Domain Model Refinement

Dalibor Dvorski (ddvorski@conestogac.on.ca)
School of Engineering and Information Technology
Conestoga College Institute of Technology and Advanced Learning

When to Define a Conceptual Subclass?

e.g. `Customer` may be correctly partitioned (or subclassed) into `FemaleCustomer` and `MaleCustomer`. But is it relevant or useful to show this in our model?

Create a conceptual subclass of a superclass when:

1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.
4. The subclass concept represents an animate thing (e.g. animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.

Based on the above criteria, it is not compelling to partition `Customer` into the subclasses `FemaleCustomer` and `MaleCustomer` because they have no additional attributes or associations, are not operated on (treated) differently, and do not behave differently in ways that are of interest.

When to Define a Conceptual Superclass?

Create a superclass in a generalization relationship to subclasses when:

1. The potential conceptual subclasses represent variations of a similar concept.
2. The subclasses will conform to the *100%* and *is-a* rules.
3. All subclasses have the same attribute that can be factored out and expressed in the superclass.
4. All subclasses have the same association that can be factored out and related to the superclass.

Based on the above criteria for partitioning the `Payment` class, it is useful to create a class hierarchy of various kinds of payments. Credit and cheque authorization services are variations on a similar concept, and have common attributes of interest. Refer to figures 31.7 and 31.8 in *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* for updated class diagrams that take into account the aforementioned.

Degree of Generalization

Study figure 31.9 in *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Also recall the Condor eLibrary case study from lab exercises last term and the interesting question that came up: *Should a modeler illustrate every variation of something?* The answer to this will depend on the system under discussion. For the NextGen POS system, the class diagram that is shown in figure 31.10 is preferred over the one shown in figure 31.9. This is because the excessive generalizations, just like in the Condor eLibrary case, do not add any obvious value. The hierarchy of figure 31.9 expresses a finer granularity of generalization that does not significantly enhance our understanding of the concepts and business rules, but it does make the model more complex – and added complexity is undesirable unless it confers other benefits.

Association Classes

The following domain requirements set the stage for association classes:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.
- Furthermore, a store has a different merchant ID for each service.

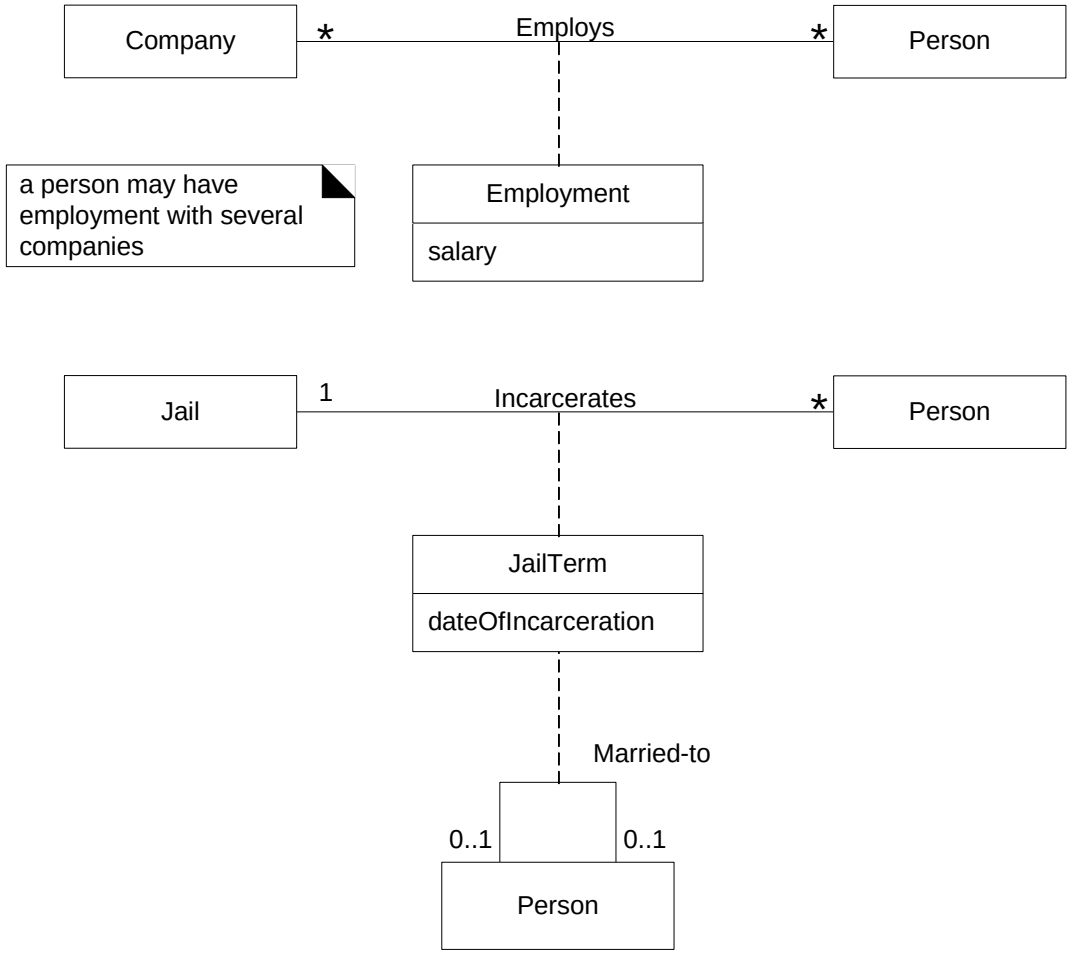
Where in the UP Domain Model should the merchant ID attribute reside? Placing `merchantID` in `Store` is incorrect because a `Store` can have more than one value for `merchantID`. The same is true with placing it in `AuthorizationService`.

Association Classes

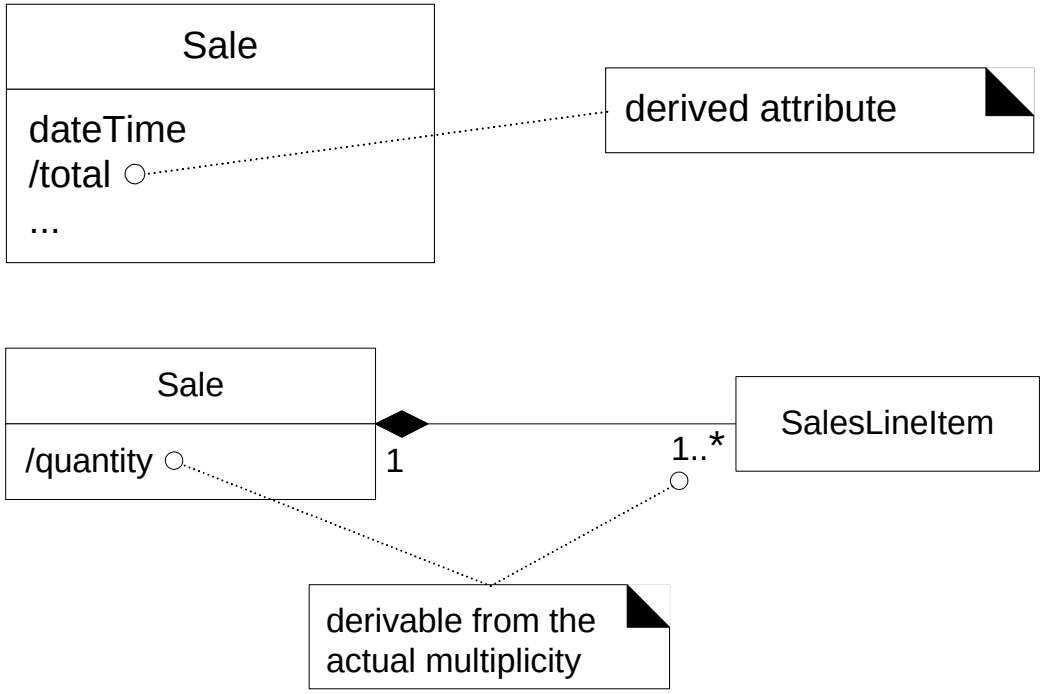
Guideline: In a domain model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another class that is associated with C. e.g. A `Person` may have many phone numbers. Place phone number in another class, such as `PhoneNumber` or `ContactInformation`, and associate many of these to `Person`.

Guideline: Clues that an association class might be useful in a domain model: an attribute is related to an association; instances of the association class have a lifetime dependency on the association; there is a many-to-many association between two concepts and information associated with the association itself.

Association Classes

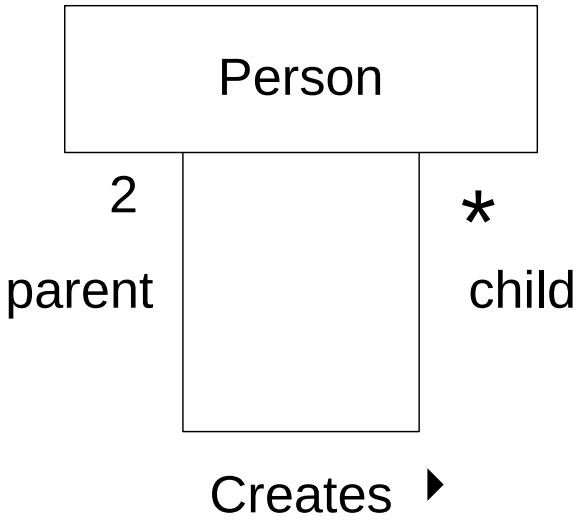


Derived Elements



A derived element can be determined from others. Attributes and associations are the most common derived elements. When should derived elements be shown? Avoid showing derived elements in a diagram, since they add complexity without new information. However, add a derived element when it is prominent in the terminology, and excluding it impairs comprehension. e.g. `Sale total` can be derived from information in `SalesLineItem` and `ProductDescriptions`. e.g. `SalesLineItem` quantity is actually derivable from the number of instances of `Items` associated with the line item.

Reflexive Associations



How to Partition the Domain Model

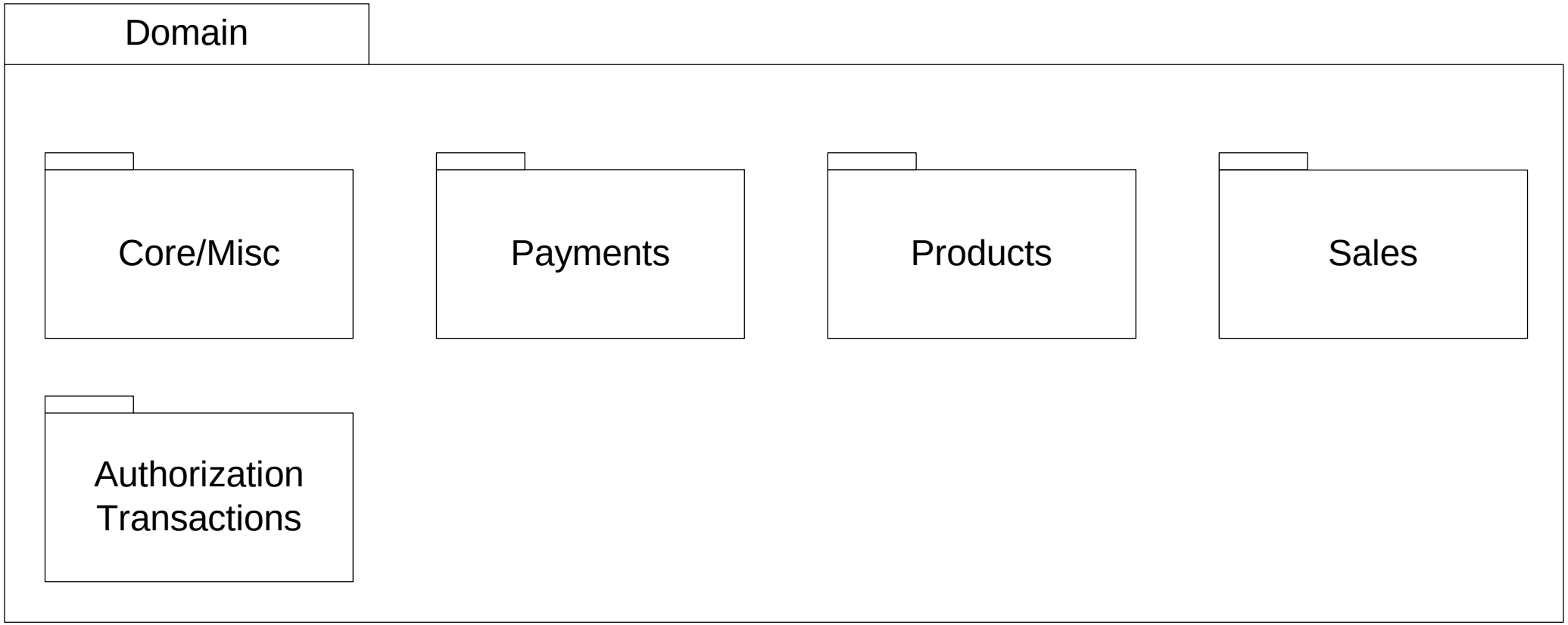
How should the classes in a domain model be organized within packages?

To partition the domain model into packages, place elements together that:

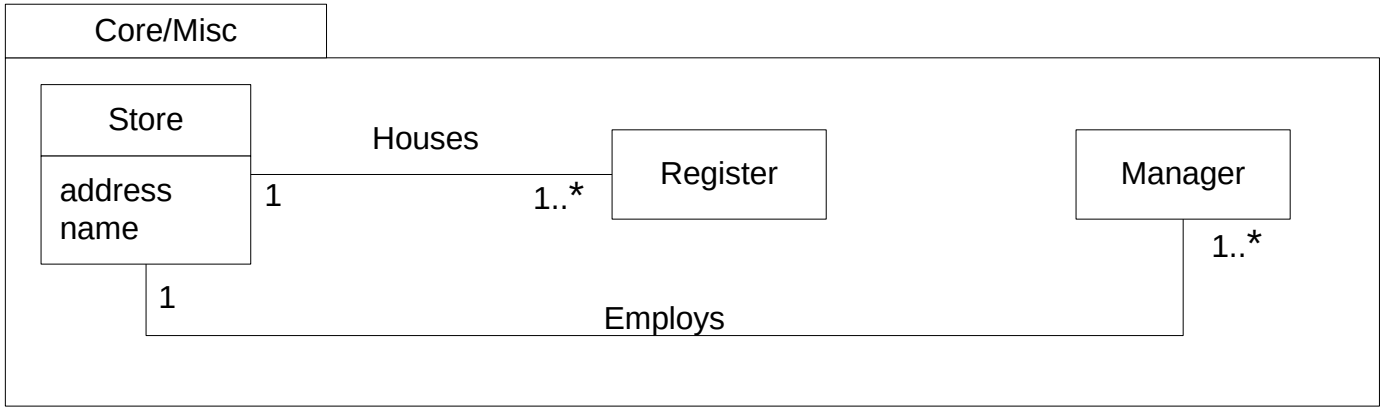
- are in the same subject area – closely related by concept or purpose
- are in a class hierarchy together
- participate in the same use cases
- are strongly associated

It is useful if all elements related to the domain model are rooted in a package called *Domain*, and all widely shared, common, core concepts are defined in a packaged named something like *Core Elements* or *Common Concepts*, in the absence of any other meaningful package within which to place them.

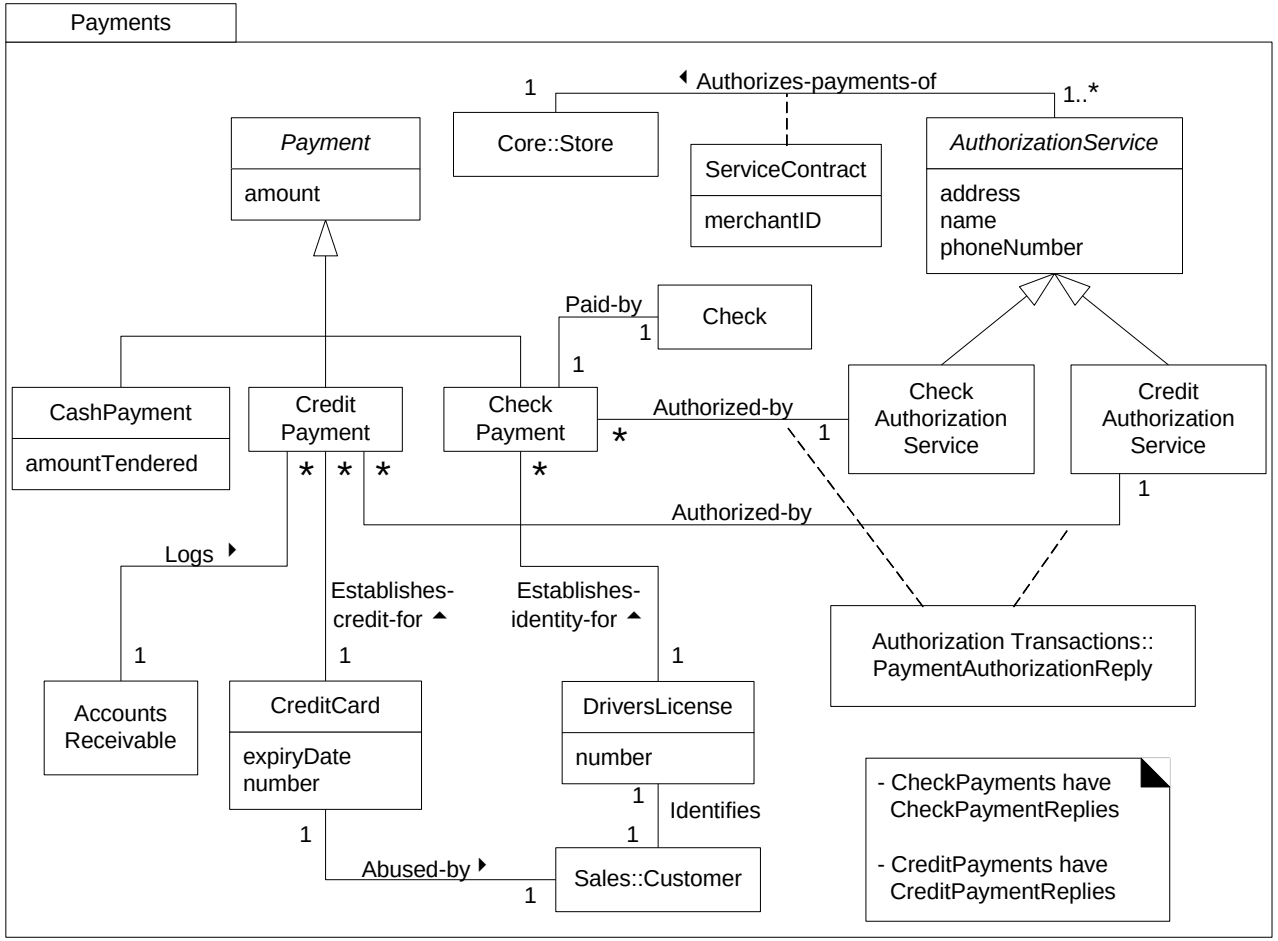
NextGen POS Domain Packages



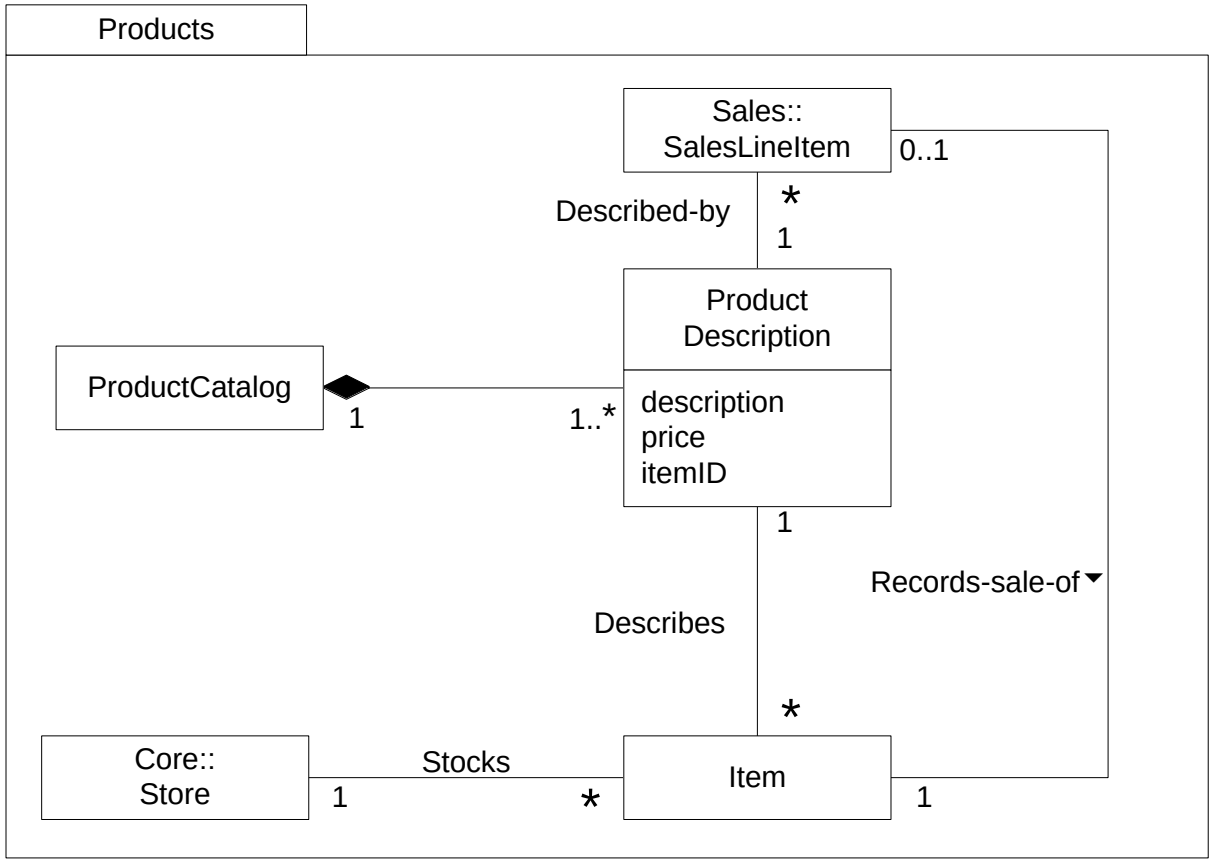
NextGen POS Core/Misc Package



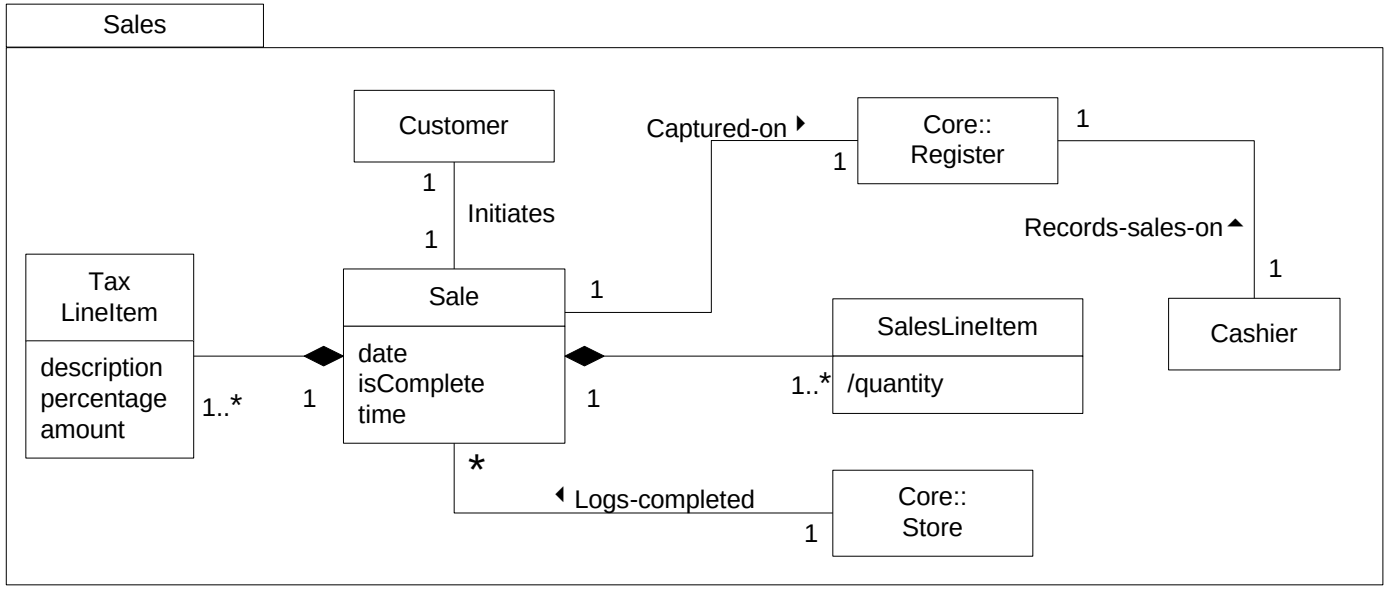
NextGen POS Payments Package



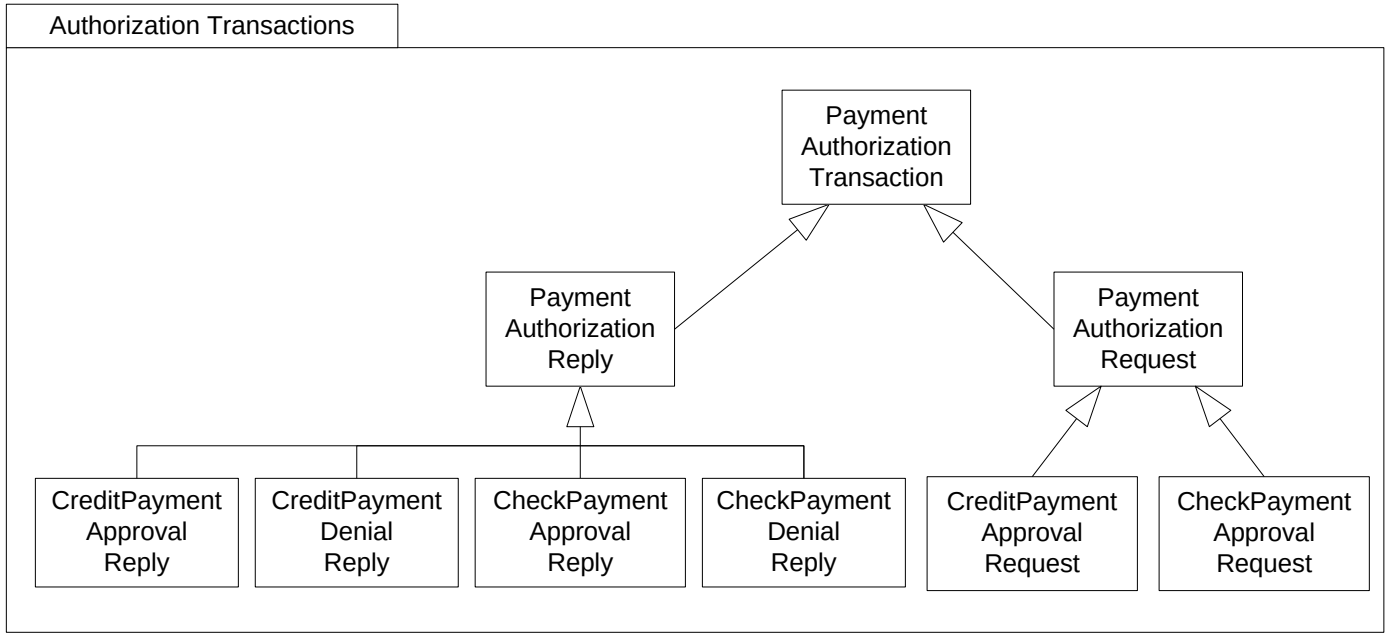
NextGen POS Products Package



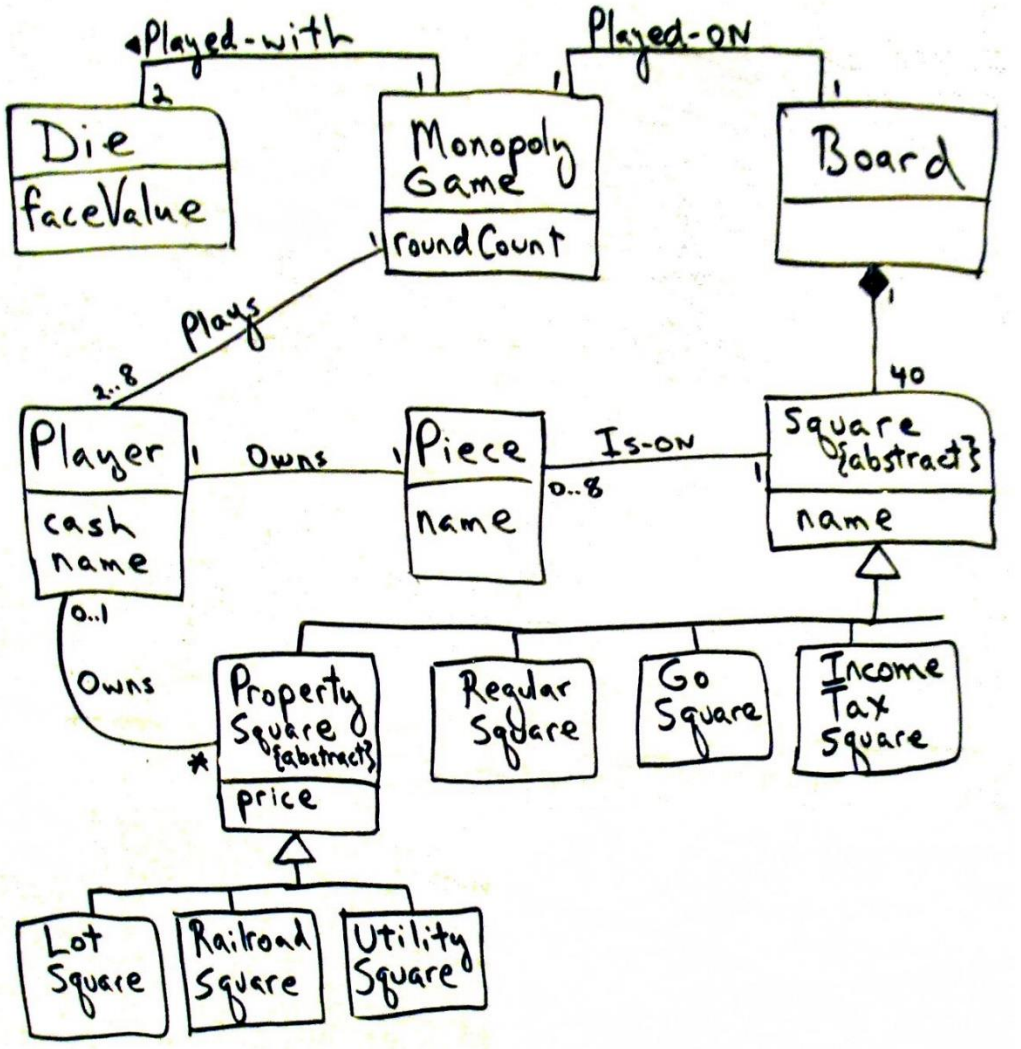
NextGen POS Sales Package



NextGen POS AuthorizationTransactions Package



Monopoly Domain Model Update



Assigned Readings

1. Chapter 30: Relating Use Cases from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
2. Chapter 31: Domain Model Refinement from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.