The analysis and design of a Smalltalk application.

The OO A&D Process

The Spiral Model: No one will dispute that o-o programming is a different way to develop programs from traditional methods.

However, experts disagree about the best way in which to develop designs that will be implemented in OO languages.

Many different design methodologies are popular, each starting from a slightly different perspective, but (hopefully) arriving at the same goal — a detailed design that can be turned into code.

Many of the more popular design methods explain the process of OO design as a series of successive refinements:

- A project starts from high-level understanding of the problem in analysis.
- The objects and responsibilities found in these earlier stages are refined and added to throughout the process, eventually being implemented in code.

Another common theme found in many OO processes is the notion that OO A&D is not done in a single “waterfall” progression from analysis through design to coding.

Instead, lessons learned at each stage feed back into the process.

In fact, it may be that a large system can be tackled in stages, with all three stages being carried on in a cycle.

This kind of iterative development is sometimes referred to as the “spiral model” of software development.

While not unique to OO development, this model has found most of its adherents among OO enthusiasts.

We will be using UML, the Uniform Modeling Language.

Our process will consist of several steps.
1. We will review the business rules and problem statement for our example

2. We will discuss the use cases that comprise our example

3. We will perform an analysis using CRC cards on our problem domain

4. We will UML notation and methodology to flesh out our design to the point that we could code it in Smalltalk.

   Specifically we will use Class Diagrams to show the relationship between instances of our classes, and Collaboration Diagrams to show how objects send messages to each other.

Over the next three lectures we will be using a single running example. We will start with an OO analysis and design, then proceed to coding the domain classes and the user interface in Smalltalk.

Problem Statement: The example is the “Mom & Pop’s Video Club.”. The statement of work is shown below.

Statement of Work

Mom & Pop’s Video Stores, Inc. has decided to replace its error-prone manual system of keeping track of video rentals with a more accurate automated one.

They have contracted with us to build this new system. Interviews with the customer have shown us the following requirements for the system:

1. A customer must be a member of the video club to rent a video.
2. At the time of joining the video club the club takes the customer’s name, his/her address and a credit card number. Currently we do not bill the credit card for overdue fines, although we would like to add this in the future. Each customer is assigned a unique club number and a printed club ID card bearing that number.

3. Customers can rent as many videos at a time as they like

4. All videos are identified by a unique tape number. We may have several copies of the same movie, in which case each video will have its own number.

5. Any overdue fines that are due must be paid before renting any more videos. Paying overdue fines can be part of a transaction in which more videos are rented.

6. Overdue fines are $1 per day per video after the due date of the video. There is no maximum.

7. “New releases” rent for $2.50 per night and are due the next day.

8. All other videos rent for $2.00 per night and are due two days after the day of rental.

9. All videos belong to a category. They begin in the “new releases” category and are recategorized later.

10. Rental fees are taxable with 5% state sales tax. Overdue fines are not taxable.

11. The video rental system will not act as a Point-Of-Sale (POS) system. There is a separate set of POS terminals not connected to the system.

OO Analysis:

Use-case scenarios: The process of setting down requirements from which to do an OO analysis is itself a complicated thing.

Some developers have had success using traditional methods of requirement specification, but others have found that the assumptions in these methods make it difficult to make the transition to OO A&D.
One of the more popular ways of setting down detailed requirements is the idea of scenarios.

A use case is a conversation between the system and the external actors. It has three parts — the preconditions, the detailed steps, and the postconditions.

Example use cases: In our system, conversations with the customer have resulted in the following use cases:

- Rent videos
- Pay for videos and overdue fees
- Join Mom & Pop’s video club
- Return a video
- Enter a new video into the system
- Search for a video

These are the different paths through the system. They each are initiated by some external actor (a person or another system) and represent one way that the system will be used.

From the use-case names above, we can develop more detailed scenarios of how the actor interacts with the system.

For instance, let’s look at one basic use case, renting videos.

**Name:** Rent a Video

**Preconditions:**
The customer has selected the video(s) they want to purchase.
The customer has an account with Mom & Pop’s Video Stores, Inc.

**Detailed steps:**
1. The customer walks up to the clerk with the video.
2. The clerk tells the system to start a new rental.
3. The system asks the clerk for the customer’s card number.
4. The system provides the clerk with the customer information.
5. The clerk tells the system to add videos to the rental.
6. The clerk asks the system to check for unpaid overdue fines.
7. The clerk tells the system to complete the rental.

**Postconditions:**
The rental is created in the system.
The customer is ready to pay for the rental.

The other scenarios are listed in the appendix. We will refer to some of them during the course of our example.

**CRC cards:** The next step is to take the scenarios developed during the previous step and use them to start the CRC process. Remember that CRC is generally concerned with two things:

1. Identifying the objects that comprise the system.
2. Identifying the responsibilities of those objects and the way in which objects interact to fulfill those responsibilities.

We will be concerned with identifying classes, responsibilities, and collaborators in our system.

Note that CRC is a high-level analysis technique. It is not meant to produce a detailed design. However, the CRC cards can be used to help specify the next level in a design.

With this in mind, there are two things to remember about the CRC technique:

1. CRC intentionally blurs the distinction between data and function.
2. CRC intentionally blurs the distinction between class and instance.

CRC focuses on

• distributing responsibility between objects,
• understanding complex interactions, and
• identifying the candidate objects, including the non-obvious ones.

**Identify the objects to be modeled**

The first step of CRC is to identify the classes that are in our system and for which we will find responsibilities.

In our system there are four obvious classes, and one almost-obvious one.

• Movie – A movie is just that; a movie. Examples are “The Bridges of Madison County” or “Jaws”.

• __________ – A __________ is a particular copy of a movie. A video store may have many videotapes of each movie.

• __________ – A __________ is the system’s representation of a video-club member.

• Address – An address is used by the system to know where the customer lives.

• __________ – This one is not as obvious; it represents a category of movies like “comedy” or “new release.”

We would at this point make CRC cards for these classes. If there are some responsibilities we know of immediately, we can fill them in now. For instance, what are some of the responsibilities of Address?

<table>
<thead>
<tr>
<th>Class Name: Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Not all classes in a system are as obvious as the previous ones. In a real CRC session, the need would become obvious for two more classes:
• CustomerTransaction – This represents a single transaction with a customer.

For instance, we would need to record the fact that a customer came in on a date and rented four videos and paid the overdue fine on one he kept a day late.

• TapeRental – Even more subtly, this is the connection between a particular videotape being rented to a particular customer. A customer can rent four videos and return them at different times; thus we need a distinction between this and the CustomerTransaction.

The difference between these two sets of objects is that the first are “physical” objects and the second are not.

The second kind of objects are more challenging to discover because they don’t have counterparts in the “real world”.

Finding responsibilities for classes

The use-case scenarios that we looked at earlier can now be used to guide the assignment of responsibility to the objects in our system.

It’s often useful to record what happens in a CRC session as some sort of script as we go along. This script will change regularly!

For example, let’s look at the following script of our use case.

<table>
<thead>
<tr>
<th>Step</th>
<th>Sender</th>
<th>Receiver</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clerk</td>
<td>Customer</td>
<td>find customer for ID#</td>
</tr>
<tr>
<td>2</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>make a customer transaction</td>
</tr>
<tr>
<td>3</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>set customer</td>
</tr>
<tr>
<td>4</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>rent a video to a customer</td>
</tr>
<tr>
<td>5</td>
<td>CustomerTransaction</td>
<td>Videotape</td>
<td>find a tape for an ID#</td>
</tr>
<tr>
<td>6</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>find overdue tapes</td>
</tr>
<tr>
<td>7</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>get total due</td>
</tr>
<tr>
<td>8</td>
<td>Clerk</td>
<td>CustomerTransaction</td>
<td>finish out transaction</td>
</tr>
</tbody>
</table>

Note that this version of the script is at a high level of abstraction—it is not very detailed. It mainly concentrates around one or two major objects in the system.
We haven’t yet fleshed out the collaborators with those objects.

It is a common mistake among novice OO designers to stop their design at this level of detail. If your system only has a few, large-grained objects it will be more difficult to find reusable components within it, and harder to maintain!

**Review completed CRC cards**

Let’s now look at the completed set of CRC cards for our system. These would be found as a result of doing CRC exercises for all of the scenarios identified in the previous steps.

<table>
<thead>
<tr>
<th>Class Name: Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>know your name</td>
</tr>
<tr>
<td>know your customer number</td>
</tr>
<tr>
<td>know your address</td>
</tr>
<tr>
<td>know your current rentals</td>
</tr>
<tr>
<td>return a customer for an ID#</td>
</tr>
<tr>
<td>tell what rentals are overdue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Name: Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>know your name</td>
</tr>
<tr>
<td>know your movies</td>
</tr>
<tr>
<td>provide your rental fee</td>
</tr>
<tr>
<td>provide your rental period</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Name: Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>know your tapes</td>
</tr>
<tr>
<td>know your title</td>
</tr>
<tr>
<td>know your year of production</td>
</tr>
<tr>
<td>create new tapes for yourself</td>
</tr>
</tbody>
</table>
**Class Name: Videotape**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>know your tape id</td>
<td></td>
</tr>
<tr>
<td>know the rental you are participating in</td>
<td></td>
</tr>
<tr>
<td>return yourself to service (be rentable)</td>
<td></td>
</tr>
<tr>
<td>return a tape for an ID#</td>
<td></td>
</tr>
</tbody>
</table>

**Class Name: TapeRental**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>know your tape</td>
<td></td>
</tr>
<tr>
<td>know the date you were ret’d</td>
<td></td>
</tr>
<tr>
<td>know your originating order</td>
<td></td>
</tr>
<tr>
<td>determine if you are overdue</td>
<td>CustomerTransaction</td>
</tr>
<tr>
<td>calculate overdue fine</td>
<td>CustomerTransaction</td>
</tr>
<tr>
<td>calculate rental fee</td>
<td>Videotape</td>
</tr>
<tr>
<td>handle tape returns</td>
<td>CustomerTransaction</td>
</tr>
</tbody>
</table>

**Class Name: CustomerTransaction**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>know your customer</td>
<td></td>
</tr>
<tr>
<td>know your rental date</td>
<td></td>
</tr>
<tr>
<td>know your rentals</td>
<td></td>
</tr>
<tr>
<td>hold a customer’s overdue fines</td>
<td>Customer</td>
</tr>
<tr>
<td>calculate total due</td>
<td>TapeRental</td>
</tr>
<tr>
<td>rent tapes</td>
<td>Videotape</td>
</tr>
<tr>
<td></td>
<td>TapeRental</td>
</tr>
</tbody>
</table>

*Class diagrams:* UML class diagrams show the composition and inheritance relationships between the objects in a system.
If you are designing a system to be built in Smalltalk many parts of the UML notation will not be needed.

- UML has special visibility options to show whether attributes are public, private, or protected. There is no way to specify this in Smalltalk.

In general, a subset of the UML class notation consisting of class rectangles, attributes, inheritance, and composition and aggregation relationships is enough.

After completing a set of CRC cards, it is straightforward to build a UML class diagram for the classes represented by the cards.

Each card will become one class rectangle with the same name.

If a class has a responsibility that says it “has a something” (composition) or “knows its something” (aggregation) then that class has a has-a relationship with the other class.

This can be represented by drawing a has-relationship relationship line starting from the class that implements the responsibility.

For instance, in our system a Customer “knows his address.” We would connect the Customer and Address clouds with a has-a association starting from customer.

The direction of the relationship is indicated by the black dot on the starting class.

The completed class diagram is shown below.

Note that we did not use inheritance in our design. That is just because the nature of our design did not dictate it. If we had found that several CRC cards implemented similar responsibilities, we could have created an abstract class from which our concrete classes would descend.

This sort of stepwise refinement is the nature of the OOA&D process.
Collaboration diagrams: UML class diagrams only capture part of the nature of an OO software design.

They capture the “static” relationships between the classes, represented by the values of the instance variables.

The sequence diagram or collaboration diagram captures the interaction of objects in a scenario: Sequence diagrams “freeze” the dynamic interactions of objects from CRC. They show the sequence of messages between objects.

In UML notation, the sequence and collaboration diagrams are basically equivalent.

Let’s take the steps of our use case one at a time and match what we found previously in the CRC cards onto an object-message diagram. We will begin with creating a Customer Transaction.
Note that the sequence of operations in this diagram follow what we found in the CRC session. However, there are a few differences.

- Instead of simply naming the general responsibility that each object has, we have chosen a specific message name in valid Smalltalk syntax.

  Each responsibility on a CRC card may end up corresponding to several message names.

  For instance, the responsibility of Customer to “know your address” would correspond to the Smalltalk messages `#address` and `#address:`.

- Another slight difference is that in the CRC script we referred to the initiator of external messages as being the "Clerk". Here we have replaced that with another object (yet to be fleshed out) called the Rental UI.

  In a later lecture we will see how this class is fleshed out and implemented in Smalltalk.

To see the difference between a UML collaboration diagram and a sequence diagram, let’s look at the sequence diagram for the previous sequence of messages.

The major difference is that the objects are vertical lines, while the messages are horizontal lines. This makes it easier to see the sequencing of the messages. Which diagram you use is largely a matter of preference.
Continuing along our path, let’s look at the next point in our script – renting a video.

Let’s walk through the steps of this one and see how it differs from what we saw earlier.

- The first step is the same as in the CRC – tell the CustomerTransaction to add a tape for an ID.
- The second step is also the same – look up the instance of Videotape for the provided ID.

From then on, it becomes more detailed. We must next create an instance of TapeRental and set up its tape and originatingTransaction attributes. Finally, we add this new rental to the CustomerTransaction.
It is easy to see why a TapeRental would need to know its tape; after all, that is its reason for existing. However, why does it know a CustomerTransaction?

Well, let’s think about what we’ve already done. Each CustomerTransaction knows the Customer that started that transaction. It also knows the date on which that transaction occurred.

We could duplicate both of these pieces of information in each TapeRental, but it would be simpler and less error-prone to just let each TapeRental know which CustomerTransaction created it.

That way when the TapeRental needs to know those pieces of information, it can query its originating transaction for them.

Now let’s look at one of the last pieces, calculating the total due.

Here we start again with asking the CustomerTransaction to calculate the totalDue. However, it does that by asking each of its VideoRentals for its “feeOrFine”.

If a rental is overdue, it would calculate its overdue fine. In the case where it is a new rental it would return the rental fee.
Since the rental fee of all copies of a movie is the same, it would ask its Videotape, which would, in turn, ask its category, since that is where the fees are actually determined.

No object had all the information necessary to determine the rental fee. It wouldn’t make sense to have the CustomerTransaction or some other object send all the messages—this would violate encapsulation.

As the last step in the above process, it would also calculate the sales tax. Note that this is a separate set of calculations because the Clerk would probably like to see both the sales tax and the total due printed on separate lines of a receipt.

In any case, it is very similar to the preceding.

Note the above collaboration diagram shows the RentalUI asking for the sales tax.

The message flow would be the same even if it were the CustomerTransaction asking for the sales tax as in the previous diagram.

Remember that one of our rules was that overdue fines are not taxable. Thus the message #salesTax in VideoRental must check to see if it is overdue; if it is then it returns 0 for the sales tax.
Summary: In this lecture we have looked at an overall A&D process using use cases, CRC, and the UML notation.

We have followed the progress of our video store example from high-level requirements gathering through analysis using CRC cards through detailed design using UML class diagrams and collaboration diagrams.

In the next lecture we will see how this information is used to develop Smalltalk code for our system.

Appendix:

Use-case scenarios: The following use-case scenarios are not covered in the lecture, but form part of the requirements for our problem. The CRC cards incorporate the responsibilities found in them.

Name: Pay for rental and overdue fees

Preconditions:
The rental has been completed.
The system knows who is renting the videos.

Detailed Steps:
1. The system calculates the total amount of rental fees.
2. The system calculates the sales tax for the rental fees.
3. The system calculates the overdue fees due for the customer.
4. The system informs the clerk of the total amount due from the customer.
5. The customer pays the clerk
6. The clerk enters the payment information into the system.

Postconditions:
The rental transaction is completed.
Any overdue fees are paid.

Name: Customer Joins Video Club

Preconditions:
The potential customer is not a video-club member.

Detailed Steps:
1. The clerk asks the system to create a new customer.
2. The system asks the clerk for the customer's name.
3. The clerk enters the customer's name into the system.
4. The system asks the clerk for the customer's address.
5. The clerk enters the customer's address into the system.
6. The system generates a new video club number.
7. The system prints out a membership card with the new number.

**Postconditions:**
The system is aware of the new customer.

**Name:** Return a video

**Preconditions:**
A customer has previously rented a video

**Detailed Steps:**
1. The clerk tells the system a video has been returned.
2. The system asks the clerk for the video number.
3. The system updates the customer account information.

**Postconditions:**
The videotape is available for rental

**Name:** Enter a new movie into the system

**Preconditions:**
No videos with the same title made in the same year exist.

**Detailed Steps:**
1. The clerk tells the system he wants to create a new video.
2. The system asks the clerk for the title of the video.
3. The system asks the clerk for the year the movie was made.
4. The system asks the clerk for the category of the video.
5. The system asks the clerk for the number of tapes available.
6. The system prints video labels for the videos.

**Postconditions:**
The system is aware of the new videos.

**Name:** Search for a Video

**Preconditions:**
The video exists in the system
**Detailed Steps:**

1. The user (clerk or customer) asks the system to show the categories available.
2. The system displays the categories of the videos.
3. The user asks the system to show all the titles in the category.
4. The system displays the video titles in the category.
5. The user selects a video title to view in more detail.
6. The system displays the number of tapes available, and the number rented.

**Postconditions:**

The system has shown the user the tape category, title and dispositions.

**Other collaboration diagrams:** The following collaboration diagrams show part of the detailed design for the above use cases. While we may not cover them in class, they form part of the design of our system.

The following diagram shows the process to find any overdue rentals that a customer has. Remember that a customer must pay all overdue fines before he can rent new videos.

The following diagram shows the sequence of events in returning a video before its due date. Returning a video after its due date would differ slightly in that step 7 would not occur.
1. tapeForId
2. returnTape
3. currentRental
4. returnTape
5. isOverdue
6. customer
7. removeRental