Software Design Best Practices
A Real-World Example

Naresh Bhatia
Domain Expert
Sapient® Global Markets
Agenda

Bullsfirst	Domain-Driven Design	Executable Specifications
Agenda

Bullsfirst Demo
http://archfirst.org/apps/bfclient-silverlight/bullsfirst.html
Bullsfirst Architecture

http://archfirst.org/books/bullsfirst
What’s wrong with this code?

class public TradingService {

    public Money calculateOrderEstimate(Account account, Order order) {

        // Calculate basic order estimate
        Money unitPrice;
        if (order.getType() == OrderType.Market) {
            unitPrice = marketDataService.getMarketPrice(order.getSymbol());
        } else {
            unitPrice = order.getLimitPrice();
        }
        Money estimate = unitPrice.times(order.getQuantity());

        // Calculate fees
        Money fees = (account.getOwner().isPreferred()) ? DISCOUNTED_FEES : REGULAR_FEES;

        // Return estimate plus fees
        return estimate.plus(fees);
    }
}

These objects are “Anemic”!

class public TradingService {

    public Money calculateOrderEstimate(Account account, Order order) {

        // Calculate basic order estimate
        Money unitPrice;
        if (order.getType() == OrderType.Market) {
            unitPrice = marketDataService.getMarketPrice(order.getSymbol());
        } else {
            unitPrice = order.getLimitPrice();
        }
        Money estimate = unitPrice.times(order.getQuantity());

        // Calculate fees
        Money fees = (account.getOwner().isPreferred()) ? DISCOUNTED_FEES : REGULAR_FEES;

        // Return estimate plus fees
        return estimate.plus(fees);
    }
}
What’s wrong with Anemic Objects?

• Violation of basic object-oriented design principles
• Require global access to internals
• Procedural code duplication
• Use of if-then-else logic
Let's fix them

```java
public class TradingService {
    public Money calculateOrderEstimate(
        Long accountId, OrderParams orderParams) {
        Account account = accountRepository.findAccount(accountId);
        return account.calculateOrderEstimate(new Order(orderParams));
    }
}

public class Account {
    public Money calculateOrderEstimate(Order order) {
        Money estimate = order.estimate();
        Money fees = this.isEligibleForDiscount() ?
            DISCOUNTED_FEES : REGULAR_FEES;
        return estimate.plus(fees);
    }
}
```
Let’s fix them (continued)

public class Order {
    public Money estimate() {
        Money unitPrice;
        if (this.type == OrderType.Market)
            unitPrice = marketDataService.getMarketPrice(this.symbol);
        else
            unitPrice = this.limitPrice;
        return unitPrice.times(this.quantity);
    }
}
Agenda

Domain-Driven Design

- Ubiquitous Language
- Bounded Contexts
- Command-Query Separation
- Layered Architecture
- Core Domain
- Aggregates
Ubiquitous Language

• Developers think in terms of code – domain experts know nothing about that!

• Ubiquitous Language – common language used by everyone on the project

As you know, we maintain a list of positions for each brokerage account. A position contains the number of shares owned for a specific security.
Ubiquitous Language

• Developers think in terms of code – domain experts know nothing about that!

• Ubiquitous Language – common language used by everyone on the project

But that’s not enough. For regulatory reasons we need to keep track of lots. Every time a security is purchased, a new lot must be created. Whenever the security is sold, shares must be taken away from existing lots, allowing us to calculate the gain.
Ubiquitous Language

• Developers think in terms of code – domain experts know nothing about that!

• Ubiquitous Language – common language used by everyone on the project

Hmmm, **Lot** seems like a pretty important concept. Let’s add it to our model.
Bounded Contexts

- Sheer volume of concepts on a large project can be daunting
- Divide large domains into smaller manageable contexts
Bounded Contexts – Commands, Queries and Events

- Commands **tell** a bounded context to do something
- Queries **ask** for information from it
- Bounded context generates events to **inform** that something happened
Command-Query Separation (CQS)

- An approach to simplify designs by separating reads from writes

- In a nutshell, every method should either be
  - a **command** that performs an action, or
  - a **query** that returns data to the caller,
  - but not both

- Asking a question should not change the answer

Example

```java
public interface TradingService {

    // ----- Commands ----- 
    void placeOrder(Long accountId, OrderParams params);
    void cancelOrder(Long orderId);

    // ----- Queries ----- 
    List<Order> getOrders(OrderCriteria criteria);
    OrderEstimate getOrderEstimate(Long accountId, OrderParams params);
}
```
BEST PRACTICES IN APPLICATION ARCHITECTURE

TODAY: USE LAYERS TO DECOUPLE

... AND EVERY YEAR WE MOUNT A NEW LAYER TO DECOUPLE US FROM THE CRAP WE'VE DONE THE YEAR BEFORE

ANNUAL RINGS
Layered Architecture (continued)

- Software applications are more than just a domain model
- Handling all the functionality cleanly requires *separation of concerns*

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface Layer</td>
<td>Accepts user commands and presents information back to the user</td>
</tr>
<tr>
<td>Application Layer</td>
<td>Manages transactions, translates DTOs, coordinates application activities, creates and accesses domain objects</td>
</tr>
<tr>
<td>Domain Layer</td>
<td>Contains the state and behavior of the domain</td>
</tr>
<tr>
<td>Infrastructure Layer</td>
<td>Supports all other layers, includes repositories, adapters, frameworks etc.</td>
</tr>
</tbody>
</table>
Layered Architecture (continued)

- Coordinates application activity
- Creates and accessed domain objects

User Interface Layer:
- Web Service
  - SecurityWebService
- JMS Listener
  - MarketPriceListener
- Web Service
  - TradingWebService
- JMS Listener
  - ExchangeMessageListener

Application Layer:
- Service
  - AccountService

Domain Layer:
- MarketPrice
- User
- Person
- BaseAccount
- Transaction
- ExternalAccount
- BrokerageAccount
- Order
- Lot

Infrastructure Layer:
- UserRepository
- ReferenceDataAdapter
- MarketDataAdapter
- ExchangeTradingService
- AccountRepository

State and behavior of domain
Supporting code for all other layers
Core Domain

Entities

- **BaseAccount**
  - name: String
  - status: AccountStatus
  - changeName(newName: String): void
  - transferCash(transfer: CashTransfer): void
  - transferSecurities(transfer: SecuritiesTransfer): void
  - isCashAvailable(amount: Money): boolean
  - isSecurityAvailable(symbol: String, quantity: DecimalQuantity): boolean

- **BrokerageAccount**
  - ownershipType: OwnershipType
  - cashPosition: Money
  - placeOrder(params: OrderParams): Order
  - getAccountSummary(): BrokerageAccountSummary
  - getPositions(): Position[]
  - calculateOrderEstimate(params: OrderParams): OrderEstimate

- **ExternalAccount**
  - routingNumber: String
  - accountNumber: String
  - getAccountSummary(): ExternalAccountSummary

Value Objects

- **Money**
  - amount: BigDecimal
  - currency: Currency

Domain Services

- **MatchingEngine**
  - placeOrder(order: Order): void

- **OrderBook**
  - add(order: Order): void

- **Order**
  - buyStack
  - sellStack

- **OrderEventPublisher**
  - publish(event: OrderAccepted): void
  - publish(event: OrderExecuted): void
  - publish(event: OrderCanceled): void
  - publish(event: OrderCancelRejected): void
  - publish(event: OrderDoneForDay): void

- **MarketDataEventPublisher**
  - publish(event: MarketPriceChanged): void

Domain Events

- **MarketPriceChanged**
  - marketPrice: MarketPrice

© COPYRIGHT 2011 SAPIENT CORPORATION | CONFIDENTIAL
Aggregate

• Pattern used to define object ownerships and boundaries

• A group of associated objects acting as a unit for the purpose of data changes

• An Aggregate has one root – “owner” of all the other objects
  ▪ External objects can only hold a reference to the root, using commands and queries
  ▪ Allows the root to maintain a consistence state for the entire aggregate
Executable Specifications

• Complementary approach to Domain-Driven Design
  ▪ Specify systems using domain language and examples (the Intent-Example pattern)
  ▪ Specifications can be used to drive acceptance tests using hidden tags

When \(<span concordion:set="#quantity">100</span>\) shares of \(<span concordion:set="#symbol">CSCO</span>\) are bought into a brokerage account @ \(\$<span concordion:set="#price">20.0000</span>\)/share, the following lot is created:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCO</td>
<td>100</td>
<td>$20.0000</td>
</tr>
</tbody>
</table>

If we buy another 200 shares of CSCO @ \(\$21.0000\)/share, the account will now have two lots of CSCO as shown below:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCO</td>
<td>100</td>
<td>$20.0000</td>
</tr>
<tr>
<td>CSCO</td>
<td>200</td>
<td>$21.0000</td>
</tr>
</tbody>
</table>
Concordian Test Results

Lot Creation - Trade

Whenever a security is bought into a brokerage account it creates a new lot.

Example

When 100 shares of CSCO are bought into a brokerage account @ $20.0000/share, the following lot is created:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCO</td>
<td>100</td>
<td>$20.0000</td>
</tr>
</tbody>
</table>

If we buy another 200 shares of CSCO @ $21.0000/share, the account will now have two lots of CSCO as shown below:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCO</td>
<td>100</td>
<td>$20.0000</td>
</tr>
<tr>
<td>CSCO</td>
<td>200</td>
<td>$21.0000</td>
</tr>
</tbody>
</table>

For a detailed discussion of executable specifications, go to http://archfirst.org/books/executable-specifications
Summary

• Domain-Driven Design
  ▪ Use a well thought-out domain model to promote *encapsulation* and *reuse*
  ▪ Establish a *Ubiquitous Language* to discuss domain related concepts
  ▪ Use *Bounded Context* to break down complex domains into manageable parts
  ▪ Use *Command-Query Separation* to simplify your designs and improve performance
  ▪ Implement a *Layered Architecture* to focus on different aspects of the application

• Executable Specifications
  ▪ Valuable part of the system documentation
  ▪ Help with automated regression testing
  ▪ Business and technology should working together to define specifications
  ▪ Break down a complex system into smaller, simpler, crisp requirements
  ▪ Use the Intent-Example pattern to specify and test each requirement
Thank You!

To continue the discussion, visit http://archfirst.org/forum