DOMAIN DRIVEN DESIGN

Moving beyond Data Processing
and COBOL Oriented Java

by Mark Windholtz

THE POINT

- Data Processing is not good or bad
  - It fits into some context
- OO is not good or bad
  - It fits into some context
- To move from DP to OO
  - First understand the context

Audience

- You are the most informed developers in town
  - May not be telling you much new
- A goal would be to provide a more descriptive vocabulary to talk about development choices
- Data Processing & Object Oriented
  - Are separate choices
  - Most a J2EE is implemented in DP style

What is Data Processing

- The execution of a systematic sequence of operations performed on data.
- Operations performed on data to provide useful information to users.

COBOL DIVISIONS

- COBOL programs are divided into four structural elements ...
- **IDENTIFICATION DIVISION**
  - Contains names of the program and programmer
- **ENVIRONMENT DIVISION**
  - Contains environment information device or encoding sequence.
  - Aliases are assigned to external devices, files or command sequences
- **DATA DIVISION**
  - Contains data descriptions

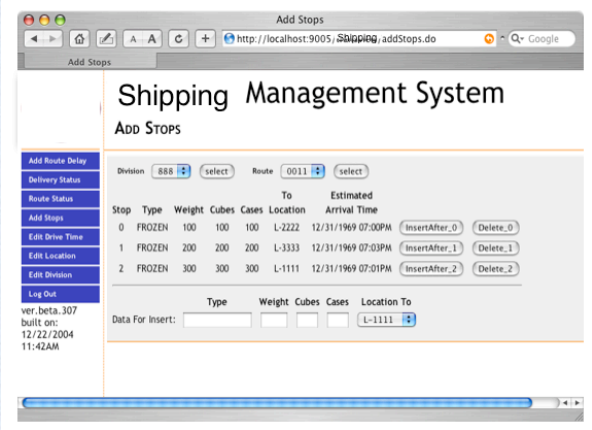
```
DATA DIVISION.
WORKING-STORAGE SECTION.
01   Num1              PIC 9  VALUE ZEROS.
01   Num2              PIC 9  VALUE ZEROS.
01   Result            PIC 99 VALUE ZEROS.
```

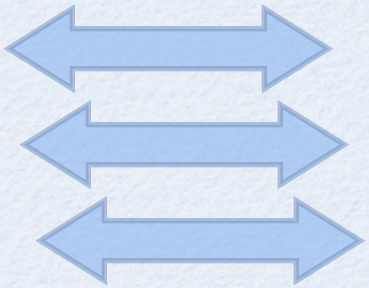
- **PROCEDURE DIVISION.**
  - Contains the program algorithm

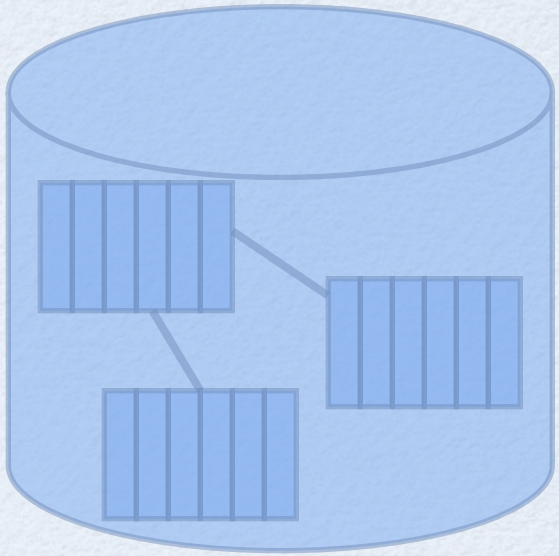
COBOL ORIENTED JAVA

- Many J2EE programs are divided into three distinct structural elements...
- **ENVIRONMENT DIVISION**
  - Contains environment information (i.e. properties files, xml config files)
  - Aliases are assigned in a Constants class
- **DATA DIVISION**
  - Contains data descriptions
  - "Business Objects" with primitive types and getters & setters
  - *Mutable Value Objects* which duplicate BOs used to transfer data
- **PROCEDURE DIVISION**
  - Contains the program algorithm
  - Services, Manager, or Applications Facades, DAOs, and Util classes that
    - Load a BO from database
    - Get data from it
    - Calculate stuff
    - Set result into BOs
    - Save the BOs back in database

Data Processing Design







- Starts with screens & database fields
- Considers the middle as distinct scripts
  - One per transaction (create, delete, list, etc …)
- J2EE blueprints and Patterns encourage a DP approach
- Details are coded into Screens, Script, & DB
- Fowler calls this the Transaction Script Pattern

Why do Data Processing?

- DP is simple
  - Simplest thing that becomes expensive to extend
- Natural way to organize small amounts of logic
- Little overhead in processing or understanding
- Compatible with Multi-phase Waterfall Software process
- When your team lacks Object Oriented Experience
- Company Culture ...

DP CULTURE

- My First Client-Server Application
  - Relational Database
  - Requirements from the Director
- Software people with DP background talk about Data and Processing separately
  - Invoice **num**, Policy type **flag**, Status **code**, Dodad **switch**
- Language of Data and Process indicate that the DP-Software people have taught Business people to talk in DP terms.

DP Across Depts

- App Dev job is to build applications
- Also must play well with others
  - Other Depts: Database, QA, Report-Dev
- Interface between departments
  - Language of least-common-denominator
  - Data fields & Database Schema
  - Phased-process and *Frozen Signed-off* Documents
- DP and COBOL Oriented J2EE
  - path of least resistance

OO CULTURE

- Business people talk naturally in Business terms
  - Unapproved Invoice, Delivered Item,
  - Extended E&O Policy, Gold Medallion Account
- Objects shift language from *data* to *behavior*
  - Away from *flags* and *if* conditions
  - Toward *responsibilities* and *naming*
- Iterative, Evolutionary Development stream
  - One Team - Communicating
    - not multiple departments to coordinate
  - Phased processes inhibit OO designs
  - Stream processes accelerate OO designs

DP INDICATORS

- Data Structures
  - “Beans” with only get/set gunk
- Util classes
- Code Smells
  - Shot gun Surgery
  - Primitive Obsession - too many *String* and *Date* objects
  - Long methods, Long argument lists
  - others
- Constants

DP WITH CONSTANTS

- Public Global Constants are Data!
- Something uses that data, right?
- That something is separate from the Constant
  - i.e. Un-Encapsulated
- If that something is in one place
  - Encapsulate the constant inside the processing
- If that something is in multiple places
  - ... Danger ... Danger ... Danger ...
  - Conditional logic will proliferate
  - Adding a new value easily causes defects

DP DOWNERS

- As application grows
  - Duplication increases
  - Common code is repeated
    - Moved into vaguely named Utility classes
- Large transaction scripts are error prone & hard to test
- Maintenance becomes increasingly difficult
  - Sometimes becomes impossible

Domain Modeling Used to be Called...

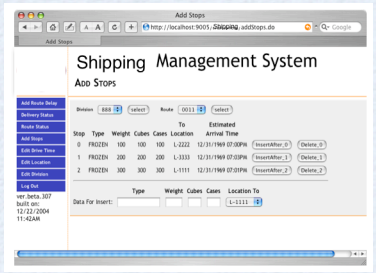
Object Oriented Programming

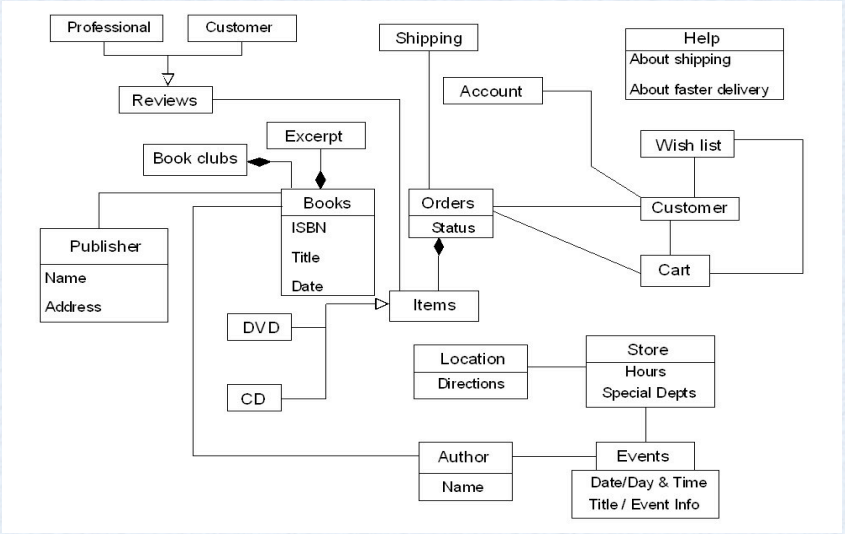
Domain Model

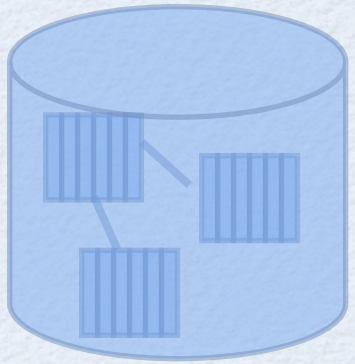
- Business logic can be complex
  - Rules in DM describe the many cases & variations
- DM creates a web of interconnected objects where each represents a meaningful concept some as large as an entire company or as small as a house number

DOMAIN MODELING

We think in generalities, but we live in details.
-Alfred North Whitehead







- Models the business domain concepts
- Next map it to Screens and DB
- Easier to test, and change
- More Agile

Styles of Domain Model

- None
  - Transaction Script, Data Structure
  - COBOL oriented J2EE
- Simple
  - Similar layout to the Database schema
- Rich
  - Easier to capture complex business logic
  - Different from the DB schema
  - Can be harder to map to Relational DB

Anemic Domain (Anti-Pattern)

- Data structures
  - Named like objects with relationships
  - But no behavior but maybe equals(), toString()
- Behavior provided by Service Classes
  - http://www.martinfowler.com/bliki/AnemicDomainModel.html

TS || DM ?

- DM
  - Complicated, changing business rules
  - Validations, calculations, derivations
  - Interdependent Objects involving Design Patterns
- TS
  - Simple null checks & summation
  - Simple Create, Read, Update, Delete
  - Deeply rooted DP culture and language

Recommendation - current apps

- Easy: Encapsulate Constants
  - Combine with Util classes
  - Real Value Objects
- Harder: Add XP Practices
  - Without XP Practices, Domain Design (i.e. OO is difficult or impossible)

Recommendation - New Apps

- Don't use DP & Transaction Script
- Start with a Simple Domain Model
  - Similar to the Database schema
  - Should be acceptable even in DP culture
- Minimize use of Constants
  - Util classes,
- Evolve a Rich DM as needed
- Use XP Practices

XP Primary Practices

- Sit Together
- Incremental Design
- Test-First Programming
- Continuous Integration
- Ten-Minute Build
- Slack
- Quarterly Cycle
- Weekly Cycle
- Stories
- Pair Programming
- Energized Work
- Informative Workspace
- Whole Team

XP Corollary Practices

- Real Customer Involvement
- Pay-Per-Use
- Negotiated Scope Contract
- Daily Deployment
- Single Code Base
- Code and Tests
- Shared Code
- Root-Cause Analysis
- Shrinking Teams
- Team Continuity
- Incremental Deployment

XP PRINCIPLES

- Humanity
- Accepted Responsibility
- Baby Steps
- Quality
- Failure
- Redundancy
- Opportunity
- Flow
- Reflection
- Diversity
- Improvement
- Self-Similarity
- Mutual Benefit
- Economics

XP Values

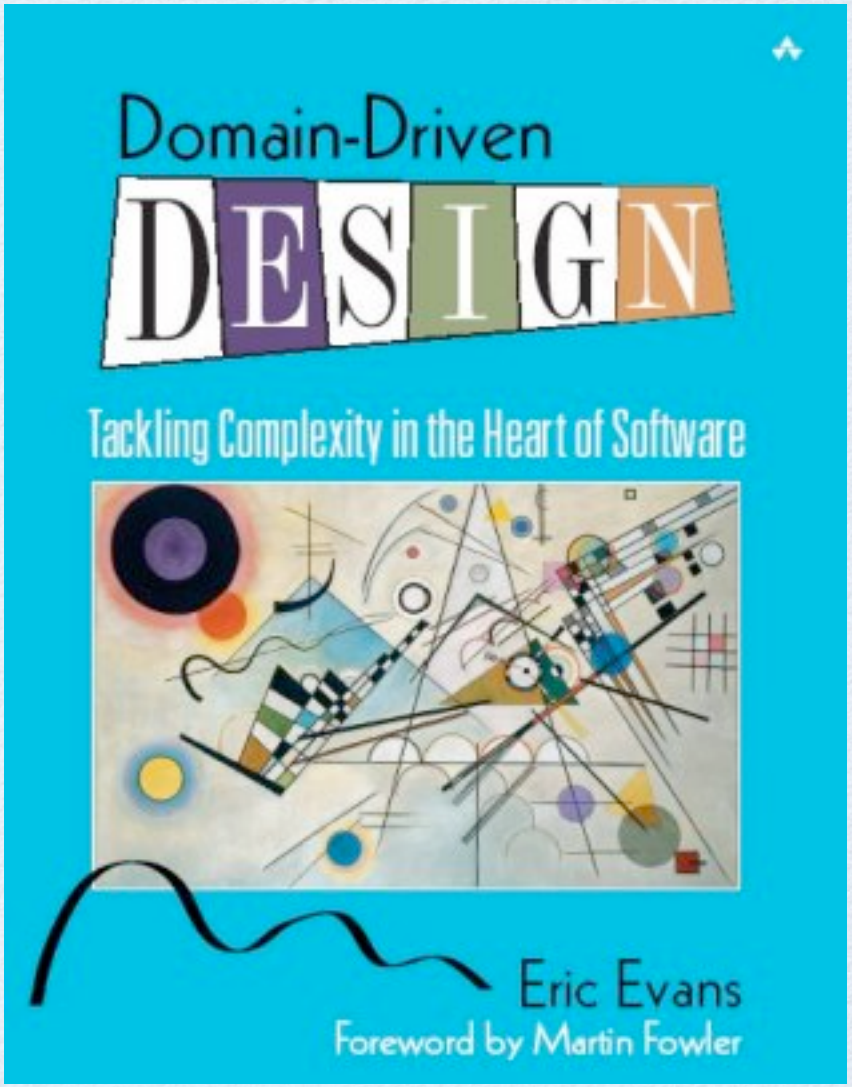
- Communication
- Respect
- Courage
- Feedback
- Simplicity

BUILDING THE DM

- Bind model with implementation
  - Paper only models lack rigor
  - Iteratively test & build the code
- Discover a common language to talk with your customer
  - Ubiquitous Language
- Knowledge rich model
  - Model should enforce behavior and rules
  - Model is not merely a data schema
- Distill the Model
  - Adding concepts to a model is good
  - Reducing the complexity with abstractions is vital
- Brainstorming and experimenting
  - Try variations of ideas
  - See what fits best
  - Experiment in code

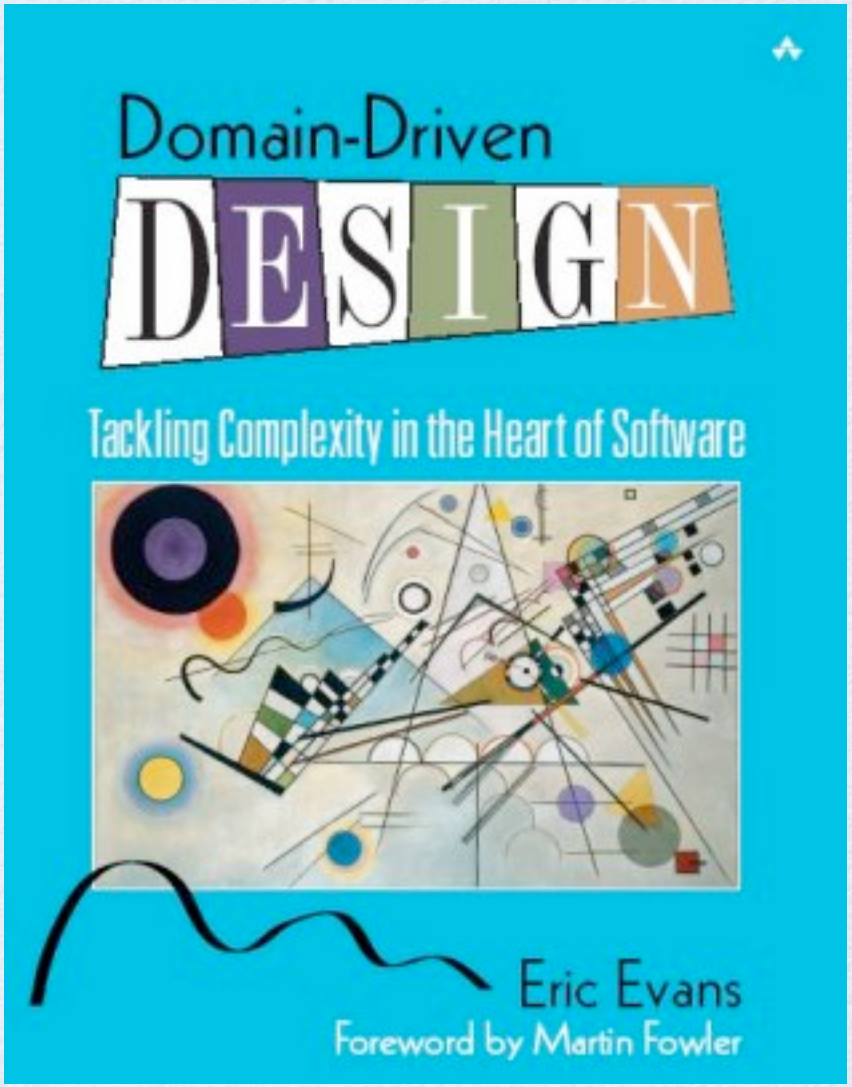
Patterns of Domain Design

- Layered Architecture
- Model-driven Design
- Entities
- Value Objects
- Services
- Modules
- Aggregates
- Factories
- Repositories
- domaindrivendesign.org



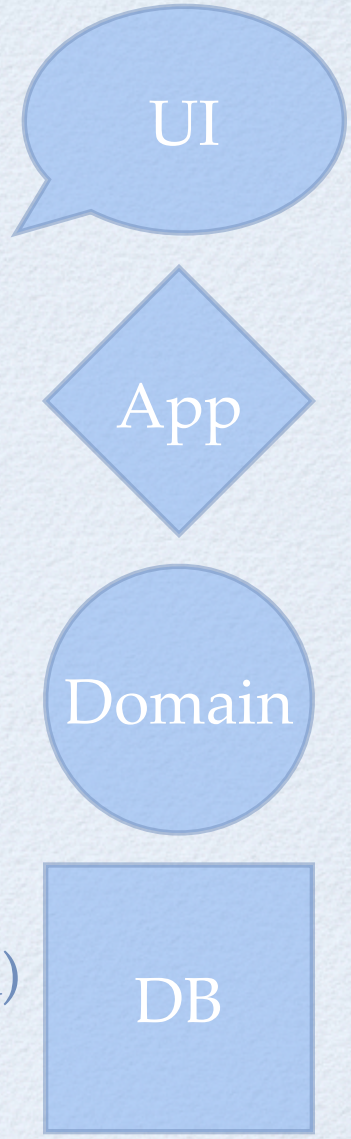
Patterns of Domain Design

- Layered Architecture
- Model-driven Design
- Entities
- Value Objects
- Services
- Modules
- Aggregates
- Factories
- Repositories
- domaindrivendesign.org



Layered Architecture

- UI
- Application
  - Coordinates tasks and delegates work to Domain
- Domain (or Model)
  - Business concepts, rules, state
  - The Heart of the Business
- Infrastructure
  - Persistence in DB
  - Message sending
  - Support for other layers (ie. Spring framework)



Model-Driven Design

- Associations need to be implemented
  - Constraining them makes implementation easier. It gives the Model more focus and allows less code implementation
- Refining Associations
  - Impose Transversal direction
  - Add qualifiers (reduce multiplicity)
  - Eliminate Non-essential Associations

Entities

- Not talking about *J2EE Entity Beans*
- Many objects are not defined by attributes but a thread of continuity and identity
- Mistaken identity can lead to data corruption
- Entities manage life-cycle and identity
- Strip Entities down to the attributes that are used to find or match them
- Only holds behavior that is essential
- Extract Value-Objects where ever possible

Value Objects

- **Not** *Sun's "Value Objects"*
  - J2EE VOs are data-structures. (period)
- Domain Value Objects are immutable
  - have no setters, no modifiers
  - They have no identity fields
  - Conceptual whole
    - Address: street, city, state
  - VOs can be shared freely
- Other examples

  junit.samples.Money

  xpcinci.domain.Day

Day - Value Object

- Xp-Cinci calendar application
  - Day is a Value Object
  - java.util.Date is encapsulated
- Day objects are immutable
  - set and forget
- Day operations mostly return other Day objects
  - plus(int days)
  - yesterday()
  - tomorrow()
  - prevSundayOrToday()

Services

- An operation offered as a stateless interface
- Should not be overused!
- Characteristics
  - Not a part of an Entity or Value Object
  - Interface contains elements of Domain Model
  - Operation is stateless

Example services by layer

- Application p107
- Mail utilities
- Managing Background processing

Modules

- To Localize changes and additions ...
- Create packages according to the motto
  - *classes that play together stay together*
- Group according to their likelihood of changing together

  `xapp.invoicing`

  `xapp.accountspayable`
- Not according to their hierarchy trees, or framework functionality

  `xapp.formbeans`

  `xapp.actions`

  `xapp.valueobjects`
- Avoid the Code Smell: *Shotgun Surgery*

LIFE CYCLE

Aggregates

- Entities are the main target of queries
- Not all Entities should be directly queried
- Limit query access to Aggregates of Entities that make sense

FACTORIES

- Day day = new WorkDay();
  - Creates a dependency from the client to the Day class.
- Day day = DayFactory.createDay()
  - Allows client to reduce dependencies and change more gracefully.
- Spring or Abstract Factory pattern
  - eliminates dependency on Factory Implementation

Repositories

- Aka "DAO"
- Repositories provide the way to persist and query Domain objects and Aggregates
- Don't query for Value Objects since they have no identity, you can just create them
  - Day aDay = new Day("30 June 1955");
- Therefore, convert reduce as much as possible to Value Objects
  - Shrink your core Entites to a smaller and smaller set

Patterns of Domain Design

- Layered Architecture
- Model-driven Design
- Entities
- Value Objects
- Services
- Modules
- (Life Cycle)
- Aggregates
- Factories
- Repositories

Styles of Domain Model

- None
  - Transaction Script, Data Structure
  - COBOL oriented J2EE
- Simple
  - Similar layout to the Database schema
- Rich
  - Easier to capture complex business logic
  - Different from the DB schema
  - Can be harder to map to Relational DB

TIPS

- Minimize Contants
- Minimize Utils classes
- Minimize getter/setter - Data structures
  - Ask: where's the Behavior?
- Minimize passing primative data types
  - Use real Value Objects
- Encourage Stream Processes like XP

My Questions

- Can Java / J2EE survive
  - ... as a Data Processing Language?
- What forces or events could possibly
  - move the dominant DP culture to Objects?
- Or is it time to move to a language
  - with a culture rooted in Object Thinking?

THE END

Moving beyond Data Processing
and COBOL Oriented Java