Object-Oriented Design

Two Worlds

- Object-Oriented Analysis: since late 80s
  - Domain modeling based on "concepts"
- Structured Analysis: since 70s, still used
  - Functional model and data model
- Functional model: e.g. data flow diagrams
- Data model: e.g. entity-relationship diagrams
  - Similar to the domain model

Data Flow Diagram

- Functions and the flow of data

Customer

Receive Order

Invalid orders
orders
billing info

INVOICES

Prepare Invoice

Customer name/address
Invoice details

Printer

CUSTOMERS

Entity-Relationship Diagram

- Represents data objects and their relationships

customer

placed

order

same as 0..*

same as 1

cardinality

modality

woman

married

man

Structured Design

- Closely related to structured analysis
- Data flow diagrams are mapped to design
  - Modules & hierarchical control structure

Object-Oriented Design

- Objects, with attributes and operations
- Classes define the blueprint for objects
  - Objects are instances of classes
- Messages among objects
  - Object x sends message to object y to activate one of m's operations
- Operation in design -> method in code
- Message in design -> method call in code
“Flavor” of OO Design/Code

- Many small methods, many calls
- “Distributed” control: processing is split among many participants
  - In structured design: centralized control
- Distributed vs. centralized
  - Centralized is easier to understand
    - In OO: “chasing around the objects, trying to find the program”
  - Distributed is more flexible
    - Reduces the impact of change

Example: Total of a Sale w/ Discount

- Centralized
  - Sale asks each SalesLineItem for its quantity and ProductSpecification
  - Sale asks each ProductSpecification for the price
  - Sale computes total := sum(quantity * price)
  - Sale asks Customer object for discount info
  - Sale calculates discounted price using discount info

- Distributed
  - Sale asks each SalesLineItem to compute its subtotal
  - Sale computes total := sum(subtotals)
  - Sale gives the total to the Customer object and asks it to compute the discounted price
  - More delegation, less coupling

Core Principles of OO Design

1. Identify the responsibilities that are needed to satisfy the requirements
2. Assign these responsibilities to objects
   - Add the appropriate operations (methods)
3. Design the interactions among objects
   - Add the appropriate messages (i.e. calls)

UP Artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Incep</th>
<th>Elab</th>
<th>Const</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-Case Model</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supplm. Spec</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain Model</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Model</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Implem. Model</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Requirements analysis: Use-Case Model + Supplementary Specification

Domain analysis: Domain Model
Design: Design Model
Coding: Implementation Model

Relationships Among Models
**Design in the Unified Process**

- Goal: create a design model that can be used to build the system
- No design during inception
- Elaboration: each iteration has requirements analysis + some design + some implementation
- Early iterations target high-risk design issues

**Design in the Unified Process**

- At the end of elaboration
  - Almost all requirements are clarified
  - High-risk design aspects are stabilized
- Construction: iterative design and coding for the remaining requirements
  - Interaction diagrams: sequence diagrams, collaboration diagrams
  - Design class diagrams

**Interaction Diagrams**

- Key tool in object-oriented design
- Show objects and their interactions
- While creating the interaction diagrams, the designer makes decisions about object responsibilities and object interactions
- Interaction diagrams require creativity
  - Harder than use cases and domain models

**A Simple Collaboration Diagram**

```
message1() → :ClassAInstance

message order ①: message2() ↓
object

2: message3() ↓
link

:ClassBInstance
```

**Links**

- Links: shows that messages may flow between two objects
  - Many messages may flow along the same link
  - Messages may flow in both directions

```
1: makePayment(amount) →
:Register
2: foo() →
:Sage
2.1: bar() ←
```
**Messages**

- UML notation for messages
  
  \[
  \text{return:=message(parameter:Type):ReturnType}
  \]

- Sometimes there is no return value
  
  \[
  \text{activate(level:StartLevel,num_users:Integer)}
  \]

- Type information may be excluded if obvious or unimportant
  
  \[
  \text{spec:=getProductSpec(id)}
  \]
  
  \[
  \text{spec:=getProductSpec(id:ItemID)}
  \]
  
  \[
  \text{spec:=getProductSpec(id:ItemID):ProductSpec}
  \]

**Messages (cont)**

- A small arrow indicates the direction
  
  Sequence number which describes the ordering
  
  - e.g. 1, 2.1, 3.4.1

- Messages to "self" ("this") are possible

```plaintext
:Register
1.2:clear()
```

**Creation of Instances**

- UML convention: message named `create`
  
  - May include parameters
    
    - Indicates the passing of initial values
  
  - `{new}` may optionally be added

```plaintext
:Register 1:create(cashier) → :Sale (new)
```

**Message Numbers**

- The first message is not numbered
  
  - Represents an action/event that triggers the collaboration

- If a message m with number x.y.z is received, the messages that are sent during the processing of m are numbered x.y.z.1, x.y.z.2, etc.
  
  - And the rule is applied recursively to these outgoing messages

**Example**

```plaintext
msg1():ClassA 1:msg2():ClassB
1.1:msg3()↓
2.1:msg5()↑
2:msg4():ClassC
2.2:msg6():ClassD
```

**Iteration**

- An iteration clause is appended to the sequence number

```plaintext
msg1():ClassA 1*[i:=1..N]:msg2():ClassB
```

- If the details of the iteration clause are not important: just "*"
  
  - e.g. 3.1*:update()
Iteration over a Collection

- Very common: iterate over all objects in a collection (e.g. list, map, etc.)
  - e.g. in Java: done through java.util.Iterator
  - UML: the set of objects is called a multiobject
- The message is sent to each object in the collection (not to the collection object)

```
:Sale st:=getSubtotal() :SalesLineItem
```

Conditional Messages

- A condition clause is appended to the sequence number
  - The message is sent only if the condition is true

```
3:ClassX 3.1[color=red]:nn() :ClassY
```

Another Example

```
prepare()
:Order
  1.2.1:needsReorder:= needToReorder()
  * prepare()
:OrderLine
  1.1:hasStock:=check() :StockItem
  1.2[hasStock]:remove() :
  1.3[hasStock]:create() :ReorderItem
:DeliveryItem
```

Names of Objects

- Representation of parameters and return values that are objects
  - More precisely, references (pointers) to objects
```
:Register
  1: makePayment(amount:Money) :Sale
  2: summary := recordSale(s:Sale):Summary :X
  3: [amount>$100] setImportanceFlag() summary:Summary
```

Messages to Classes

- It is possible to send a message to a class rather than to an object
  - This invokes a class operation
  - e.g. in Java/C++ such operation are implemented by static methods

```
msg1() :
:ClassA
  1:list:= synchronizedList(aList) :Collections
```

Sequence Diagrams

- Another kind of UML iteration diagram
Elements of a Sequence Diagram

- **Object lifeline**: vertical dashed line
- **Messages**: time ordering is organized top to bottom
- **Returns**: optional, represented by dashed arrows
  - Some exclude them, others show only returns that return a value
- **Activation box**: shows that the method is active on the run-time call stack

Another Example

![Sequence Diagram Example](image)

Messages to “self”

![Messages to "self" Diagram](image)

Object Creation and Destruction

![Object Creation and Destruction Diagram](image)

Conditions and Iteration

- A condition indicates when a message in sent
  
  \[\text{[color=red]} \text{msg()} \]

- Iteration for one message
  
  \[\text{*[i:=1..N]:msg()} \]

Iteration for Multiple Messages

![Iteration for Multiple Messages Diagram](image)
Iteration for a Multiobject

```
:Sale
  st := getSubtotal()
  t := getTotal()

:SalesLineItem
```

Names of Objects

- Representation of parameters and return values that are objects

```
:Register

smsm := recordSale(ss:Sale)

amount > $100 setImportanceFlag()

s := Sale

: X

sm := S
```

Messages to Classes

```
: ClassA

list := synchronizedList(aList)

msg1()
```

Collaboration vs. Sequence Diagrams

- Represent the same information
  - Could be used interchangeably
- Advantages and disadvantages
  - Collaboration diagrams: harder to keep track of the flow of control
  - Sequence diagrams are more "verbose"
- In practice people seem to prefer sequence diagrams
  - I will mostly show collaboration diagrams to save space

UML Resources

- "UML Distilled", by Martin Fowler
  - 2nd edition - on reserve at SEL
    - Based on UML 1.4
  - 3rd edition - came out recently
    - Based on UML 2.0
  - Short book (150 pages); highly recommended
- Larman’s book: UML 1.4
  - In the context of object-oriented design
- Countless other books

UML 2.0 versus UML 1.x

- In this course we consider first-generation UML: versions 1.1 through 1.4
- Major development in 2003: second-generation UML: version 2.0
  - Substantial changes: more powerful, more general
  - New, adoption will take time
Basic Patterns for Object-Oriented Design

The Goal of Object-Oriented Design
- Goal: produce a design model
  - Two major categories of artifacts
- Interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams
- Design classes and the corresponding class diagrams
  - Often based on conceptual classes from the domain model

Basics of Object-Oriented Design
- Core idea: identify responsibilities and assign them to classes and objects
- Responsibilities for doing
  - Create an object, perform calculations, invoke operations on other objects, ...
- Responsibilities for knowing
  - Private encapsulated data, related objects, things that can be derived or calculated, ...

Responsibilities
- Doing: “a Sale object is responsible for creating its SalesLineItem objects”
- Knowing: “a Sale object is responsible for knowing its total”
  - Often inferable from the domain model
- Implemented by operations: act alone or in collaboration with other objects
  - Sale: operation getTotal, which collaborates with all SalesLineItem objects

Responsibilities and Interaction Diagrams
- Assignment of responsibilities occurs during the creation of interaction diagrams
  - Decisions are encoded in the diagrams

Sale has the responsibility to handle message makePayment: to fulfill this responsibility, Sale is responsible for creating a new object Payment and for collaborating with it

A Sample Design Pattern
Name: Information Expert

Solution: Assign a responsibility to the class that has the information needed to fulfill that responsibility

Problem it solves: How do we assign responsibilities to classes and objects?

Example: ...

Discussion: when to use it, how to use it, ...
Contraindications: when not to use it, ...
Some Basic Patterns

- For now, we will focus on five basic patterns for assigning responsibilities
  - Information Expert
  - Creator
  - High Cohesion
  - Low Coupling
  - Controller

Pattern 1: Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill the responsibility
- Example in the POS system: who should be responsible for knowing the total of a sale?
- Look for candidates among the existing design classes in the design model
  - If nothing applicable, look at the domain model

The Domain Model

- Suppose we haven’t created any design classes yet; look at the domain model

Design Class “Sale”

- In the domain model, Sale looks like an expert: it knows about all line items, and can compute the sum of their subtotals
- So we create a design class Sale
  - Part of the design class diagram

Interaction Diagram

- Now need to create an interaction diagram for the responsibility
  
\[
t := \text{getTotal()} : \text{Sale}
\]
- Need subtotal for each item (i.e. quantity \times price)
- Who is responsible for knowing the subtotal for a line item?
- SalesLineItem is the expert, so we create a corresponding design class

SalesLineItem in the Design Model
### ProductSpec in the Design Model

- Who is responsible for knowing item price?

<table>
<thead>
<tr>
<th>Sale</th>
<th>SalesLineItem</th>
<th>Product Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>quantity</td>
<td>description</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td>itemID</td>
</tr>
<tr>
<td>getTotal()</td>
<td></td>
<td>getPrice()</td>
</tr>
</tbody>
</table>

### Assigned Responsibilities

- To know a sale’s total, three responsibilities were assigned
  - Sale: knows sale total
  - SalesLineItem: knows subtotal for line item
  - ProductSpecification: knows product price

- Interaction diagram: shows the dynamic behavior
  - Design classes were created as necessary
  - Show the static structure of the software

### Wasn’t That Solution Obvious?

- Consider an alternative
  - Some object X asks the Sale for all of its SalesLineItem objects
  - X asks each line item for the quantity and the ProductSpecification
  - X asks each spec for the price
  - X computes sum(quantity * price)

- X has several responsibilities related to data that lives in other objects
  - Not uncommon for OO novices to do this

### Summary

- Information Expert: objects do things related to the information they have
  - Often requires collaboration among objects
  - Enables information hiding: objects use their own info to fulfill tasks
  - Low coupling, more robust and maintainable system, better opportunities for reuse

### Summary (cont)

- Contraindications: may create problems with coupling and cohesion
  - E.g. should a Sale be responsible for saving itself to a database?
  - Increased coupling: the code in Sale depends on DB services (SQL, JDBC, etc)
  - Duplicated code: similar DB logic will be duplicated in many persistent classes (bad for maintenance)

### Pattern 2: Creator

- Assign class B the responsibility of creating an instance of class A if:
  - B aggregates A objects
    - Whole-Part: Assembly-Part (e.g. Body-Leg)
  - B contains A objects
  - B records A objects
  - B closely uses A objects
  - B has initializing data that will be passed to a new A object
  - B is an expert w.r.t. creating A objects
Who should be responsible for creating a SalesLineItem?
- Sale aggregates SalesLineItem objects

**Example**

```plaintext
:Register
makeLineItem(quantity)

:Sale
create(quantity)
:Sales
LineItem
```

**Summary**
- The creating object will have to be connected with the new object anyway, so we just add some extra work there ...
  - If some other object were to do it: extra coupling, reduced maintainability/reuse
  - E.g.: the enclosing container or recorder is a natural candidate for a creator
- Contraindications: complex creation
  - E.g. using recycled objects for performance
  - Better to use the Factory pattern

**Pattern 3: Low Coupling**
- Assign responsibilities so that coupling remains low
  - Goal: low dependency, low change impact, increased reuse
- Coupling: measure of how strongly one class is connected to, has knowledge of, or relies on other classes
  - Changes in related classes force local changes
  - Harder to understand classes in isolation
  - Harder to reuse because a class requires the presence of other classes it depends on

**Example**
- Classes Payment, Register, Sale
- Need to create a Payment instance and associate it with the Sale
  - Which class creates Payment instances?
  - Register has the info necessary to create a payment, so we can use Creator

```
makePayment(x)
:Register
1: create(x)
1.1: makePayment(x)
:Sale
2: addPayment(p)
```

**Examples of Coupling**
- Class A has an attribute (field) of class B
- An instance of A calls an instance of B
- A has a method that references instances of B
  - local variable/parameter/return value is a reference (i.e. pointer) to a B object
- A is a direct or indirect subclass of B

**Example**
- But this couples class Register to knowledge of the Payment class
  - Alternative

```
makePayment(x)
:Register
1: makePayment(x)
1.1: create(x)
:Payment
```
- Basic idea: Sale will need to know about Payment, so this coupling is already there
  - But Register does not need to know
Another Example

After this, who should be responsible for computing the change?

- Option 1: Sale asks Payment about the amount tendered & computes diff
- Option 2: Payment asks Sale about the total & computes diff

Which option has lower coupling?

Summary

- Low coupling: general principle for design
  - Should be considered together with other patterns, and some trade-offs may be needed
- Classes that are inherently generic in nature and have high probability of reuse should have especially low coupling
- Some degree of coupling is necessary, the goal is to avoid unnecessary coupling

Pattern 4: High Cohesion

- Cohesion: how strongly related and focused are the responsibilities of a class
- A low-cohesion class does unrelated things, or just does too many things
- Problem: responsibilities should have been delegated to other classes

Example

Who creates Payment objects?

- If Register does the work for all system events, it will become bloated and incohesive

Degrees of Cohesion

- Very low: a class is solely responsible for many tasks in different areas
  - E.g. RDB-RPC-Interface for interacting with relational databases (RDB) and handling of remote procedure calls (RPC)
- Low: sole responsibility for a complex task in one area
  - E.g. RDBInterface for interacting with relational databases: still too much code
Degrees of Cohesion

- **High:** moderate responsibilities in one area
  - Collaborates with other classes
  - E.g. RDBInterface, but partially responsible
    - Collaborates with a dozen other classes related to RDB access
- The "feel" of a highly cohesive class
  - Relatively small # of operations, w/ highly related functionality
  - Shares the work w/ other classes

Benefits

- Clarity and ease of comprehension
- Maintenance and enhancements are simplified
- Often results in low coupling
- Cohesive classes are easier to reuse
- Contraindications: distributed objects need to be larger, w/ coarse-grain operations
  - Reduces the number of remote calls

System Events

- Who should be responsible for handling an input system event?
  - An event generated by an external actor
  - E.g. word processor: "spell check" button triggers event "perform spell check"
- System sequence diagrams from analysis: conceptual class System handles events
- In design: handling by instances of controller classes

Pattern 5: Controller

- **Facade controller:** a class representing the entire system or device
  - Use case controller: a class representing a use case within which the event occurs
    - e.g. XyzHandler, XyzCoordinator, XyzSession
    - Xyz=name of the use case
    - Handles all system events in the use case

Example

- System events in POS system
  - endSale(), enterItem(), makeNewSale(), makePayment(), ...
  - Typically generated by the GUI

```
[Interface Layer]
:SaleWindow

: ???
[Domain Layer]
enterItem(itemID,qty)
```

Controller Classes

- **Entry points** into the domain layer
  - Isolate the internals of the domain layer
- Facade controller: entire system/device
  - POS_System, Register
- Use case controller: handler for all events in a use case
  - ProcessSaleHandler, ProcessSaleSession
- Can track the state of interactions
  - e.g. order of events
Using Controller Classes

- Facade controller: used when there are not “too many” system events
  - Avoid “bloated” controllers (e.g., too many responsibilities, has too much data)
- Use-case controllers
  - Good when there are many system events
  - Several manageable controller classes
  - Tracking of the state of the current use-case scenario: e.g. to enforce sequencing constraints

Interface Layer

- Interface objects (windows, etc.) should not handle system events
  - The domain layer has the application logic
  - Good for reuse of application logic and UI

Client/Server Applications

- GUI + controllers on the client side
  - GUI sends requests to the controller
    - In the same OS process
    - Controller forwards the request to a remote server
- Systems with Web interface
  - Server-side use-case controllers
    - e.g. for Enterprise Java Beans: often there is a session bean per use case

Examples of Object-Oriented Design

Relationships between Artifacts

- Use cases suggest system events
  - Events are shown in system sequence diagrams
- Effects of events could be described by operation contracts
  - Changes to objects from the Domain Model
- Events correspond to messages that initiate interaction diagrams
- The interactions involve objects that are instances of design classes

SSD for Process Sale

- this event invokes operation makeNewSale
- enterItem(itemID,quantity)
  - description, total
  - *[more items]*
- endSale()
  - total with taxes
- makePayment(amount)
  - change, receipt
**Conceptual vs. Design Classes**

- Conceptual classes represent domain objects; design classes represent software objects.
- Conceptual classes are often mapped to corresponding design classes.
  - But not all design classes are related to the domain model: e.g., use-case controllers.

**Operation makeNewSale**

- Postconditions in the operation contract:
  - A Sale instance s was created
  - s was associated with the Register
  - attributes of s were initialized

  ![Operation Diagram]

- *Who handles the event?*
  - *Controller* pattern

**Controller Pattern**

- Simplified example: only 4 events
  - Use a facade controller
  - If later it gets bloated: use-case controllers
- Design class Register: represents the register from the Domain Model
  - Or: System, POS_System, etc.

  ![Controller Pattern Diagram]

**New Sale Instance**

- Postcondition: new instance of conceptual class Sale was created
- Corresponding design class Sale
- *Who is responsible for creating it?*
  - Two choices
    - Register does it (takes the responsibility)
    - Register asks someone else to do it (delegates the responsibility)

**Creator Pattern**

- A register records (captures) a sale
  - *Creator* pattern: Register is a good candidate
    - Has to know about a sale anyway, so the coupling already exists

  ![Creator Pattern Diagram]

**Container**

- Domain model: a Sale contains a set of SalesLineItem instances
- Design model: the association could be implemented with a container
  - The most common implementation
Creating the Container

- Who creates the container? When?
  - By Creator, Sale is a good candidate
  - Do it when creating a Sale object
  - Alternative: do it when the first SalesLineItem is added (on-demand)

```
makeNewSale():Register 1:create():Sale
  1.1:create():SalesLineItem
```

enterItem(itemId:ItemID, quantity:Integer)

- Postconditions
  - A SalesLineItem instance \( s \) was created
  - \( s \) was associated with the current Sale
  - \( s.quantity \) became \( quantity \)
  - based on \( itemId \) match, \( s \) was associated with a ProductSpecification

- Design class SalesLineItem
  - Based on conceptual class SalesLineItem

Sequence Diagram

- Same semantics, different representation
  - creates an empty container; happens in the constructor of Sale

```
makeNewSale():Register
  1:create():Sale
  1.1:create():SalesLineItem
```

Responsibilities

- Register is the controller
  - Sale creates the new SalesLineItem object (by Creator)
  - Better than creation by Register
  - Register provides \( quantity \) to Sale
  - Sale creates the association
  - By adding the new object to the container

Initial Collaboration Diagram

```
enterItem(id, qty):Register
  1:makeLineItem(qty):Sale
    1.2:add(s):SalesLineItem
  s:SalesLineItem
```

Product Specification

- Based on \( itemId \) match, \( s \) was associated with a ProductSpecification

```
Sale
  date
time

1
Contains
1..* Described-by

SalesLineItem
  quantity

ProductSpecification
  description
  price
  itemID
```
Finding a Product Specification

- Conceptual class → design class
- Who finds the product spec with this id?
  - Expert: who knows all the product specs?

**Domain Model**

<table>
<thead>
<tr>
<th>Product Catalog</th>
<th>Contains 1 Product Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>description price itemID</td>
</tr>
</tbody>
</table>

Product Catalog

- Design class ProductCatalog
- Possible implementation of the association: map ItemID → ProdSpec
  - Message find(id): returns the spec for id
  - Implementation: hash table, binary tree, ...

```
:ProductCatalog p:=find(id) :ProductSpecification
```

Finding a Product Specification

- Who asks ProductCatalog for the spec?
  - Register or Sale
- Possible decision: Register asks
  - No need for each Sale to know about the ProductCatalog
- Connection between Register and ProductCatalog
  - Created during system initialization

Final Collaboration Diagram

```
enterItem(id, qty) :Register

:ProductCatalog 1:p:=getSpec(id) 1.1:p:=find(id) :ProductSpecification

2:makeLineItem(p, qty) :Sale

2.1:create(p, qty) :SalesLineItem

2.2:add(s) :SalesLineItem
```

Observations

- For each system operation
  - Design classes that will implement it
  - Assign responsibilities, operations and attributes
  - Design the interactions, and show with interaction diagrams
- Interaction diagrams do not have to start with system event messages
  - Start with any message for which the designer wants to show interactions

Visibility between Objects

- Object A sends a message to object B
  - B should be visible to A
  - i.e., A should have access to a reference (pointer) to B
- Ensure the necessary visibility
  - If the interaction diagram shows a message, need to choose the appropriate visibility mechanism to make the message possible
Attribute Visibility
- Reference to B is an attribute of A
  - Relatively permanent: often exists for the lifetime of the objects
  - E.g. Register needs to send getSpec(id) to ProductCatalog

```java
class Register {
    private ProductCatalog catalog; ...
}
```

- Attribute visibility from Register to ProductCatalog

Parameter Visibility
- Reference to B is passed as a parameter to a method of A
  - Relatively temporary: exists only for the scope of the method
  - Often transformed into attribute visibility

Example of Parameter Visibility
```java
enterItem(id,qty) :Register
1:p:=getSpec(id) ↓ :ProductCatalog
2:makeLineItem(p,qty) ↓ :Sale

parameter visibility from Sale to ProductSpecification
void makeLineItem(p,qty) {
    s = new SalesLineItem(p,qty);
    inside the constructor, p is assigned to an attribute . . .
}
```

Local Visibility
- B is a local object within a method of A
  - A new B object is created and a reference to it is assigned to a local variable
  - An object reference returned by a call is assigned to a local variable
  - Relatively temporary: only within the scope of the method
  - Often transformed to attribute visibility

Example of Local Visibility
```java
enterItem(id,qty) :Register
1:p:=getSpec(id) ↓ :ProductCatalog

enterItem(id,qty) {
    local visibility from Register to ProductSpecification
    p = catalog.getSpec(id);
    . . .
}
```

Global Visibility
- B is defined in a scope that encloses A’s scope
  - For example: a static field is “global” for all methods inside its declaring class
  - Relatively permanent: typically persists as long as A and B exist
  - Should be used cautiously: may violate the principles of object orientation
Design Class Diagrams

Elements

- **Design classes**: identified while creating the interaction diagrams
  - Often “inspired” by the domain model
- **Attributes**: in the domain model ⇒ attributes in the design classes
- **Methods (operations)**: determined from the messages in the interaction diagrams

Type Information

- **Types of attributes**
  - Usually it is useful to show them
- **Types of method parameters/returns**
  - Could be omitted

“create” messages

- **create** messages: language-independent
- No **create** methods in the design classes
- For many languages: **constructor(s)**
  - Sometimes people do not show constructors in the DCD: reduces the clutter

“get” and “set” methods for attributes

- E.g. for **price** attribute of type **Money**
  - getPrice(): Money
  - setPrice(amt: Money)
- Internal variables that implement the attribute are private and hidden
  - e.g. internally a Point attribute may be a pair of floating-point numbers
- **Get/set** methods are typically not shown
  - Just show the attribute
**Associations in the DCD**

- Based on the interaction diagrams and the domain model
  - Often the associations already exist in the domain model
  - **Will there be an ongoing, somewhat permanent connection between an instance of X and an instance of Y in order to satisfy the interactions?**
- Common cases to consider: X sends a message to Y, or X creates Y

---

**Partial Collab. Diagram for enterItem**

```
enterItem(id, qty): Register
makeLineItem(p, qty): Sale
```

1. `p/:getSpec(id)`
2. `1: create(p, qty)`

---

**Part of the DCD**

```
Register
... enterItem(..) 1 0..1

Sale
date
isComplete: bool
makeLineItem(..)
```

```
ProductCatalog
... getSpec(..)
```

**Navigability**

- Property of an association
  - Shows how it will be implemented
  - **Who is responsible for knowing the association?**
  - **Not part of the domain model**
  - Navigability from Register to Sale: should be able to traverse the association in that direction
    - Register is responsible for knowing the associated Sale, but not vice versa

---

**Navigability**

- Could be 1-way or 2-way
  - 2-way: X \(\rightarrow\) Y
- Not mandatory, but most associations in the DCD should have it
- Implies attribute visibility
  - Will be implemented by an attribute in class Register
  - The attribute is not shown in the DCD; it is implied by the navigability

---

**System Operation makeNewSale**

```
makeNewSale(): Register
create(): Sale
```

```
create(): SalesLineItem
```
Creating a Container

- When creating this interaction diagram, we also considered the domain model

```
Sale
  date
  time
  Contains
    Sales LineItem
      quantity
```

Creating a Container

- Based on the domain model: decided to use a container for SalesLineItems
  - Sale will create the container
  - This will happen when Sale is created
  - Very common case for one-to-many associations: an attribute of Sale refers to the container
    - Attribute visibility from Sale to the container

Representation in the DCD

- Not necessary to show a separate container class
- The navigability implies that Sale has an attribute that refers to a set of SalesLineItem objects
  - i.e. to a container storing these objects

```
Sale
  ...
  Contains
    Sales LineItem
      quantity
      ...
```

Another Example: `enterItem(id, qty)`

- Postconditions from the contract
  - ... new SalesLineItem `s` was created
  - `s` was associated with the current Sale
  - `s` was associated with a ProductSpec
- Sale-SalesLineItem: just insert the SalesLineItem in the container for Sale
  - one-to-many association implemented through a container

SalesLineItem and ProductSpecification

- Domain Model:
  - Based on the interaction diagrams: relatively permanent connection
  - Decision: attribute visibility from SalesLineItem to ProductSpec

```
Sales LineItem
  quantity: Integer

Product Specification
  descr: String
  price: Money
  id: ItemID
```

Design Class Diagram

- Looks a lot like the domain model, but has more details worked out

```
Sale
  ...
  Contains
    Sales LineItem
      quantity: Integer
      ...

Product Specification
  descr: String
  price: Money
  id: ItemID
```
Generalization

- Superclass-subclass relationships
- Used in the domain model and in the design model

![Class Diagram]

Basic Idea

- Domain model: general concept and some specialization
  - "An instance of CashPayment is also an instance of Payment"
- Design: the subclass interface conforms to the interface of the superclass
  - "interface" = public methods and fields
  - The subclass can be used at any place where the superclass is allowed

Abstract Classes and Methods

- Abstract class: instances of it cannot be created
  - Only instances of its subclasses
- Abstract methods (operations)
  - No code: just name, parameter types, and return type
  - Abstract methods must be overridden in subclasses, by concrete methods
    - "concrete" = "non-abstract"
- Various language mechanisms

![UML Diagram]

Accessibility of Methods and Fields

- Public: can be accessed by any code
  - UML notation: +foo
- Private: can be accessed only by code inside the class
  - UML notation: -foo
- Protected: can be accessed only by code in the class and in its subclasses
  - UML notation: #foo
- Fields usually are not public

Instance Methods and Fields

- Instance field: each object of the class has its own copy
- Instance method: invoked on an object
  - Keyword "this" in C++ and Java: a pointer to the object on which the method was invoked
  - Commonly used term: "receiver object"
  - "the object which received the message that invoked the operation (method)"
- All class diagrams seen up to this point show instance fields and instance methods
Static Methods and Fields

- Static field: a single copy for the entire class
- Static method: not invoked on an object
  - Just like a regular procedure (function) in a procedural language (e.g. C, Pascal, etc.)
- Terminology
  - "static method/field" = "class method/field"
  - "instance method/field" = "non-static method/field"

Classic Example (Java)

```java
class X {
    private static int num = 0;
    // constructor
    public X() { num++; }
    public static int numInstances()
        { return num; }
}
```

in main:
```java
X x1 = new X(); X x2 = new X();
int n = X.numInstances(); --> returns 2
```

Classic Example (C++)

```cpp
class X {
    private: static int num;
    public: X();
    public: static int numInstances();
}
```

```cpp
text: X::X() { num++; }
int X::numInstances() { return num; }
in main:
X* x1 = new X; X* x2 = new X;
int num = X.numInstances(); --> returns 2
```

UML Notation

- Private static field
- Public static method
- Constructor

note: "static constructor" is meaningless: by definition, a constructor is invoked on an object

Implementation Model

- UP: code, database definitions, HTML pages, etc.
- Built from the design model: interaction diagrams and DCDs
- Design a little, code a little
- May deviate from the design
  - The design is not perfect
  - In the next iteration: the design will be modified based on the code
- Reverse engineering

Mapping Design to Code

- DCDs -> classes in code
  - DCD: class names, methods, attributes, superclasses, associations, etc.
  - Tools can do this automatically
    - Forward engineering
- Interaction diagrams -> method bodies
  - Interactions in the design model imply that certain statements should be included in a method's body
Example

```java
public class SalesLineItem {
    private int quantity;
    private ProductSpecification productSpec;
    public SalesLineItem(ProductSpecification s, int q) {
        // Constructor implementation
    }
    public Money getSubtotal() {
        // Method implementation
    }
}
```

Another Example

```java
public class Register {
    private Sale sale;
    private ProductCatalog catalog;
    public Register(ProductCatalog c) {
        this.catalog = c; // Constructor implementation
    }
    public void makeNewSale() {
        // Method implementation
    }
    public void enterItem(ItemID id, int qty) {
        ProductSpecification p = this.catalog.getSpec(id);
        this.sale.makeLineItem(p, qty);
    }
    public void endSale() {
        // Method implementation
    }
    public void makePayment(Money amt) {
        // Method implementation
    }
}
```

Java class “Register”

```java
public class Register {
    private Sale sale;
    private ProductCatalog catalog;
    public Register(ProductCatalog c) {
        this.catalog = c; // Constructor implementation
    }
    public void makeNewSale() {
        this.sale = new Sale(); // Method implementation
    }
    public void enterItem(ItemID id, int qty) {
        ProductSpecification p = this.catalog.getSpec(id);
        this.sale.makeLineItem(p, qty);
    }
    public void endSale() {
        // Method implementation
    }
    public void makePayment(Money amt) {
        // Method implementation
    }
}
```

Method makeNewSale

```java
makeNewSale():Register 1:create():Sale
    1.1:*create():SalesLineItem
```

Method enterItem

```java
enterItem(id,qty):Register 2:makeLineItem(p,qty)
    1:p:=getSpec(id):ProductCatalog
```

Reading Assignment

- Detailed example: Ch. 20 of Larman’s book
- For a simplified version of Process Sale
- Important: must read the code and completely understand its meaning
  - See how design decisions have been implemented in code
  - If there is anything in the code that you do not understand, talk with me