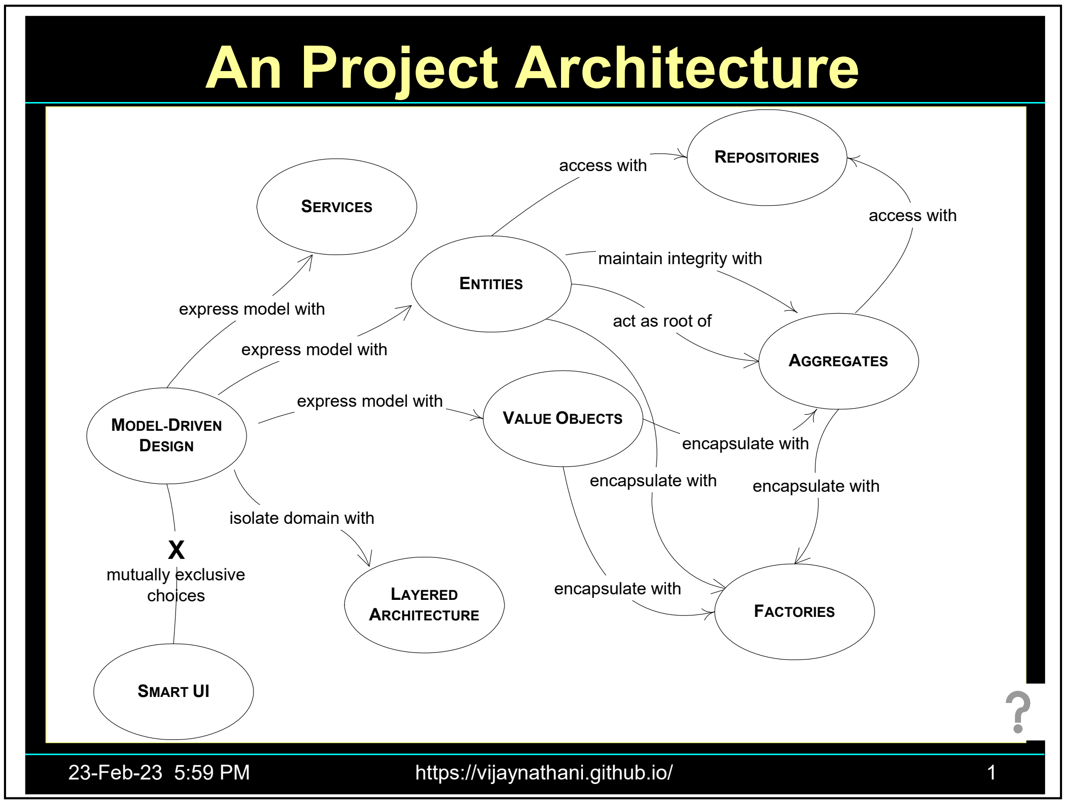
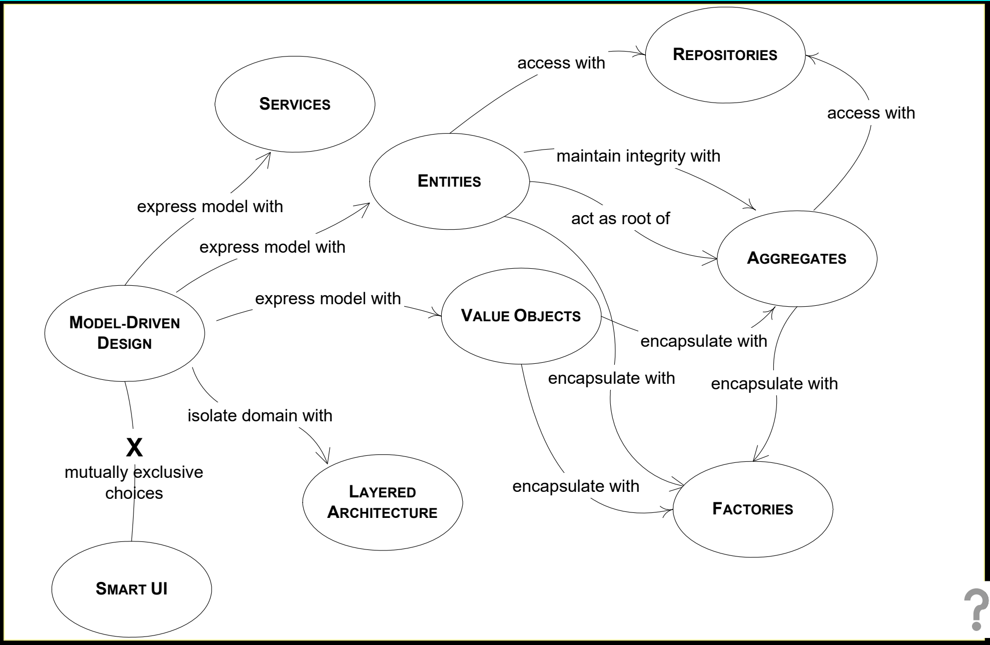


An Project Architecture



QRestaurant

Roles in a domain model:

Entity – Objects with an distinct identity. These are different from J2EE Entity Beans. Try to build Entities only out of value objects.

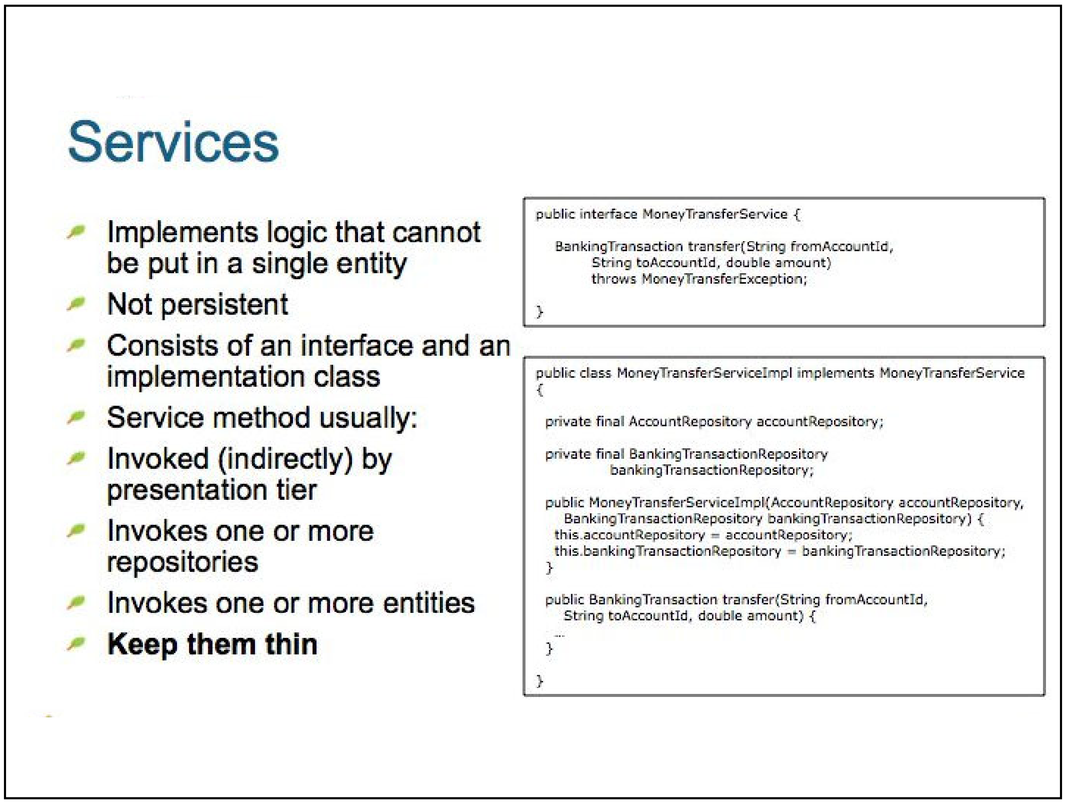
Value – Objects with no distinct identity. These are different from J2EE/Sun Value objects.

Factories – Responsible for creation of Entities.

Repositories – Manage collection of entities and encapsulate the persistence mechanism

Services – Implement responsibilities that can’t be assigned to a single class and encapsulate the domain model. They contain an operation offered as a stateless service.

Repositories – Similar to J2EE DAO. Don’t query for value objects, since they have no identity.



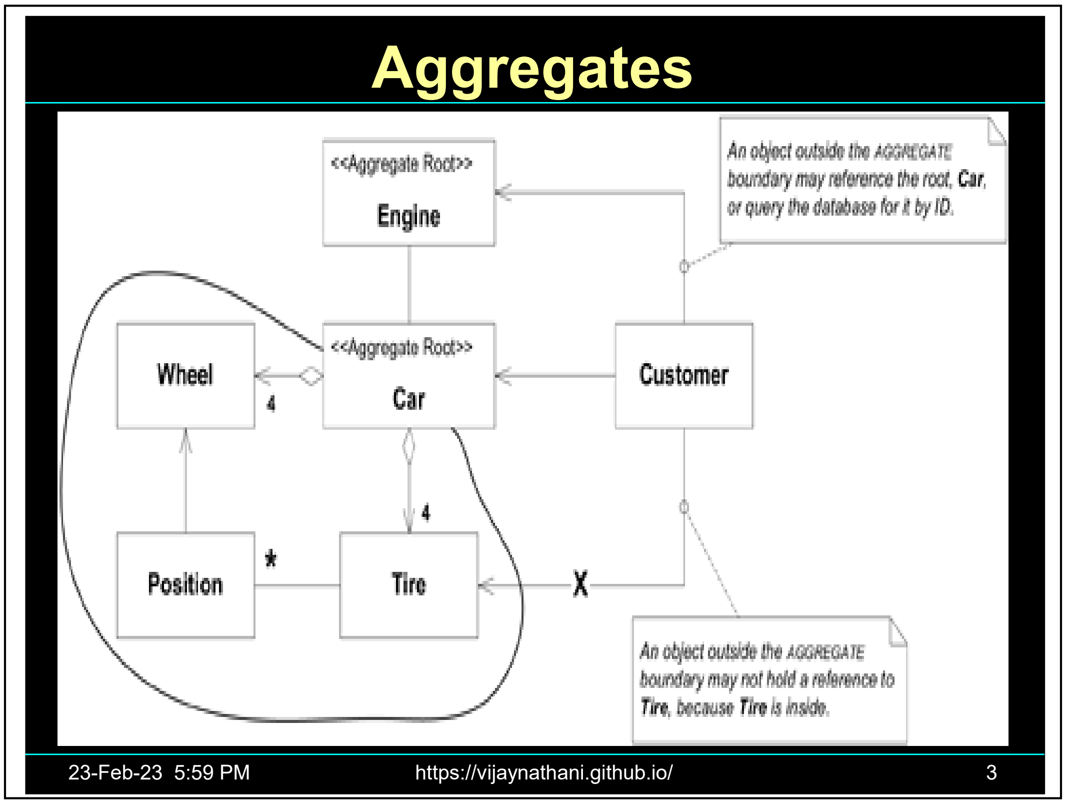
Services

- Implements logic that cannot be put in a single entity
- Not persistent
- Consists of an interface and an implementation class
- Service method usually:
- Invoked (indirectly) by presentation tier
- Invokes one or more repositories
- Invokes one or more entities
- **Keep them thin**

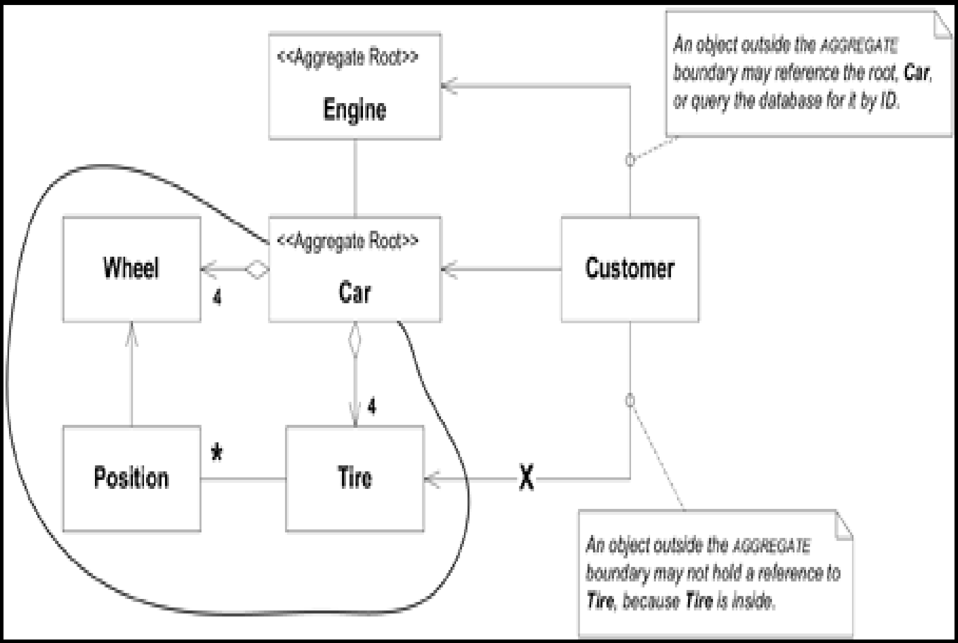
```
public interface MoneyTransferService {

   BankingTransaction transfer(String fromAccountId,
          String toAccountId, double amount)
          throws MoneyTransferException;

}
```

```
public class MoneyTransferServiceImpl implements MoneyTransferService
{

  private final AccountRepository accountRepository;

  private final BankingTransactionRepository
            bankingTransactionRepository;

  public MoneyTransferServiceImpl(AccountRepository accountRepository,
     BankingTransactionRepository bankingTransactionRepository) {
   this.accountRepository = accountRepository;
   this.bankingTransactionRepository = bankingTransactionRepository;
  }

  public BankingTransaction transfer(String fromAccountId,
     String toAccountId, double amount) {
   ...
  }

}
```

Used for transactions, Logging, etc. Aspects used here.



Aggregates



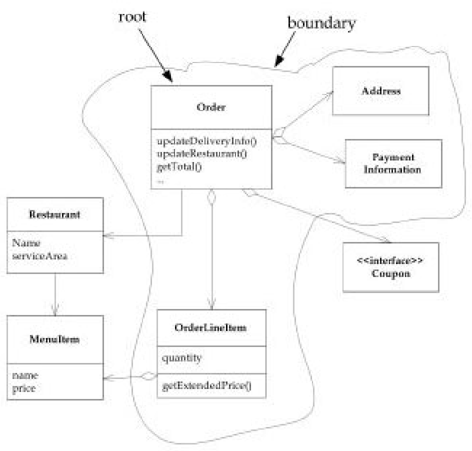
<<Aggregate Root>>
Engine
An object outside the AGGREGATE boundary may reference the root, Car, or query the database for it by ID.
Wheel
4
<<Aggregate Root>>
Car
Customer
4
Position
*
Tire
X
An object outside the AGGREGATE boundary may not hold a reference to Tire, because Tire is inside.

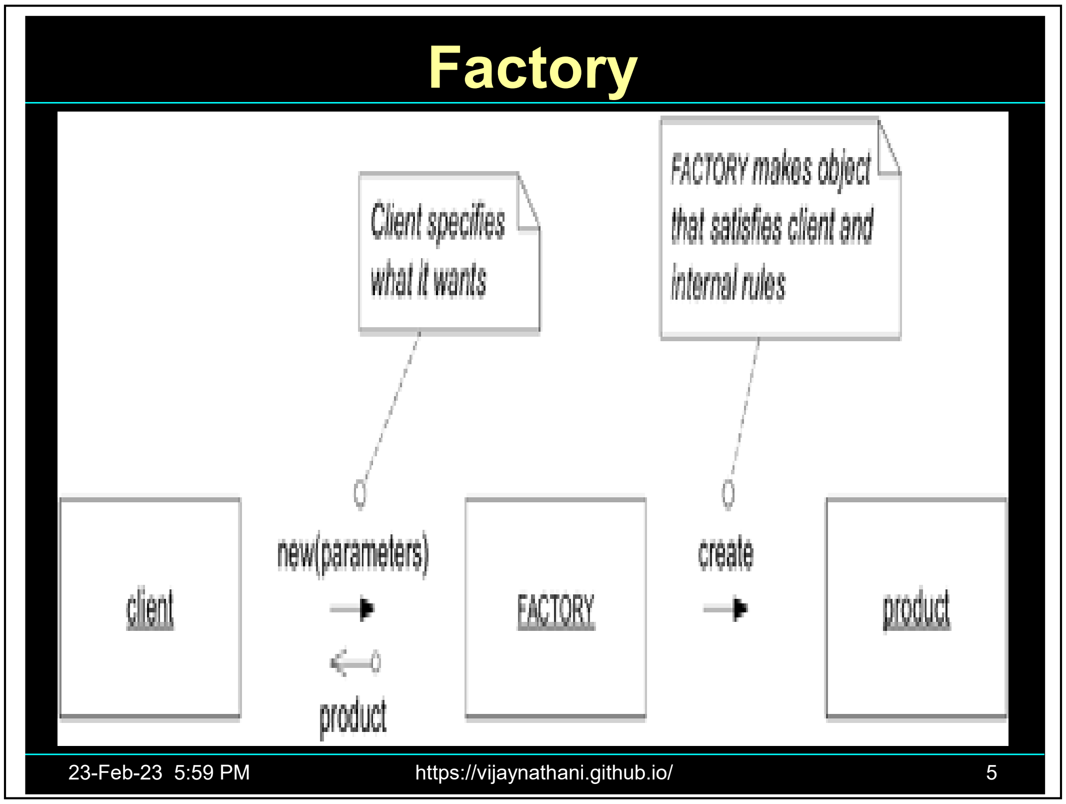
A cluster of related entities and values.
Behaves as an unit.
Has a root.
Has a Boundary.
Objects outside the aggregate can only reference the root
Deleting the root removes everything.

Aggregates provide consistency and transactional boundary.

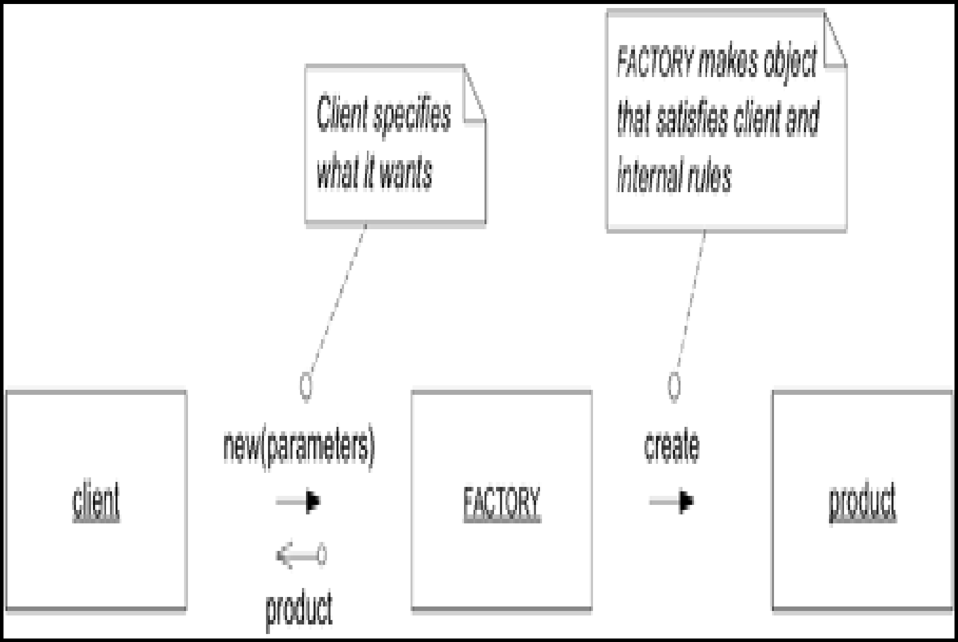
Aggregates

- A cluster of related entities and values
- Behaves as a unit
- Has a root
- Has a boundary
- Objects outside the aggregate can only reference the root
- Deleting the root removes everything





Factory


Client specifies what it wants
FACTORY makes object that satisfies client and internal rules
client
new(parameters)
product
FACTORY
create
product

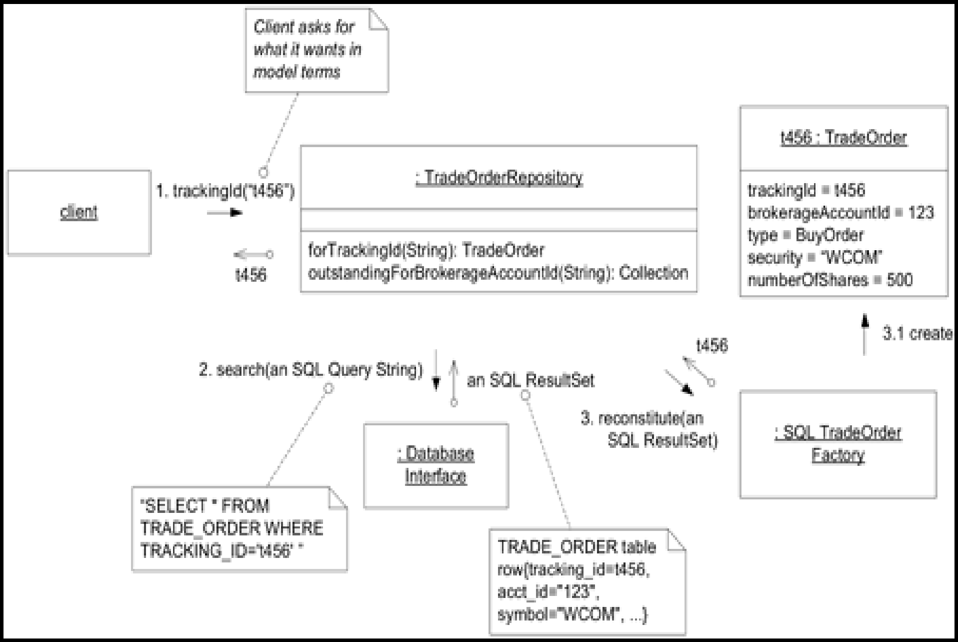
23-Feb-23 5:59 PM https://vijaynathani.github.io/ 5

Constructor's should be dead simple. Otherwise, use a factory. E.g. thread pool of Java, sessionfactory of Hibernate

Factories

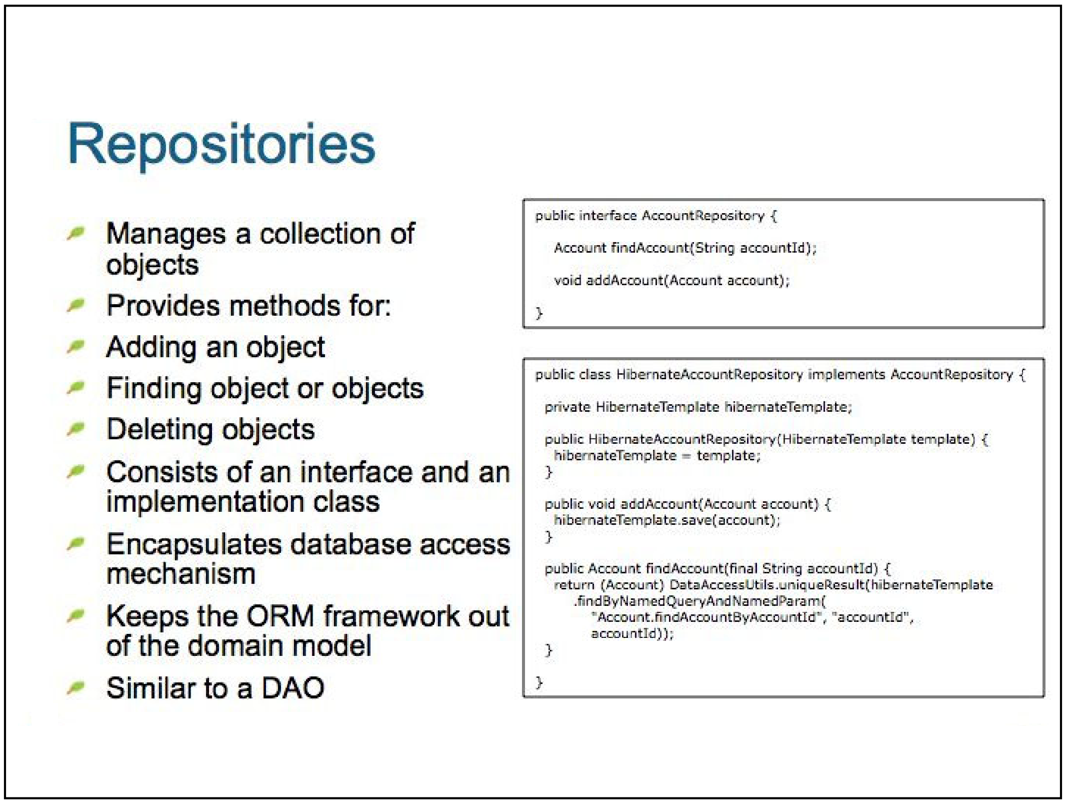
- Use when a constructor is insufficient
  - Encapsulates complex object creation logic
  - Varying products
- Different kinds of factories
  - Factory classes
  - Factory methods
- Example: OrderFactory
  - Creates Order from a shopping cart
  - Adds line items

Repository



Client asks for what it wants in model terms
client
1. trackingId("t456")
t456
: TradeOrderRepository
forTrackingId(String): TradeOrder
outstandingForBrokerageAccountId(String): Collection
t456 : TradeOrder
trackingId = t456
brokerageAccountId = 123
type = BuyOrder
security = "WCOM"
numberOfShares = 500
3.1 create
t456
2. search(an SQL Query String)
an SQL ResultSet
3. reconstitute(an SQL ResultSet)
: SQL TradeOrder Factory
: Database Interface
"SELECT * FROM TRADE_ORDER WHERE TRACKING_ID='t456' "
TRADE_ORDER table row{tracking_id=t456, acct_id="123", symbol="WCOM", ...}

The FACTORY makes new objects; the REPOSITORY finds old objects. The client of a REPOSITORY should be given the illusion that the objects are in memory.

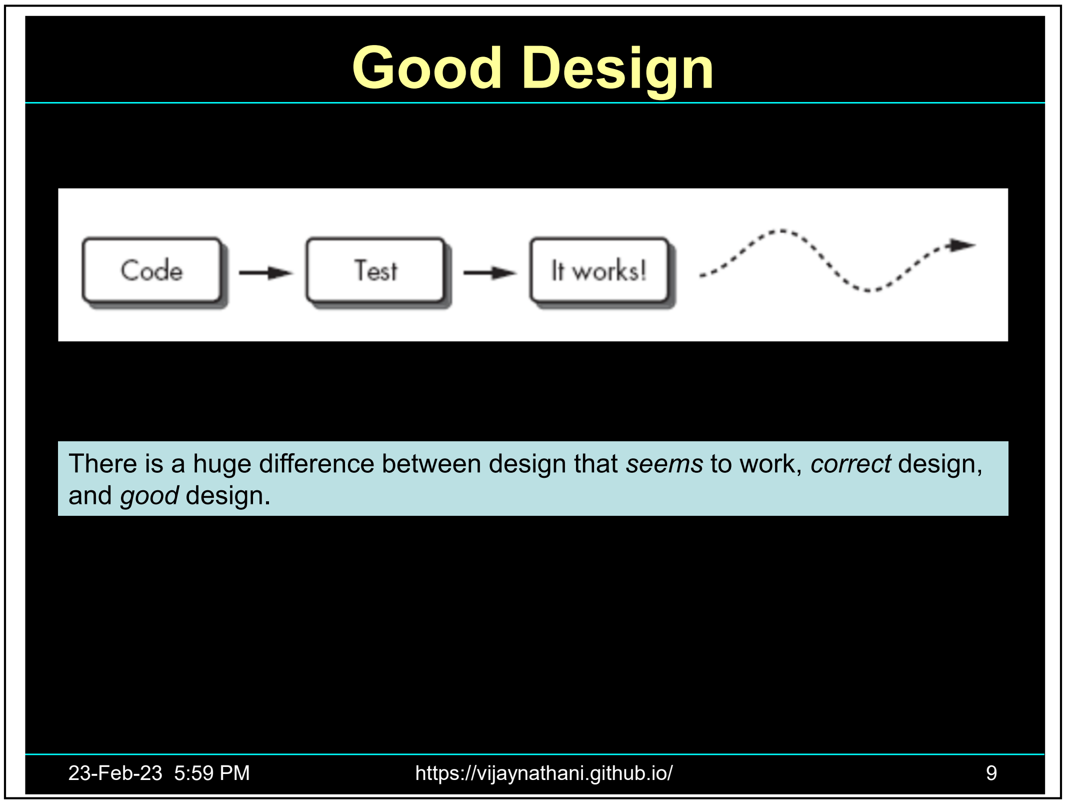


Repositories

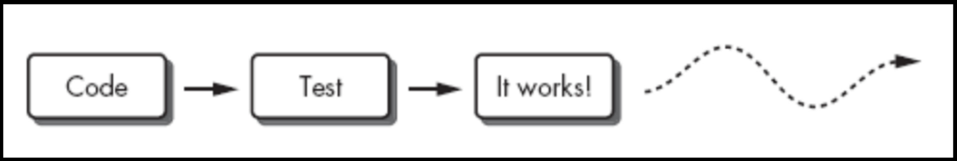
- Manages a collection of objects
- Provides methods for:
- Adding an object
- Finding object or objects
- Deleting objects
- Consists of an interface and an implementation class
- Encapsulates database access mechanism
- Keeps the ORM framework out of the domain model
- Similar to a DAO

```
public interface AccountRepository {

   Account findAccount(String accountId);

   void addAccount(Account account);

}
```

```
public class HibernateAccountRepository implements AccountRepository {

  private HibernateTemplate hibernateTemplate;

  public HibernateAccountRepository(HibernateTemplate template) {
    hibernateTemplate = template;
  }

  public void addAccount(Account account) {
    hibernateTemplate.save(account);
  }

  public Account findAccount(final String accountId) {
    return (Account) DataAccessUtils.uniqueResult(hibernateTemplate
        .findByNamedQueryAndNamedParam(
           "Account.findAccountByAccountId", "accountId",
           accountId));
  }

}
```



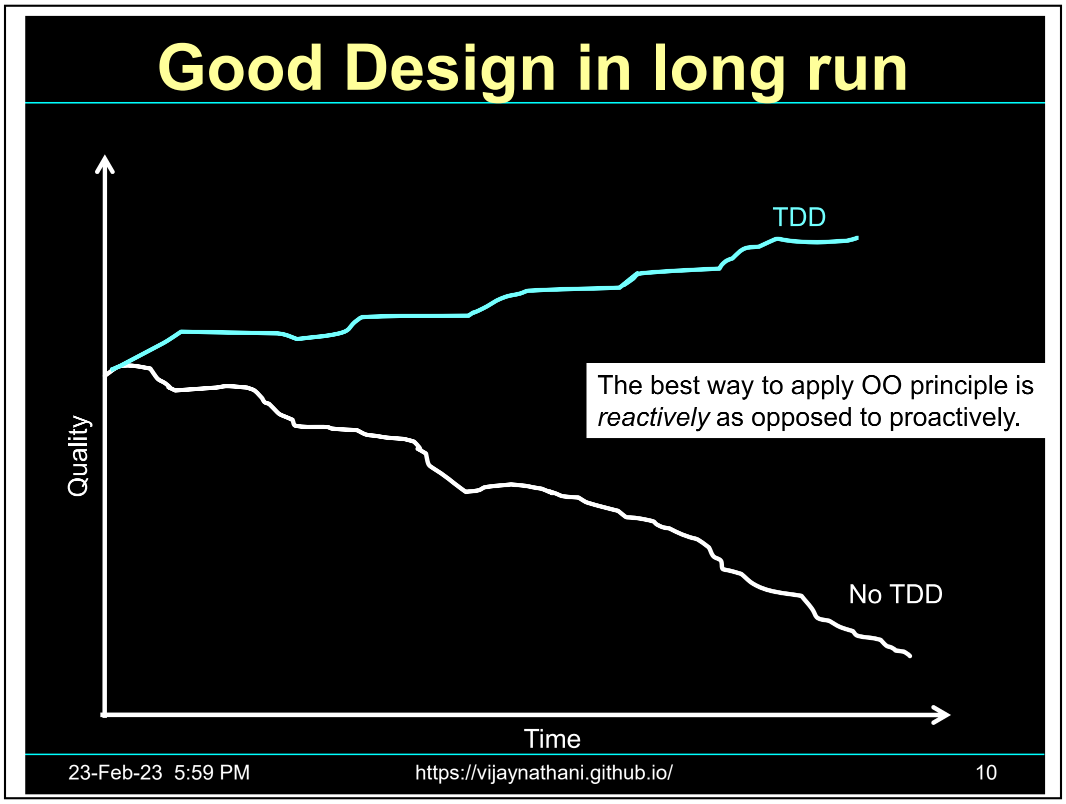
Good Design



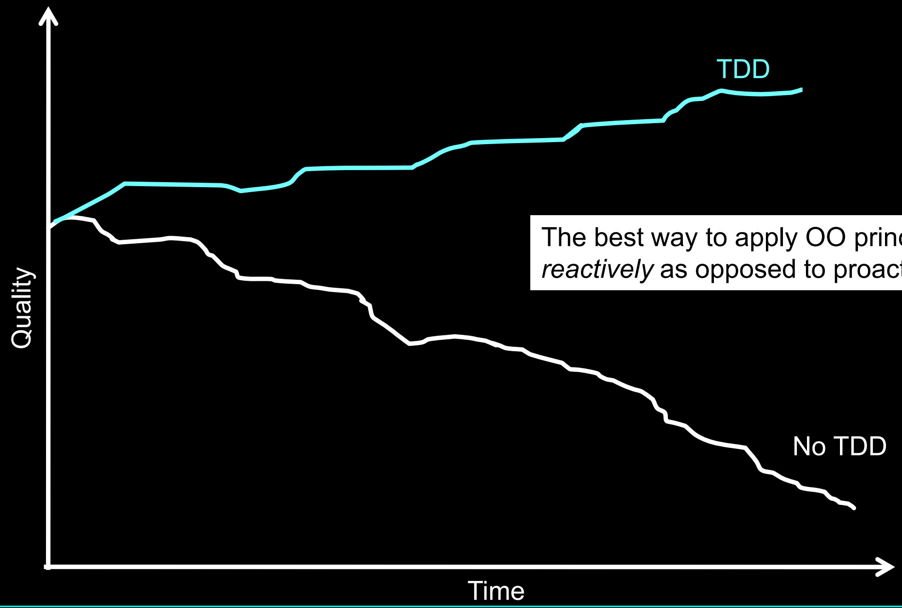
There is a huge difference between design that *seems* to work, *correct* design, and *good* design.

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 9

. Defensive programming is a method of prevention.
Debugging is all about finding a cure.



Good Design in long run

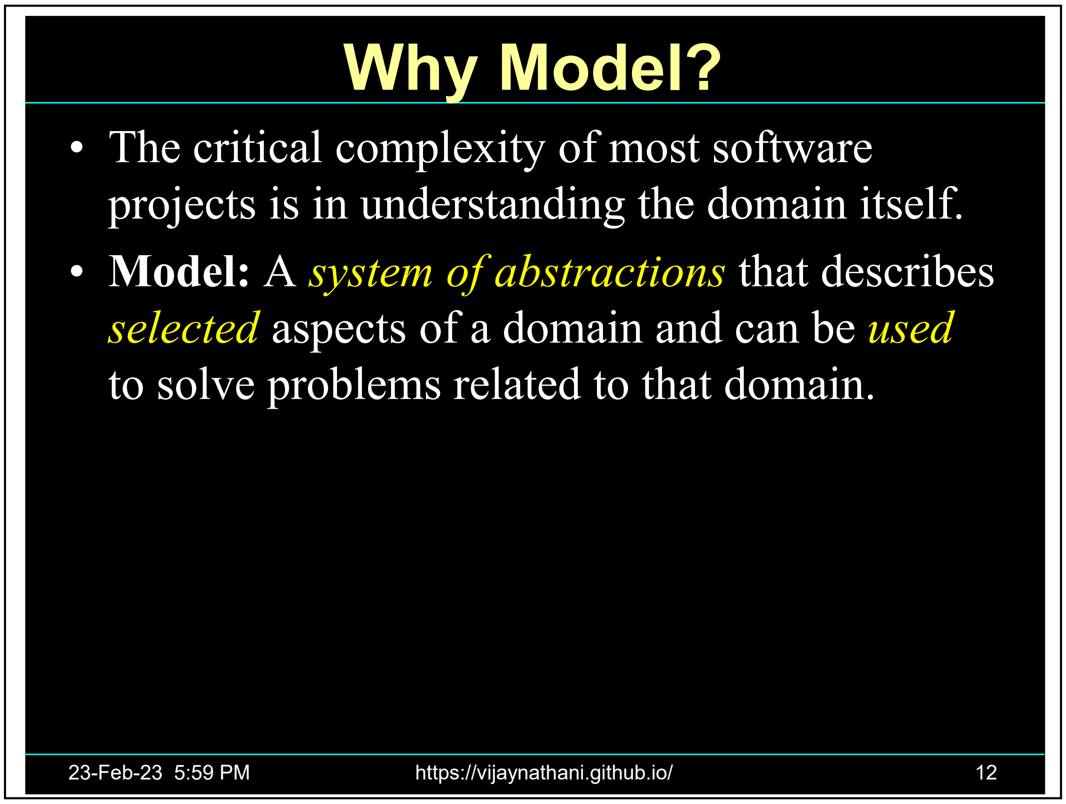


The best way to apply OO principle is *reactively* as opposed to proactively.

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 10

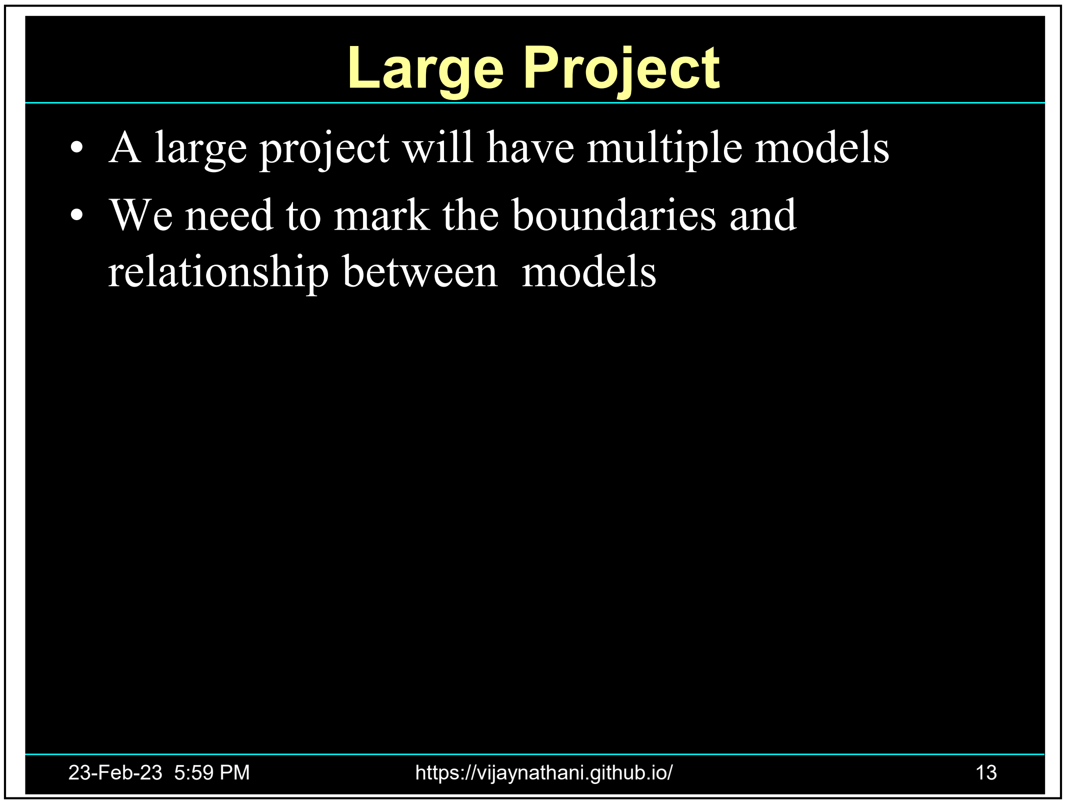
The best way to apply OO principle is *reactively* as opposed to proactively. When you first detect that there is a structural problem with the code, or when you first realize that a module is being impacted by changes in another, *then* you should see whether one or more of these principles can be brought to bear to address the problem

Continuous Integration



Why Model?

- The critical complexity of most software projects is in understanding the domain itself.
- **Model:** A *system of abstractions* that describes *selected* aspects of a domain and can be *used* to solve problems related to that domain.



Large Project

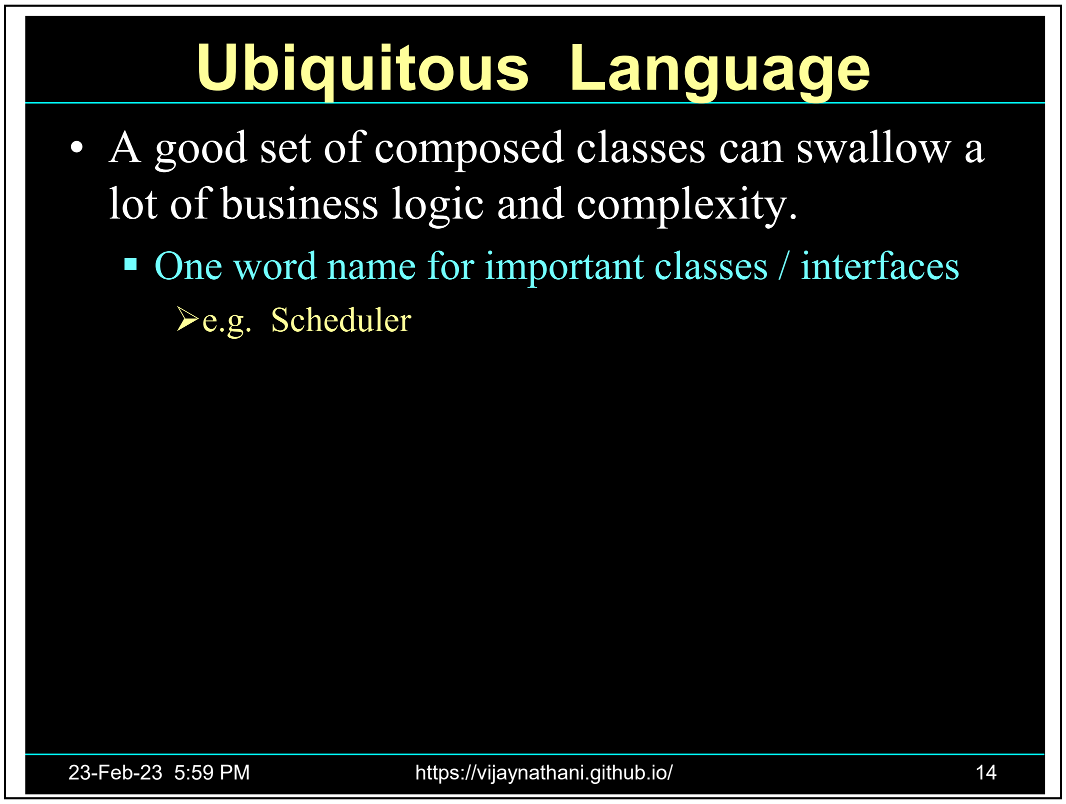
- A large project will have multiple models
- We need to mark the boundaries and relationship between models

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 13

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.

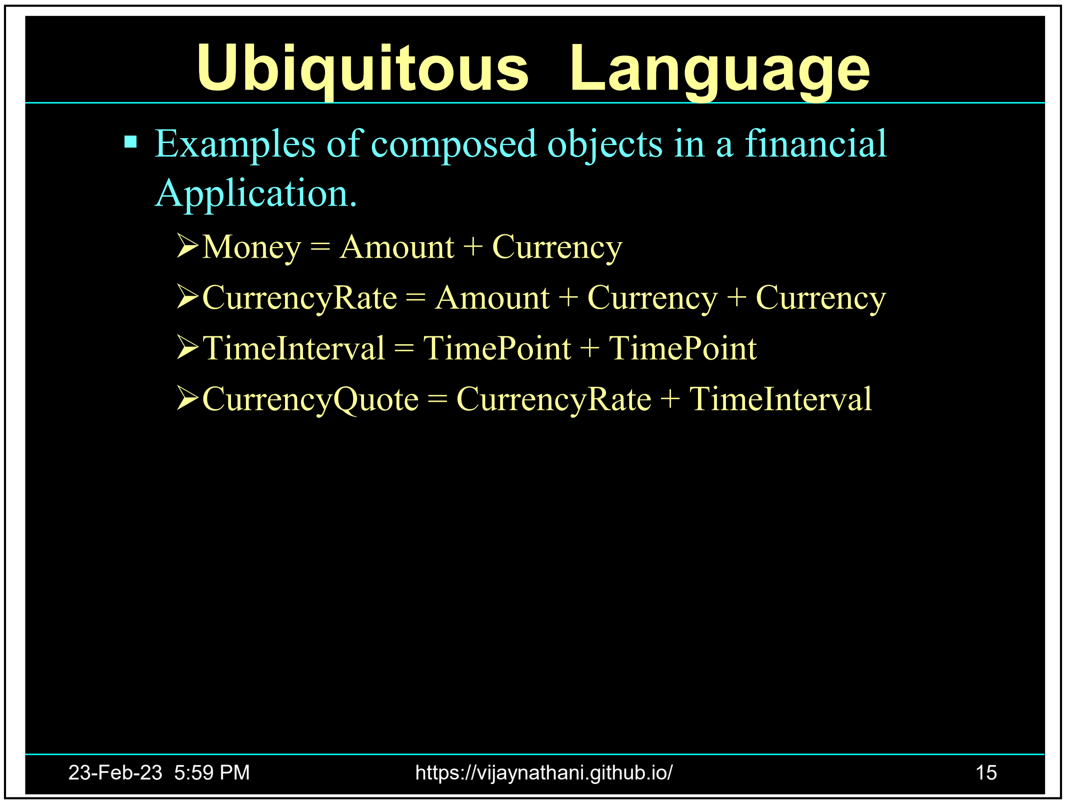
A large project will have multiple models. Combining code based on different models is error-prone and hard to understand.

Explicitely define the context within which a model applies. Set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations like code and database schemas.



Ubiquitous Language

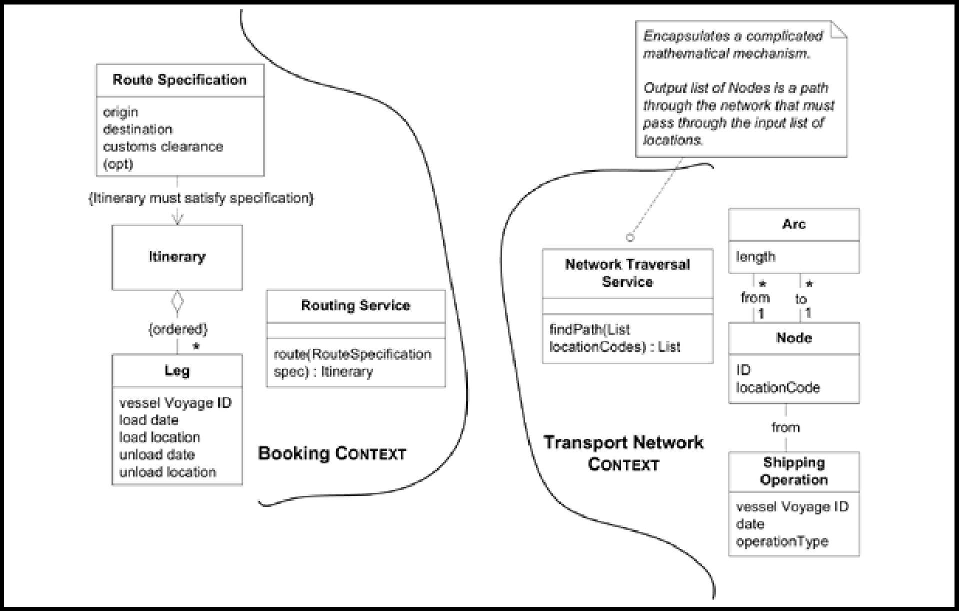
- A good set of composed classes can swallow a lot of business logic and complexity.
  - One word name for important classes / interfaces
    - e.g. Scheduler



Ubiquitous Language

- Examples of composed objects in a financial Application.
  - ➢Money = Amount + Currency
  - ➢CurrencyRate = Amount + Currency + Currency
  - ➢TimeInterval = TimePoint + TimePoint
  - ➢CurrencyQuote = CurrencyRate + TimeInterval

Bounded Context



Route Specification
origin
destination
customs clearance (opt)
{Itinerary must satisfy specification}
Itinerary
{ordered}
*
Leg
vessel Voyage ID
load date
load location
unload date
unload location
Routing Service
route(RouteSpecification spec) : Itinerary
Booking CONTEXT
Encapsulates a complicated mathematical mechanism.
Output list of Nodes is a path through the network that must pass through the input list of locations.
Network Traversal Service
findPath(List locationCodes) : List
Transport Network CONTEXT
Arc
length
from
to
Node
ID
locationCode
Shipping Operation
vessel Voyage ID
date
operationType

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 16

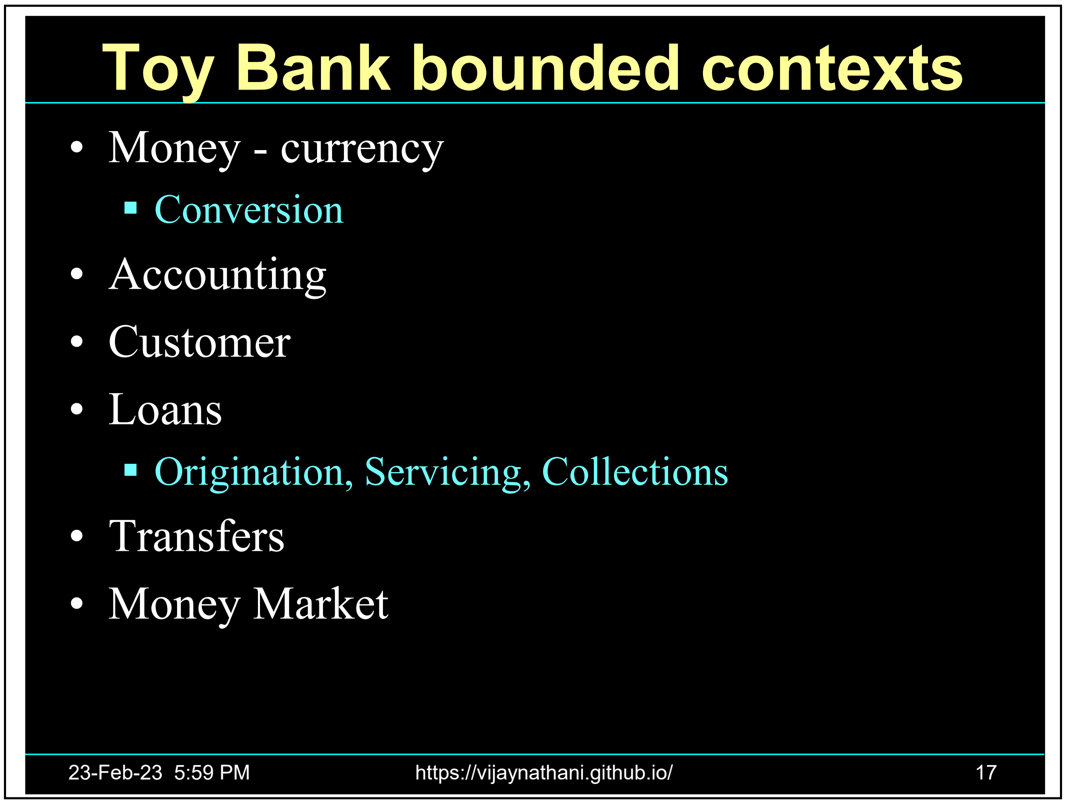
Defiance the context within which a model applies.

Explicitly set boundaries in terms usage within specific parts of the application.

Keep the model strictly consistent within these bounds. Don’t be distracted by issues outside the model.

Bounded Context are not packages.

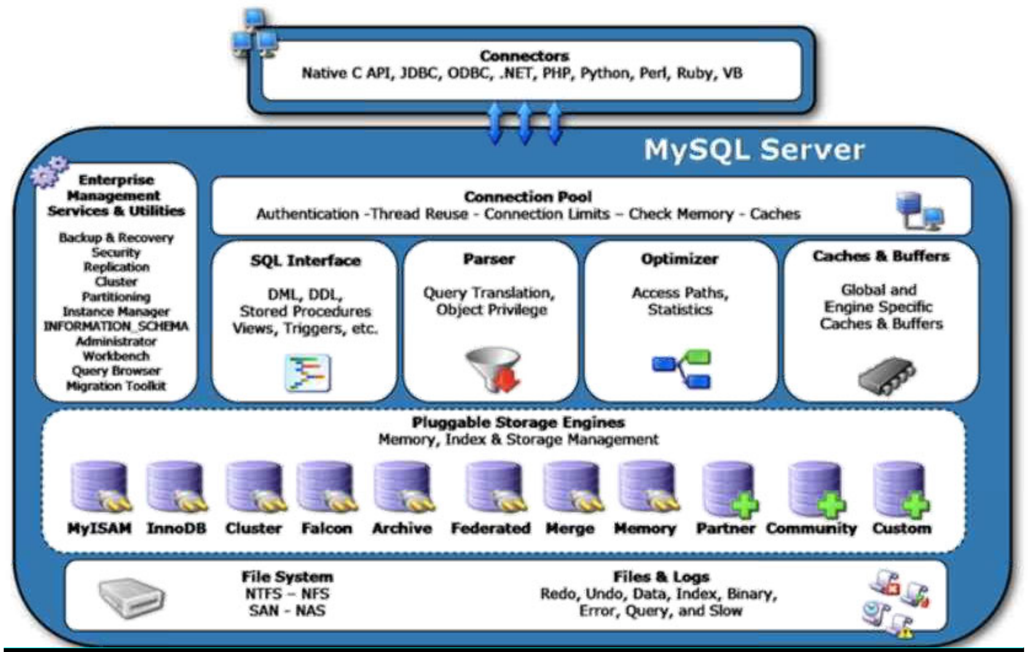
So, what has been gained by defining this BOUNDED CONTEXT? For the teams working in CONTEXT: clarity. Those two teams know they must stay consistent with one model. They make design decisions in that knowledge and watch for fractures. For the teams outside: freedom. They don't have to walk in the gray zone, not using the same model, yet somehow feeling they should.

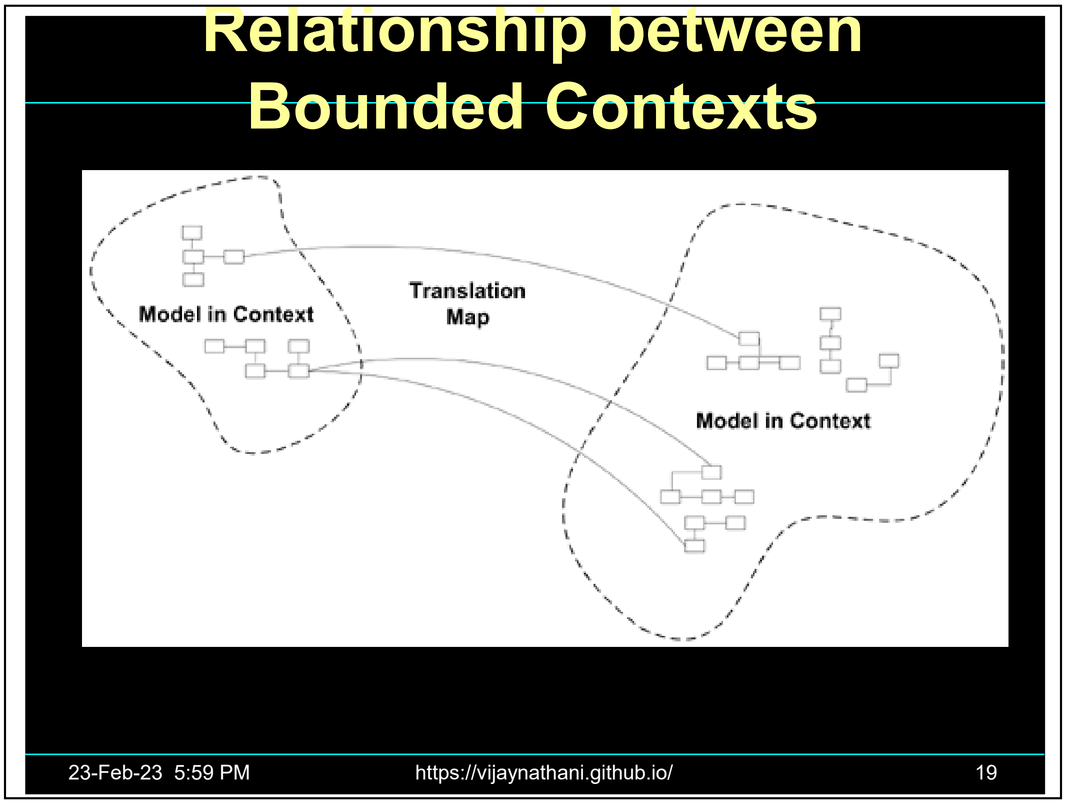


Toy Bank bounded contexts

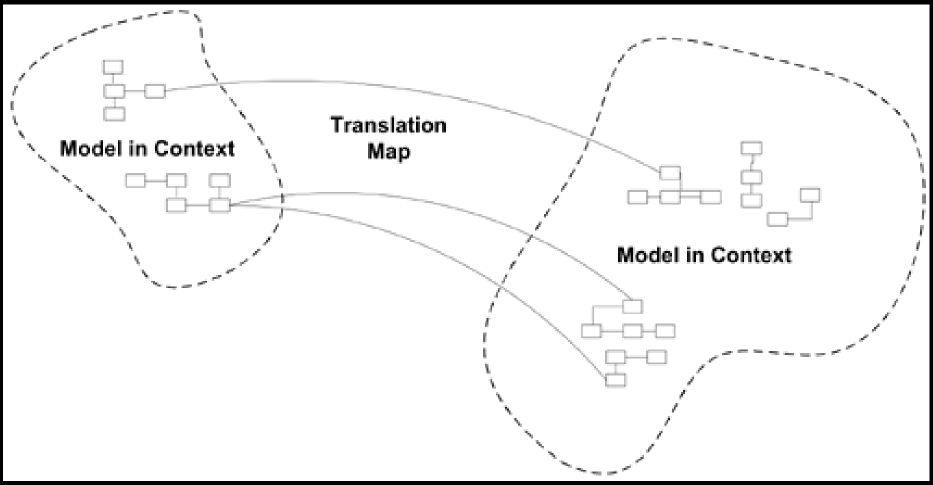
- Money - currency
  - Conversion
- Accounting
- Customer
- Loans
  - Origination, Servicing, Collections
- Transfers
- Money Market

MySQL Architecture


Connectors
Native C API, JDBC, ODBC, .NET, PHP, Python, Perl, Ruby, VB
MySQL Server
Enterprise Management Services & Utilities
Backup & Recovery
Security
Replication
Cluster
Partitioning
Instance Manager
INFORMATION_SCHEMA
Administrator
Workbench
Query Browser
Migration Toolkit
Connection Pool
Authentication -Thread Reuse - Connection Limits – Check Memory - Caches
SQL Interface
DML, DDL, Stored Procedures Views, Triggers, etc.
Parser
Query Translation, Object Privilege
Optimizer
Access Paths, Statistics
Caches & Buffers
Global and Engine Specific Caches & Buffers
Pluggable Storage Engines
Memory, Index & Storage Management
MyISAM
InnoDB
Cluster
Falcon
Archive
Federated
Merge
Memory
Partner
Community
Custom
File System
NTFS – NFS
SAN - NAS
Files & Logs
Redo, Undo, Data, Index, Binary, Error, Query, and Slow



Relationship between Bounded Contexts

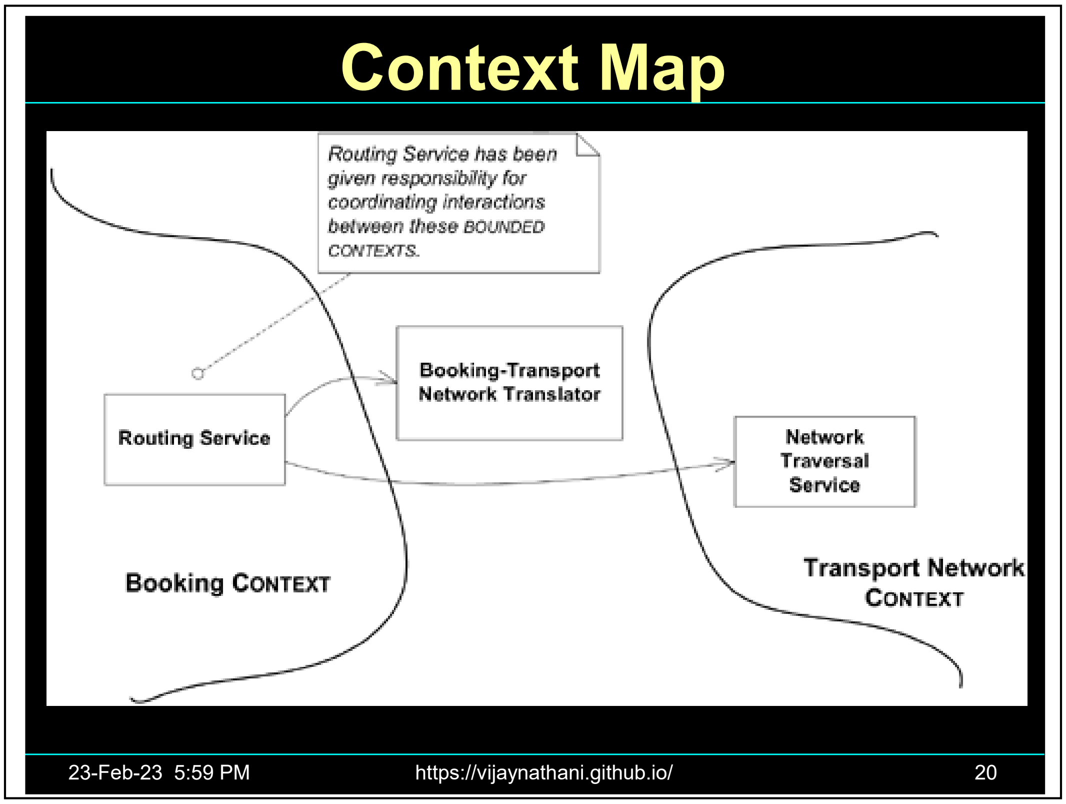


Context Map: Name each Bounded Context.

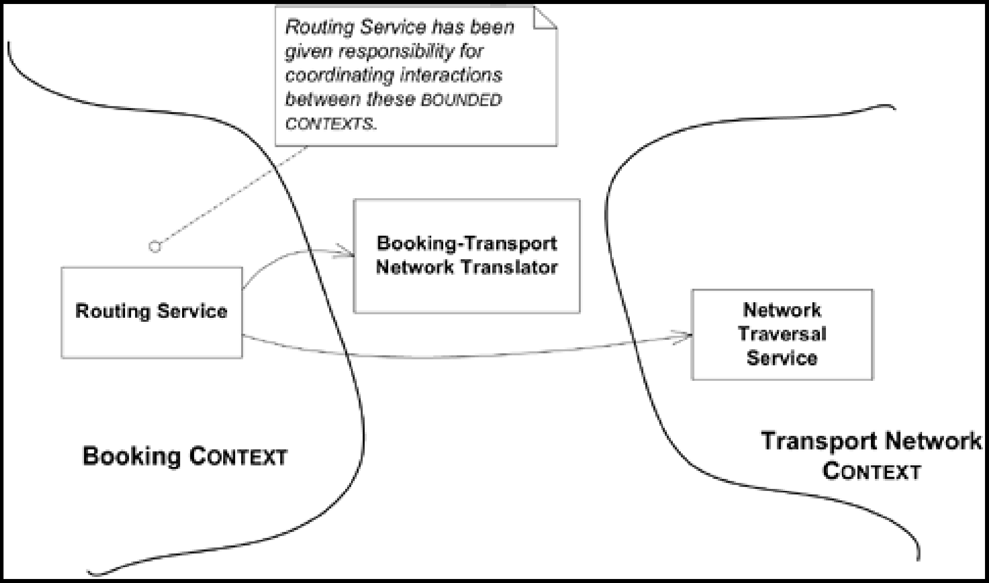
Describe the points of contact between the models

- communication (requires translation)
- sharing (makes assumptions)

Different bounded contexts usually are in different modules, have different development teams, have their own tests,



Context Map


Routing Service has been given responsibility for coordinating interactions between these BOUNDED CONTEXTS.
Booking-Transport Network Translator
Routing Service
Network Traversal Service
Booking CONTEXT
Transport Network CONTEXT

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 20

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.

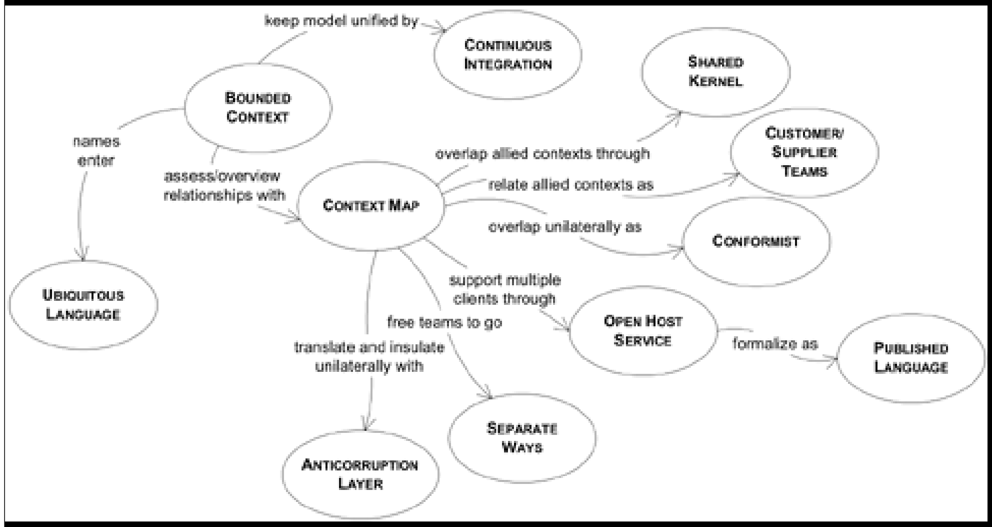
Identify each model in play on the project and define its BOUNDED CONTEXT. This includes the implicit models of non-object-oriented subsystems. Name each BOUNDED CONTEXT, and make the names part of the UBIQUITOUS LANGUAGE.

Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing.

Map what is and not what should be.

One Bounded context should have a single unified Model.

Model Integrity patterns



23-Feb-23 5:59 PM https://vijaynathani.github.io/ 21

Shared Kernel - Some sharing between two Bounded Context.

Customer / Supplier – Consumer / Producer relationship.

Conformist – The consumer slavishly adheres to the model of the Producer. This is used if the consumer cannot impact the Producer.

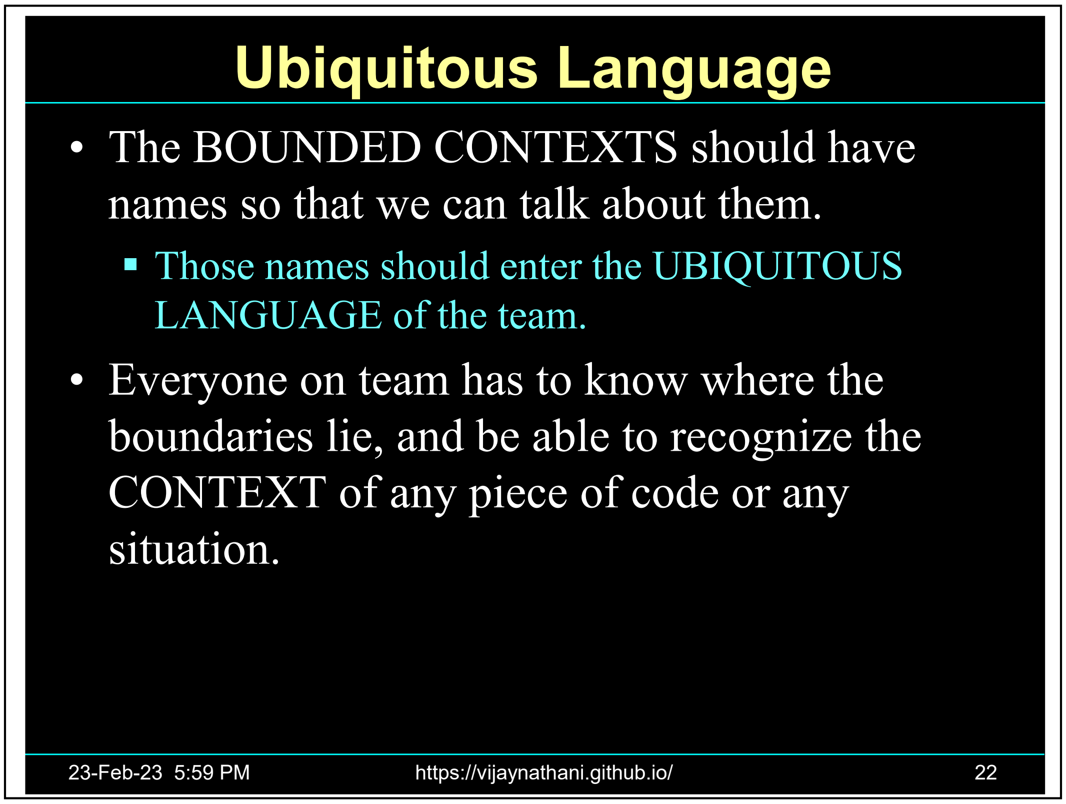
Anti-Corruption Layer – Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions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

Separate Ways - 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 – 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.

Published Language – XMLSchema / DTD for XML

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.



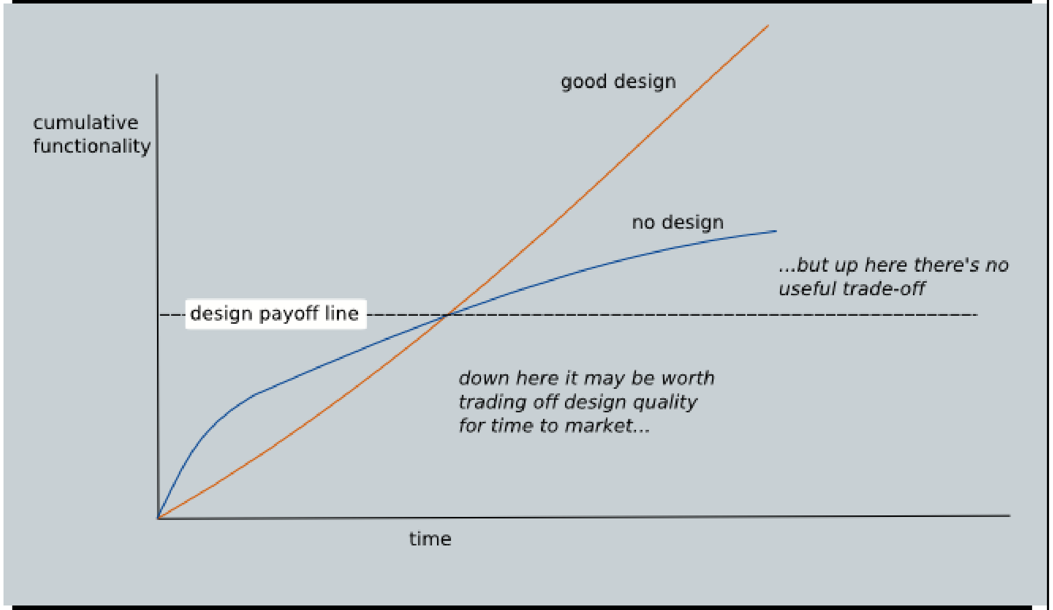
Ubiquitous Language

- The BOUNDED CONTEXTS should have names so that we can talk about them.
  - Those names should enter the UBIQUITOUS LANGUAGE of the team.
- Everyone on team has to know where the boundaries lie, and be able to recognize the CONTEXT of any piece of code or any situation.

23-Feb-23 5:59 PM https://vijaynathani.github.io/ 22

In the 12 practices of Extreme Programming, the role of a SYSTEM METAPHOR could be fulfilled by a UBIQUITOUS LANGUAGE.

Is Good Design worth it?


good design
cumulative functionality
no design
...but up here there's no useful trade-off
design payoff line
down here it may be worth trading off design quality for time to market...
time