Object-Oriented Analysis and Design

PART2: DESIGN
UML class diagrams
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int ...

officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common

plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0 ...

getFont(name : String) : Font
getName() : String ...

interface?
Runnable

run() ...

interface implementation and subclassing

SuperclassFoo
or
SuperClassFoo { abstract }

-classOrStaticAttribute : Int
+publicAttribute : String
-attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }

-derivedAttribute
+classOrStaticMethod()
+publicMethod()
assumedPublicMethod()
#protectedMethod()
-packageVisibleMethod()

-interface? SuperclassFoo( Long )

methodWithParms( parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() (exception IOException)
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

SubclassFoo

-ellipsis 摴? means there may be elements, but not shown
- a blank compartment officially means 摻unknown? but as a convention will be used to mean 摻no members?

3 common compartments
1. classifier name
2. attributes
3. operations

an interface shown with a keyword

interface?
Runnable

run() ...

interface implementation and subclassing

SuperclassFoo
or
SuperClassFoo { abstract }

- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
toBeTruthyAttribute : [Bool] = true
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }

-derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
# protectedMethod()
-packageVisibleMethod()

-interface? SuperclassFoo( Long )

methodWithParms( parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() (exception IOException)
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

SubclassFoo

-ellipsis 摴? means there may be elements, but not shown
- a blank compartment officially means 摻unknown? but as a convention will be used to mean 摻no members?

3 common compartments
1. classifier name
2. attributes
3. operations

an interface shown with a keyword

interface?
Runnable

run() ...

interface implementation and subclassing

SubclassFoo
or
SuperClassFoo { abstract }

- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
.truthAttribute : Bool] = true
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }

-derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
# protectedMethod()
-packageVisibleMethod()

-interface? SuperclassFoo( Long )

methodWithParms( parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() (exception IOException)
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

SubclassFoo

-ellipsis 摴? means there may be elements, but not shown
- a blank compartment officially means 摻unknown? but as a convention will be used to mean 摻no members?
UML: Notating Attributes

using the attribute text notation to indicate Register has a reference to one Sale instance

OBSERVE: this style visually emphasizes the connection between these classes

thorough and unambiguous, but some people dislike the possible redundancy

using the association notation to indicate Register has a reference to one Sale instance
Attributes As Associations

Notice that there are subtle differences between the conceptual perspective (Domain Model) and software perspective (Design Model) for attributes that are defined as associations.

For DCDs, there is usually

- A navigability arrow
- A multiplicity at the target end, but not the source
- A role name
- No association name
UML: Attributes as Associations

The association name, common when drawing a domain model, is often excluded (though still legal) when using class diagrams for a software perspective in a DCD.

UP Domain Model
conceptual perspective

<table>
<thead>
<tr>
<th>Register</th>
<th>1</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>id : Int</td>
<td></td>
<td>time : DateTime</td>
</tr>
</tbody>
</table>

UP Design Model
DCD
software perspective

<table>
<thead>
<tr>
<th>Register</th>
<th>1</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: Int</td>
<td></td>
<td>time: DateTime</td>
</tr>
<tr>
<td>...</td>
<td>1</td>
<td>currentSale</td>
</tr>
</tbody>
</table>

Captures-current-sale
Attributes: Text versus Associations

We explored the idea of data type objects earlier (Domain Models)

- Data types referred to objects for which individual identity is not important
- Recall a *Person* object versus a *Name* data type

One guideline is to use the text for data types (basically primitive types) and associations for more complicated classes

Note that this is a diagram preference – does not matter in the final code
UML: Attributes as Associations

Register
- id: Int
- ...

Sale
- time: DateTime
- ...

Store
- address: Address
- phone: PhoneNumber
- ...

Register has THREE attributes:
1. id
2. currentSale
3. location

applying the guideline to show attributes as attribute text versus as association lines
Attributes: Lists

How do we notate a list of attributes, e.g. an ArrayList in Java?

Two ways to show a collection attribute

<table>
<thead>
<tr>
<th>Sale</th>
<th>SalesLineItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>time: DateTime</td>
<td>...</td>
</tr>
<tr>
<td>lineItems : SalesLineItem [1..<em>] or lineItems : SalesLineItem [1..</em>] {ordered}</td>
<td>...</td>
</tr>
</tbody>
</table>

notice that an association end can optionally also have a property string such as {ordered, List}
Operations and Methods

Operations are usually displayed in the class box with the notation:

\[ \text{visibility name (parameter-list) \{property-string\}} \]

Sometime a return-type is value is added

Assume public if no visibility is shown

An operation is a declaration (name, parameters, return type, exceptions list)

A method is an implementation of an operation

- in sequence diagrams, may show the details and sequence of messages
- In class diagram, usually include some pseudo-code in a note with the \(<<\text{method}>>\) tag
Method Notation in UML

```
method?
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
   ProductDescription desc = catalog.getProductDescription(id);
   sale.makeLineItem(desc, qty);
}
```

Often times constructors (if included) are notated with the <<constructor>> tag.

Usually, getters and setters are ignored in class diagrams.

They are assumed to exist, or are added to the code on an as-needed basis.
Keywords

*Keywords* are textual adornments used to categorize a model element – they provide some additional information about the element.

Usually notated 

```
<<keyword>>
```

, and sometimes 

```
{keyword}
```

Some examples:

```
<<actor>> - this entity is an actor
<<interface>> - this entity is an interface
{abstract} – this is an abstract element, it can’t be instantiated
{ordered} – this set of objects is ordered, e.g. the ArrayList example shown earlier
Abstract Classes

As we saw earlier in Domain Models, UML has the ability to denote *generalization*

Solid line with open, fat arrow; can also notate `{abstract}` in super-class

It represents a relationship between more general classifier and more specific classifier. The specific classifier indirectly has the features of the more general classifier.
In UML, dependency lines can be used in any diagram, but they are especially common in class and package diagrams.

In UML, a general dependency relationship indicates that a client element (class, package, use case, etc.) has knowledge of a supplier element and that a change in the supplier could affect the client.

Indicated by a dashed arrow from the client to the supplier.

Note that we often associate elements with associations (e.g. super- and sub-classes as we just saw), so we do not need to add dependency arrows if an association already exists.

Often used when a class has an attribute of another class type, or if one class sends a message to another class.
Dependency in UML

Guideline: Use dependency in UML to depict global parameter variable, local variable, and static-method call to another class.

```java
public class Sale {
    public void updatePriceFor(ProductDescription description) {
        Money basePrice = description.getPrice();
        //...}
    }
}
```

The `Sale` has parameter visibility to a `ProductDescription`, and thus some kind of dependency.
the `doX` method invokes the `runFinalization` static method, and thus has a dependency on the `System` class.
Dependency Labels

A dependency on calling on operations of the operations of a Clock

A dependency that A objects create B objects
Composition and Aggregation

We saw this earlier in Domain Models ...

*Composition* is a whole-part relationship between model entities, such that
- an instance of the part belongs to only one instance of the composite
- a part must belong to a composite
- the composite is responsible for creating/deleting the parts. (So if the composite is destroyed, the parts are destroyed or become attached to another composite.)

*Aggregation* is a weaker form of composition, where the above requirements are not necessarily true
- Aggregation does not imply ownership of the parts

Composition involves instantiating objects, aggregation involves pointers to other objects.
Composition: Example

- A part instance (Square) can only be part of one composite (Board) at a time.
- The composite has sole responsibility for management of its parts, especially creation and deletion.

Generally look for “has a” associations.
Association Classes

In UML, an association may be considered a class, with attributes, operations, and other features. Include this when the association itself has attributes associated with it.

A person may have employment with several companies.

- Company
- Employs
- Person
- Employment
  - salary
  - startDate
class Company {
    Set<Employment> employments;
}

class Employment{
    Company company ;
    Person person;
    Date startDate;
    Money Salary;
}

class Person{
    Set<Employment> employments;
}

public Set<Person> getPersonnels () {
    Set<Person> result = new HashSet<Person>();
    for (Employment e: employments) {
        result.add(e.getPerson());
    }
    return result;
}
UML interaction diagrams
What will we learn?

UML Interaction Diagrams – What are they, how to create them
UML Interaction Diagrams

There are two types: Sequence and Communication diagrams

We will first look at the notation used to represent these, and then later look at important principles in OO design

We’ll look at various examples here to learn how to create the diagrams
UML Sequence Diagrams

Sequence diagrams are more detailed than communication diagrams.

They often represent a series of method calls between objects in a system.

The sequence is represented in what is called “fence format”, and each new object in the sequence is added to the right in the diagram.

Interactions between objects are usually method calls, but may also be object creation/deletion.

Especially useful for message flow diagrams, with request-reply pairs.
Example: Sequence Diagram

public class A {
    private B myB = new B();
    public void doOne() {
        myB.doTwo();
        myB.doThree();
    }
}

```
```
We would say “The message *makePayment* is sent to an instance of *Register*. The Register instance sends the *makePayment* message to the *Sale* instance. The *Sale* instance creates an instance of a *Payment*.” Here, “message” is a method call.
Interaction Diagrams Are Important

Often left out in favor of class definition diagrams, but these diagrams are important and should be done early.

They describe how the objects interact, and may give clues to the operations and attributes needed in the class diagrams.

These diagrams are part of the *Design Model* artifact, and are started in the Elaboration phase in Agile UP.
Sequence Diagrams: Lifeline Box Notation

Basic notation for the entities that make up the sequence diagram – they are called *lifeline* boxes and represent the *participants* in the particular sequence being modeled.

Note that a participant does not need to be a software class, but it usually is for our purposes.

The standard format for messages between participants is:

```
return = message(parameter: parameterType) : returnType
```

Type information is usually omitted, as are parameters.
sales: ArrayList<Sale>

:Sale

s1 : Sale

Font

eta class?

List is an interface

in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal

sales[i] : Sale

related example

lifeline box representing an instance of an ArrayList class, parameterized (templatized) to hold Sale objects

lifeline box representing a named instance

lifeline box representing the class Font, or more precisely, that Font is an instance of class Class – an instance of a metaclass

lifeline box representing one instance of class Sale, selected from the sales ArrayList <Sale> collection

x : List
Sequence Diagrams: Messages

Messages are notated as solid arrows with filled in arrowheads between lifelines.

The lifelines are the dotted lines that extend below each participant box, and literally show the lifespan of the participant.

The first message may come from an unspecified participant, and is called a “found message”. It is indicated with a ball at the source.

Messages can be synchronous (sender waits until receiver as finished processing the message, and then continues – blocking call) or asynchronous (sender does not wait, more rare in OO designs).

Dashed arrow is used to indicate return of control, e.g. after receipt of synchronous message. May contain a value.
: Register

: Sale

doX

doA

doB

doC

doD

typical synchronous message shown with a filled-arrow line

a found message whose sender will not be specified

execution specification bar indicates focus of control

typical synchronous message shown with a filled-arrow line
Sequence Diagrams: Specifics

The execution specification bar or activation bar indicates that the operation is on the call stack.

Usually replies to messages are indicated with a value or a dotted line (see next slide).

It is possible to have a message to “self” (or “this”).

Sequence diagrams can also indicate instance creation (see later slide).

Likewise, instances can be destroyed (indicated by “X” at the end of lifeline).
Register

Sale

note that newly created objects are placed at their creation "height"

makePayment(cashTendered)

create(cashTendered)

Payement

authorize

: Sale

create(cashTendered)

: Payment

the destroy? stereotyped message, with the large X and short lifeline indicates explicit object destruction
Sequence Diagrams: Specifics

Diagram frames may be used in sequence diagrams to show:

- Loops
- Conditional (optional) messages
- Nesting (a conditional loop)
- Relationships between diagrams

See next slides for examples
a UML loop frame, with a boolean guard expression

: A

makeNewSale

: B

loop [ more items ]

total

description, total

endSale

: A

enterItem(itemID, quantity)

: B

a UML loop frame, with a boolean guard expression, total

describe

makeNewSale

: B

enterItem(itemID, quantity)

: A

calculate

: Bar

opt [ color = red ]

calculate

: Foo

xx

yy
This lifeline box represents one instance from a collection of many salesLineItem objects. The expression \( \text{lineItems}[i] \) is the expression to select one element from the collection of many salesLineItems; the value refers to the same in the guard in the LOOP frame.

An action box may contain arbitrary language statements (in this case, incrementing \( i \))

It is placed over the lifeline to which it applies.

\[
\begin{align*}
\text{lineItems}[i] : & \text{SalesLineItem} \\
\text{lineItems}[i] < & \text{lineItems.size} \\
\text{t} = & \text{getTotal} \\
\text{i}++ & \\
\text{st} = & \text{getSubtotal}
\end{align*}
\]
calculate: Bar
\[ \text{opt}: \text{Foo} \quad \text{loop}(n) \quad \text{calculate} \]
\[ \text{color} = \text{red} \]
interaction occurrence
note it covers a set of lifelines
note that the sd frame it relates to has the same lifelines: B and C
Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation.

---

Payment (abstract)
authorize() (abstract)
...

CreditPayment
authorize()
...

DebitPayment
authorize()
...

---

polymorphic message

object in role of abstract superclass

---

:Register
authorize

stop at this point – don’t show any further details for this message

---

:Payment (abstract)

---

separate diagrams for each polymorphic concrete case

---

:DebitPayment
authorize

doA

:Foo

:CreditPayment
authorize

doX

:Bar

---

Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation.
Example:
Library Information System (LIS)
LIS Requirements and Use cases

R1. The LIS must allow a patron to check out documents.
R2. The LIS must allow a patron to return documents.


How about Allow a Patron? Is it a use case? Who is the actor? What is the goal or business task for the actor? Does it start and end with an actor?
Domain Model

User
- uid : String

Loan
- dueDate : Date

Document
- callNum : String
- available : boolean
LIS UC.1 Text

<table>
<thead>
<tr>
<th>UC1 : Checkout Document</th>
<th>System: LIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor: Patron</td>
<td></td>
</tr>
<tr>
<td>0. The LIS displays the main menu.</td>
<td></td>
</tr>
<tr>
<td>1. Patron clicks the checkout Document button on the main menu.</td>
<td>2. The system displays the checkout menu.</td>
</tr>
<tr>
<td>3. The Patron enters the call numbers of documents to be checked out and clicks the Submit button.</td>
<td>4. The system displays the document details for confirmation.</td>
</tr>
<tr>
<td>5. The patron click the OK button to confirm the checkout.</td>
<td>6. The system displays a confirmation message to patron.</td>
</tr>
<tr>
<td>7. The patron clicks OK button on the confirmation dialog.</td>
<td></td>
</tr>
</tbody>
</table>
msg := verify (uid: String, password: Password) : String
Identify Classes Used in Sequence Diagrams

Identify objects that send or receive messages, passed as parameters or return type.
Classes Identified

- CheckoutGUI
- User
- CheckoutController
- DBMgr
- Loan
- Document
Identify Methods

Methods of CheckoutController:
- checkout(uid, cnList)

Methods of Document:
- setAvailable(false)
- save(d)
- create(u, d)
- save(l)

User(uid): User
- get

Document(cn):
- get
- setAvailable(false)
- save

Loan(cn): Loan
- process(cnList)
- [u! = null]

Controller:
- check-out(uid, cnList)
- msg

Loop (for each cn in cnList):
- process(cnList)
## Fill In Identified Methods

<table>
<thead>
<tr>
<th>CheckoutGUI</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CheckoutController</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkout(uid,cnList)</td>
</tr>
<tr>
<td>process(cn:String[])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;singleton&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMgr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>getUser(uid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDocument(callNo)</td>
</tr>
<tr>
<td>saveLoan(loan)</td>
</tr>
<tr>
<td>saveDocument(book)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isAvailable() : boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAvailable(a:boolean)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loan</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>create(u:User, d:Document)</th>
</tr>
</thead>
</table>

11-51
Identify Attributes

**Checkout GUI**
<<uid, cnList>>

**Checkout Controller**
msg:=checkout(uid, cnList)
Loop (for each cn in cnList)

**DBMgr**
u:=get User(uid):User
[u!=null]
process(cnList)

**Loan**
d:=get Document(cn)
a:=sAvailable()
[a]create(u,d)
[a]save(l)
[a]save(d)
[a]setAvailable(false)

**Document**
d:Document
setAvailable(false)
save(d)

<<msg>>

<<uid, cnList>>
Identify Relationships

GUI

Checkout Controller

DBMgr

Loan

Document

<<uid, cnList>>

msg:=checkout(uid, cnList)

u:=getUser(uid):User

[u!=null]

process(cnList)

d:=getDocument(cn)

disableAvailable()

create(u, d)

[a]save(l)

[a]setAvailable(false)

[a]save(d)

<<msg>>

Loop (for each cn in cnList)

CheckoutController and DBMgr use User.
The dashed arrow lines denote uses or dependence relationships.

```
CheckoutGUI
  display(msg: String)

CheckoutController
  checkout(uid, cnList)
  process(cn: String)

<<singleton>>
DBMgr
  getUser(uid)
  getDocument(callNo)
  saveLoan(loan)
  saveDocument(book)

<<create>>
User
  uid: String

Loan
  dueDate: Date
  create(u: User, d: Document)

Document
  callNum: String
  available: boolean
  isAvailable(): boolean
  setAvailable(a: boolean)
```
From Sequence Diagram to Implementation

```java
public class CheckoutController {
    DBMgr dbm = new DBMgr();
    public void process(String[] cnList) {
        for(int i=0; i<cnList.length; i++) {
            Document d = dbm.getDocument(cnList[i]);
            if (d.isAvailable()) {
                Loan l = new Loan(u, d);
                dbm.saveLoan(l);
                d.setAvailable(false);
                dbm.saveDocument(d);
            }
        }
    }
}
```