CS 619 Introduction to OO Design and Development

Domain Models (Part 1)

Fall 2012

UP: From inception to Elaboration

- Elaboration is the initial series of iterations
  - the core, risky software architecture is programmed and tested
  - the majority of requirements are discovered and stabilized
  - the major risks are mitigated or retired
- Elaboration in one sentence:
  - **Build** the core architecture, **resolve** the high-risk elements, **define** most requirements, and **estimate** the overall schedule and resources.

UP: From inception to Elaboration

- CAUTION:
  - Elaboration **is not** a design phase (such as in waterfall) or a phase when the models are fully developed in preparation for implementation in the construction step,
- Do not superimpose waterfall ideas on iterative development and the UP.
- The production quality programming and its output is not discardable, but a part of your final system to be delivered to customer.

Artifacts produced in Elaboratoin

Table 8.1. Sample elaboration artifacts, excluding those started in inception.

Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

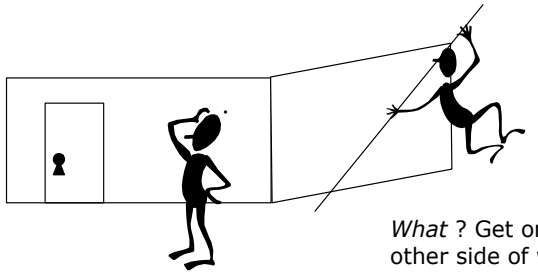
Domain Model

- A requirements specification must be **validated**
  - Are we building the right system?
- A requirements specification must be **analyzed**
  - Did we understand the problem correctly ?
    = Are we modeling the problem domain adequately?
- Why?

  The 30++ years of software development taught us one fundamental lesson...
  - The customer doesn't know what he wants!
  - And if he does, he will certainly change his mind.

How Domain Modeling ?

- Domain Models help to anticipate changes, are more robust.
- Focus on the what (goal), not on the how (procedure)!



What ? Get on the other side of wall.

How ? Open door, break lock, …
… jump over the wall

Problem Decomposition

Object-Oriented Decomposition	Functional Decomposition
Decompose according to the objects a system must manipulate. ⇒ several coupled "is-a" hierarchies	Decompose according to the functions a system must perform. ⇒ single "subfunction-of" hierarchy

Example: Order-processing software for mail-order company	
Order - place - price - cancel Customer - name - address LoyalCustomer - reduction	OrderProcessing - OrderMangement • placeOrder • computePrice • cancelOrder - CustomerMangement • add/delete/update

Problem Decomposition

Object-Oriented Decomposition	Functional Decomposition
⇒ distributed responsibilities	⇒ centralized responsibilities

Example: Order-processing software for mail-order company	

`Order::price(): Amount`
`{sum := 0`
`FORALL this.items do`
`{sum := sum + item. price}`
`sum:=sum-(sum*customer.reduction)`
`RETURN sum`
`}`
	`computeprice(): Amount`
	`{sum := 0`
	`FORALL this.items do`
	`sum := sum + item. price`
	`IF customer isLoyalCustomer THEN`
	`sum := sum - (sum * 5%)`
	`RETURN sum`
	`}`

`Customer::reduction(): Amount`
`{ RETURN 0%}`
`LoyalCustomer::reduction(): Amount`
`{ RETURN 5%}`	

Functional vs. Object-Oriented

Functional Decomposition

Good with stable requirements or single function (i.e., "waterfall") is a clear problem decomposition strategy

• However

+ Naive: Modern systems perform more than one function

e.g. What about "produceQuarterlyTaxForm"?

+ Maintainability: system functions evolve => cross-cuts whole system

e.g. How to transform telephone ordering into web order-processing?

+ Interoperability: interfacing with other system is difficult

e.g. How to merge two systems maintaining customer addresses?

Functional vs. Object-Oriented

Object-Oriented Decomposition

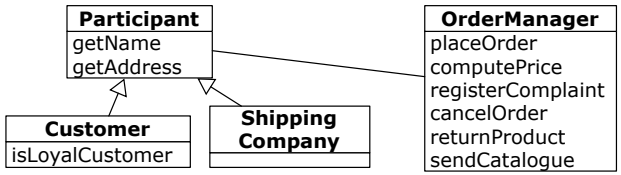
Better for complex and evolving systems
Encapsulation provides robustness against typical changes

- But, how to find classes?

Identifying a rich set of **conceptual classes** is at the heart of OO analysis.

But, not God Classes

How to do functional decomposition with an object-oriented syntax



Symptoms

- Lots of tiny "provider" classes, mainly providing accessor operations + most of operations have prefix "get", "set"
- Inheritance hierarchy is geared towards data and code-reuse + "Top-heavy" inheritance hierarchies
- Few large "god" classes doing the bulk of the work
- + suffix "System", "Subsystem", "Manager", "Driver", "Controller"

A Domain Model

- illustrates meaningful conceptual classes in a problem domain.
- is a representation of real-world concepts, not software components.
- is **NOT** a set of diagrams describing software classes, or software objects and their responsibilities.
- In other words, it has nothing to do with programming.

Decomposition:

- A central distinction between Object-oriented analysis and structured analysis is the division by objects rather than by functions during decomposition.
- During each iteration, only objects in current scenarios are considered for addition to the domain model.

A Domain Model is the most important OO artifact

- Its development entails identifying a rich set of conceptual classes, and is at the heart of object oriented analysis.
- It is a visual representation of the decomposition of a domain into individual conceptual classes or objects.
- It is a visual dictionary of noteworthy abstractions.

Domain Model with UML Notation

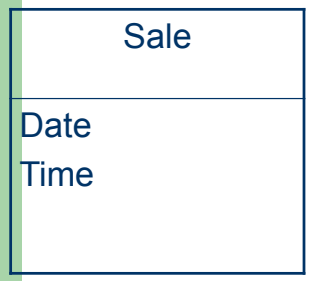
- Illustrated using a set of class diagrams for which no operations are defined.

It may contain:

- Domain Objects or Conceptual Classes
- Associations between conceptual classes
- Attributes of conceptual classes

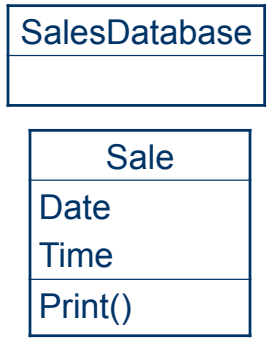
A Domain Model is not a Software Artifact

A Conceptual class:



vs.

Software Artifacts:



- **Important:**
  - Domain Model is a **visualization** of things in a real-situation domain of interest, ***not*** of software objects such as Java or C# classes, or software objects with responsibilities .
- Therefore, the following elements are not suitable in a domain model:
  - Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
  - Responsibilities or methods.

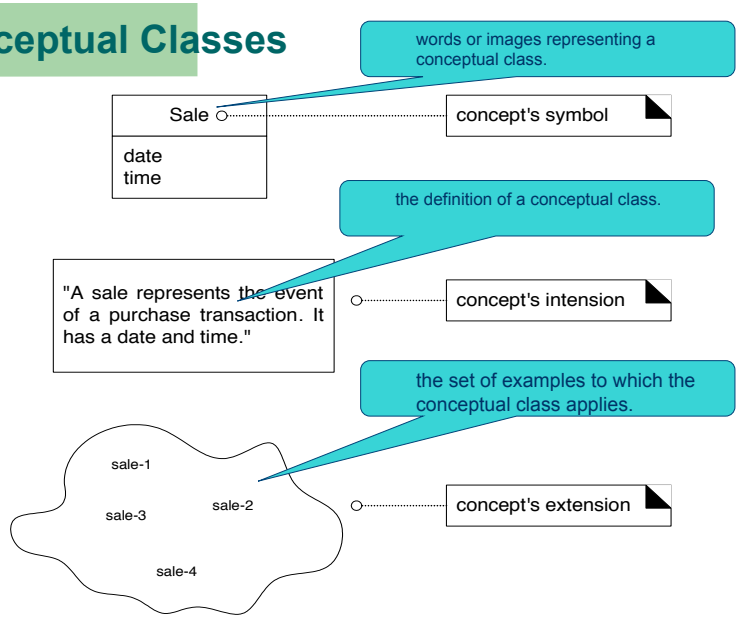
Conceptual Classes

Domain model describes conceptual classes in a domain. So what is a conceptual class?

- Informally, a conceptual class is an idea, thing, or object.
- Formally, we think about conceptual classes in terms of:
- **Symbols** – words or images
- **Intensions** – its definition
- **Extensions** – the set of examples to which it applies

Symbols and Intensions are the practical considerations when creating a domain model.

Conceptual Classes



Conceptual Class Identification:

Three Strategies to Find Conceptual Classes

- Reuse or modify existing models.
  1) This is the first, best, and usually easiest approach. There are published, well-crafted domain models and data models (which can be modified into domain models) for many common domains, such as inventory, finance, health, and so forth. Example books include ***Analysis Patterns*** by Martin Fowler, ***Data Model Patterns*** by David Hay, and the ***Data Model Resource Book*** (volumes 1 and 2) by Len Silverston.
- Use a category list.
- Identify noun phrases.

Conceptual Class Identification:

- It is better to overspecify a domain with lots of fine-grained conceptual classes than it is to underspecify it.
- Discover classes up front rather than later.
- Unlike data modeling, it is valid to include concepts for which there are no attributes, or which have a purely behavioral role rather than an informational role.

Use a category list

Table 9.1. Conceptual Class Category List.

Conceptual Class Category	Examples
business transactions *Guideline*: These are critical (they involve money), so start with transactions.	*Sale, Payment* *Reservation*
transaction line items *Guideline*: Transactions often come with related line items, so consider these next.	*SalesLineItem*
product or service related to a transaction or transaction line item *Guideline*: Transactions are *for* something (a product or service). Consider these next.	*Item* *Flight, Seat, Meal*
where is the transaction recorded? *Guideline*: Important.	*Register, Ledger* *FlightManifest*
roles of people or organizations related to the transaction; actors in the use case *Guideline*: We usually need to know about the parties involved in a transaction.	*Cashier, Customer, Store MonopolyPlayer Passenger, Airline*
place of transaction; place of service	*Store* *Airport, Plane, Seat*
noteworthy events, often with a time or place we need to remember	*Sale, Payment MonopolyGame Flight*

Use a category list

physical objects *Guideline*: This is especially relevant when creating device-control software, or simulations.	*Item, Register Board, Piece, Die Airplane*
descriptions of things *Guideline*: See p. 147 for discussion.	*ProductDescription* *FlightDescription*
catalogs *Guideline*: Descriptions are often in a catalog.	*ProductCatalog* *FlightCatalog*
containers of things (physical or information)	*Store, Bin Board Airplane*
things in a container	*Item Square (in a Board) Passenger*
other collaborating systems	*CreditAuthorizationSystem* *AirTrafficControl*
records of finance, work, contracts, legal matters	*Receipt, Ledger* *MaintenanceLog*
financial instruments	*Cash, Check, LineOfCredit* *TicketCredit*
schedules, manuals, documents that are regularly referred to in order to perform work	*DailyPriceChangeList* *RepairSchedule*

Identify Conceptual Classes by Noun Phrase:

- Identify Nouns and Noun Phrases in textual descriptions of the domain.
- Fully dressed Use Cases are good for this type of linguistic analysis.

  Also in other documents, or the minds of experts.

It's **not** strictly a mechanical process:

- Words may be ambiguous
- Different phrases may represent the same concepts.

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters **item identifier**.
4. System records **sale line item** and presents **item description, price**, and running **total**. Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment**.
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions**) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

. . .

Noun Phrase Identification

- Some of these noun phrases are candidate conceptual classes,
- some may refer to conceptual classes that are ignored in this iteration (for example, "Accounting" and "commissions"), and
- some may be simply attributes of conceptual classes.
- it is recommended in combination with the Conceptual Class Category List technique.

Example: Case Study: POS Domain

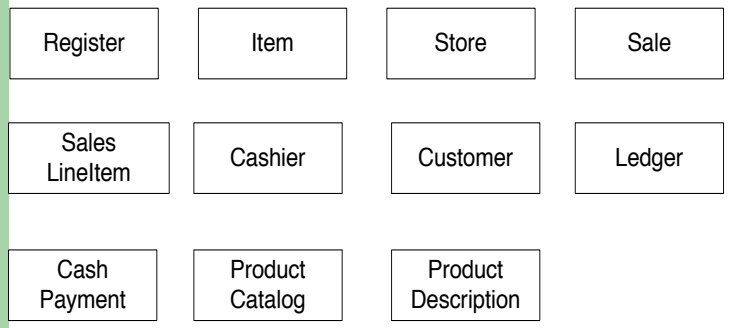
Sale	**Cashier**
CashPayment	**Customer**
SalesLineItem	**Store**
Item	**ProductDescription**
Register	**ProductCatalog**
Ledger	

Example: Case Study: POS Domain

- Is this list correct?
- The good news/or bad news is
  - There is no such thing as a "correct" list.
  - It is a somewhat arbitrary collection of abstractions and domain vocabulary that the modelers consider noteworthy.
  - Nevertheless, by following the identification strategies, different modelers will produce similar lists.

Example: Case Study: POS Domain

- In practice, one can immediately draw a UML class diagram of the conceptual classes as we uncover them.



Example: Case Study: POS Domain

- We find concepts such as Register, Sale, Item, Customer, Receipt etc. in this use case.

Should we include Receipt in the Model?

- Con: As a report of a sale, it's duplicate info.
- Pro: Business Rules for a Return require that the customer has a receipt.
- Suggestion: Include it in the iteration where the Return Use Case is covered.

Guideline: Think Like a Mapmaker

- Use the existing names in the territory.
  - For example, if developing a model for a library, name the customer a "Borrower" or "Patron" the terms used by the library staff.
- Exclude irrelevant or out-of-scope features.
  - For example, in the Monopoly domain model for iteration-1, cards (such as the "Get out of Jail Free" card) are not used, so don't show a Card in the model this iteration.
- Do not add things that are not there.

Guideline: A Common Mistake with Attributes vs. Classes

- Perhaps **the most common mistake** when creating a domain model is to represent something as an attribute when it should have been a conceptual class.
- A rule of thumb to help prevent this mistake is:
  - *If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute*

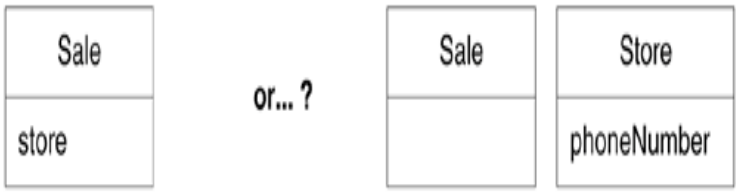
Guideline: A Common Mistake with Attributes vs. Classes

- Should "**destination**" be an attribute of "**Flight**", or a separate conceptual class "**Airport**"?

  - In the real world, a destination airport is ***not*** considered a number or text - it is a massive thing that occupies space. Therefore, Airport should be a concept.

Guideline: A Common Mistake with Attributes vs. Classes

- As an example, should "**store**" be an attribute of "**Sale**", or a separate conceptual class "Store"?



  - In the real world, a store is ***not*** considered a number or text - the term suggests a legal entity, an organization, and something that occupies space. Therefore, Store should be a conceptual class.

Guideline: When to Model with 'Description' Classes?

- A description class contains information that describes something else.
  - For example, a *ProductDescription* that records the price, picture, and text description of an Item.
- Why bother with description class?
- The same situation that calls for a description class happens a lot….

Guideline: When to Model with 'Description' Classes?

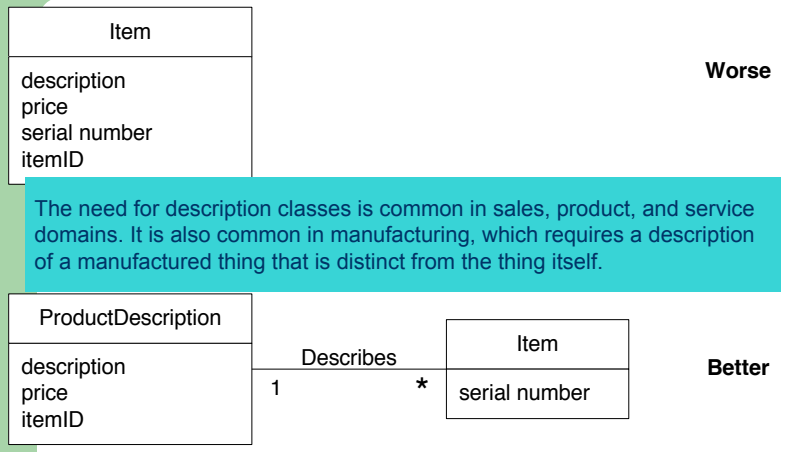
- Assume the following:
  - An Item instance represents a physical item in a store; as such, it may even have a serial number.
  - An Item has a description, price, and itemID, which are not recorded anywhere else.
  - Everyone working in the store has amnesia.
  - Every time a real physical item is sold, a corresponding software instance of Item is deleted from "software land."
- With these assumptions, what happens in the following scenario?

Guideline: When to Model with 'Description' Classes?

- There is strong demand for the popular new vegetarian burger - ObjectBurger. The store sells out, implying that all Item instances of ObjectBurgers are deleted from computer memory.

- Now, here is one problem: If someone asks, "*How much do ObjectBurgers cost?*", **no one can answer**, because the memory of their price was attached to inventoried instances, which were deleted as they were sold.

Guideline: When to Model with 'Description' Classes?



Switching from a conceptual to a software perspective, note that even if all inventoried items are sold and their corresponding Item software instances are deleted, the ProductDescription still remains.

Guideline: When are Description Classes useful?

- **Add a description class (for example, ProductDescription) when:**
  - There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
  - Deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained, but was incorrectly associated with the deleted thing.
  - It reduces redundant or duplicated information.

Example: Descriptions in the Airline Domain

- Consider an airline company that suffers a fatal crash of one of its planes.

- Assume that all the flights are cancelled for six months pending completion of an investigation.

- Also assume that when flights are cancelled, their corresponding Flight software objects are deleted from computer memory. Therefore, after the crash, all Flight software objects are deleted.

Example: Descriptions in the Airline Domain

- If the only record of what airport a flight goes to is in the Flight software instances, which represent specific flights for a particular date and time, then ***there is no longer a record of what flight routes the airline has.***

- The problem can be solved, both from a purely conceptual perspective in a domain model and from a software perspective in the software designs, with a ***"FlightDescription" class*** that describes a flight and its route, even when a particular flight is not scheduled

Example: Descriptions in the Airline Domain