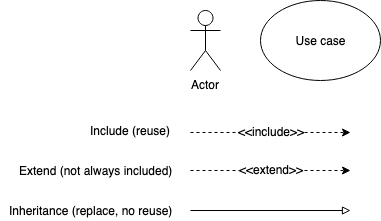
**Use case diagram**

Scenario-oriented description of expected functionality for the system

* Show “what” can be done (instead of “how” it works
* Don’t care about class, object, package, etc.
* Triggered by an actor
* UML syntax
  + Actor: anything external to the program that interacts with the program
  + Use case: Provides value to actors
  + Interaction
  + Relationship between use cases: include, extend, inheritance
  + Relationship between actors: inheritance



| **dt** | **Data** | **ps** | **Static** | **Dynamic** | **Integration** | **lod** | **sc** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Domain | x |  | x |  |  | l | sys |
| Use case |  |  |  |  | x | l | sys |
| Activity |  | x |  | x |  | m | uc |
| Robustness | x | x |  | x | x | h | uc |
| Sequence | x | x |  | x | x | h | uc |
| Data Flow | x |  |  | x |  | l | sys |

dt/data/ps/int/lod/sc=diagram type/-/-integration/level of disclosure/scope

sys/uc=system/use case

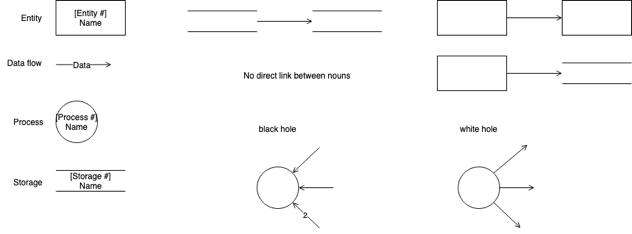
h/m/l = high/medium/low

**Use Case Description**

* Required: use case name, scenarious, alternatives
* Optional: purpose/objective, actors, preconditoins, postconditions, system behavior
* Always have actor and action

**Data Flow Diagrams**

* Not UML
* Shows how a system’s entities (external), processes, and storages (internal) are interconnected
* Syntax
  + Entity (People, roles, organizations, external systems, etc.)
  + Data flow (input, output)
  + Process
  + Storage (Databases, files, folders, etc.)
* Leveled
  + Level 0 (Context diagram) - system level (only one process, no storage) / Level 1 / etc.
* Rules
  + No direct link between nouns (entity, storage)
  + Storage introduced at level 1+ (not at context)
  + Process
* No black hole (all in, no out)/no white hole (no in, all out)

Add numbers to processes for tracking across levels (in the same level, no order)  


**Robustness Analysis/Design**

* Purposes

Identify objects (view, entities)/processes (controllers) needed to implement the use case

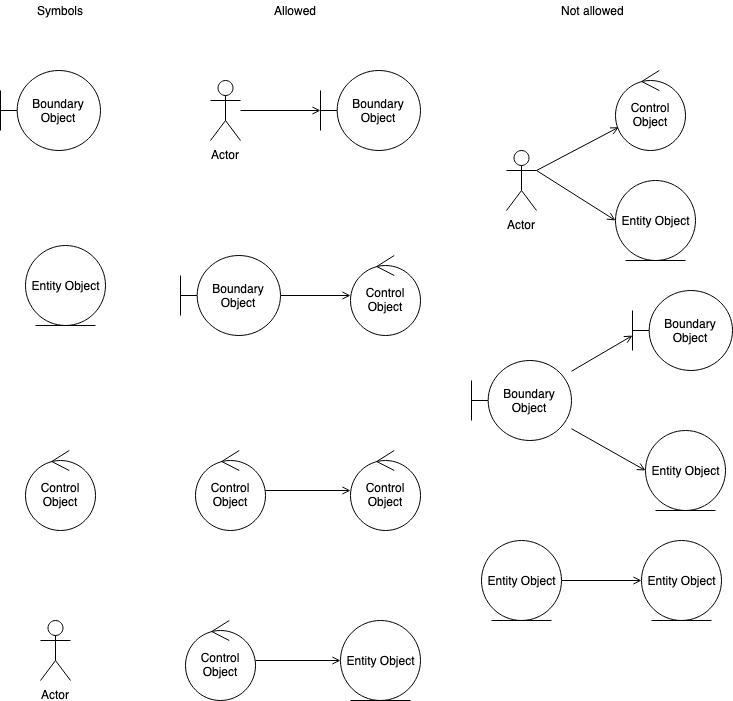
* Objects

**Actors** // **View/Interface** objects (What various actors use to interact with the system; e.g. pages, dialog boxes) // **Model/Entity** objects (temporary & persistent data that are used/generated by the system e.g. temporary data structures, database tables) // **Processes/Controller** object (To state loosely the ones that facilitate interactions between the above)

* Construction

one diagram for each use case // created sentence-by-sentence from the text of the use case

* Rules

**Actors can only talk to boundary objects** (Users can only deal with View objects) // **Boundary objects can only talk to controllers and actors**. (Views only talk with Users and controllers) // **Entity objects can only talk to controllers**. (Model objects can only talk with controllers) // **Controllers can talk to boundary objects, entity objects, and other controllers, but not to actors** (Controllers are always middleware and so don’t do anything outside the system)  
  


Sequence Diagram

Purpose

Help assign operations to classes/instances while drawing messages

Drive the detaield design - sequencing of assigned calls

Further refine set of required classess and operatrions needed

How

One sequence diagram per use case

Both basic and alternate courses

Focus on behavior allocation