Domain-Driven Design

Stephen P Levitt

School of Electrical and Information Engineering
University of the Witwatersrand

2012

Outline

Outline

Outline

What is Domain-Driven Design?

"Domain-driven design flows from the premise that the heart of software development is knowledge of the subject matter and finding useful ways of understanding that subject matter. The complexity that we should be tackling is the complexity of the domain itself – not the technical architecture, not the user interface, not even specific features. This means designing everything around our understanding and conception of the most essential concepts of the business and justifying any other development by how it supports that core." — Eric Evans

Domain Modelling

- DDD centres around domain modelling
- A diagram can represent and communicate the model, as can carefully written code, as can an English sentence
- Domain modelling is not about making as "realistic" a model as possible (even in a domain of tangible things).
- Successful models loosely represent reality for a particular purpose
- Models rigourously organise and selectively abstract knowledge.
- *There are systematic ways of thinking that can be employed to produce effective, insightful models*

Knowledge Crunching

- Domain modeling requires processing (crunching) knowledge
  - In the same way that financial analysts crunch numbers to understand the quarterly performance of a corporation
- While speaking with domain experts, a domain modeler will
  - Try one organising idea (set of concepts) after another
  - Create models, try them out, reject some, transform others
- Success is achieved when the modeler has created a set of abstract concepts that makes sense of all the details provided by the domain experts
  - Domain experts are a critical part of the process
  - Without them, developers tend to create models with concepts that seem naive and shallow to domain experts

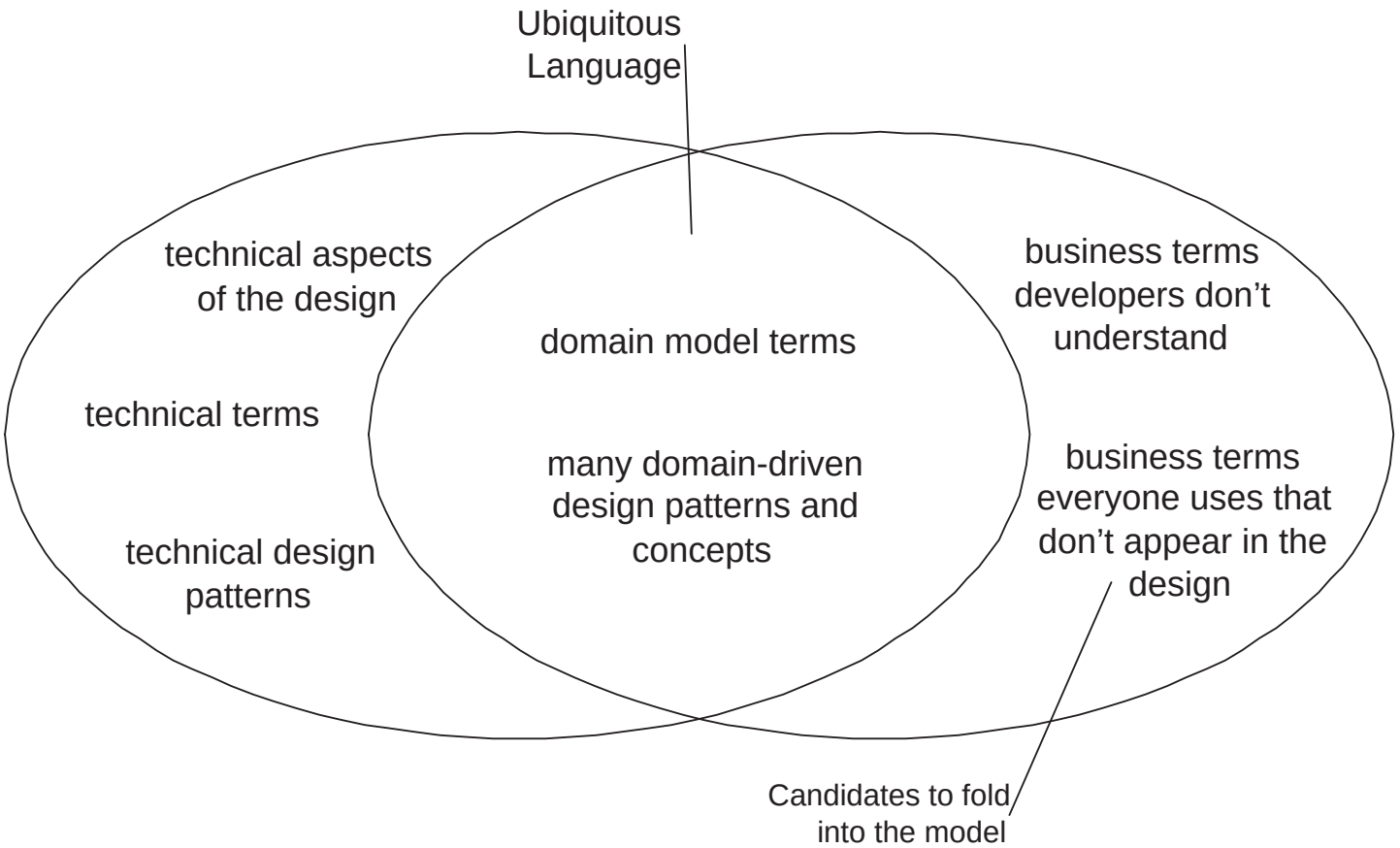
Iterative Development, Continuous Learning

- Development is iterative (as advocated by Evans)
- Knowledge crunching is continuous throughout the lifetime of the project
- Often modelling attempts produced in early iterations are superficial
- Subtle abstractions emerge over time and must be (re)factored into the model.
- To achieve this requires an ongoing close relationship with the domain experts

Communication - Aim for a Ubiquitous Language

- Domain experts use their jargon while developers have their own language for discussing the design and implementation
- The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project)
- Translation is then needed which hampers understanding
- To avoid this create a model which serves as a backbone of a language which is used relentlessly within the team *and* within the code
- With a ubiquitous language, the model is not just a design artifact. It becomes integral to everything the developers and domain experts do together.

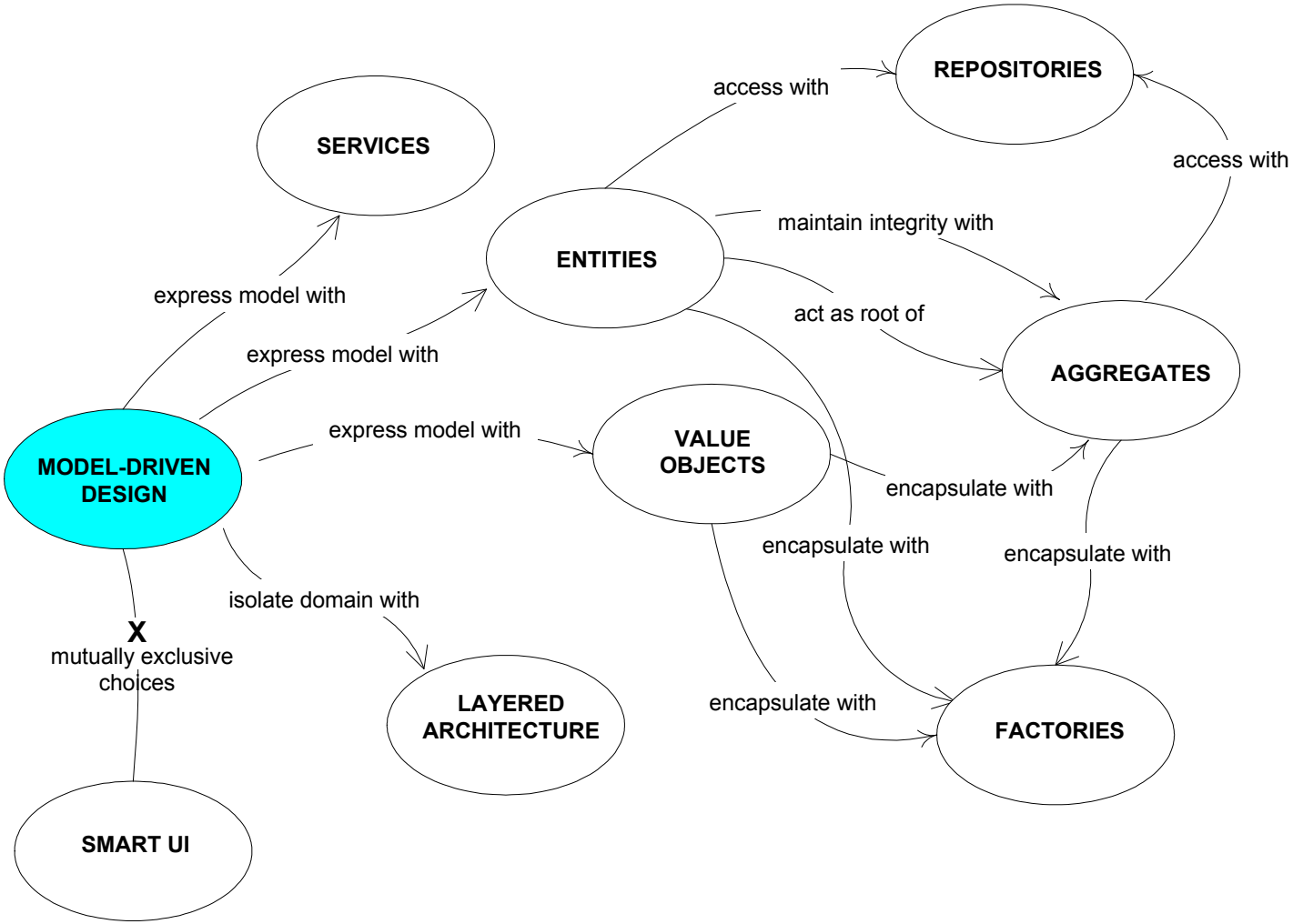
Ubiquitous Language is Cultivated in the Intersection of Jargons



Ubiquitous Language Example

Cargo Routing

Read the two dialogs (scenarios 1 and 2) which present alternative conversations between a developer and a domain expert about the cargo routing domain. Identify subtle (and not so subtle) differences in the conversations. Is the language used rich enough to support a discussion of what the application must do?


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Domain Models and System Design

- Projects with no domain model, in which code is written to fulfill one function after another, will be overwhelmed by complex domains.
- Many complex projects do attempt a domain model but the connection to the design is weak and over time the model becomes increasingly irrelevant.

Separate Analysis and Design

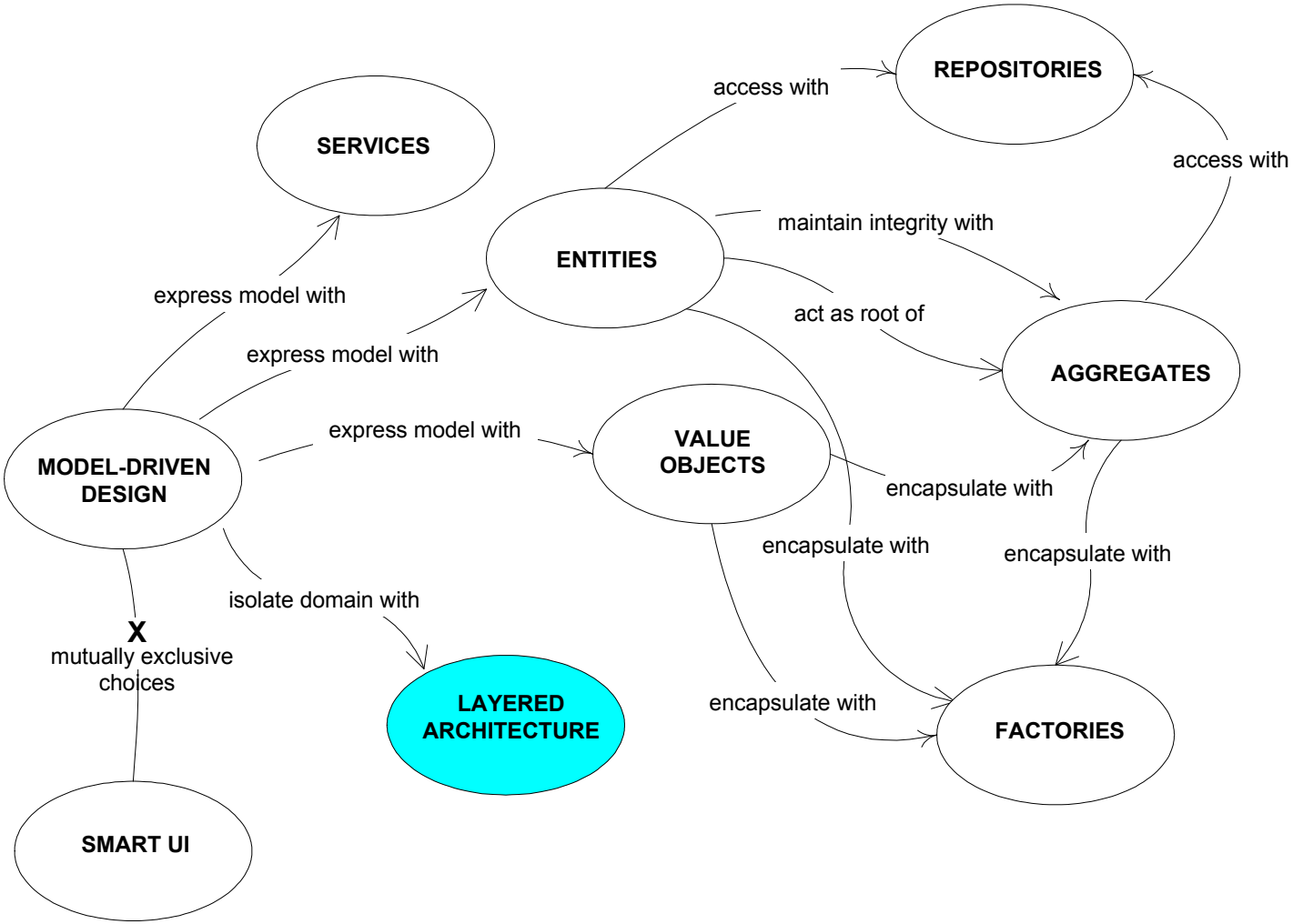
- This may be a *conscious* choice — separate analysis from design:
  - Analysis is seen as tool for understanding only, distinct from design and performed by different people.
  - Business domain concepts are organised without being "contaminated" by implementation concerns.
  - Because of this the model is unlikely to be practical for design needs, resulting in complex mappings. There is no guarantee that the insight gained during analysis will be retained.
  - A pure analysis model falls short of its main goal of understanding the domain as crucial discoveries emerge during design/implementation, and by this time the model is abandoned.

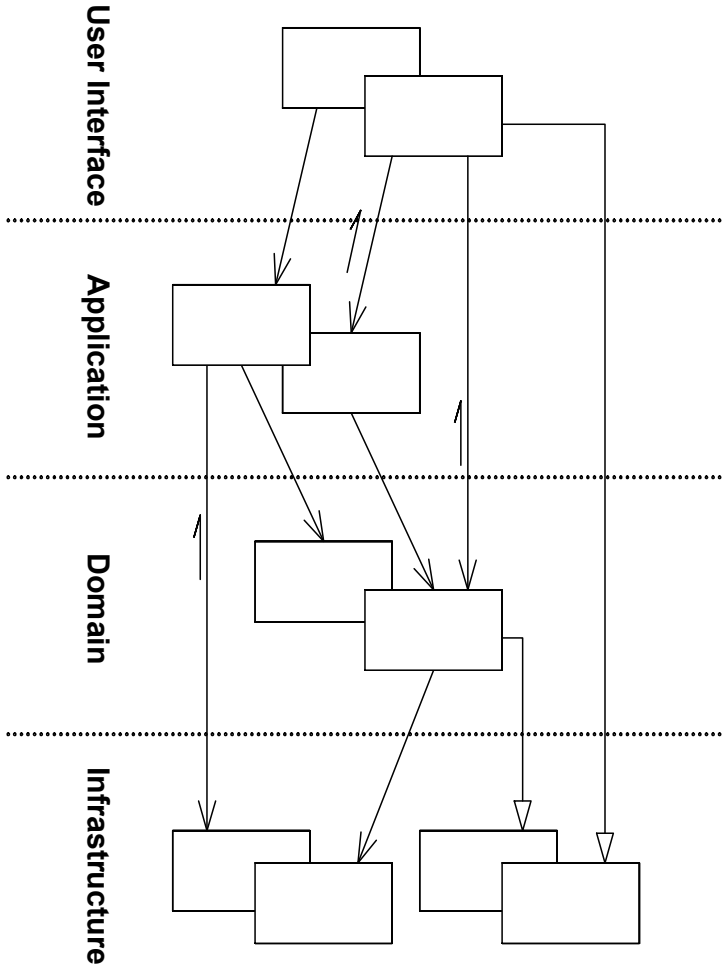
(Domain) Model-Driven Design

- Model-driven design discards the dichotomy of analysis model + design to find a single model serving both purposes
- Many ways of abstracting a domain, many possible designs
- Find a model that
  - Does not come at the cost of weakened analysis, fatally compromised by technical considerations
  - Does not eschew software design principles
- Design a portion of the system to reflect the domain in a very literal way, so the mapping is obvious
- A change in the model implies a change in code and vice-versa

Why Models Matter to Users

- In theory you could present the user with any view of the system regardless of what lies beneath
- In practice the design/implementation model leaks through to the *user model* in subtle ways, creating confusion
- For example, Internet Explorer Favorites and the Scott Adams Meltdown
- When the design is based on a model reflecting the basic concerns of the users and domain experts, the "bones" of the design can be revealed to the user to a greater extent


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI


User Interface
Application
Domain
Infrastructure

Layers

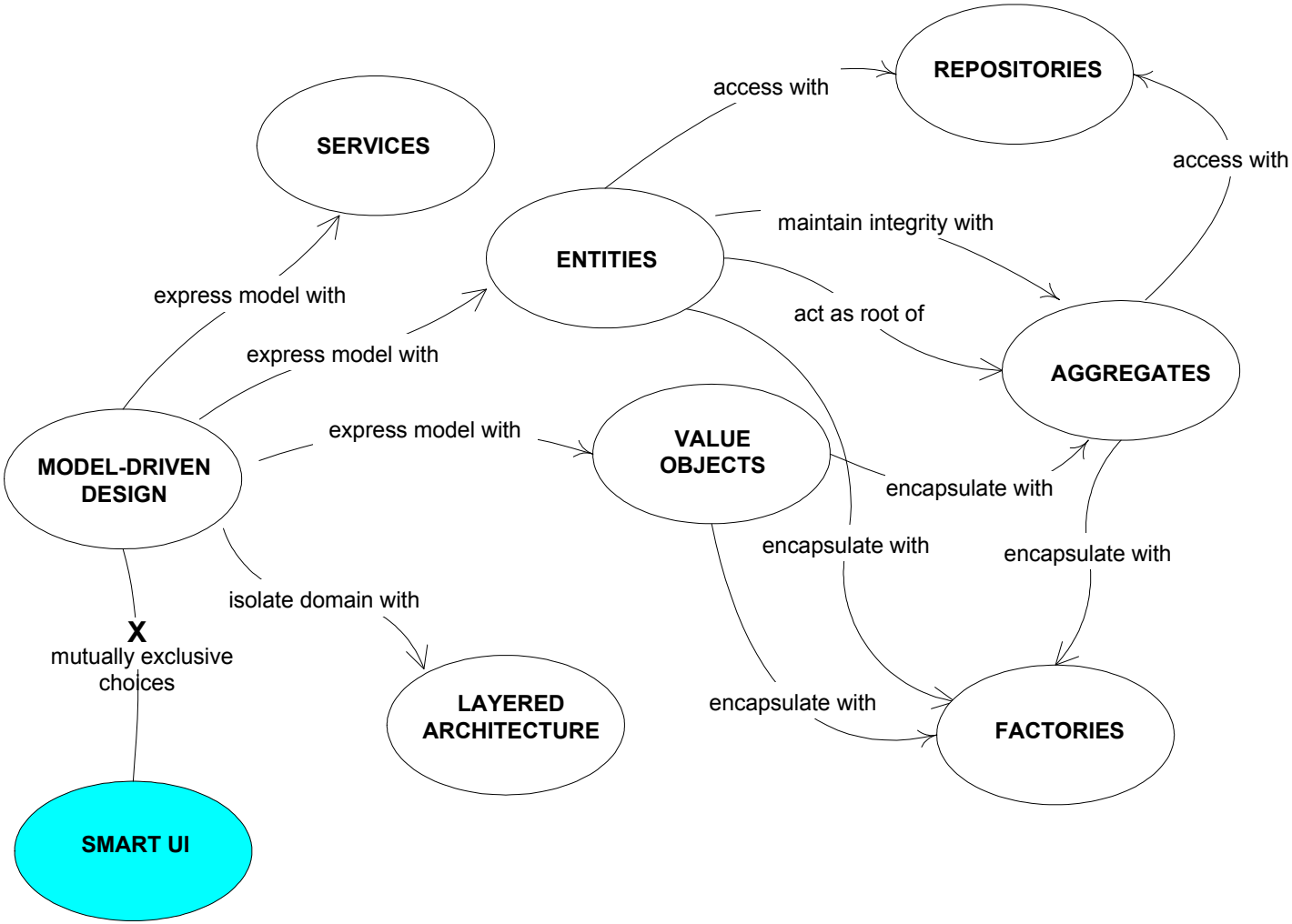
- User Interface (Presentation Layer) – responsible for displaying information and interpreting user commands (user may be a computer system)
- Application Layer – This is a thin layer which *coordinates* the application activity. It does not contain business logic. It does not hold the state of the business objects, but it can hold the state of an application task progress.

Layers

- Domain Layer – This layer contains information about the domain. This is the heart of the business software. The state of business objects is held here. Persistence of the business objects is delegated to the infrastructure layer.
- Infrastructure Layer – Provides generic technical capabilities for the higher layers: message sending for the application, persistence for the domain, drawing widgets for UI etc

Design Principles for Layers

- Partition a complex programme into cohesive layers
- A layer should only depend on a layer or layers below it
- Follow standard patterns to provide loose coupling to upper layers if upward communication is required
  - Callbacks
  - Observer pattern
  - Model-View Controller
- Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application and infrastructure code so that it can focus on expressing the domain.


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Smart UI "Anti-Pattern"

- Put all business logic as small functions into the UI, use a relational database as a shared repository of data, use the most automated UI building tools possible
- Advantages
  - High productivity
  - Suitable for lower skilled developers
  - Relational databases work well providing integration at a data level
  - Easy to redo portions that are not understood when maintaining

Smart UI "Anti-Pattern"

- Disadvantages
  - Growth path is strictly limited to simple applications
  - No reuse of behaviour or abstraction of the business problem, code duplication
  - Integration of applications is difficult except through the database

Associations

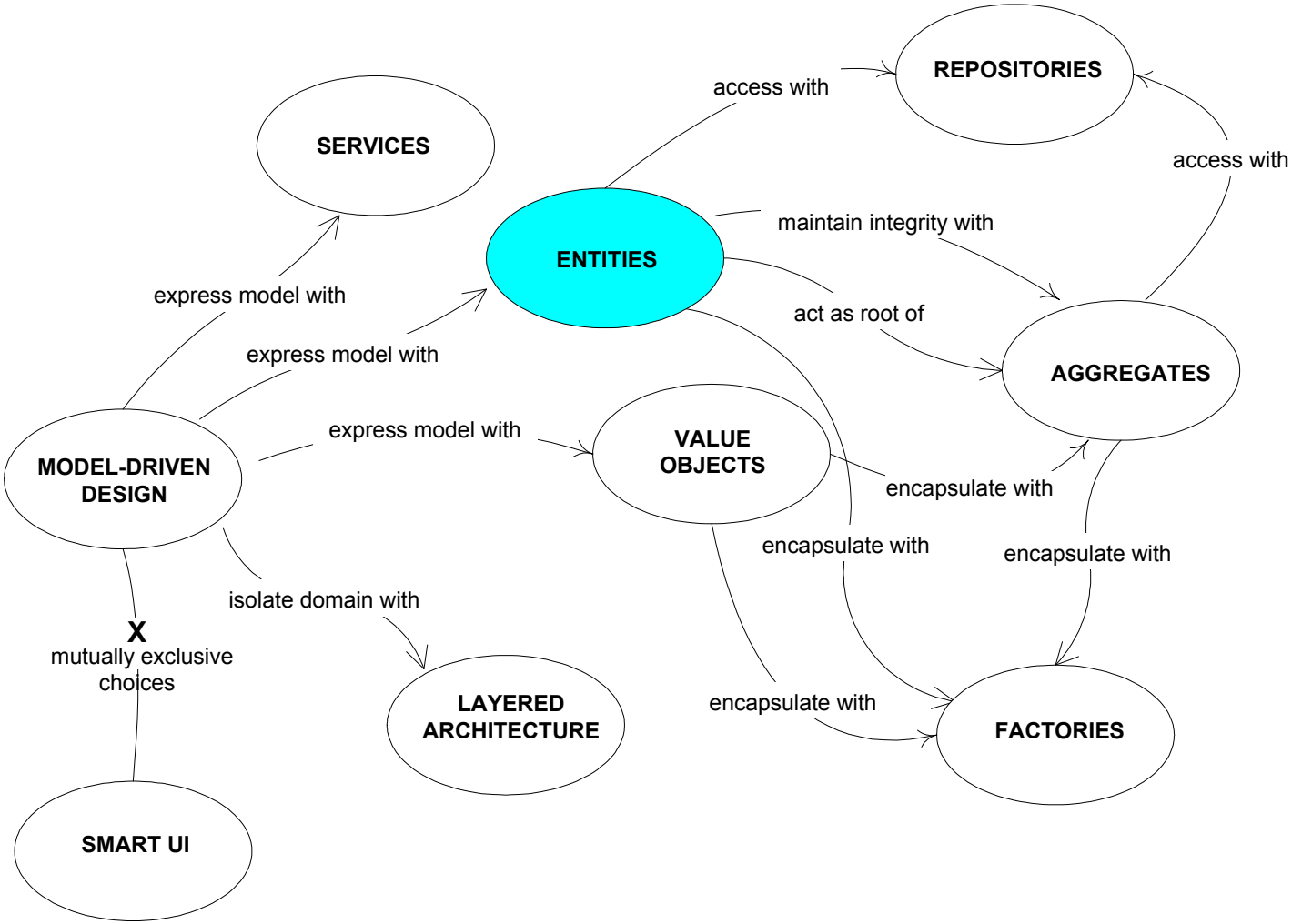
- For every traversable association in the model there is a mechanism in the software with the same properties
- Associations may be implemented with pointers/references or other mechanisms such as a database lookup
- Associations introduce coupling and complexity to the model, especially bi-directional associations
- Many associations are superfluous, constrain by
  - Imposing a traversal direction
  - Adding a qualifier, effectively reducing multiplicity
  - Eliminating non-essential associations

Association Example

Car Ownership

In understanding a particular problem domain you have realised that it is important to model the following *bi-directional* association: A person may own zero or more cars, a car may be owned by zero or one person. You require the ability to get/set a car's owner, as well as, add cars to the collection owned by a person.

Implement this bi-directional association in an OO language of your choice. Take care to ensure that the properties of both classes are synchronised.


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Entities

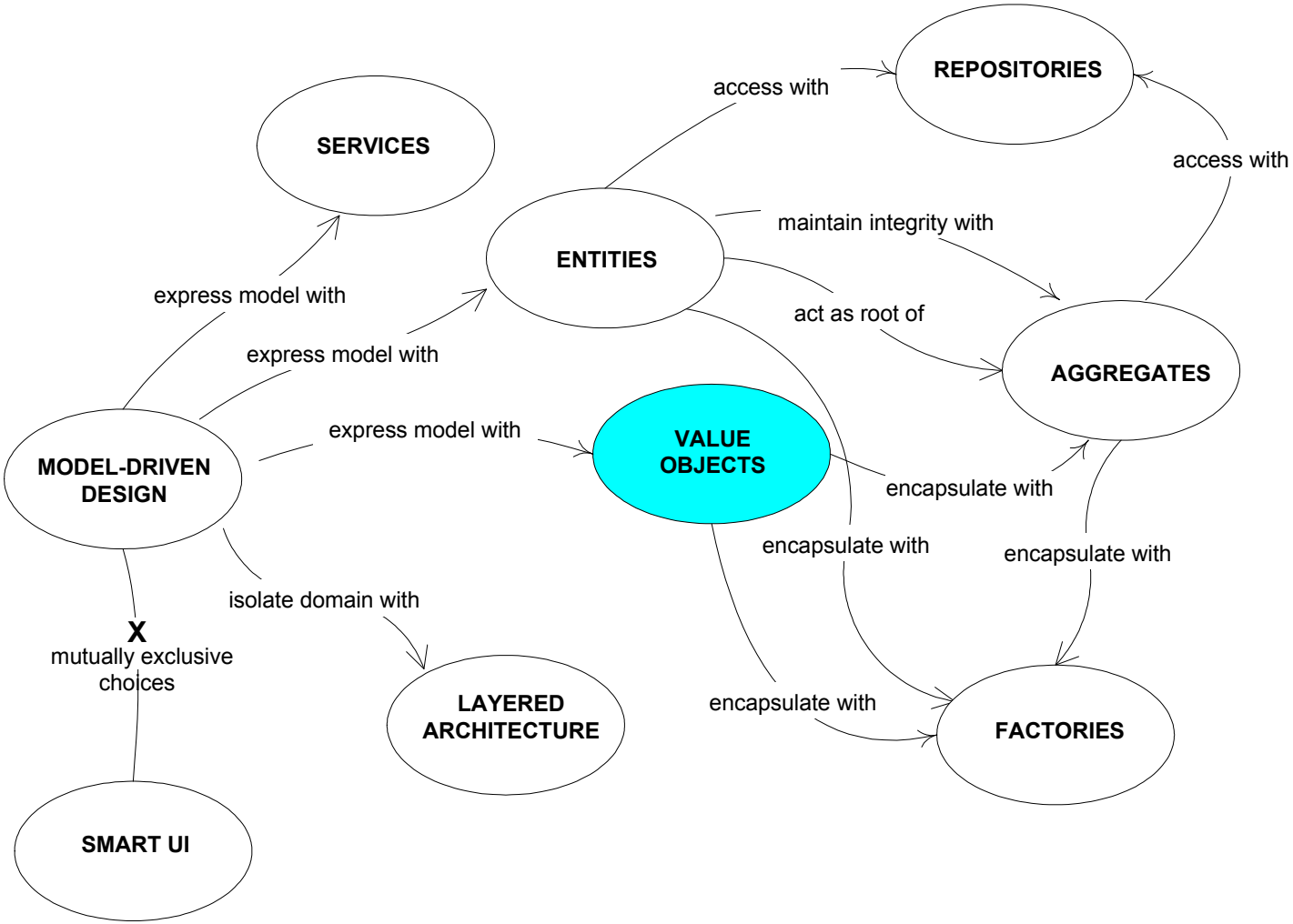
- Many things are defined by their identity and not by any specific attribute
- Consider, a person, they have an identity from birth to death and even beyond. Every attribute of a person can change over their lifetime, yet there identity persists
- In software terms, a Customer object may exist in memory, database tables, XML files, other software systems and may eventually be archived
- Therefore, some objects are not defined primarily by their attributes but by a thread of identity that runs through time and across distinct representations - these are called Entities

Entities

- If an object is distinguished by its identity make this primary to its definition in the model
- Define a means of distinguishing each object regardless of its form or history
- Be alert to requirements that call for matching by attributes
- Define an operation that is guaranteed to produce a unique result for each object
- The means of identification
  - may be significant outside the system, eg. a bank account number
  - may be only used within the system, eg. a computer process ID
- Users may or may not need to see the ID

Modelling Entities

- Focus on modelling of identity — not attributes or behaviour
- Pare an entity's definition down to its most intrinsic characteristics, particularly those that identify it or are commonly used to find or match it
- Look to remove behaviour and attributes into other objects associated with the core entity


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Value Objects

- Entities are conspicuous in the domain model — it is tempting to assign an identity to all domain objects
- However, artificial entities muddle the model and add overhead in tracking and maintaining the identity
- Reserve the special handling required for entities only where necessary
- When you only care about the attributes of a model element classify it as a value object

Money — an Object of Value

```
public class Money implements Comparable{
    private BigInteger amount;
    private Currency currency;

    public Money (long amount, Currency currency) {
        this.amount = BigInteger.valueOf(amount * 100);
        this.currency = currency;
    }
    public double amount() {
        return amount.doubleValue() / 100;
    }
    public Currency currency() {
        return currency;
    }
```

Value Object Example

Modelling Money

For the Money class write methods which add and subtract money.

Think carefully about what it means to model a *value*.

Money Addition and Subtraction

```
class Money...
 public Money add (Money arg) {
  assertSameCurrencyAs(arg);
  return new Money (amount.add(arg.amount), currency);
  }

 public Money subtract (Money arg) {
   return this.add(arg.negate()); }

 void assertSameCurrencyAs(Money arg) {
   Assert.equals("money math mismatch",
                 currency, arg.currency); }

 public Money negate() {
   return new Money (amount.negate(), currency); }
```

Money's Private Constructor

```
class Money...
  private Money (BigInteger amountInCents,
        Currency currency) {
      Assert.notNull(amountInCents);
      Assert.notNull(currency);
      this.amount = amountInCents;
      this.currency = currency;
  }
```

Designing Value Objects

- Treat the value object as immutable in most cases
  - No modifiers or 'set' functions
  - Potential modifiers produce a *new* value object with the modification applied
- This greatly simplifies development as there are no side effects (referencing a changed object)
- Can lead to performance issues, see the *flyweight* pattern
- Immutability and value-like behaviour can be enforced in some languages which helps communicate the design decision

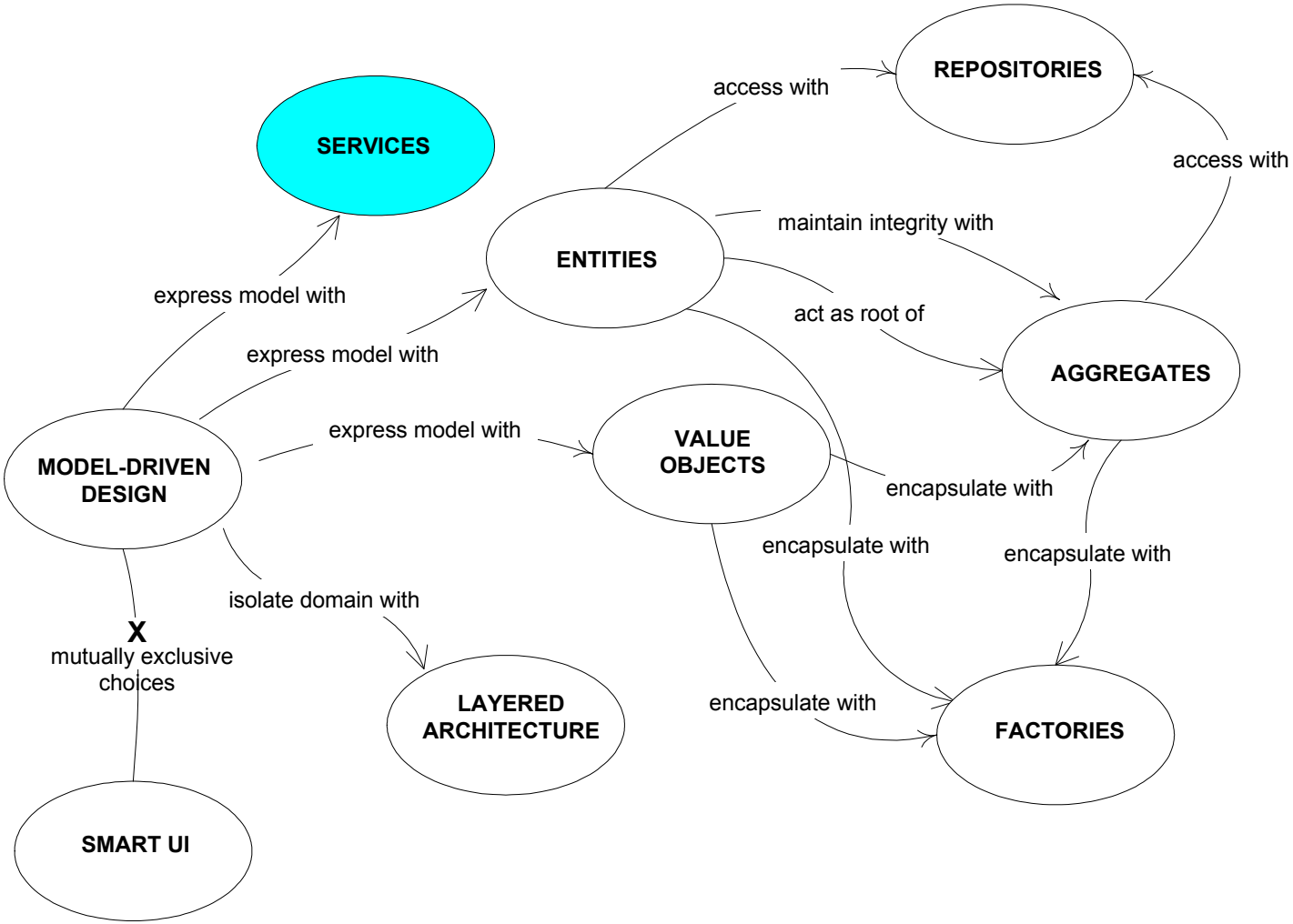
Immutable Value Objects are Important

"Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code."

— The Java Tutorials, Sun Microsystems

" I believe that immutable objects are the way of the future in C#."

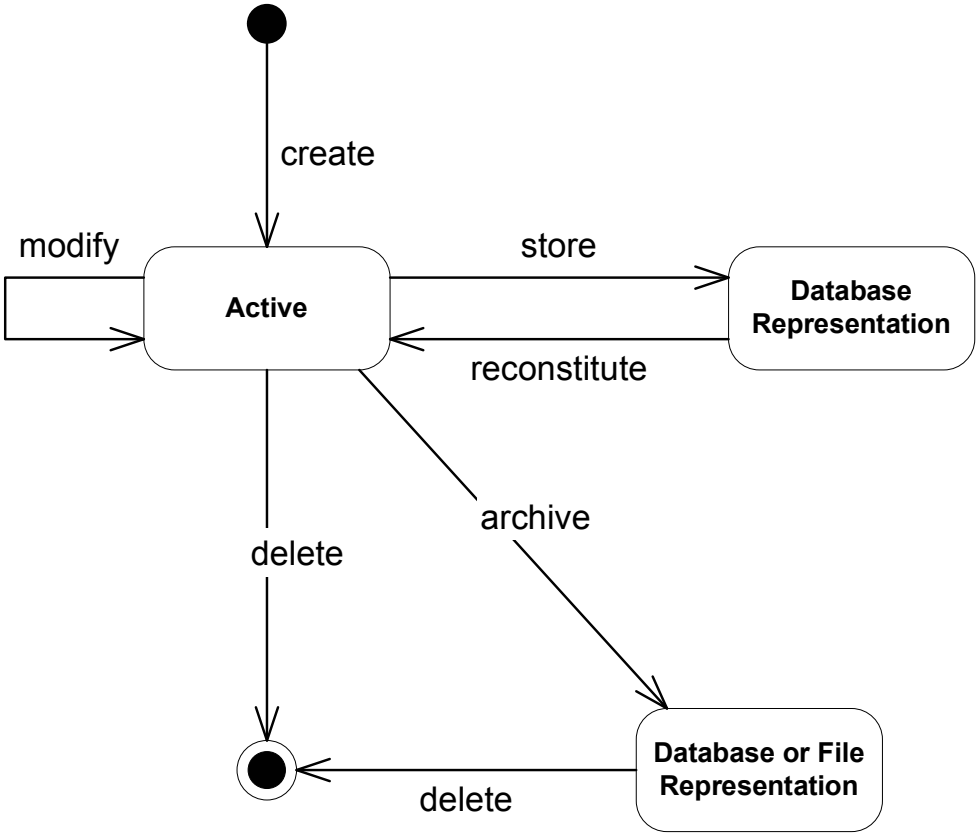
— Eric Lippert, senior software design engineer at Microsoft, November 2007


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Services

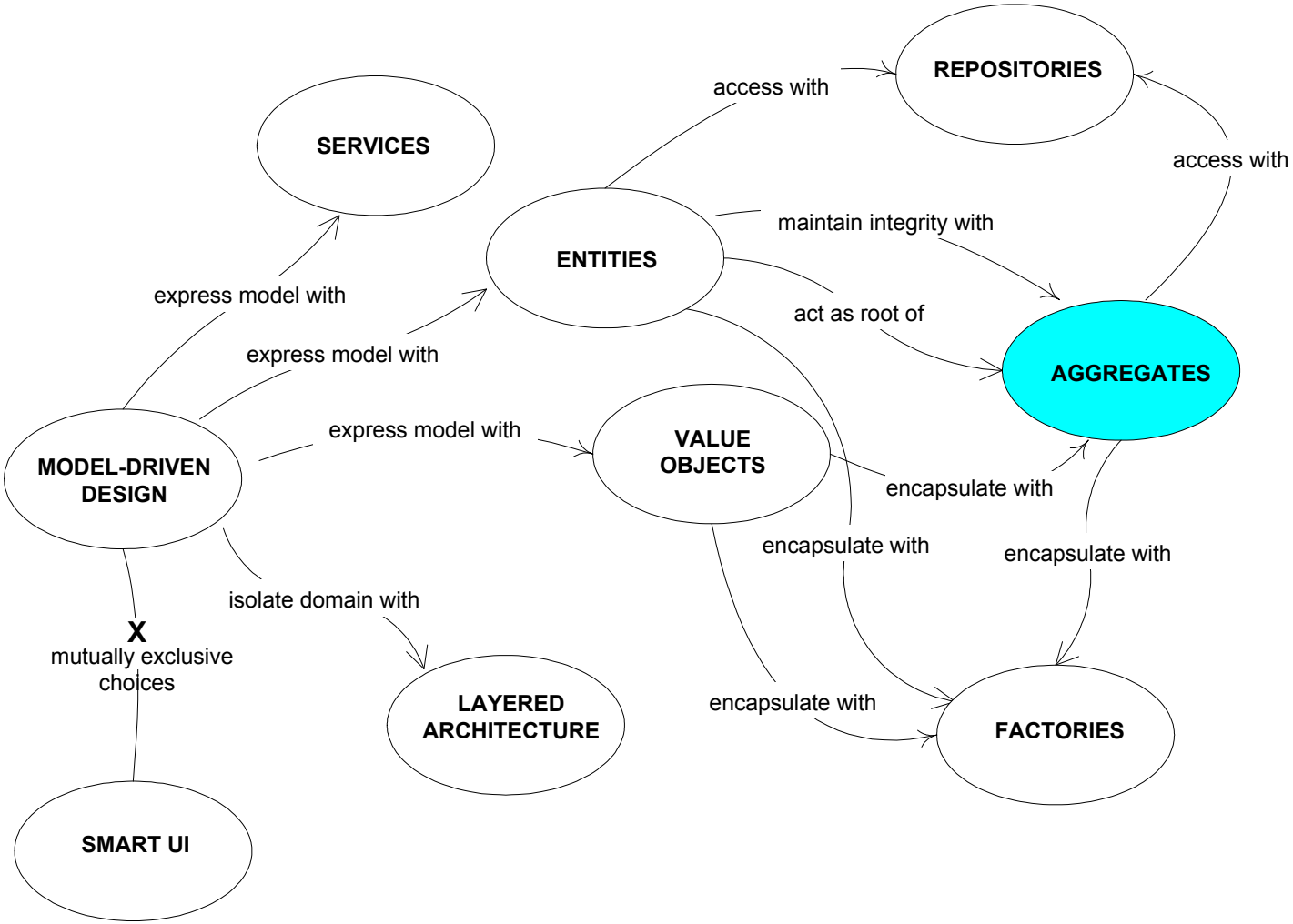
- Some concepts or operations from the domain model are not naturally modelled as objects
- These operations often involve many domain objects and co-ordinate their behaviour
- Forcing such operations onto domain objects dilutes their focus and makes them difficult to understand and refactor.
- When a significant operation is not the natural responsibility of an Entity or Value object, model it as a Service
- Declare services as standalone interfaces defined in terms of other elements of the domain model.
- Make services *stateless* although they may depend on globally accessible information

The Life Cycle of a Domain Object



The Life Cycle of a Domain Object

- Every object has a lifecycle
- Simple objects:
  - are relatively transient
  - easily constructed through direct calls
  - used and then abandoned to the garbage collector
- More complex objects:
  - have longer lives not all of which are spent in memory
  - have complex interdependencies with other objects
  - require invariants to be maintained among these objects
- Complex objects provide challenges to model-driven design in
  - maintaining integrity
  - swamping the model with life-cycle management issues


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

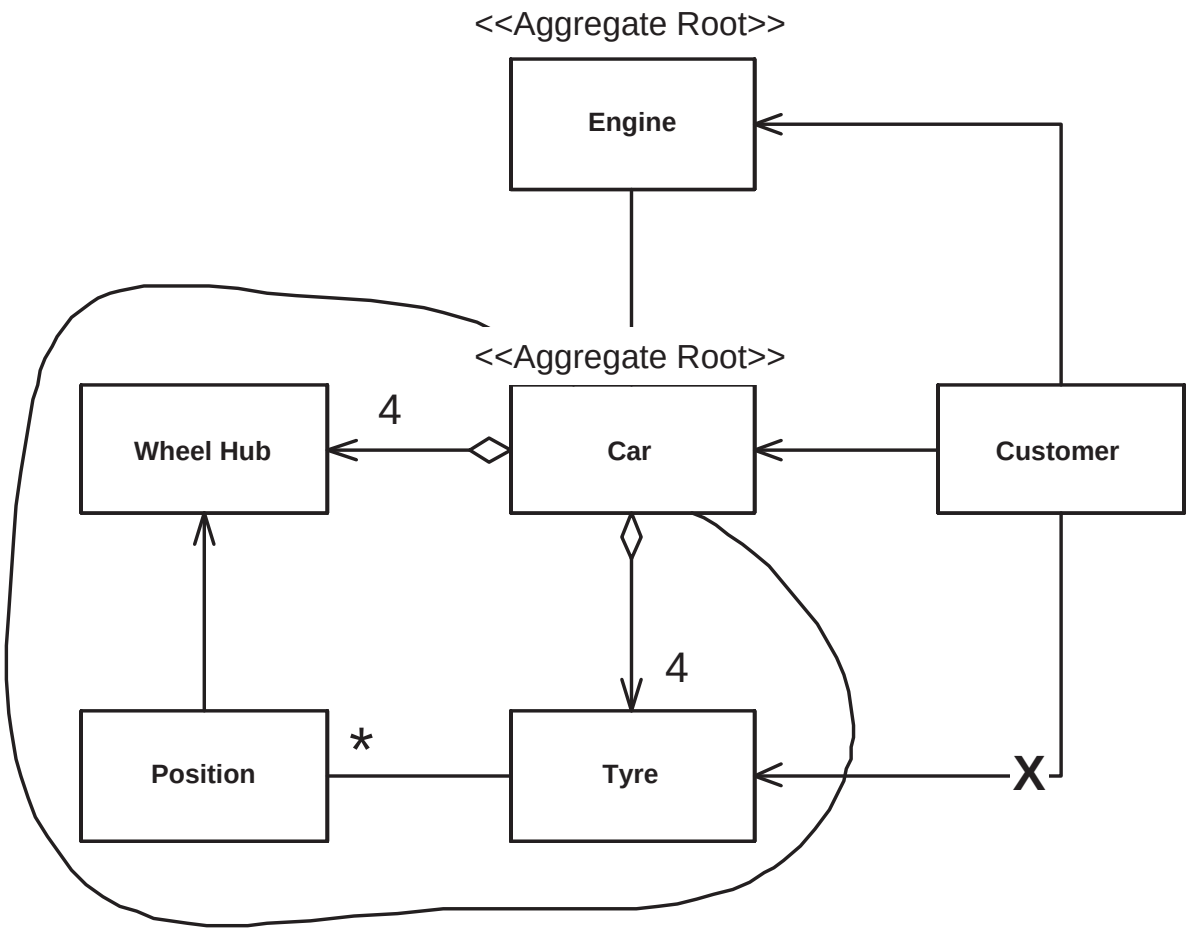
Aggregates — A Motivation

- Constraining associations where appropriate simplifies traversal and limits the explosion of relationships
- Nevertheless, complex objects exist which are interconnected with others
- This web of relationships give no clear limit to the potential effect of a change
- The problem is acute in systems offering concurrent access to the same objects by multiple clients
  - simultaneous changes to interdependent objects have to be avoided because invariants need to be maintained among these objects
  - cautious locking schemes cause multiple users to interfere pointlessly with each making the system unusable

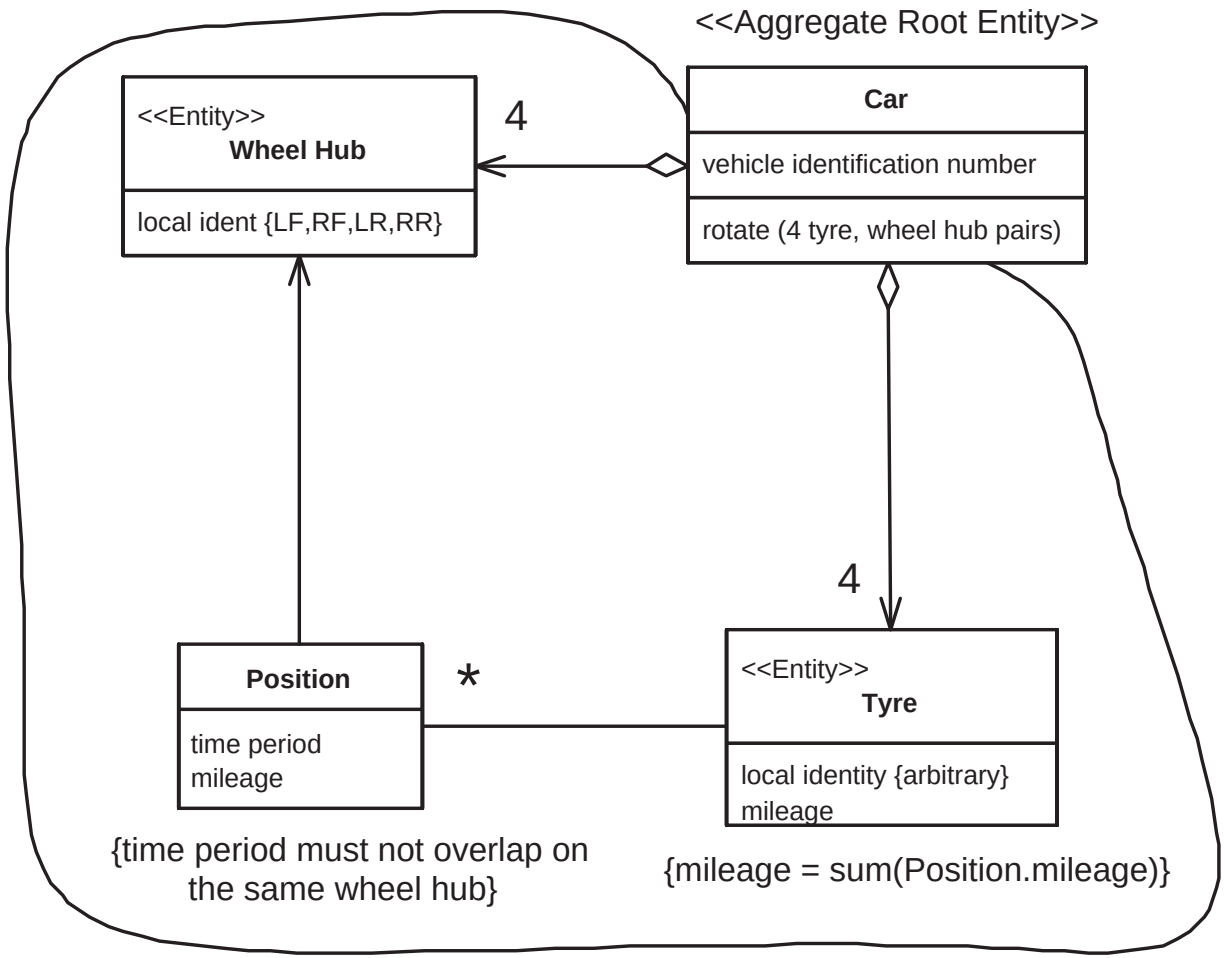
Aggregates

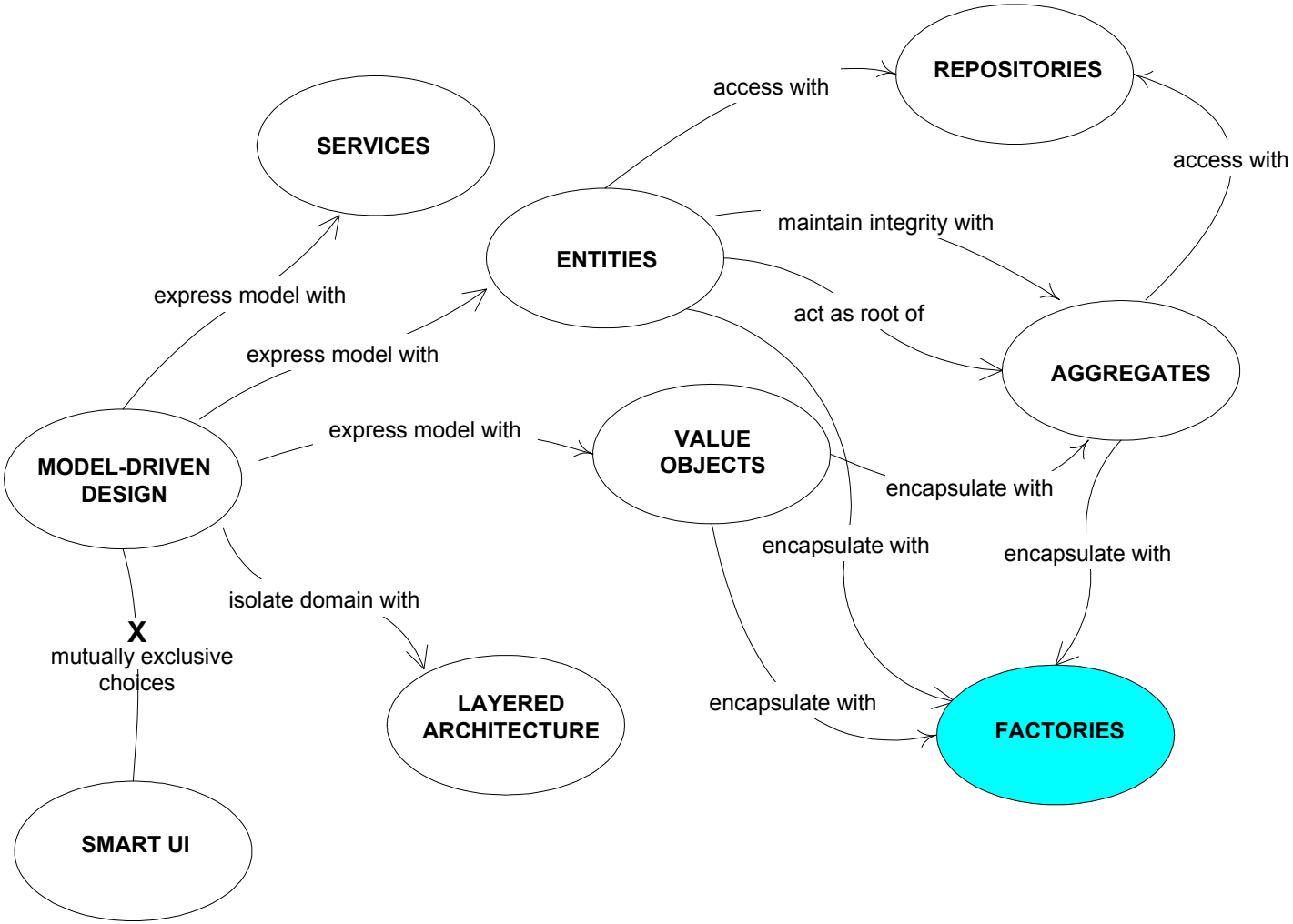
- How do we know where an object made up of other objects begins and ends?
- An aggregate is a cluster of associated objects that are treated as a unit for the purpose of data changes
- Each aggregate has a *root* and a *boundary*
- The root is a single specific entity; the boundary defines what is inside the aggregate
- The root is the only member of the aggregate that is exposed, i.e objects outside the boundary can hold references to it
- Within the boundary, other entities have local identity

Car Aggregate



Car Aggregate Invariants




SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

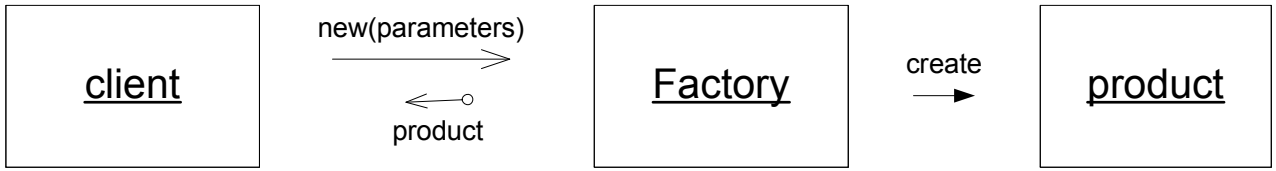
The Need for Factories

- Consider a car
  - Cars which are able to assemble themselves would be far more complex, and probably less efficient and reliable
  - Cars are never assembled and driven at the same time so there is no value in combining these mechanisms
- Likewise, assembling a complex, compound object is a task best separated from whatever job the object will do when constructed.
- Shifting responsibility to the client leads to worse problems — coupling the client to the internals of the domain objects it uses, and blurring its responsibility

Factories

- Complex object creation is a responsibility of the domain layer, yet the task does not belong to objects that express the model
- Often, object creation and assembly have no special meaning for the domain and are a necessity of the implementation — elements are added to the design which do not correspond to to anything in the domain model, but are still part of the domain layer's responsibility
- Sometimes object construction and assembly correspond to domain milestones, such as "open a bank account"

Factories

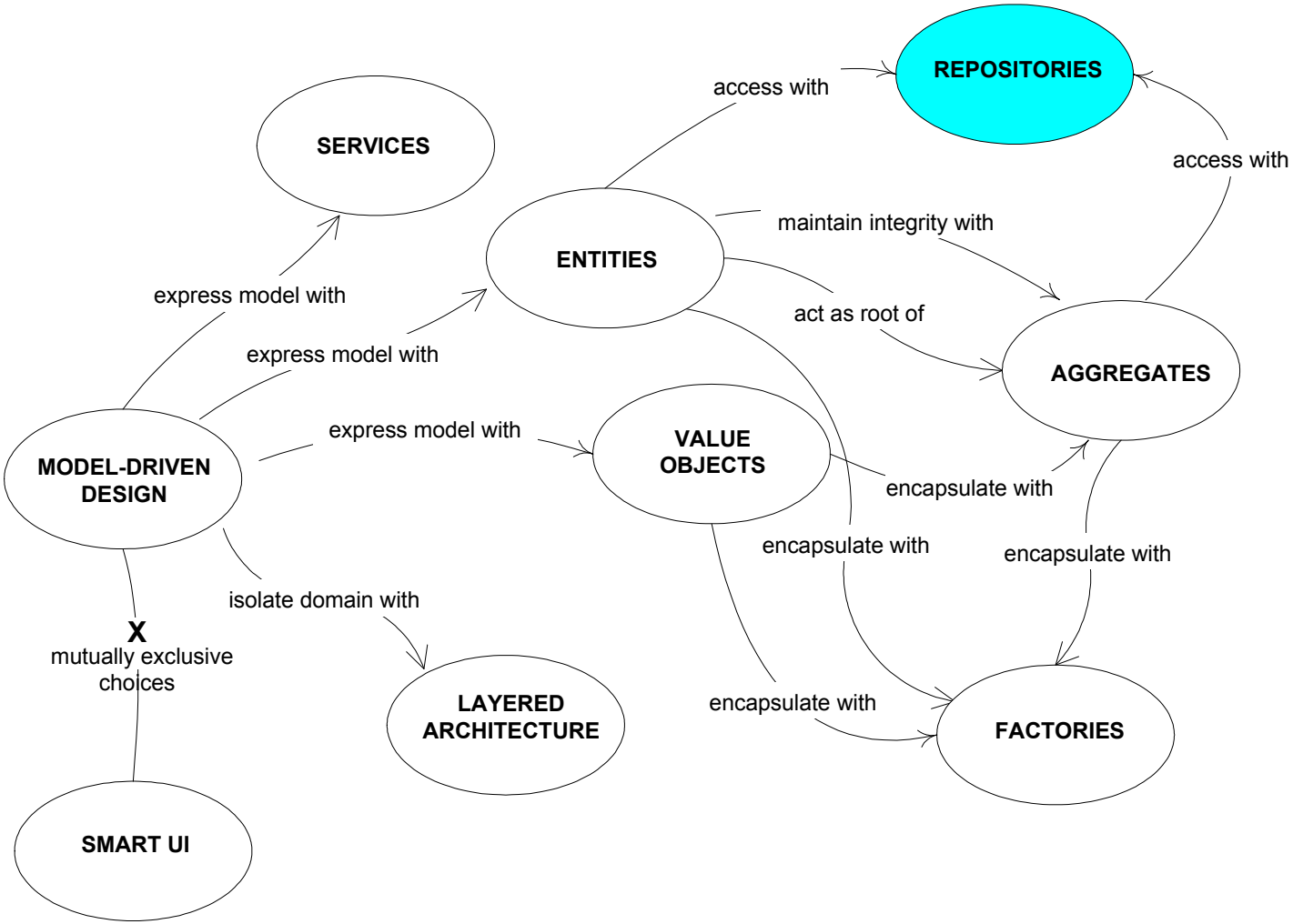


Factory Design

- Creation methods must be atomic and enforce all invariants of the created object or aggregate
- Factories should only produce created objects in a consistent state
  - For Entities this means creation of the entire aggregate with all invariants satisfied and possibly optional elements to be added later
  - For immutable Value Objects, all attributes need to be initialised to their correct final state
- A factory should be abstracted to the desired type, rather than the concrete classes created
- Use classic patterns eg. GoF Factory Method and Abstract Factory

Reconstituting Stored Objects

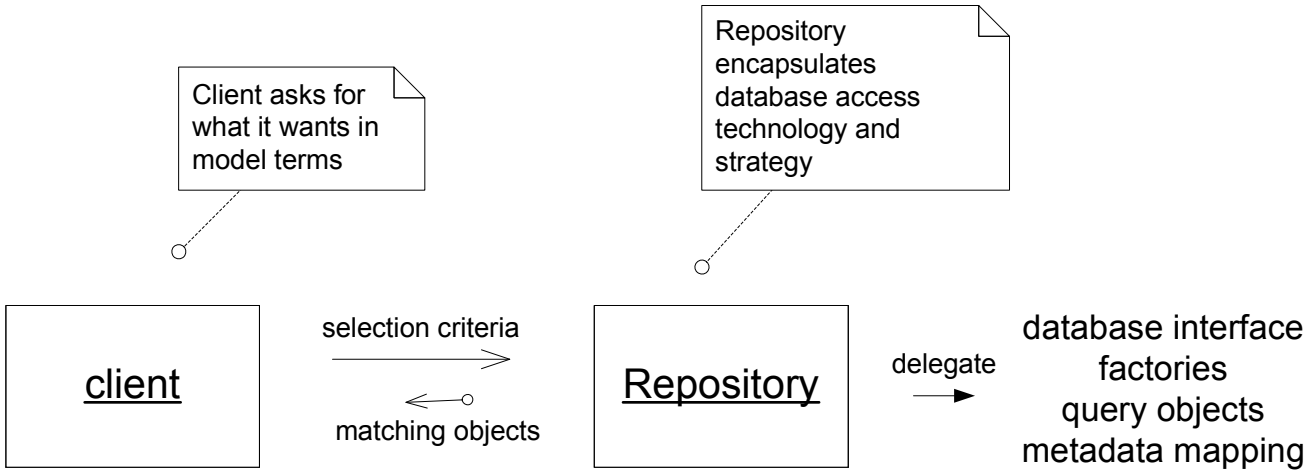
- A Factory used for reconstitution is similar to one used for creation with the following differences
  - An Entity Factory used for reconstitution does not assign a new tracking ID
  - Identifying attributes must be part of the input parameters to a factory during object reconstitution
  - A Factory reconstituting objects will handle violations of invariants differently
    - During creation invariant violations are strictly rejected
    - During reconstitution a more flexible approach is needed as the object does exist in some form in the system


REPOSITORIES
access with
access with
SERVICES
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Repositories

- Clients need a practical means of acquiring references to pre-existing domain objects
- For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type
- Provide methods to add and remove objects which will encapsulate the actual insertion and removal of data from the data store
- Provide methods that select objects based on some criteria and return fully instantiated objects or collections of objects whose values meet the criteria, thereby encapsulating the actual storage and querying technology

Repositories

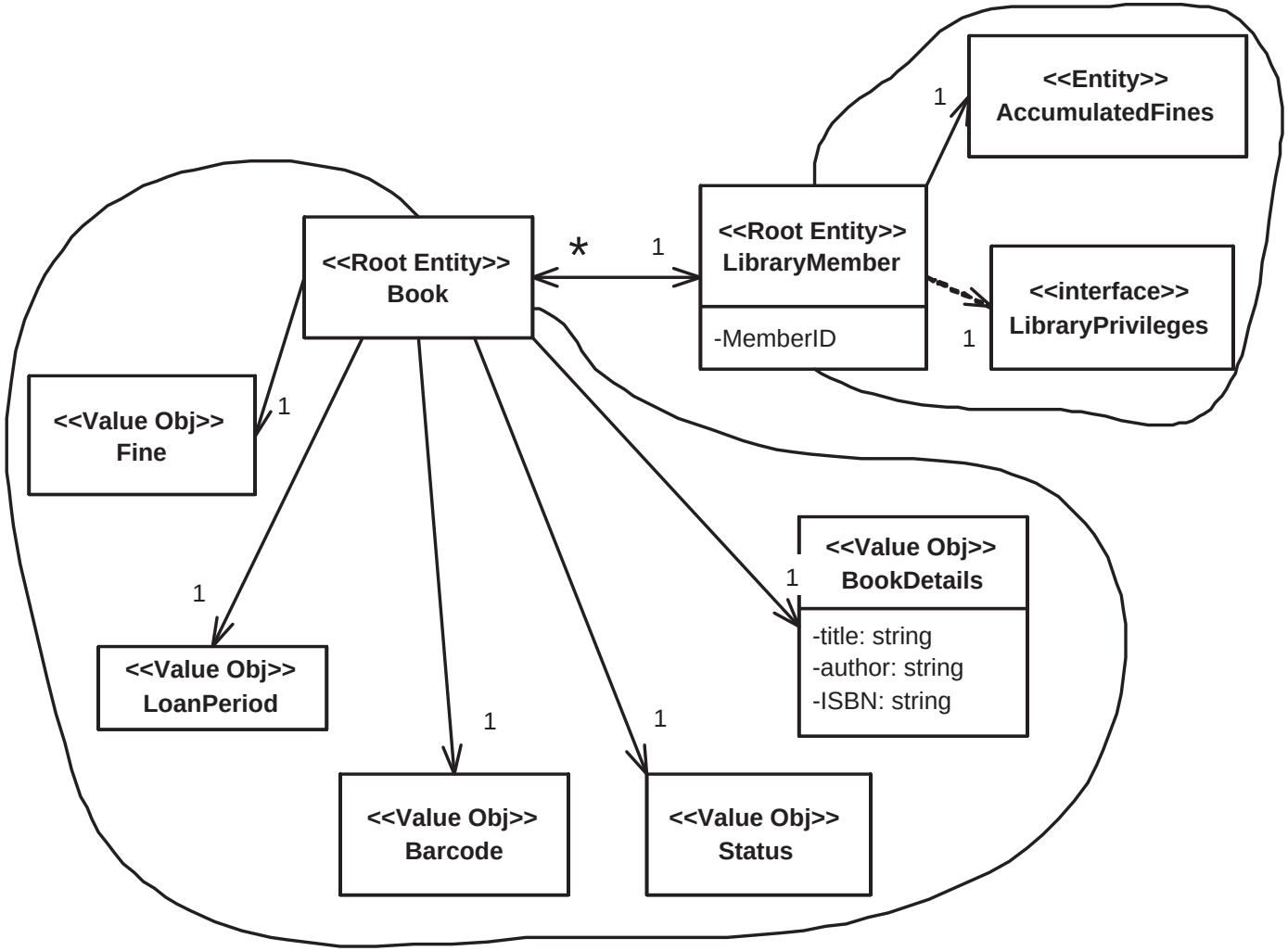


Modelling a Library System Using DDD Building Blocks

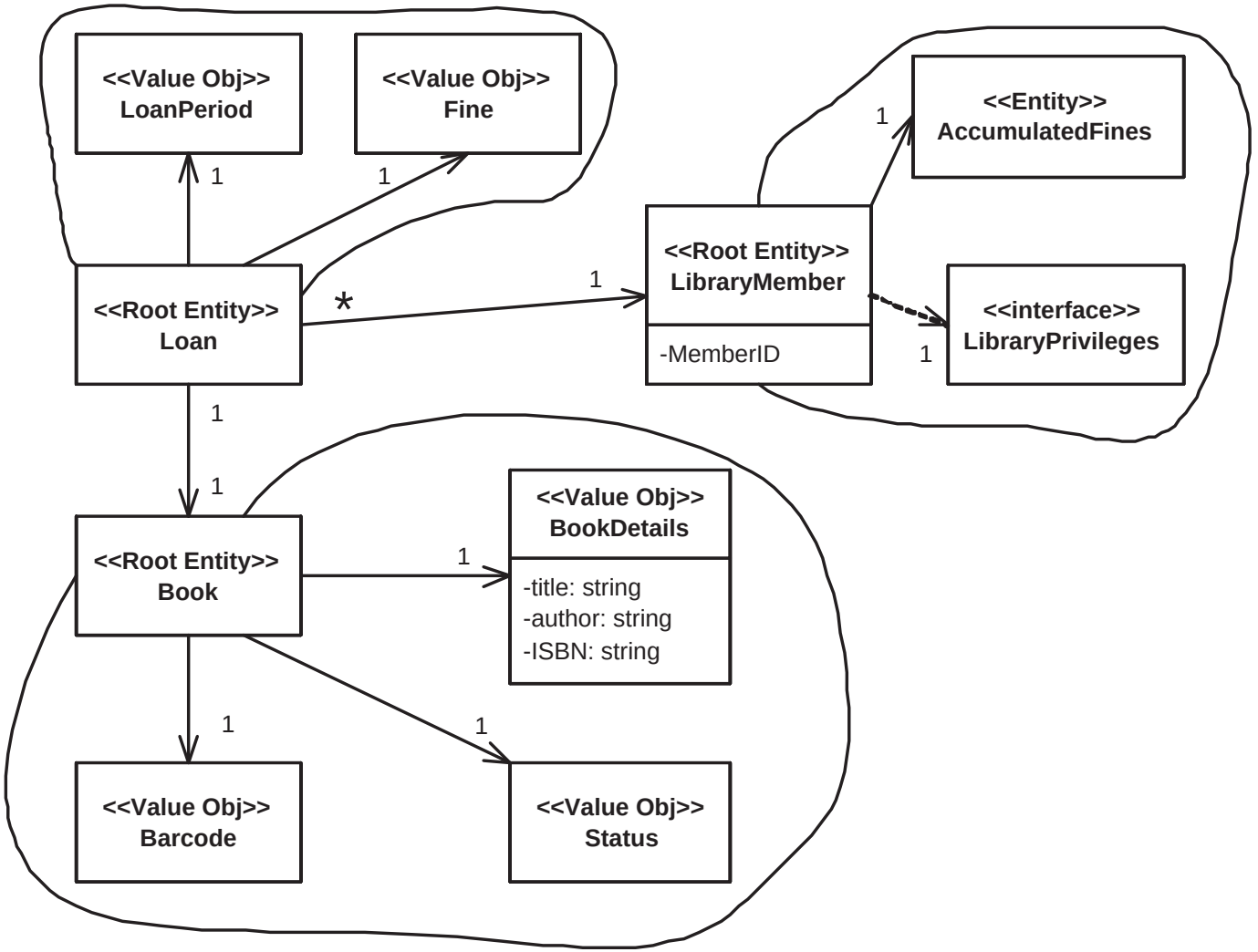
Read the handout describing the library system and:

1. Give a UML diagram illustrating the classes required for modelling the system. Only provide *key* attributes and/or operations for your classes. Think of your classes in terms of the building blocks of domain-driven design, and identify these on your diagram.
2. Give a sequence diagram showing the object interactions when a book is checked out of the library.

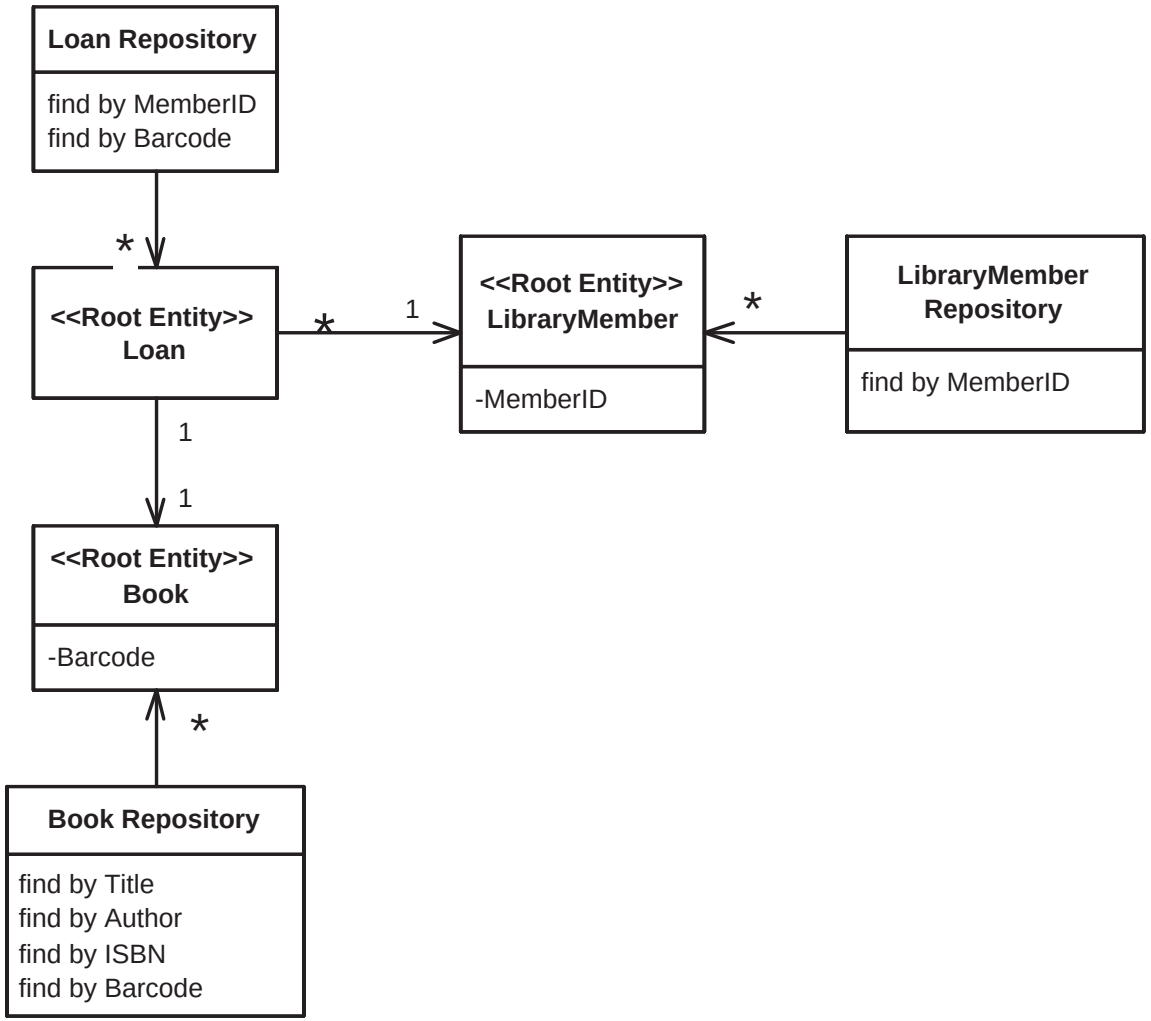
Library System Missing Loan



Library System With Loan



Library System Repositories



Dealing With Business Rules — The Specification Pattern

```
class Invoice
{
    public boolean isOverdue() {
        Date currentDate = new Date();
        return currentDate.after(dueDate);
    }
}
```

Usage: `anInvoice.isOverdue()`

Boolean test methods are useful for modelling *simple* rules

Complex Rules can Obscure the Domain Model

- Rules can be complex, for example, `anInvoice.isDelinquent()`:
  - First test if overdue, then check grace period for customer
  - Grace period depends on the customer's account status, payment history, and company policy on the products being invoiced.
  - Some delinquent invoices will prompt a second notice, others will be sent to a collection agency
- An `Invoice` is essentially a request for payment — the clarity of this abstraction will be obscured by complex rule evaluation code
- Nonetheless, domain rules should live within the Domain Layer (not the Application Layer)

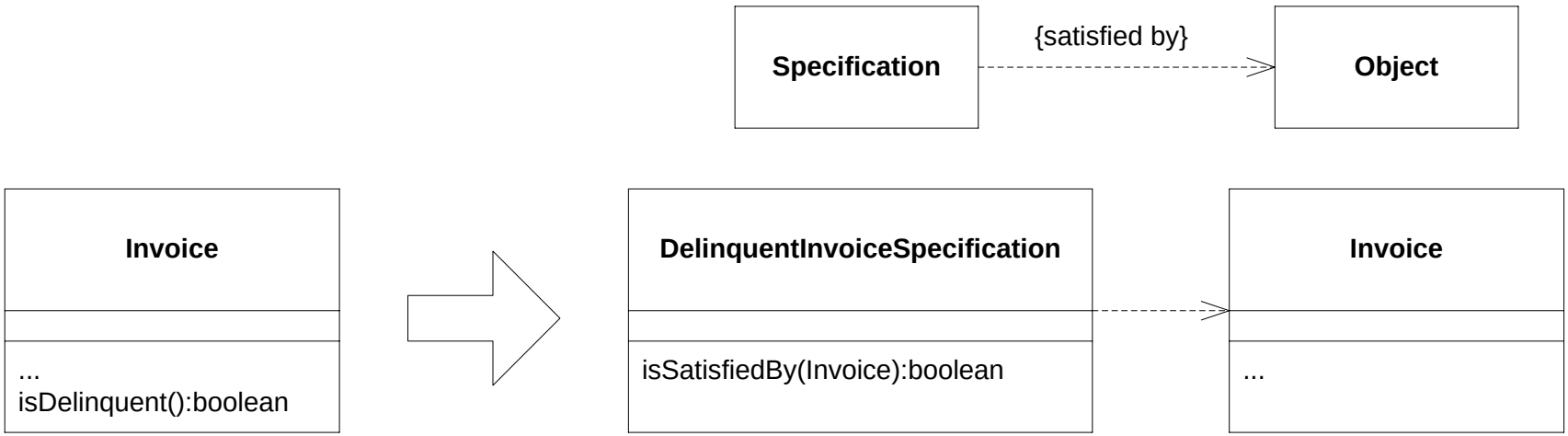
Logic Programming — An Option?

- Logic programming provides the concept of separate, combinable, *predicates*, eg. Prolog
- Full implementation of this paradigm with objects is cumbersome
- A general implementation does not communicate intent as well as a more specialised design

Specifications — Inspired by Logic Programming

- Borrow the concept of predicates and create specialised objects that evaluate to a Boolean.
- Test methods expand into their own value objects, termed specifications, and are used to evaluate another object to see if the predicate is true for that object
- More formally: “A Specification is a predicate that determines if an object does or does not satisfy some criteria”

Delinquent Invoice Specification



Implementation of Delinquent Invoice Specification

```
class DelinquentInvoiceSpecification extends
    InvoiceSpecification
{
    private Date currentDate;

    public DelinquentInvoiceSpecification(Date
        currentDate) {
        this.currentDate = currentDate;
    }

    public boolean isSatisfiedBy(Invoice candidate) {
        int gracePeriod =
            candidate.customer().getPaymentGracePeriod();
        Date firmDeadline =
            DateUtility.addDaysToDate(candidate.dueDate,
            gracePeriod);
        return currentDate.after(firmDeadline);
    }
}
```

Using the Delinquent Invoice Specification

```
public boolean accountIsDelinquent(Customer customer) {
    Date today = new Date();
    Specification delinquentSpec = new
        DelinquentInvoiceSpecification(today);

    Iterator it = customer.getInvoices().iterator();
    while (it.hasNext()) {
        Invoice candidate = (Invoice) it.next();
        if (delinquentSpec.isSatisfiedBy(candidate))
            return true;
    }
    return false;
}
```

Specifications can be Used for Different Purposes

- Validating an object to see if fulfills some need or is ready for some purpose (see the previous example)
- Selection of objects from a collection
- Specifying the creation of a new object to fit some criteria (DDD book, p.234)

Specification for Selection of Invoices in Memory

In the InvoiceRepository class:

```
public Set selectSatisfying(InvoiceSpecification spec) {

    Set results = new HashSet();
    Iterator it = invoices.iterator();

    while (it.hasNext()) {
        Invoice candidate = (Invoice) it.next();
        if (spec.isSatisfiedBy(candidate))
            results.add(candidate);
    }
    return results;
}
```

Client code:

```
Set delinquentInvoices =
    invoiceRepository.selectSatisfying(
    new DelinquentInvoiceSpecification(currentDate));
```

Specification for Selection of Invoices from a DataBase

Add to the DelinquentInvoiceSpecification class:

```
public string asSQL() {
 return
    " SELECT * FROM INVOICE, CUSTOMER" +
    " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
    " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
    " < " + SQLUtility.dateAsSQL(currentDate);
 }
```

Specification for Selection of Invoices from a DataBase

Add to the `DelinquentInvoiceSpecification` class:

```
public string asSQL() {
 return
    " SELECT * FROM INVOICE, CUSTOMER" +
    " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
    " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
    " < " + SQLUtility.dateAsSQL(currentDate);
 }
```

This solution is messy, details of the table structure have leaked into the domain layer. This makes both `Invoice` and `Customer` less modifiable.

An Improved Version, First the InvoiceRepository

```
public class InvoiceRepository {
    public Set selectWhereGracePeriodPast(Date aDate) {
    // this is a specialised query
    String sql = whereGracePeriodPast_SQL(aDate);
    ResultSet queryResultSet =
        SQLDatabaseInterface.instance.executeQuery(sql);
    return buildInvoicesFromResultSet(queryResultSet);
    }

    public String whereGracePeriodPast_SQL(Date aDate) {
        return
            " SELECT * FROM INVOICE, CUSTOMER" +
            " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
            " AND INVOICE.DUE_DATE +
              CUSTOMER.GRACE_PERIOD" +
            " < " + SQLUtility.dateAsSQL(aDate);
        }
```

An Improved Version, First the InvoiceRepository cont.

```
// this solution makes use of double dispatch
public Set selectSatisfying(InvoiceSpecification spec) {
    return spec.satisfyingElementsFrom(this);
   }
}
```

Now for the DelinquentInvoiceSpecification

```
class DelinquentInvoiceSpecification extends
    InvoiceSpecification
{
    // base code for DelinquentInvoiceSpecification here

    public Set selectSatisfyingFrom(InvoiceRepository
        repo) {
        // delinquency rule defined as: "grace period
            past as of current date"
        return
            repo.selectWhereGracePeriodPast(currentDate);
    }
}
```

This repository selection method is very specific, we may want a more generic selection mechanism, i.e. all overdue invoices.

Conclusion

Domain-Driven Design offers a compelling vision for software development — the ability to create systems which directly express the problem domain and are not fettered by technical concerns. This allows us to focus our efforts on tackling the complexity of the business and thereby produce more meaningful and flexible systems.