Domain Modeling

• A representation of the conceptual classes in problem domain
  - UML (conceptual) class diagrams
    • What’s a conceptual class?
      - What are its attributes?
      - What are description classes?
    • What are inter-class associations?
    • Implementation issues

Domain Modeling

• Idea: identify the important concepts in the problem domain
  - These concepts later will serve as basis for the design and the implementation

• Domain modeling (domain analysis)
  - We will consider object-oriented domain modeling in the context of the Unified Process
The Domain Model

- Representation of real-world conceptual classes in the problem domain
  - With class attributes
- Representation of relationships between conceptual classes
  - Associations between classes
  - Generalization relationships
- Represented by a UML class diagram
  - But it could also be described in text

Models of Domain Concepts

Conceptual class:
No operations; part of domain model

Implementation class: created during design; not part of domain model

Of course, it is not always this simple ...
A Conceptual Class Diagram

Sales
  LineItem
    quantity
      0..1
        1..*
            Contained-in
                1
    date
        1
        1
            Paid-by
                1
    time
        1
        1
    Payment
      amount

Records-sale-of
    Item
      1

Stocked-in
    *
      1

Sale
  captured-on
    address
      name
      1
      1..*

Houses
  1
  1

Register
  1

UML Diagrams

- UML is just notation
- Different diagrams mean different things in different contexts
  - Conceptual perspective: description of the problem domain
  - Specification perspective: description of software abstractions or components
    - e.g., no commitment to a particular language
  - Implementation perspective: description of Java classes
- Can have UML class diagrams in each perspective; used for different purposes
Conceptual Classes

• Abstractions of concepts from the problem domain
  - Concepts such as Sale, Register, Item, ...

• UML representation

<table>
<thead>
<tr>
<th>Sale</th>
<th>Class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Attributes</td>
</tr>
<tr>
<td>time</td>
<td>- only attributes relevant to the problem</td>
</tr>
</tbody>
</table>

Building the Domain Model

• Over several iterations during elaboration
• Driven by the use cases
  - In each iteration, the use case model is enriched, and the domain model is extended accordingly

• How to identify conceptual classes?
  - Consider common categories (see next slide)
  - Identify nouns and noun phrases from the fully dressed use case
  - Use analysis patterns: existing partial domain models created by experts
    • “recipes” for well-known problems and domains (e.g. accounting, stock market, ...)
**Common Categories**

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical objects</td>
<td>Register, Airplane</td>
</tr>
<tr>
<td>Places</td>
<td>Store, Airport</td>
</tr>
<tr>
<td>Transactions</td>
<td>Sale, Payment, Reservation</td>
</tr>
<tr>
<td>Roles of people</td>
<td>Cashier, Manager</td>
</tr>
<tr>
<td>Scheduled Events</td>
<td>Meeting, Flight</td>
</tr>
<tr>
<td>Records</td>
<td>Receipt, Ledger</td>
</tr>
<tr>
<td>Specifications and</td>
<td>FlightDescription, Product</td>
</tr>
<tr>
<td>descriptions</td>
<td>Specification, ProductCatalog</td>
</tr>
<tr>
<td>Catalogs of descriptions</td>
<td></td>
</tr>
</tbody>
</table>

**Example: Simplified “Process Sale”**

Simplified scenario in Process Sale. No credit cards, no taxes, no external accounting system, no external inventory system, ...

- **Customer** arrives with goods
- **Cashier** starts a new sale

Possible conceptual classes: **Customer**, **Cashier**, **Item** (i.e., goods), **Sale**
Simplified “Process Sale”, cont.

- **Cashier enters item ID**
- **System records sale line item and presents item description, price, and running total**
- **At the end, Cashier tells Customer the total and asks for payment**

Possible conceptual classes: **SalesLineItem**, **ProductSpecification** (description + price + item ID), **Payment**
- item ID, description, price, total: probably too simple to be classes but will be class attributes

---

Simplified “Process Sale”, cont.

- **Cashier enters amount tendered (cash)**
- **System presents change due, and releases cash drawer**
- **Cashier deposits cash and returns change**
- **System presents receipt**

Possible conceptual classes: **Register** (implied by cash drawer), **Receipt**
- amount, change: probably too simple
Example (cont)

- Want a completely integrated system
  - Store: has the items and the registers
  - ProductCatalog: stores the product specifications for all items
  - Manager: starts all the registers in the morning
    - Need this for the initial implementation: to be able to start up the system
- There is no “correct solution”
  - Somewhat arbitrary collection of concepts

Possible Initial Domain Model

- Just the conceptual classes
  - Attributes and associations later
- For this particular simplified scenario
  - Will evolve as more scenarios are explored

Register  Store  Item  Sale
Customer  Cashier  Manager  Payment
Product Specification  Product Catalog  Sales LineItem
A Common Mistake

- **Example**
  
<table>
<thead>
<tr>
<th>Flight</th>
<th>OR ..?</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>Flight</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

- If in doubt, make it a conceptual class
  - Attributes should be fairly rare in a domain model and should be relevant to a use case.

  "If we do not think of some conceptual class X as a number or text in the real world, X is probably a contextual class, not an attribute." Larman Ch 9, p 146.

Description Classes

- **Class Item** represents a physical item in a store
  - unique serial number, but same ID and price as all items of same kind (e.g., JVC XV-S40 DVD player)

- Could represent ID and price as attributes of **Item**
  - Suppose we sell all items of a particular kind; we lose all price info
  - Unnecessary duplication of data

- Need a separate conceptual class that is a description of items e.g., class **ProductSpecification**

- An instance of this class represents a description of information about items
  - Even if we sell all JVC XV-S400 DVD players, we still have information about their price/item ID
The Two Alternatives

<table>
<thead>
<tr>
<th>ProductSpecification</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>price</td>
</tr>
<tr>
<td>itemID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>price</td>
</tr>
<tr>
<td>serial number</td>
</tr>
<tr>
<td>itemID</td>
</tr>
</tbody>
</table>

Describes

* Item
serial number

When Do We Need This?

- When need description of an item or a service
  - Independent of the current existence of any instances of those items or services
- When description classes would reduce amount of redundant info in model
  - e.g., many instances of the class have the same values for some attributes
- If the description alone can be in interesting relationships
  - e.g., all JVC XV-S400 DVD players are on sale until Dec 26th
Another Example

<table>
<thead>
<tr>
<th>Flight</th>
<th>Flight</th>
<th>Airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date</td>
<td>name</td>
</tr>
<tr>
<td>number</td>
<td>time</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flies-to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FlightDescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flies-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Describes-flights-to

Associations in the Domain Model

• Relationship between instances of conceptual classes
  - “connectedness” between instances
  - e.g. an order is related to the customer that placed that order

• Think of it as a mathematical relation
  - Typically a binary relation: $R \subseteq S_1 \times S_2$
  - $S_1 =$ set of instances of the first class
  - $S_2 =$ set of instances of the second class
Associations in the Domain Model

- Usually, the relation changes with time
  - For any pair \((o1, o2) \in S1 \times S2\): at some moments of time the link exists, other times it does not
- An association typically represents a relatively permanent relationship
  - Often holds for the duration of the entire lifetime of the instance(s)
  - e.g. a sale is permanently associated with the register that captures it

UML Notation

- Named to enhance understanding of the relationship
- Multiplicity: what number of instances can be associated?
- Direction arrow: just helps the reader
  - No meaning for the model; often omitted
**Multiplicity**

- One instance of Store can be associated with zero or more Item instances

```
  Store  Stocks  *  Item
  1      
```

- Intuition
  - A person may be married to many spouses during their lifetime, but at any particular moment the person is married to zero or one other person
  - Think of $R \subseteq S_1 \times S_2$ at a particular moment

**Representing Multiplicity**

- Range: $x..y$
- Common notation for ranges
  - $x..x \rightarrow x$
  - $x..\infty \rightarrow x..*$
  - $0..\infty \rightarrow *$
- Combination of ranges
  - $x..y, z..w$
  - e.g. "2,4" \rightarrow number of doors in a car
- Most common multiplicities: *, 1..*, 0..1, 1
Interpretation of Multiplicity

<table>
<thead>
<tr>
<th>Store</th>
<th>Stocks</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

*Why 1 and not 0..1?*

- E.g., an item may be sold or discontinued and then no store stocks it
- Multiplicities may encode relevant domain constraints
  - But: it is not always clear

Typical Associations

- **A is a physical/logical part of B**
  - Wing-Airplane, SalesLineItem-Sale, FlightLeg-FlightRoute, Finger-Hand
- **A is physically/logically contained in B**
  - Item-Shelf, Passenger-Airplane, Flight-FlightSchedule
- **A is recorded/reported/captured in B**
  - Sale-Register, Reservation-FlightManifest
- **A is a description of B**
  - ProductSpecification-Item
Typical Associations

- A is a member of B
  - Cashier-Store, Pilot-Airline
- A uses or manages B
  - Cashier-Register, Pilot-Airplane
- A is related to a transaction B
  - Customer-Payment, Payment-Sale, Reservation-Cancellation
- A is owned by B
  - Airplane-Airline

Finding Associations

- Consider the typical categories
  - Larman, Ch 9 p 155
- Focus on associations that are relevant with respect to the use cases
- SalesLineItem-Sale
  - A sale contains a set of line items
    - Permanent “whole-part” relationship
  - Needed in the context of the Process Sale use case (for the total and receipt)
Examples

- **ProductSpecification-ProductCatalog**
  - "contained-in" relationship
  - Given an item id, the system needs to look up the item description in the catalog

- **Payment-Sale**
  - Two related transactions: the payment is with respect to a particular sale
  - The payment info is needed to compute the change due

**Diagram**

```
Sale ? Contains ? SalesLineItem
Sale ? Paid-by ? Payment
```
**A Complicated Example**

- A store uses a set of external authorization services for payments

  * Authorizes-via 1..*  
  
  **Store**  
  **Authorization Service**

- Each service associates merchant ID with the store (different for each store)
  - The ID is provided by the store as part of the request for authorization
- A store has different merchant IDs for each service

**Stores and Services**

- A software system at headquarters: many stores, many services
  - Where should the merchantID be located?

<table>
<thead>
<tr>
<th>Store</th>
<th>AuthorizationService</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>address</td>
<td>address</td>
</tr>
<tr>
<td>merchantID</td>
<td>phoneNumber</td>
</tr>
<tr>
<td></td>
<td>merchantID</td>
</tr>
</tbody>
</table>

Option 1

Option 2
**Association Class**

- Attribute merchantID is conceptually related to the association, not to the individual classes
- **Solution:** association class
  - Represents attributes of the association

```
Store * Authorizes-via 1..* Authorization Service

ServiceContract
merchantID
```

**Association Classes**

- An association class is a generalized form of an association
  - **Association:** set of pairs \((o1, o2) \in S1 \times S2\)
  - **Association class:** set of pairs \((o1, o2) \in S1 \times S2\), where each pair has some attached info (attributes)
- The attributes of a pair may change with time (e.g., the merchant ID may change)
- Association classes may be associated with other classes (e.g., ternary relation)
When to Use Association Classes?

- When an attribute “doesn’t fit” in the classes participating in an association
- When the lifetime of the attribute depends on the lifetime of the association
- Often used with many-to-many associations

Many-to-Many Association

- A company may employ several persons
- A person may be employed by several companies
  - Many people work two or even three jobs
- Attributes: salary, starting date, …
What is the Difference?

![Diagram](image)

Company * Employs * Person

<table>
<thead>
<tr>
<th>Employment</th>
<th>Employment</th>
</tr>
</thead>
<tbody>
<tr>
<td>salary</td>
<td>salary</td>
</tr>
<tr>
<td>startDate</td>
<td>startDate</td>
</tr>
</tbody>
</table>

Company 1 * Person

Associations and Their Implementation

- In the domain model: an association is **conceptual** and does not imply that a particular implementation will be used
  - Some domain-level associations may never be implemented
- In design and coding: there are **standard mechanisms** to implement the associations
## Implementation Examples

### Class Sale
```java
class Sale {
    // set of references
    // to S.L.I. objects
    private Set items;
}
```

### Class SalesLineItem
```java
class SalesLineItem {
}
```

### Relationship
```
Sale 1 Contains 1.* SalesLineItem
```

Could even be bi-directional: fields in both classes

## Domain Model vs. Implementation

- **Key principle**: in the domain model, complex concepts should be related through associations, not through attributes

- **In design/code**, the implementation of the association may be through attributes of software classes
  - e.g. class Flight may have a field (attribute) that refers to an instance of Airport
  - But other implementations are also possible
Common Types of Attributes

- **Primitive types**: Number, String, Boolean
- **Other simple types**: Date, Time, Name, Address, Color, Phone Number, SSN, UPC (universal product code = barcode), ZIP, enumeration types, ...
- Some simple attribute types (e.g., SSN) may need to be represented as separate conceptual classes

When Attribute Types are Classes?

- The type has **separate sections**
  - e.g. address, phone number, name, item id

  ```
<table>
<thead>
<tr>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: Address</td>
</tr>
</tbody>
</table>
  ```

- The type has associated **operations**
  - e.g. parsing and validation for SSN

```
<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>line1: String</td>
</tr>
<tr>
<td>line2: String</td>
</tr>
<tr>
<td>city: String</td>
</tr>
<tr>
<td>state: Enumeration</td>
</tr>
<tr>
<td>ZIP: Integer</td>
</tr>
</tbody>
</table>
```
When Attribute Types are Classes?

- **Quantity with a unit**
  - Most quantities have units needed for conversions: price, velocity, weight, etc.
  - Represent different quantities as separate conceptual classes: Money, Weight, etc.

```
<table>
<thead>
<tr>
<th>Payment</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount: Number</td>
<td>amount: Money</td>
</tr>
<tr>
<td>Not useful</td>
<td>Better</td>
</tr>
</tbody>
</table>
```

"Process Sale" Use Case

- **Store**
  - address:Address
  - name:String

- **Payment**
  - amount:Money

- **Product Specification**
  - descr:String
  - price:Money
  - id:ItemID

- **Sales**
  - LineItem
    - quantity:Integer

- **Sale**
  - date:Date
  - time:Time

- Store name/address: for receipt
- item ID in Product Spec: for lookups
- Description/price in ProductSpec: for amount due and for display/receipt
Summary

• **Conceptual classes**
  - Special case: specification classes

• **Attributes**
  - Should be simple

• **Associations:** relationships that are relevant for the use cases
  - Multiplicity at a particular moment
  - Association classes