Interaction Diagrams
(Chapter 15)

by
Mauricio Monsalve
Design Road

Sample UP Artifact Relationships

Domain Model

<table>
<thead>
<tr>
<th>Sales</th>
<th>Sales LineItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>quantity</td>
</tr>
</tbody>
</table>

Operation Contracts

Operation: enterItem(...) 
Post-conditions: ...

Operation: enterItem(id, quantity)

Design Model

: Register
: ProductCatalog
: Sale

spec = getProductSpec(itemID)
addLineItem(spec, quantity)

System Sequence Diagrams

make NewSale() 
System events

Use Case Diagram

Use Case Text

business modeling
requirements
idea for the post-conditions
system operations

starting events to design for, and
more detailed requirements that must be satisfied by the software

the domain objects, attributes, and associations that undergo changes

Design Road

Supplementary Specification

Glossary

Vision
Interaction Diagrams

- UML interaction diagrams represent interaction (communication, collaboration) between objects/classes
- For dynamic object modeling
- UML interaction diagrams consist of
  - Sequence diagrams
  - Communication diagrams
Sequence Diagram

We have used a simplified version of these for System Sequence Diagrams
Communication Diagram

Steps are enumerated and placed in lines with arrows
## The diagrams compared

**Sequence diagram**
- clearly shows sequence or time ordering of messages
- large set of detailed notation options
- forced to extend to the right when adding new objects; consumes horizontal space

**Communication diagram**
- space economical; flexibility to add new objects in two dimensions
- more difficult to see sequence of messages
- fewer notation options
Exercise

makePayment(cashTendered)

: Register

makePayment(cashTendered)

: Sale

create(cashTendered)

: Payment
Exercise

```
Register

makePayment(cashTendered)

makePayment(cashTendered)

Sold

Payment

create(cashTendered)

: Payment

direction of message

makePayment(cashTendered)

: Register

1: makePayment(cashTendered)

: Sale

1.1: create(cashTendered)

: Payment
```
Drawing Sequence Diagrams

- Lifeline box representing an instance of an `ArrayList` class, parameterized (templatized) to hold `Sale` objects.
- Lifeline box representing an unnamed instance of class `Sale`.
- Lifeline box representing a named instance.
- Lifeline box representing one instance of class `Sale`, selected from the `sales` `ArrayList` `<Sale>` collection.
- Lifeline box representing the class `Font`, or more precisely, that `Font` is an instance of class `Class` – an instance of a metaclass.
- `sales`:
  - `ArrayList<Sale>`
  - `related example`
- `sales[i]`:
  - `Sale`
- `x`:
  - `List`

In UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal.
In the case of singleton objects/classes, we put a “1” on their boxes.

Singleton classes are the ones that only have one instance.

- Cf. Scala: singleton defined with “object”, not “class”
Drawing Sequence Diagrams

- **Register**
  - `doX` to `Sale`
  - `doA`
  - `doB`
  - `doC`
  - `doD`

- **Sale**
  - `doA`
  - `doB`
  - `doC`
  - `doD`

**Annotations:**
- A **found message** whose sender will not be specified.
- An **execution specification** bar indicates focus of control.
- A **typical synchronous message** shown with a filled-arrow line.
Two ways to specify a return value. The first one is brief. The second one allows one to describe the information contained in the returned value.
Drawing Sequence Diagrams

```
: Register

makePayment(cashTendered)

: Sale

create(cashTendered)

: Payment

authorize

note that newly created objects are placed at their creation "height"
```
Drawing Sequence Diagrams

Vertical “presence” or coverage demonstrates the life-cycle of an object
Drawing Sequence Diagrams

Types of frames:
- **loop**—for repeated statements,
- **opt**—for if-statements without else,
- **alt**—for if-statements with else or else-if,
- **par**—for parallel execution,
- **region**—for critical region (concurrency).

A UML loop frame, with a boolean guard expression.
Drawing Sequence Diagrams

: A

: B

: C

doX

alt

[ x < 10 ]
calculate

[ else ]
calculate


Drawing Sequence Diagrams

This lifeline box represents one instance from a collection of many `SalesLineItem` objects.

`lineltems[i]` is the expression to select one element from the collection of many `SalesLineItems`; the ‘i’ value refers to the same “i” in the guard in the LOOP frame.

The `t = getTotal` action box contains an arbitrary language statement (in this case, incrementing ‘i’).

Vertical “presence” or coverage demonstrates the life-cycle of an object.
Drawing Sequence Diagrams

: Foo

: Bar

xx

opt: [ color = red ]

loop(n)

calculate

Nesting of frames
**Drawing Sequence Diagrams**

Interaction occurrence:
- Note it covers a set of lifelines:
- Note that the sd frame it relates to has the same lifelines: B and C.

**sd AuthenticateUser**
- AuthenticateUser
  - authenticate(id)

**sd DoFoo**
- DoFoo
  - doY
  - doZ

**doX**: A
- doA: B
- doB: C

**authenticate(id)**
- B
- C

**doM**: 1
- B
- C

**doM1**: C
- B
- C

**doM2**: B
- C
- B
- C
Polymorphism

*Payment* is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation.

---

**Diagram:**

- **Payment (abstract)**
  - authorize() (abstract)
- **CreditPayment**
  - authorize()
- **DebitPayment**
  - authorize()

---

**Message:**

Polymorphic message

---

**Register**

**Authorize**

---

**DoX**

---

**DebitPayment**

**Authorize**

---

**Foo**

---

**CreditPayment**

**Authorize**

---

**Bar**

---

Separate diagrams for each polymorphic concrete case.
Communication Diagrams

- Numbering follows legalistic ordering
  - $1 < 2 < 2.1 < 3 < ...$
Three ways to show creation in a communication diagram

Create message, with optional initializing parameters. This will normally be interpreted as a constructor call.

1: `create(cashier)`

If an unobvious creation message name is used, the message may be stereotyped for clarity.
unconditional after either msg2 or msg4

1a and 1b are mutually exclusive conditional paths

1a [test1] : msg2

1b [not test1] : msg4

1a.1: msg3
iteration is indicated with a * and an optional iteration clause following the sequence number
Communication Diagrams

This lifeline box represents one instance from a collection of many `SalesLineItem` objects.

`linItems[i]` is the expression to select one element from the collection of many `SalesLineItem`s; the `i` value comes from the message clause.

Less precise, but usually good enough to imply iteration across the collection members.

This iteration and recurrence clause indicates we are looping across each element of the `linItems` collection.
Concurrency

A stick arrow in UML implies an asynchronous call. A filled arrow is the more common synchronous call.

In Java, for example, an asynchronous call may occur as follows:

```java
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

The asynchronous `start` call always invokes the `run` method on the `Runnable (Clock)` object.

To simplify the UML diagram, the `Thread` object and the `start` message may be avoided (they are standard “overhead”); instead, the essential detail of the `Clock` creation and the `run` message imply the asynchronous call.

Note the dependency with the programming language. For the sake of abstraction and generality, you may want to express concurrency in its simplest form here.
Concurrency

startClock

1: create

2: run

:Clock

:ClockStarter

3: runFinalization

System : Class

asynchronous message

active object
Credits

Notes and figures adapted from