Object Reloaded

Jun Hu
Department of Industrial Design
Eindhoven University of Technology
j.hu@tue.nl
http://id00243.id.tue.nl/ObjectOrientationAndDesignPatterns

7th March 2005

▶ Contents

▸ Introduction: What's the big deal?

Object-Orientation is a different way of looking at the world.

Suppose you're at the dinner table, you would like some salt, and the salt shaker is inconveniently located at the other end of the table.

You could try the *Procedural* way:

- Please remove your right hand from your wine glass;
- move it to the left until it contacts the salt shaker;
- grasp it;
- lift it from the table;
- move it in a natural arc in my direction;
- stop when it contacts my hand;
- wait until my hand has closed around it;
- then release it.

Or better, the "Object-oriented" way:

- say to your friend "Please pass me the salt", and she would do so.

▶ The basic OO principle

Objects ("salt-passers") are responsible for their own actions in responding to requests from clients ("salt-wanters").

Let's try another example: Suppose you're writing a software system for a bank, and you need to know the balance of a particular customer's account.

Procedural Starting with the balance at the start of the month, read each transaction in turn, and add or subtract its value as appropriate.

Object-oriented There will be an object for the customer's account, and you simply ask it to return you the balance.

Remember the single most important principle of object-orientation: *Objects are responsible for their own actions*. And it is called "***Encapsulation***"

▶ Encapsulation

Procedural thinking is based on performing actions on data
Object-orientation is based on the data (objects) performing actions on themselves

- Model Real-World Objects
- Isolate Knowledge
- Protect System from Change
- Use Behavior from Other Objects

▶ Encapsulation ▶ Model Real-World Objects

- we're describing objects in the real world more closely, you don't have to give detailed instructions to a friend to pass you the salt (because she knows how to do it).

- Structured analysis also models the real world. The difference is that these methods focus on *data flow*, while object-oriented techniques focus on *objects*.

- The advantage of modeling objects in the real world is that you reduce the amount of work you have to do when your requirements change

- Why? Because, as a rule, objects in the real world don't change (and even when they do, they don't change *much*)

▶ Encapsulation ▶ Isolate Knowledge

The knowledge of how a service is performed by an object is kept with the object itself.

- Your obliging friend picks up the salt and passes it to you.
- Another friend might have chosen to slide it across the table.
- Another might have picked it up, stood up, walked around the table and handed it to you.
- Yet another might have checked the salt shaker, found it empty, stood up, walked to the cupboard, refilled the shaker, and then passed it to you.

The point is that if your objective was to get the salt from your friend, you don't care exactly how your friend gets it for you.

▶ Encapsulation ▶ Protect System from Change

This separation of *what* from *how* is important for another reason - it makes the system easier to change later.

- Look how much detailed knowledge about salt-passing you're using to make your request.
- You're not only making a request, you're also insisting that it be carried out in a specific way.
- If your friend later comes up with a different way of passing the salt (e.g. balancing it on two toothpicks), then you (and everybody else) will have to change the way you make your request.
- If, on the other hand, you leave it up to her to decide how to pass the salt, if she comes up with a new, more effective, more efficient, faster, cheaper, funnier, or more creative method, you still get your salt.

Customer requirements change all the time, and your product needs to change with them.

▸ Encapsulation ▸ Use Behavior from Other Objects

One more advantage of encapsulation is something called ***polymorphism.*** More than one kind of object can fulfill a request. The requestor doesn't know and doesn't care which kind of object actually fulfilled a particular request.

- When you say "Pass the salt", your friend gives it to you.
- Or it could be that somebody from the other side of the table reaches over and hands it to you.
- Or perhaps your well-trained Labrador retriever jumps up, grabs the salt shaker in its mouth, and brings it to you.
- Or maybe you have invited the town robot over to dinner that night, and it reaches over with its metallic arm to pass you the salt.

Does it matter? No. The end result is the same - a salt shaker in your hand. That's ***polymorphism*** for you.

▸ Inheritance

In the real world, we often classify things by how similar they are to each other. By classifying kinds of objects by their similarities, we can talk about their common properties. Let's take mountains, for example.

- If you and I both understand the concept of "a mountain" even vaguely as a pretty big pile of earth with a pointy bit at the top, then we can communicate pretty effectively about mountains.
- So we understand what all mountains have in common, but of course mountains like Everest are also different from each other . So we might now want to talk about a volcanic mountain. So we simply define what makes it so special: Hot red liquid can flow out of the pointy bit.
- Here's the crucial point: To understand about volcanic mountains, we only need to add to our existing knowledge of mountains. We can say that "volcanic mountains" inherit all the properties of "mountains", and then adds its own properties.

That's ***inheritance*** for you.

▸ Inheritance

Let's look at a software example. Suppose you've been asked to write a software system to keep track of all a bank's customers, and you're in the process of designing a "Bank Customer" object.

- Suddenly you remember that in your previous system, you already have a "Customer" object, with properties like: name, postal address, phone number, fax number, and so on.
- In building your Bank Customer object, you simply re-use all of the Customer object, and then add the new properties - say, credit rating and security password..

In OO jargon, Bank Customer *inherits* the properties of Customer (or *is derived from* Customer).

- It allows us to capture the similarities and the differences between classes of objects.
- It can increase reusability greatly.

▸ Again, what's the good of it?

Object-oriented techniques can help software/product development in quite a few areas:

- Modeling of real-world objects makes it easier to describe and communicate behavior.
- Encapsulation of knowledge means that behavior can be isolated. This in turn means that changes in requirements can be accommodated without affecting the entire system.
- Inheritance allows us to re-use objects that have been created already.

These are all good things , but none of these is the most significant impact of object-orientation on the design and development process.

▸ Structured Development

Before object-oriented computing, many people used this approach in developing software:

- Start with a structured analysis.
- Develop a modular design.
- Write procedural programs.

What's the problem? Think of building your dream house using this approach ...

The crux of the problem is using different techniques for analysis, design and implementation. It's very difficult to accommodate requirements changes late in the development process, because it's just so difficult to work out what parts of the program code are affected by a change in the requirements.

▸ Object-orientation: A Consistent Model

Object-oriented computing allows us to use the same model throughout the entire software development process.

- Start with an object-oriented analysis.
- Convert this into an object-oriented design.
- Convert this into object-oriented programs.

As you can see, an object-oriented approach is used each step of the way. In fact, the same object-oriented model that was developed right at the start is used in the program code.

Yes, there are differences here and there to do with particular implementation details (just as an architect doesn't worry about how the bricklayers mix the cement), but the two models are essentially the same.

▸ Rapid Application Development

- Requirements gathering
- Analysis
- Design
- Development
- Deployment

▸ Rapid Application Development ▸ Requirements gathering

- **♡Discover Bussines Processes: Activity diagram(s).**
- **♡Perform Domain Analysis: High-level class diagram and a set of meeting notes.**
- Identify Cooperating Systems: Deployment diagram.
- Discover System Requirements: Package diagram.
- **♡Present Results to Client**

▶ Rapid Application Development ▶ Analysis

- ♡**Understand System Usage: Use case diagram(s).**
- ♡**Refine the Class Diagrams: Refined class diagram.**
- ♡**Analyze Changes of State in Objects: State diagram.**
- ♡**Define the Interactions Among Objects: Sequence and collaboration diagrams.**
- Analyze Integration with Cooperating Systems: Detailed deployment diagram and if necessary data models.

▶ Rapid Application Development ▶ Design

Design and Analysis should go back and forth until the design is complete.

- ♡**Develop and Refine Object Diagrams: Activity diagrams.**
- Develop Component Diagrams: Component diagrams.
- Plan for Deployment: Part of the deployment diagram developed earlier.
- ♡**Design and Prototype User Interface: Screen shots of the screen prototypes.**
- Design Tests: Test scripts.
- Begin Documentation: Document structure.

▶ Rapid Application Development ▶ Development

- Construct Code: The code.
- Test Code: Test results.
- Construct User Interfaces, Connect to Code and Test: Functioning system, complete with user interfaces.
- Complete Documentation: System documentation.

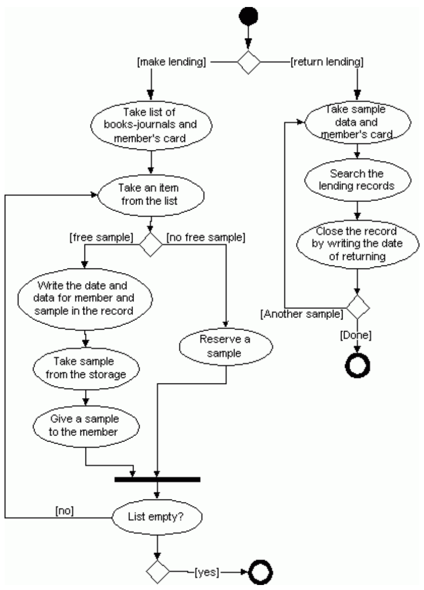
▶ Rapid Application Development ▶ Deployment

- Plan for Backup and Recovery: The crash recovery plan.
- Install the Finished System on Appropriate Hardware: Fully deployed system.
- Test the Installed System: Test results.
- ♡**Celebrate!**

▶ A Simplified Practical Process

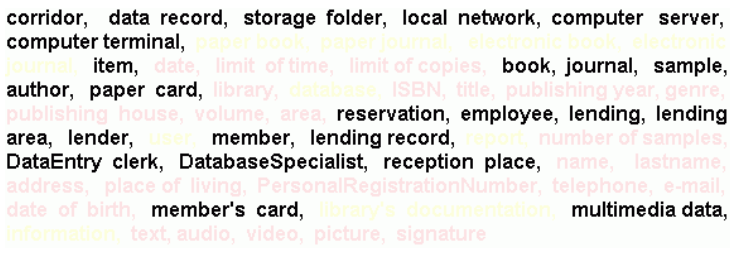
- Requirements gathering: Discover Business Processes
- Analysis: Identify objects/attributes/behaviors
- Analysis: Find out classes and their relations
- Analysis: Refine classes and their relations
- Analysis: Dynamic behavior of classes
- Design

▶ A Simplified Practical Process ▶ Requirements gathering: Discover Business Processes


[make lending]
[return lending]
Take list of books-journals and member's card
Take sample data and member's card
Search the lending records
Take an item from the list
Close the record by writing the date of returning
[free sample]
[no free sample]
Write the date and data for member and sample in the record
[Another sample]
[Done]
Reserve a sample
Take sample from the storage
Give a sample to the member
[no]
List empty?
[yes]

▶ A Simplified Practical Process ▶ Analysis: Identify objects/attributes/behaviors

Work on the nouns:

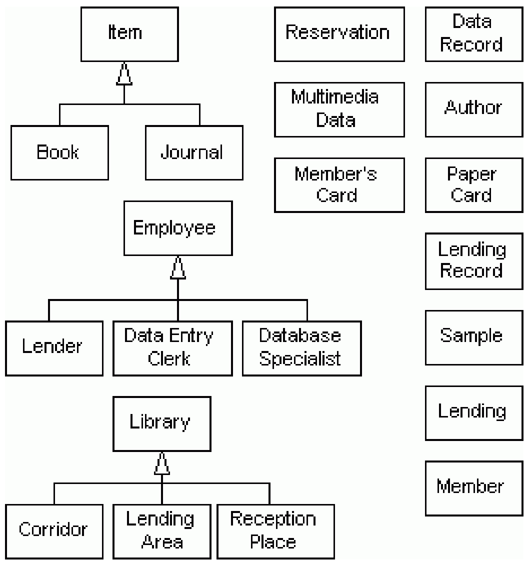

corridor, data record, storage folder, local network, computer server, computer terminal, paper book, paper journal, electronic book, electronic journal, item, date, limit of time, limit of copies, book, journal, sample, author, paper card, library, database, ISBN, title, publishing year, genre, publishing house, volume, area, reservation, employee, lending, lending area, lender, user, member, lending record, report, number of samples, DataEntry clerk, DatabaseSpecialist, reception place, name, lastname, address, place of living, PersonalRegistrationNumber, telephone, e-mail, date of birth, member's card, library's documentation, multimedia data, information, text, audio, video, picture, signature

and verbs in the requirements:

- go, search, lend, take, sort, print, put, do, has, have, convert, write, manage, access, read, preview, take a sample, reserve, give, inform, make, set free, check the data, recommend, post, buy, operate, maintain, conclude, browse, input

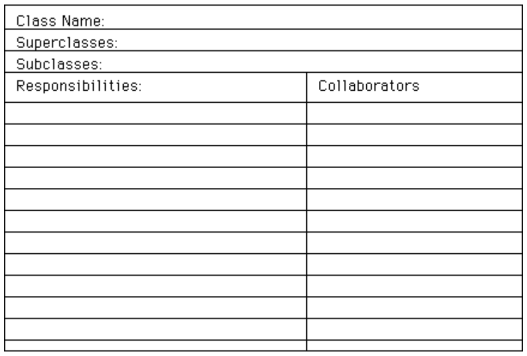
Understand System Usage: Use case diagram(s).

▶ A Simplified Practical Process ▶ Analysis: Find out classes and their relations



▶ A Simplified Practical Process ▶ Analysis: Refine classes and their relations

Class, Responsibilities, and Collaboration (CRC) Cards:



Class Name:	
Superclasses:	
Subclasses:	
Responsibilities:	Collaborators

▶ A Simplified Practical Process ▶ Analysis: Dynamic behavior of classes

- **♡Refine the Class Diagrams: Refined class diagram.**
- **♡Analyze Changes of State in Objects: State diagram.**
- **♡Define the Interactions Among Objects: Sequence and collaboration diagrams.**

▶ A Simplified Practical Process ▶ Design

- **♡Develop and Refine Object Diagrams: Activity diagrams.**
- **♡Design and Prototype User Interface: Screen shots of the screen prototypes.**

Iterate analysis and design.

▶ References

[1] Nils Brummond. *Object-Oriented Analysis and Design using CRC Cards*. 2004. Avaiable from http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b/.

[2] First Step Communications. *Object-Oriented Computing*. 2004. Examples about "passing the salt" is from http://www.firststep.com.au/education/solid_ground/oo.html.

[3] Slobodan Kalajdziski. *UML Tutorial in 7 days*. 2004. Avaiable from http://odl-skopje.etf.ukim.edu.mk/uml-help/.

[4] Bruce E. Wampler. Several chapters of this book by Wampler are available online. Chapter 2 is a very good overview of object-oriented programming. http://www.objectcentral.com/oobook/oobook.htm.