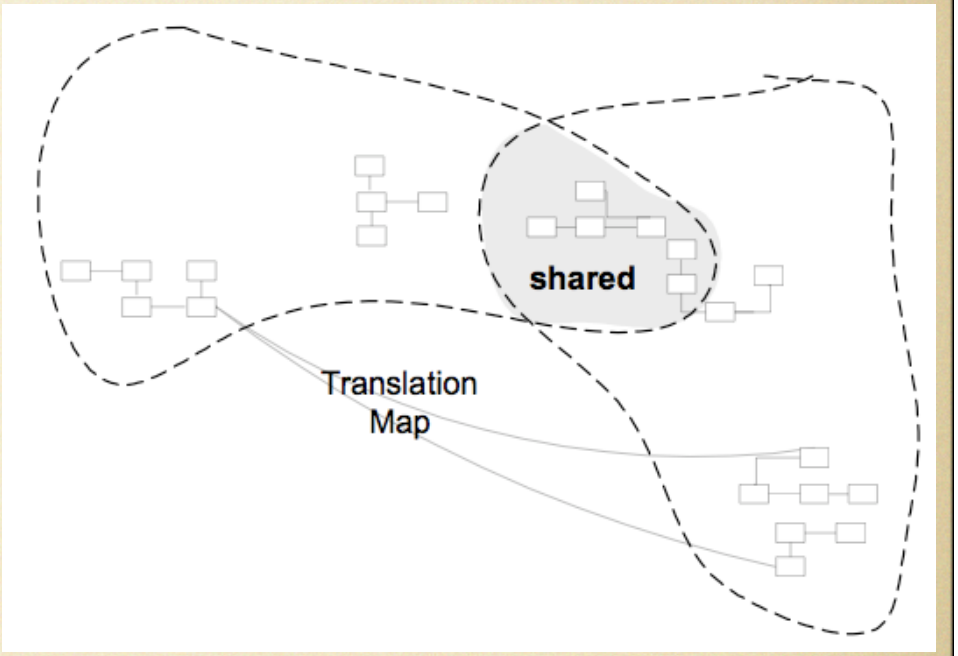
Interaction with legacy or external systems

- Separate Ways

High independence of contexts

Shared Kernel

- Large projects involved teams that work on different contexts
- Integration often is hard and error prone
- Increased risk



- Define subset of domain model that teams share
  - Includes the code and database model sharing
- Helps reduce duplication while keeping teams' contexts separate
- Not trivial, needs frequent integration
- Rigorous frequent automated tests essential

Customer-Supplier

- Establish a customer-supplier relationship between teams
- One team produces what the other team consumes without any active sharing of model
- May make use of the same database or database schema
- Stronger need for defining interface for interaction
- Automated conformity tests are essential in this case

Anticorruption Layer

- Interaction with external applications
- You may open access to database
- Data is shared, but what about semantics and constraints?
- Opening up data access may end up violating these
- Anticorruption Layer is
  - a service that controls access to protect data integrity and consistency
- Takes care of any needed translations
- This service may be a Façade or an Adapter

Open Host Service

- When your system has to interact with several applications
- Hard to provide customized interaction with each
- You may need more generalized approach
- Define a protocol for communication

Separate Ways

- Sometimes there is not much common
- Integration may be more harder and not cost effective
- You may bundle a number of applications together from the users' point of view
- Development wise, they have gone separate ways and have nothing much in common at model and code level

Agenda

- Challenges
- Domain
- Model
- Modeling
- DDD

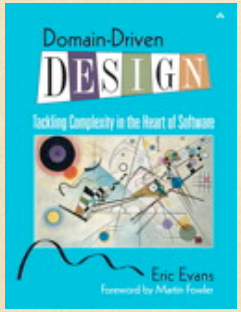
- Domain Model
- Design
- Life Cycle
- Context
- Conclusion

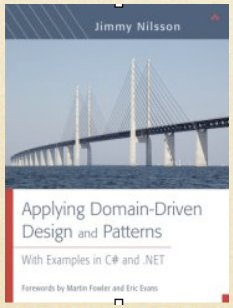
Concepts in DDD

- Layered Architecture
- Emphasis on Model which in turn focuses on Domain
  - Entities
  - Value Objects
  - Services
  - Modules
  - Aggregates
  - Factories
  - Repositories
- Context Boundaries
  - Patterns to Map Context

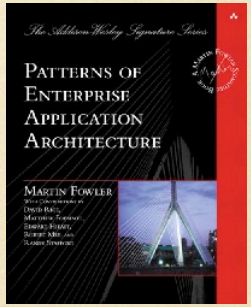
Quiz Time

References


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


Jimmy Nilsson
Applying Domain-Driven Design and Patterns
With Examples in C# and .NET
Forewords by Martin Fowler and Eric Evans


Domain-Driven Design Quickly
InfoQ


PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
MARTIN FOWLER

Thank You!

http://www.agiledeveloper.com — download