Assigning Responsibilities

Problem statement to Design

Using the POS System Statement of Needs document create a list of system "features". Note that a feature is a function initiated by the user to accomplish a goal

- . An example in a banking ATM system might be "Withdraw Cash".

Depending on the starting point for describing the system, some (or all) of these features may have been identified. This is a good starting point, but should not stop the development team from looking for functions that have not been explicitly stated. This will help to establish the "scope" of the project.

Feature Identification

POS Features that might be identified, but are out of scope for this project:

- Scan bar code – will be handled by a subsystem we will interface with
- Inventory control
- Payment authorization
- Others…?

Feature Identification

POS System Features (by priority)

- Process Sale (can explore entire transaction – helpful in initial design and construction)
- Handle Returns (might be able to develop along with Process Sale)
- Pay by Cash (system could be operational with cash payments only to start)
- Pay by Credit (not sure on connections to payment authorization service)
- Analyze Activity (defer reporting until initial modeling complete)
- Manage Security (push admin functions to later releases)
- Manage Terminals
- Others….?

Stakeholder identification

POS Stakeholders:

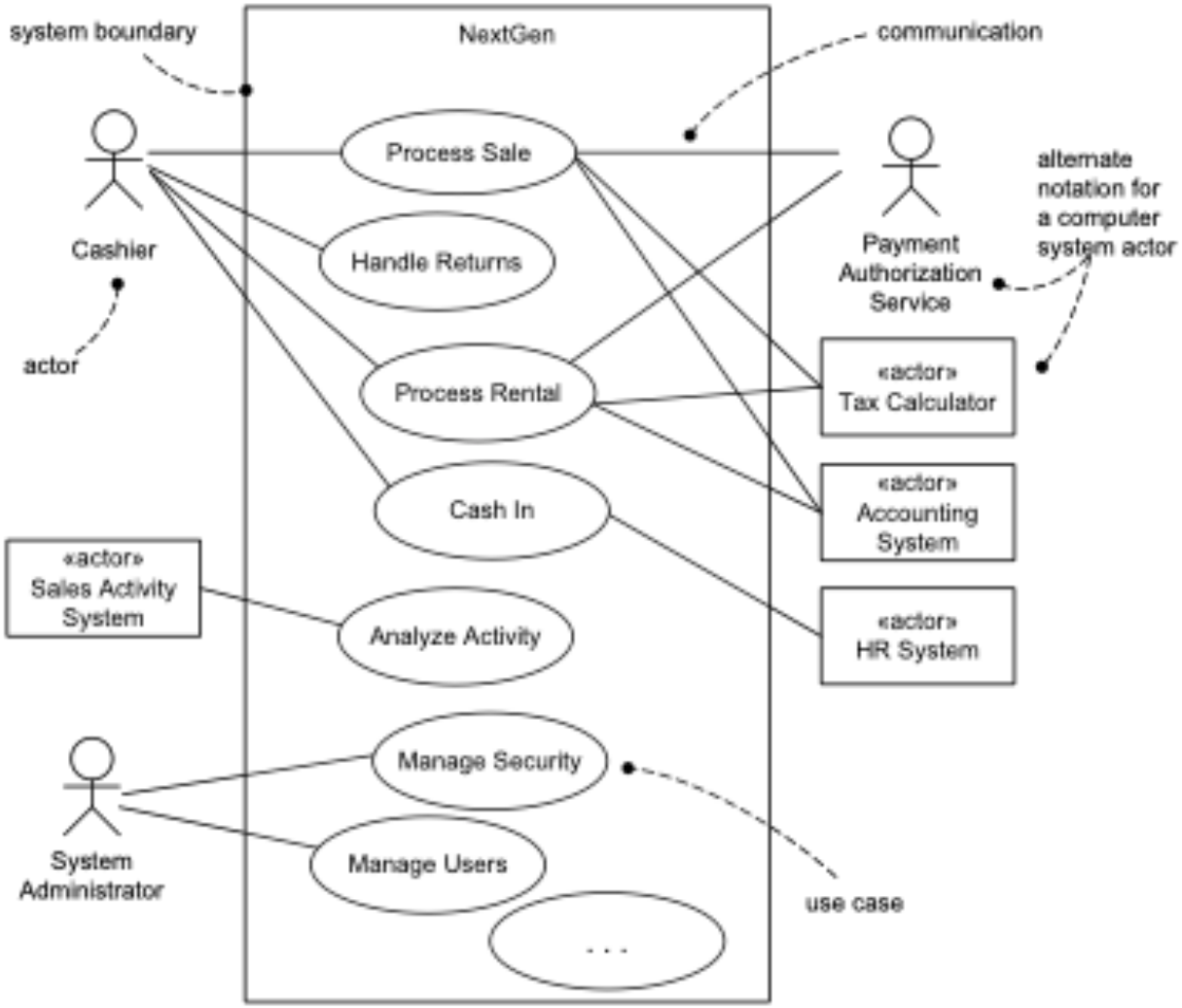
- Cashier
- Customer
- Company / Store
- Payment Authorization Service
- Store Manager (Administrator)
- Others…?

Feature Prioritization

Prioritize the order in which features should be completed and why. What influences which order features are delivered in?

- Customer priority, business value – system could be useful in a partial state
- Development priority – select a feature that helps to define the initial design, ideally a feature that touches many classes and helps to create a design skeleton, or decide on architecture.
- Features that once implemented help in making requirements clearer
- Mitigating Risk – attack technically risky features first

Use Case Diagram



Use Case Description for each use case

Process Sale success scenario use case text:

Preconditions: Cashier is identified and authenticated on a sales terminal.

- Customer arrives at POS checkout with goods and/or services to purchase.
- Cashier starts a new sale.
- Cashier enters item identifier.
- System records sales line item and presents item description, price, and running total. Price calculated from a set of price rules.
  < Cashier repeats steps 3-4 until indicates done >
- System presents total with taxes calculated.
- Cashier tells Customer the total, requests payment.
- Customer pays and System handles payment.
- System logs completed sale and sends sale and payment information to the external Accounting System and Inventory System.
- System presents receipt.
- Customer leaves with receipt and goods (if any).

Use Case to Design

Process Sale success scenario use case text:

Preconditions: Cashier is identified and authenticated on a sales terminal.

- ***Customer*** arrives at ***POS checkout*** with ***goods*** and/or ***services*** to purchase.
- ***Cashier*** starts a new ***sale***.
- Cashier enters ***item identifier***.
- System records ***sales line item*** and presents ***item description***, ***price***, and ***running total***. Price calculated from a set of price rules.
  < Cashier repeats steps 3-4 until indicates done >
- System presents total with *taxes* calculated.
- Cashier tells Customer the total, requests ***payment***.
- Customer pays and System handles payment.
- System logs completed ***sale*** and sends sale and payment information to the external ***Accounting*** System and ***Inventory*** System.
- System presents ***receipt***.
- Customer leaves with receipt and goods (if any).

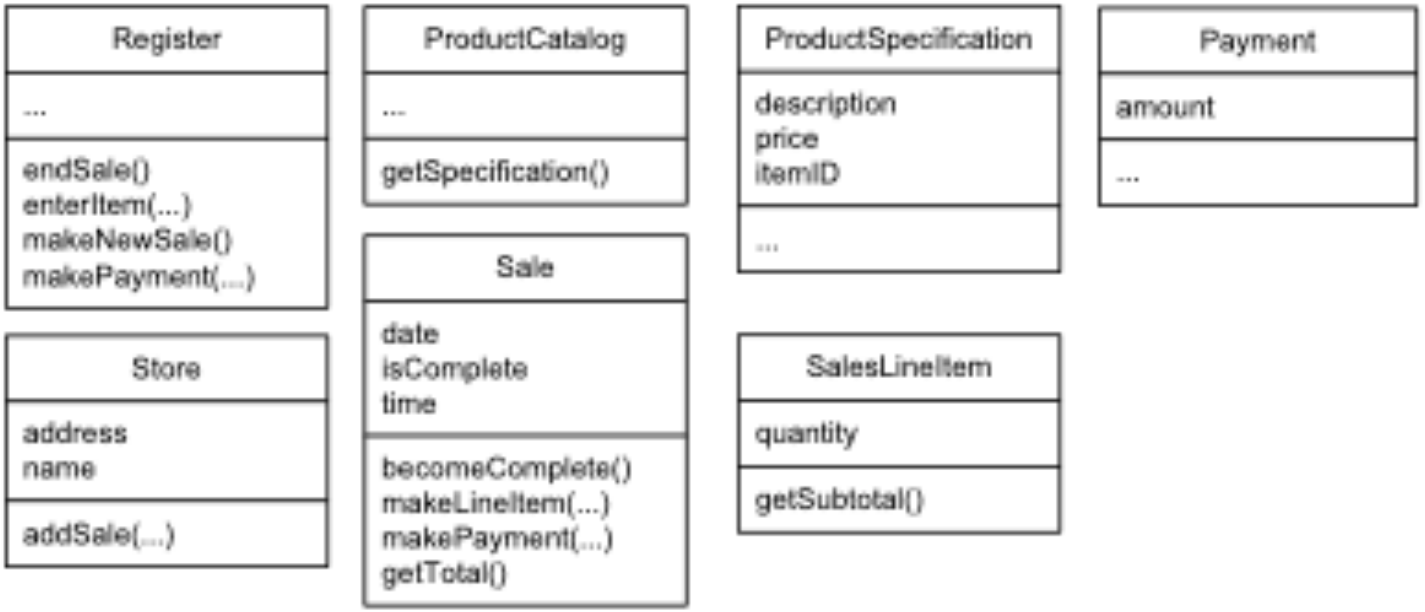
Note the differences between class and attributes – not all nouns automatically become classes

Conceptual classes for POS domain

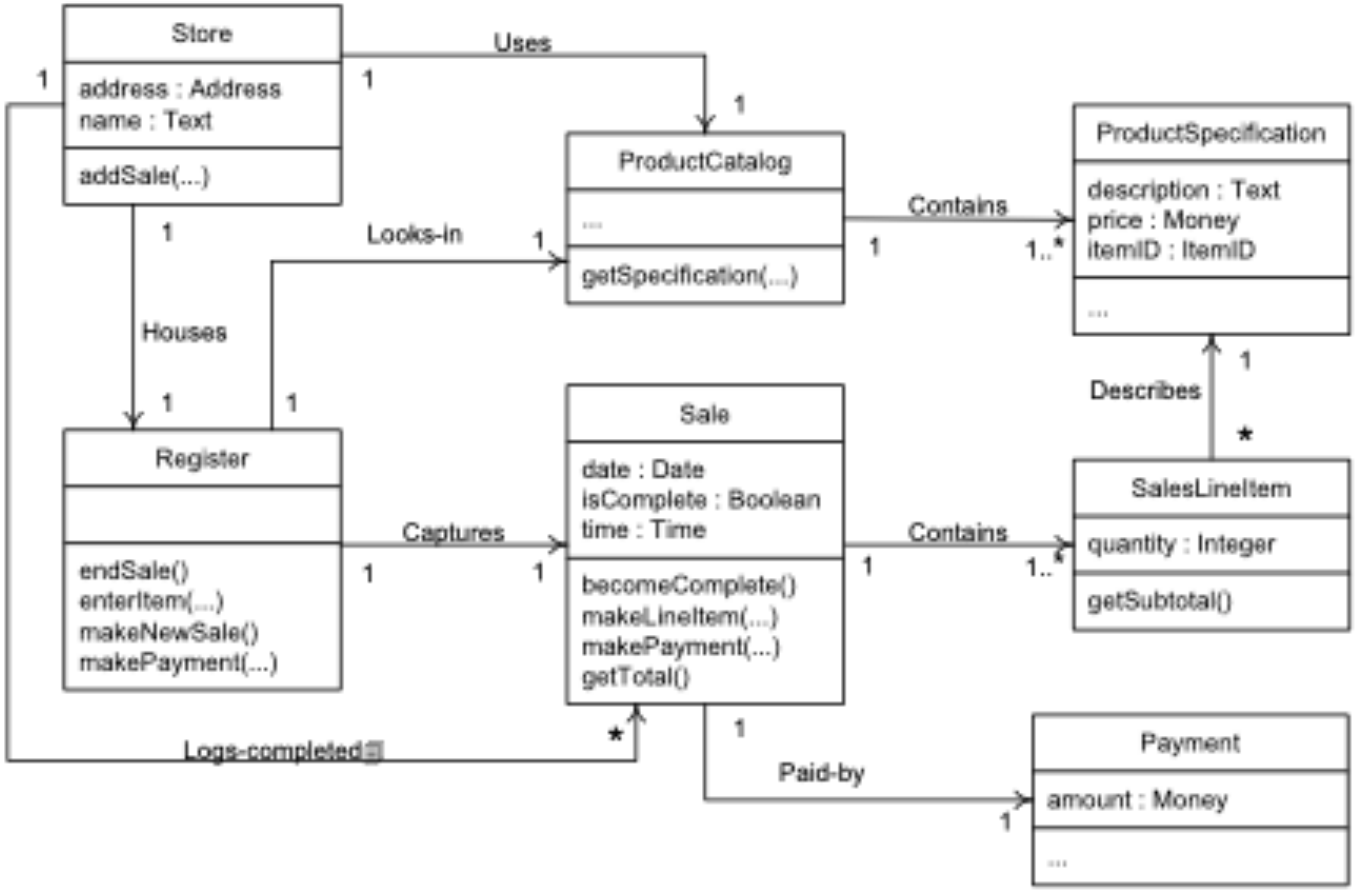
Moving to software classes, Cashier, Customer & Manager are not considered, Item potentially becomes an attribute in ProductSpecification.

Identify Attributes

Identify methods



Identify Relationships



Keep iterating (use GRASP)

Summarizing...

Object Design

- → Requirements identification and creation of a domain (concept) model
- → Add methods to the software classes
- → Define messaging between the obects to fulfill the interaction

General Responsibility Assignment Software Patterns

(GRASP)

Basic Concepts

- Responsibility: A contract or obligation of a classifer
  - Action-oriented responsibilities
    - Carrying out an activity
    - Controlling and coordinating activities in other objects
    - Delegating activities to other objects
  - Data-oriented responsibilities
    - Maintaining private information
    - Maintaining relationships to other objects
    - Maintaining derived information

- Interaction diagrams reflect decisions pertaining to assignment of responsibilities
  - Message passing between objects reflect responsibility distribution
  - Use interaction diagrams to explore possible distribution of responsibilities

Extensibility vs. Reusability

- Poor distribution can lead to inflexible designs
  - Change impact can be wide
  - Good distribution can lead to lower maintenance costs
- Tension between extensibility and reusability
  - Good distribution requires encapsulation of application-specific details in class
  - Reusability requires that software be as independent as possible from application context

Patterns of Responsibility Assignment Principles

- High Cohesion
- Information Expert
- Creator
- Low Coupling
- Controller

A total of 9 GRASP patterns (including above) have been captured in literature: Polymorphism, Protected Variations, Pure Fabrication, Indirection

High Cohesion

- Cohesion is a measure of how diverse an entity's features are.
  - A highly cohesive class has features that pertain to a single concept
  - A highly cohesive class has one general responsibility
    - Guideline: Should be able to describe responsibility of a highly cohesive class in one sentence
    - Use sentence as comment in code

- Guideline: Assign a responsibility so that parts of the class are strongly related and the class responsibility is tightly focused
  - Class easier to understand
  - Easier to maintain and reuse

Levels of Cohesion

- Very low cohesion: class is responsible for many things in different functional contexts
- Low cohesion: class is solely responsible for a complex group of tasks in a single functional area
- Moderate cohesion: class is responsible for relatively simple tasks in different but related functional areas
- High cohesion: class is responsible for a group of tasks in a single functional area and discharges its responsibility by delegating some of its responsibilities to other classes.

When to ignore high cohesion guidelines

- A class that provides a single point of entry into a system may sometimes be desirable
  - Such a class is called a Façade and provides external clients with a single point of access to services offered by a system
- For efficiency reasons it may be more appropriate to place two diverse classes in the same class
  - Rather than an object delegating responsibility for a service to another object it may carry it out itself to avoid delegation performance overhead

Expert

- Assign responsibility to the class that has the information necessary to discharge the responsibility.

- Naïve use can lead to undesirable coupling and low cohesion.
  - Giving a class the responsibility for storing its objects in a database leads to low coupling and undesirable coupling
    - Low cohesion: class contains code related to database handling between the
    - Undesirable coupling: class is tightly coupled to database services provided by another system

  - Example: Elevator Control System

Creator

- Class B can be responsible for creating objects of A in the following situations:
  - A is a part class of B
  - B is a container of A objects
  - B records A objects
  - B has the data needed to initialize A objects

Low Coupling

- Assign responsibilities to reduce high coupling to unstable classes (i.e., classes with high probability of significant changes)
  - Reduces impact of change
  - Classes can be understood in relative isolation

- Forms of coupling in OO designs
  - Class X contains a reference to Class Y objects
  - Class X operation includes calls to Class Y operations
  - Class X operation has a Class Y object as a parameter or declares a Class Y object as a local variable
  - Class X is a direct or indirect subclass of Class Y
  - Class X implements an interface Y

- Classes designed for reuse should have low coupling. Why?

Controller

- Assign responsibility for handling a system event to a class representing the system or a class that is responsible for handling the events in a group of related use cases.
  - A *system event* is an event generated by an actor. A system event results in the execution of a *system operation*.
  - A *controller* is a non-user interface class responsible for receiving and handling system events. A controller defines the method for the system operation.

- A good controller delegates the work needed to handle a system event to other objects.
  - A controller controls and coordinates the collaborating objects.
  - A controller does not do much of the actual work.

Controller Options

- Presentation objects (UI objects) should not be responsible for handling events
  - Decouple presentation layer from application processing layer. Why?
- System as controller
  - Referred to as a façade controller
  - Use when number of system events is not large
    - Large number of events can lead to a controller with low cohesion and high coupling
- Use case handlers
  - For each use case design a controller that handles the use case events
  - Use when number of system events is large

Bloated Controllers

- Signs of problematic design
  - Interface objects handle system events directly
  - Controller object handles many events
  - Controller object performs bulk of work needed to handle event.
  - Controller class has many attributes because of its many responsibilities.

General Guidelines

- Avoid dumb objects: objects that hold data and provide only get/set methods
- Avoid "god" controllers: a "god' controller is one that requests state information (e.g., using a get method) and uses the information to make decisions or perform calculations
- Avoid coupling by having services above and beyond get/set services in interface of objects
- A client should request an object to do something on its behalf, not request information about an object's state.