

# Tutorial 5: UML

## UML

Software engineers often have to communicate properties of the systems they develop with other stakeholders who may or may not be familiar with the technical details of the system that is being examined. UML provides a unified, consistent way to communicate information about software systems in a way designed to be intuitive for both technical and non-technical individuals. Many tools exist that support working with UML that serve to support the development of a UML model and expressing that model by generating graphical artifacts as specified by the UML standard.

This laboratory exercise will introduce you to one such tool, draw.io.

**Note that you are ALLOWED to use other software tools to complete this tutorial as long as the result follows the proper UML conventions and is in a .pdf file.**

We'll use draw.io to develop four commonly encountered UML artifacts:

- use case diagram
- state diagram
- class model diagram
- sequence diagram

The diagrams you create will model aspects of a simple case study wherein you've been contracted to create a new computer system for a university library. The system you are to develop will replace the library's antiquated card catalog and their manual, paper-based method for keeping track of what has been lent out to library members. For each diagram, we'll introduce the basics of working with draw.io to create that diagram and then you will be charged with translating a description of the system to be developed into the desired UML artifact.

This tutorial is based on materials from Dr. Miryung Kim.

## Software

[draw.io](https://www.draw.io/) (<https://www.draw.io/>) is an online diagram making tool which is not limited to making UML diagram, but also flowcharts, process diagrams, organization charts, ER diagrams, network diagrams and much more. And, it is free.

Other tools (but not limited to):

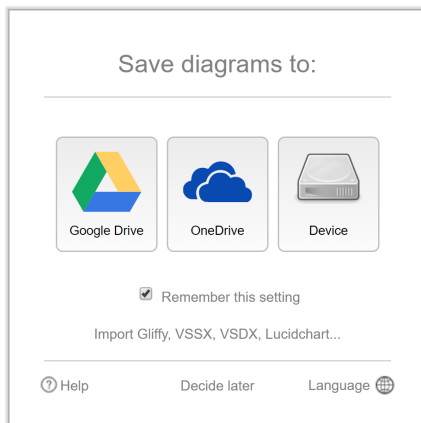
- [Lucidchart](https://www.lucidchart.com/) (<https://www.lucidchart.com/>) (paid)
- [Gliffy](https://www.gliffy.com/) (<https://www.gliffy.com/>) (free trial)

- **Microsoft Visio** [\\_ \(https://support.office.com/en-us/article/uml-diagrams-in-visio-ca4e3ae9-d413-4c94-8a7a-38dac30cbcd6\)](https://support.office.com/en-us/article/uml-diagrams-in-visio-ca4e3ae9-d413-4c94-8a7a-38dac30cbcd6) (?)
- **ArgoUML** [\\_ \(http://argouml.tigris.org/\)](http://argouml.tigris.org/) (free)
- **PlantUML** [\\_ \(http://plantuml.com/\)](http://plantuml.com/) (free)
- **Creately** [\\_ \(https://creately.com/\)](https://creately.com/) (free in demo mode)

## Use case diagram

A use case diagram is a simple drawing which documents the behavior of a system from the user's point-of-view. It identifies the external entities (known as actors) that will work with your system as well as what they will be doing with your system (the "use cases" for those actors). When you start draw.io, it will ask you to choose one of 3 methods of saving your future diagrams:

- Google Drive
- Onedrive
- Device (your local computer storage)



Let's use google drive (but it depends on your own preference). Then follow these steps:

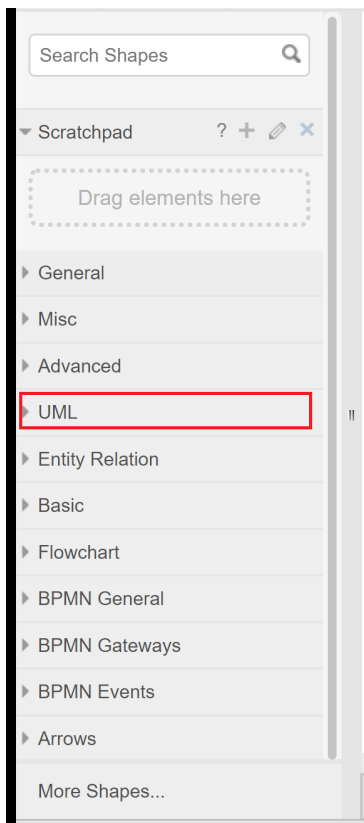
1. Authorize draw.io to access your Google Drive by signing in
2. Create new diagram
3. Add diagram name: **library\_use\_case.html**
4. Select the folder you want the file to be saved in
5. You are ready to start your drawing. But please follow the instructions.

## Familiarize with the Environment

Please consider to review [this document](https://support.draw.io/display/DO/The+draw.io+Environment)

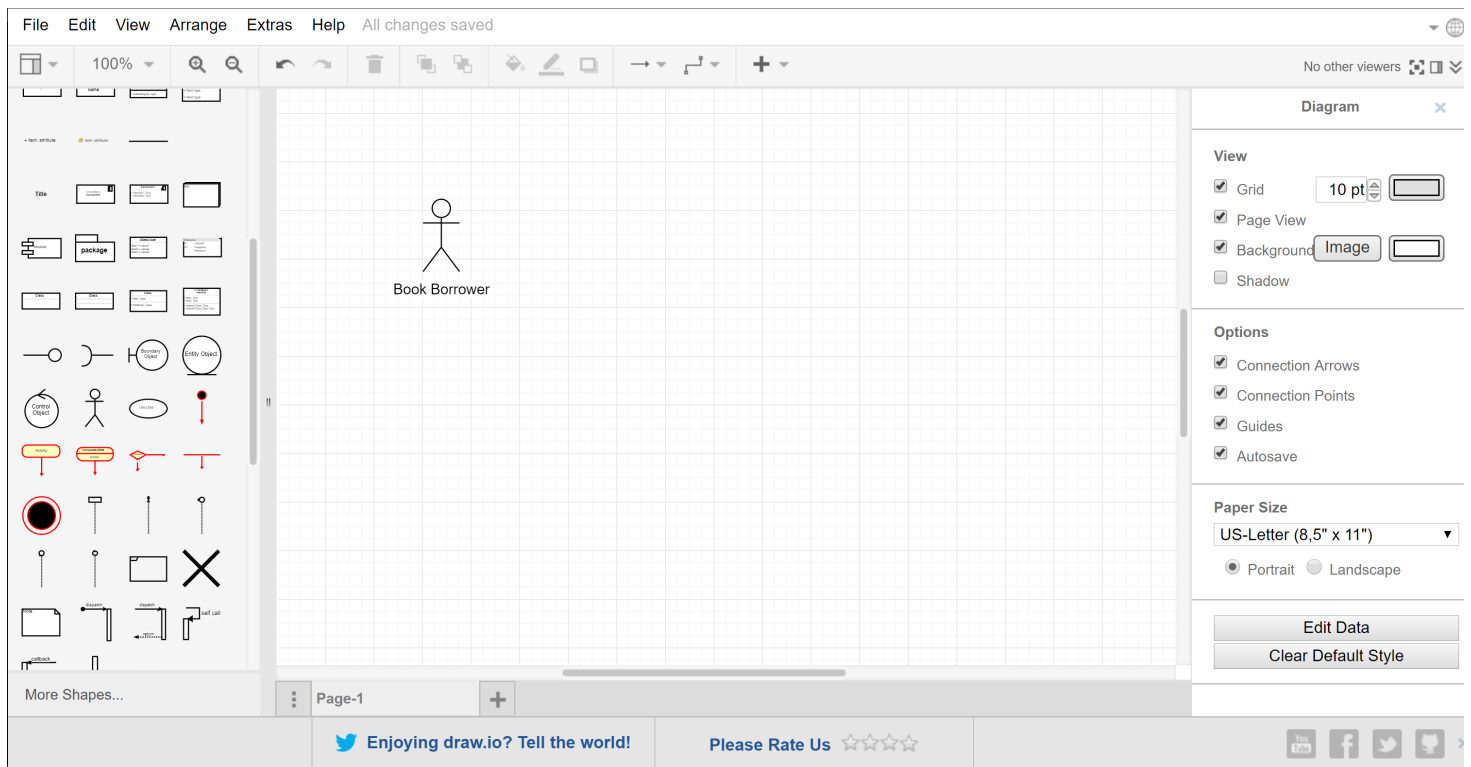
[\\_ \(https://support.draw.io/display/DO/The+draw.io+Environment\)](https://support.draw.io/display/DO/The+draw.io+Environment) to familiarize with draw.io environment.

In this tutorial, we, obviously, will use the UML section of the toolbar.



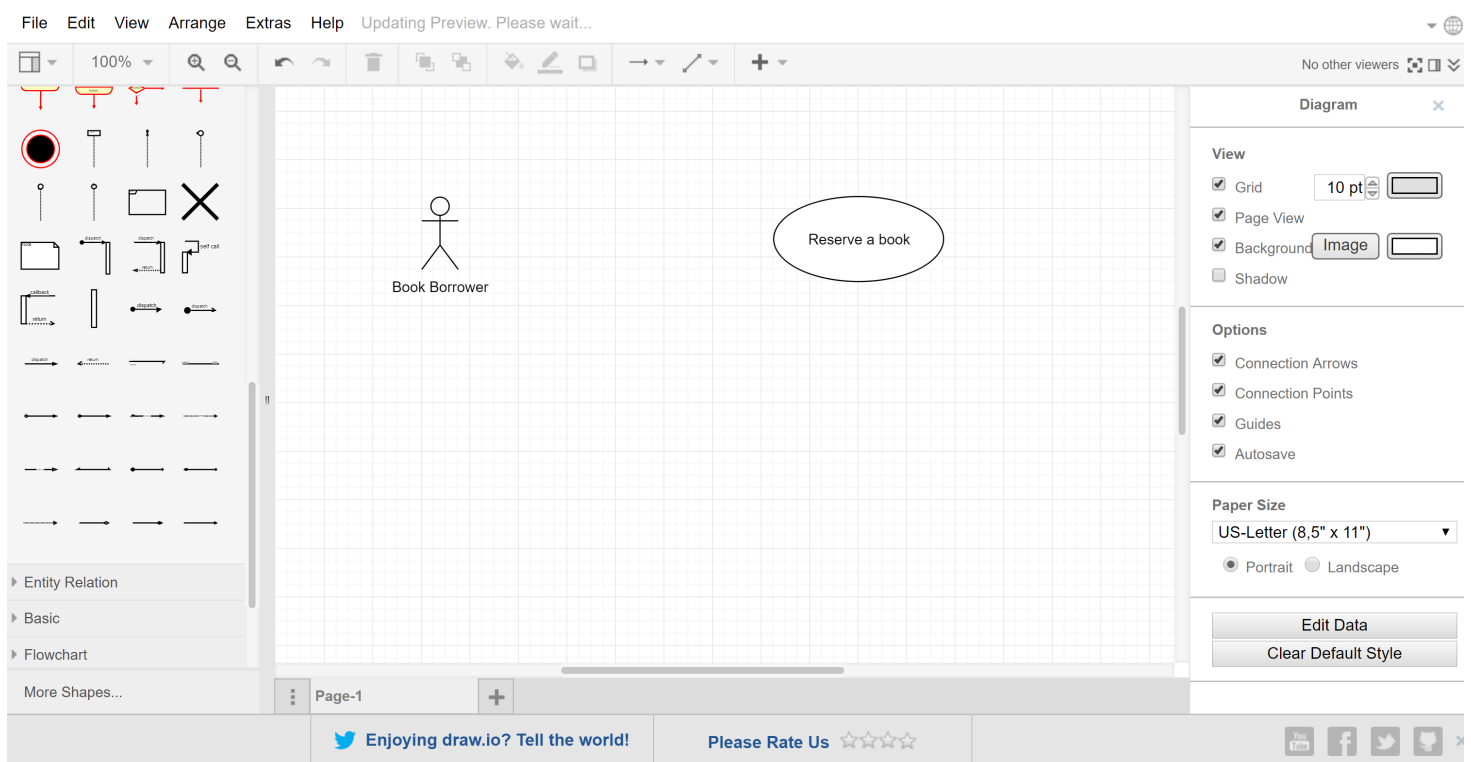
## Actors

Actors can be added to the diagram by clicking or dragging the "**Actor**" tool from the toolbar to the work area. Each actor in your system should have a short descriptive name. Add this information by selecting the instance of the Actor, then start typing the descriptive name. All these features are shown below:



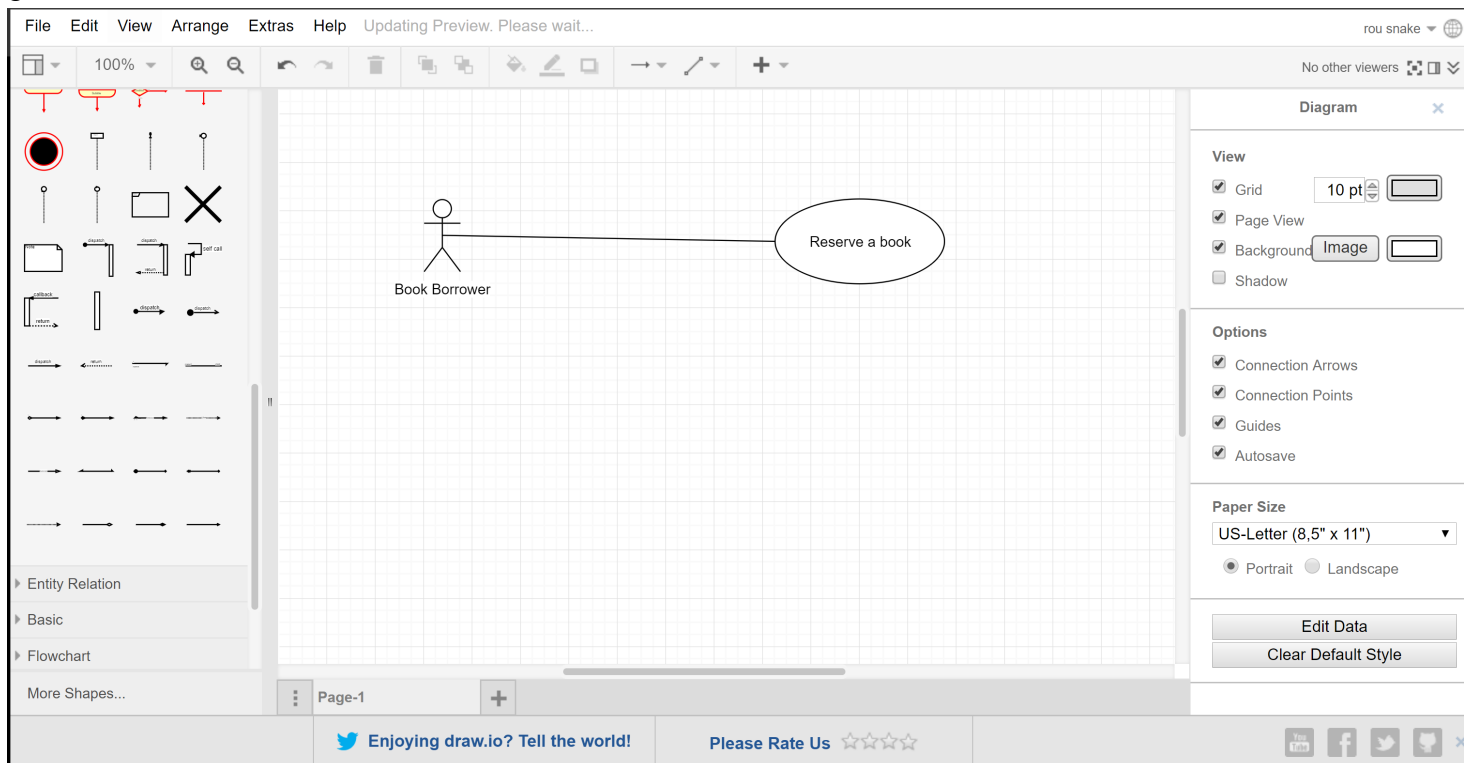
## Use cases

Use cases can be added to the diagram by selecting the **"Use Case"** tool from toolbar. Once the tool has been selected, click anywhere in the diagram window to place a use case in that location. Each use case in your system should also have a short descriptive name. Again, all these features are shown below:



## Associations

Use cases and actors are linked by an "association" which indicates that a particular actor is expected to need to use the system in a given way. To notate these association between actors and use cases you'll need to draw a line representing this association. This can be added to the diagram by hovering the actor, selecting one of the connection points, and dragging it to the use case diagram. Additionally, the line style can be changed in the format panel. You can associated multiple actors with a given use case, or multiple use cases with a single actor. You can name your associations if you like, but this is not required. An example of a simple associated actor/use case is given below:




## Exercise #1

Translate the following English statements into a representative use case diagram:

The system shall have the following four types of actors: BookBorrower, JournalBorrower, Browser, and Librarian. A user playing the role of a BookBorrower shall be able to reserve a book, borrow a copy of a book, return a copy of a book, and extend the loan of a book. A JournalBorrower shall be able to borrow a journal and return a journal. A Browser shall be able to browse, and a librarian shall be able to update the catalog.

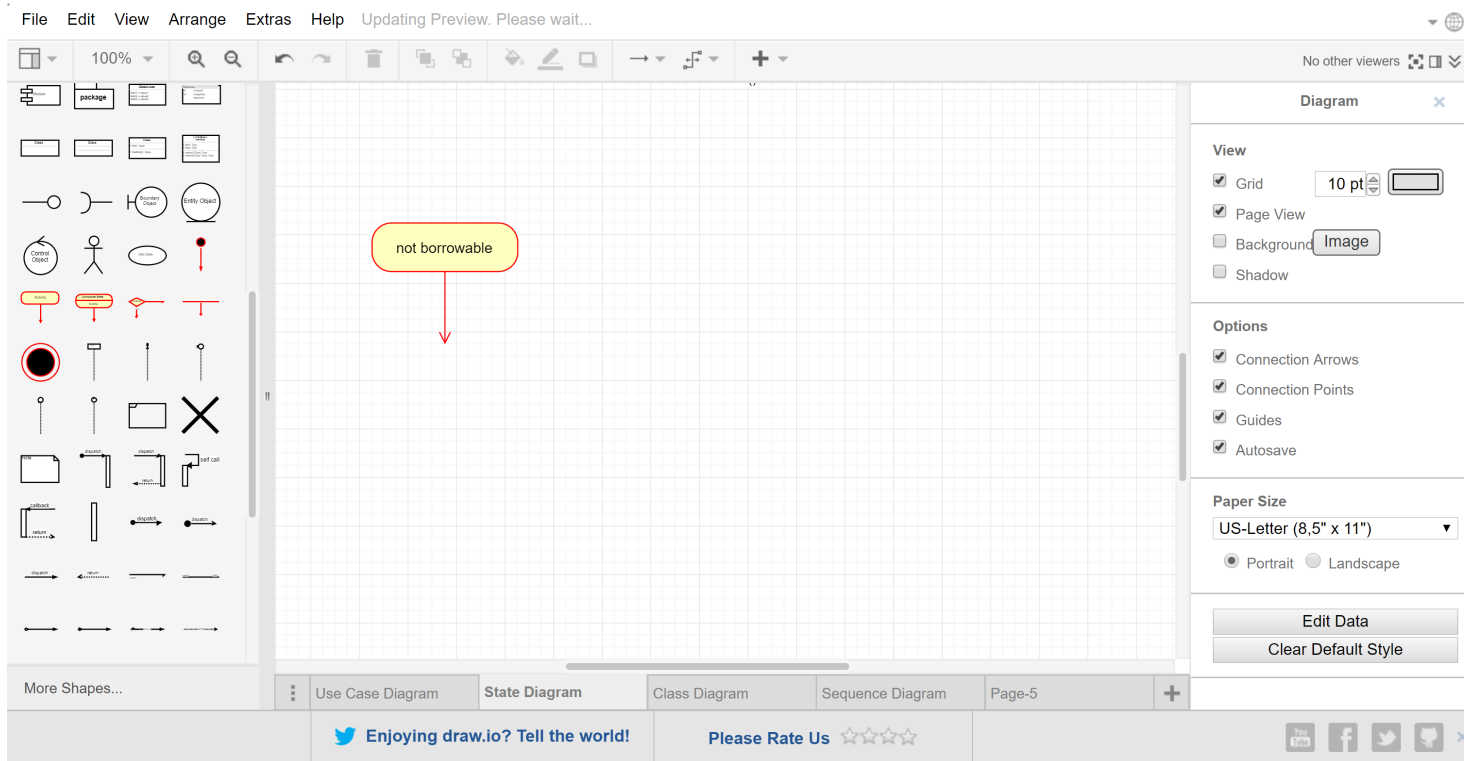
**Download the resulting diagram to a PDF file, File > Export As > PDF > Export > Download. You will turn this in at the completion of the tutorial.**

## State diagrams

Perhaps the most familiar of the UML diagram types, state diagrams are used to communicate how parts of a software system react to specific stimuli. Now, create a new page by clicking  at the bottom of the work area.

### States

States can be added by clicking or dragging on the "**Activity**" tool in toolbar. Each state should be given a short descriptive name. The essential user interface elements you'll need are called out below.



### Transitions

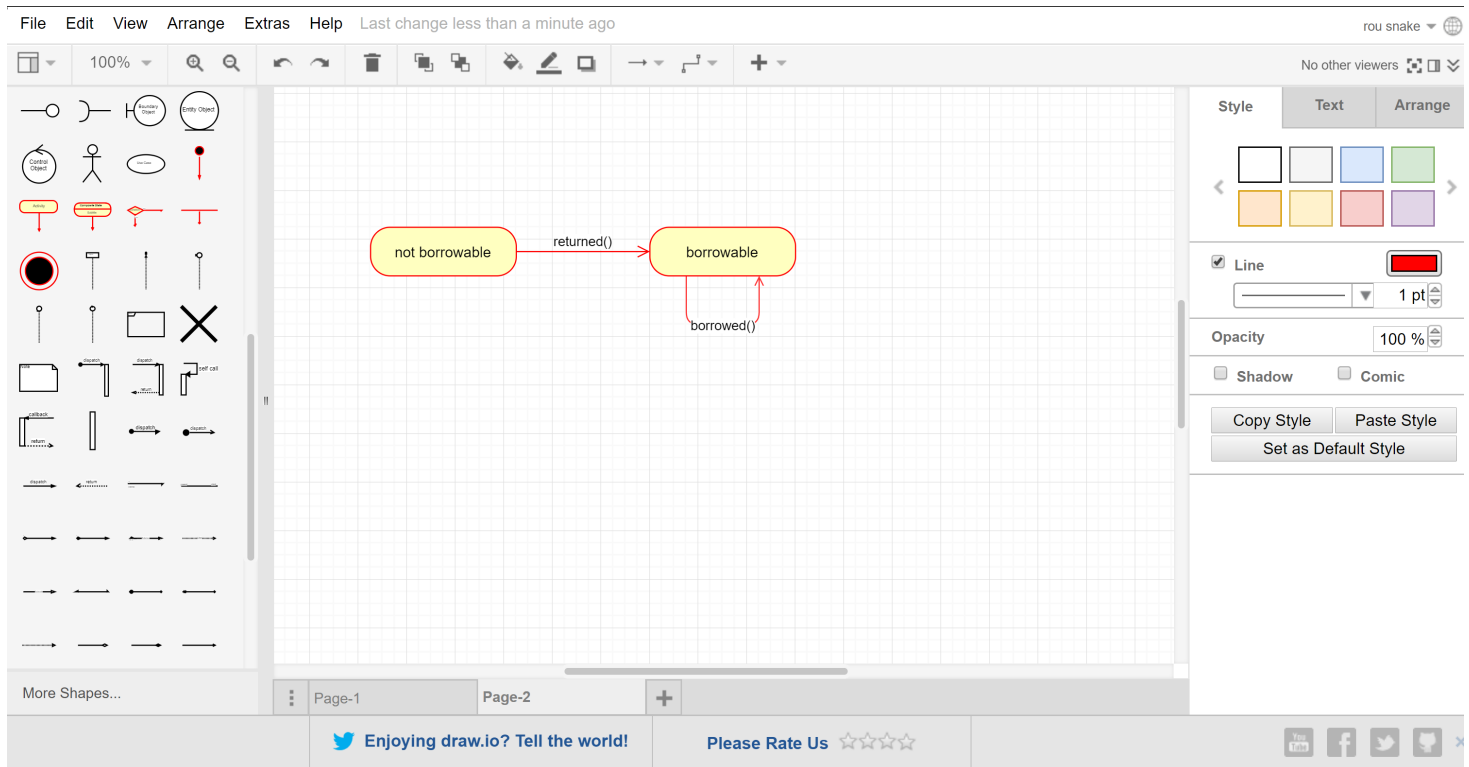
Movement by the system from one state to another is indicated by adding a transition from one state to another. To add a transition between states, select the downward arrow of the activity, select the line end with the arrow. Then click and hold on the state in which the transition begins and drag the mouse to the state to which the transition ends. You can also get the line/arrow from the toolbar.

### Annotations

Transitions are almost always annotated with additional information regarding actions that cause the transition to occur (triggers) and things that will happen when the transition occurs (effects). Additionally, a transition's triggers can be specified to only be valid under certain conditions (guards). In draw.io, adding a guard is substituted with "**Condition**". This shape could have 2 or more

transitions to another state when a condition is fulfilled. See the figure below for an example showing two transitions (incomplete):

- "not borrowable" transitioning to "borrowable": this transition represents a transition triggered by a call to "returned()"
- "borrowable" transitioning to "borrowable": this self-transition represents a transition triggered by a call to "borrowed()", but it is guarded (i.e.: won't happen) unless it is not the last copy



## Exercise #2

During the process of designing the new library system it was discovered that one class of the library system objects would be a "Book". Perhaps not as obvious is that an additional (but closely related) object class would be one for specific copies of a book. For this exercise you will be designing a simple subset of the state machine that is implemented as part of the "Book" class. Translate the following English statements into a representative state diagram that models the behavior of the "Book" class of objects:

A Book should either be borrowable or not borrowable. If a Book is currently borrowable, it should stay in borrowable when a book is returned. Additionally, if the book is borrowable and it is notified that a copy of it has been borrowed, then it should transition to not being borrowable only if it was the last copy that was just borrowed - otherwise it should remain borrowable. Conversely, if the book is not currently borrowable, then it should transition back to being borrowable whenever it is notified that a copy of the book is returned.

**Download the resulting diagram to a PDF file, File > Export As > PDF > Export > Download. You will turn this in at the completion of the tutorial.**

# Class diagrams

Class diagrams provide a way to document the structure of a system by defining what classes there are within it and how they are related.

## Classes

As you would expect, the basic building block of a class diagram is the class. To add a new class to your class diagram, click or drag "**Class**" or "**Class 2**" tool in the toolbar. At a minimum, each class should have a name.

### Methods/Operations

Classes can contain additional information about the methods/operations they support. Methods define the ways in which objects interact with each other. Notice that there is plus sign (+) in front of each method. It means that the method is a Public method. Conversely, minus sign (-) is allowed and it is to indicate a Private method.

The most common properties that you'll want to modify will be to give the operation a name and to specify information about the parameters it uses. Parameters are often used to record the arguments the operation/method is expected to take and the type (if any) it uses to return information.

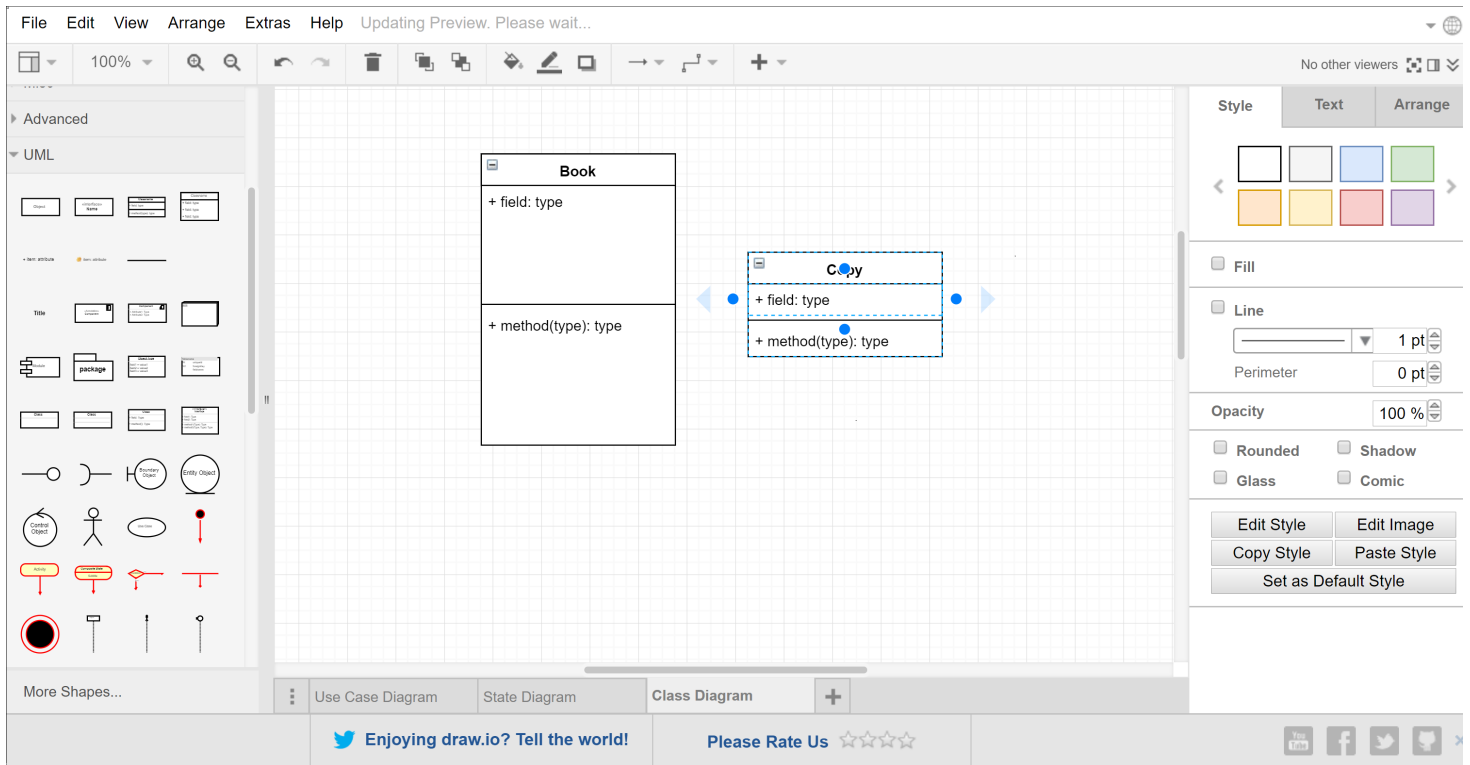
**Note:** You can also specify the type to be a class that you've added to your model (such as the Book class) as well.

### Attributes

In addition to operations, classes will also often have attributes which are used to describe the data contained in an object of the class (these attributes are often analogous to fields in many programming languages).

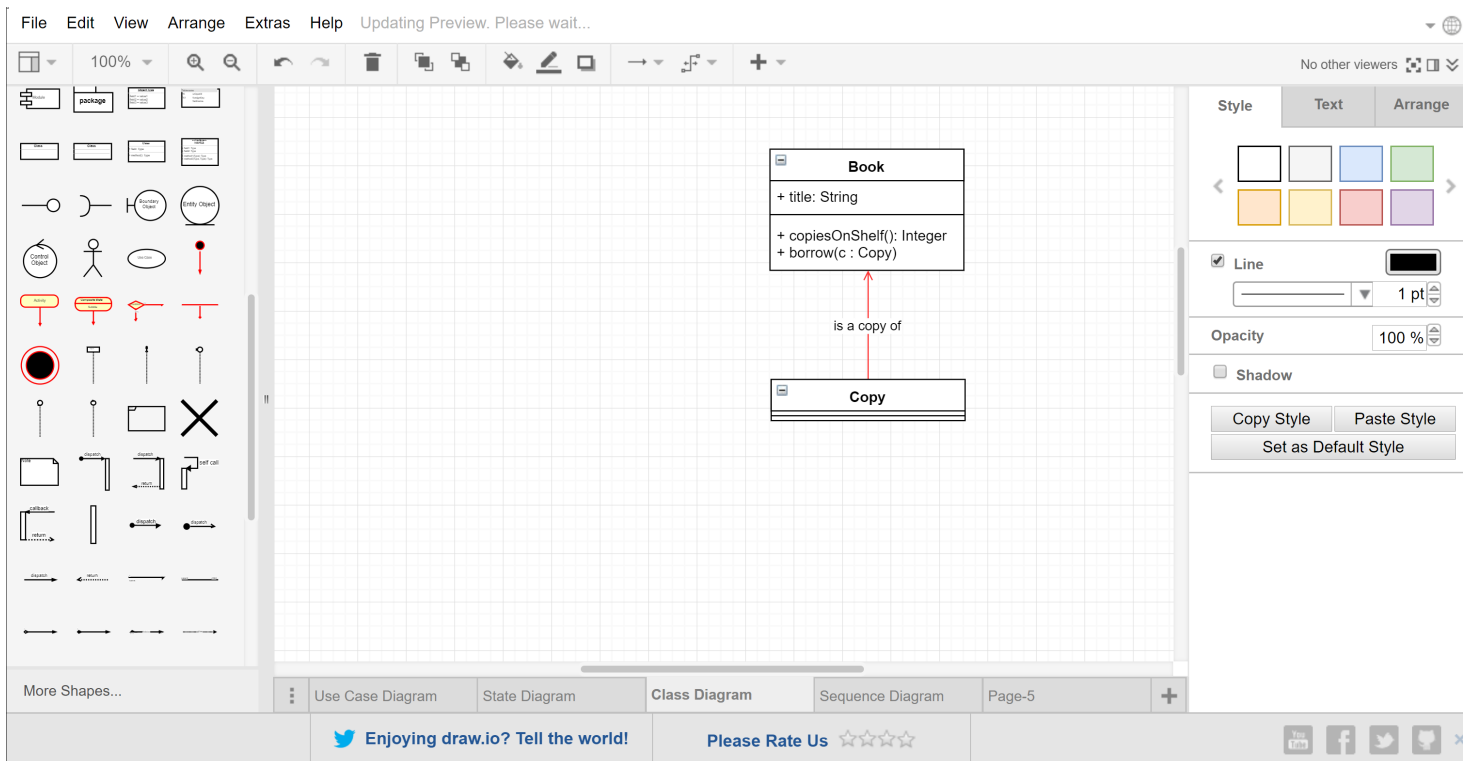
A sample class diagram using a class with attributes and properties is shown below.



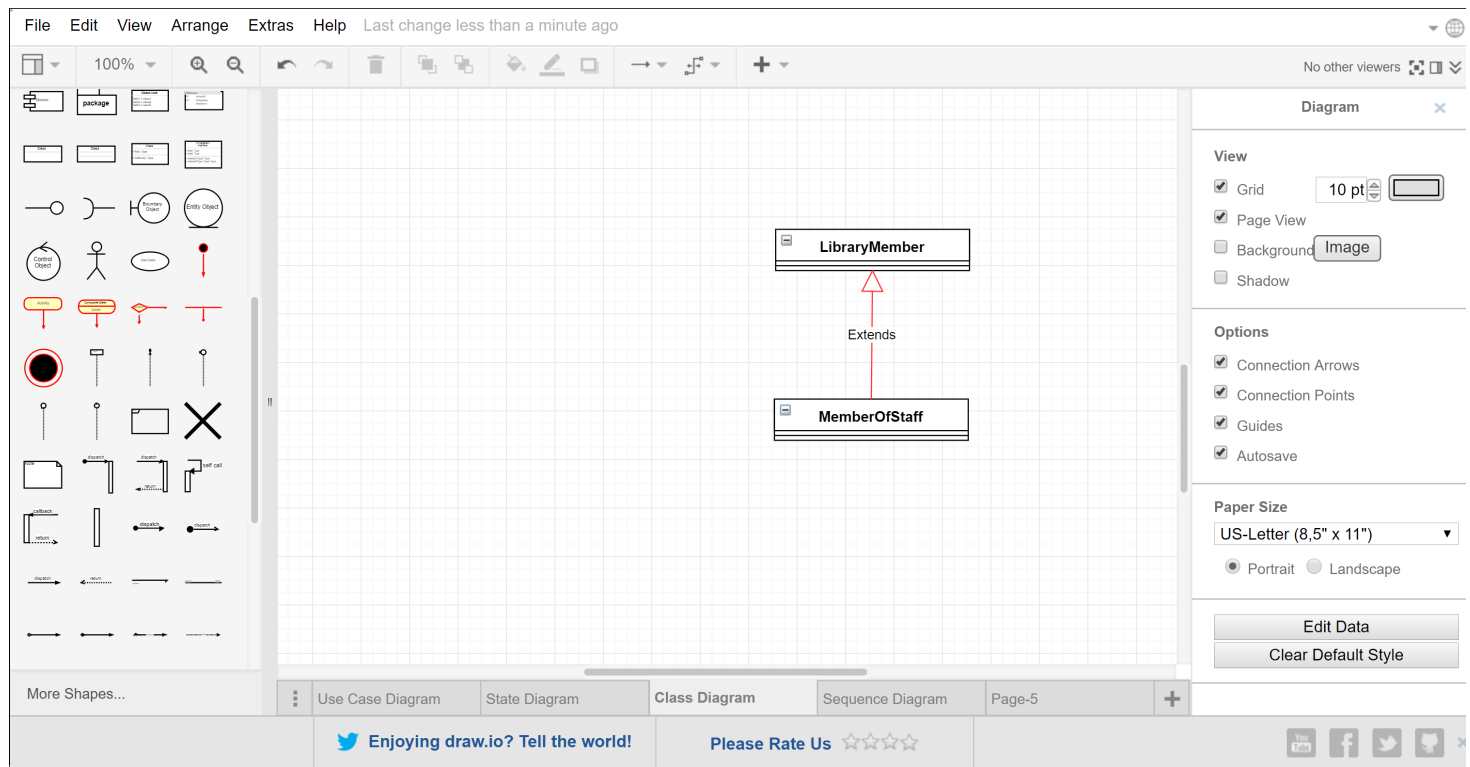


## Associations and Generalizations

The relationships between classes are often described using associations and generalizations. While classes often correspond the "nouns" in a system, associations correspond to the "verbs". They describe relationships such as "is a copy of" and "borrows/returns". Associations can be added between classes by clicking or dragging the **Association** tool in the toolbar, then adjusting the line ends to the classes to which you want to create the association. You'll generally want to give names to your associations. A sample class association is shown below.



Another important relationship between classes is a "generalization". Generalizations describe relationships that are often analogous to inheritance in programming languages. Generalizations can be added by clicking or dragging on the "**Generalization**" tool in the toolbar and then adjusting the line end without arrowhead on the more specific class and then adjusting the line end with arrow head to the class which is a generalization of it. A sample showing a generalization relationship between two classes is shown below.



### Exercise #3

Translate the following English statements into a representative class diagram that models the structure the library system:

The library system should allow for the representation of the following roles: **LibraryMember**, **MemberOfStaff**, **Journal**, **Book**, **Copy**. **LibraryMembers** can interact with **Copy** objects in that they can borrow/return them. Likewise, **MemberOfStaff** objects can borrow/return **Journals**. It's also important that we have a relationship between **Book** and **Copy** where **Copy** is a copy of a **Book**. Also, note that **MemberOfStaff** objects will always be **LibraryMembers** as well and should be able to do everything a **LibraryMember** can do.

Lastly, let's add some additional details to the **Book** class: every **Book** will keep track of its title, which is a string. It should respond to a **copiesOnShelf** message with an integer, and it should accept a **borrow** message that has an input parameter of type **Copy**.

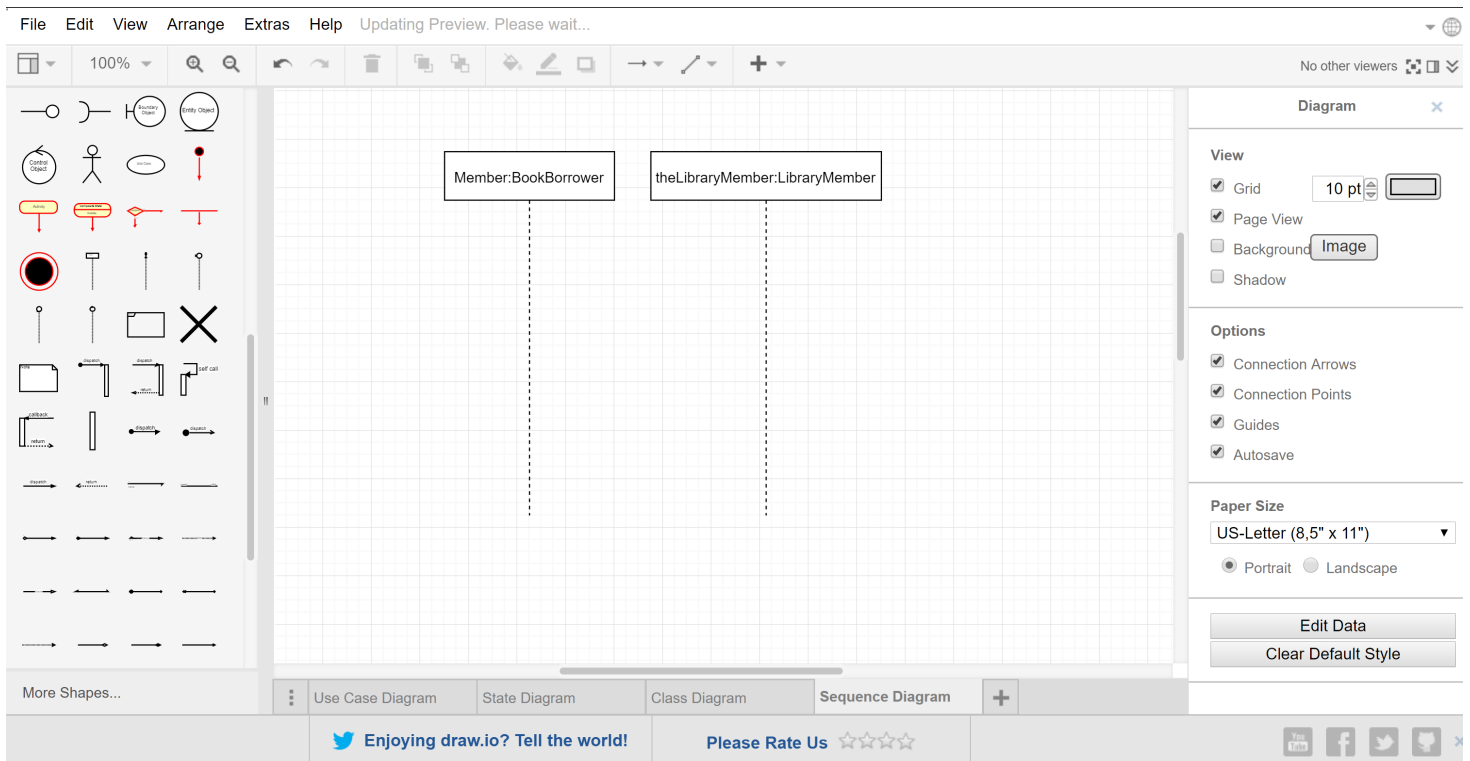
**Download the resulting diagram to a PDF file, File > Export As > PDF > Export > Download. You will turn this in at the completion of the tutorial.**

## Sequence Diagrams

The last UML diagram type we'll be working with is a sequence diagram. A sequence diagram is sub-type of a more broad category of UML diagrams called interaction diagrams. A sequence diagram shows how actors and/or objects interact with each other by providing a timeline of messages being passed from one to the other. Each object/actor has a representation of its own timeline (known as a lifeline) in which time is assumed to pass as we move from top to bottom of the diagram.

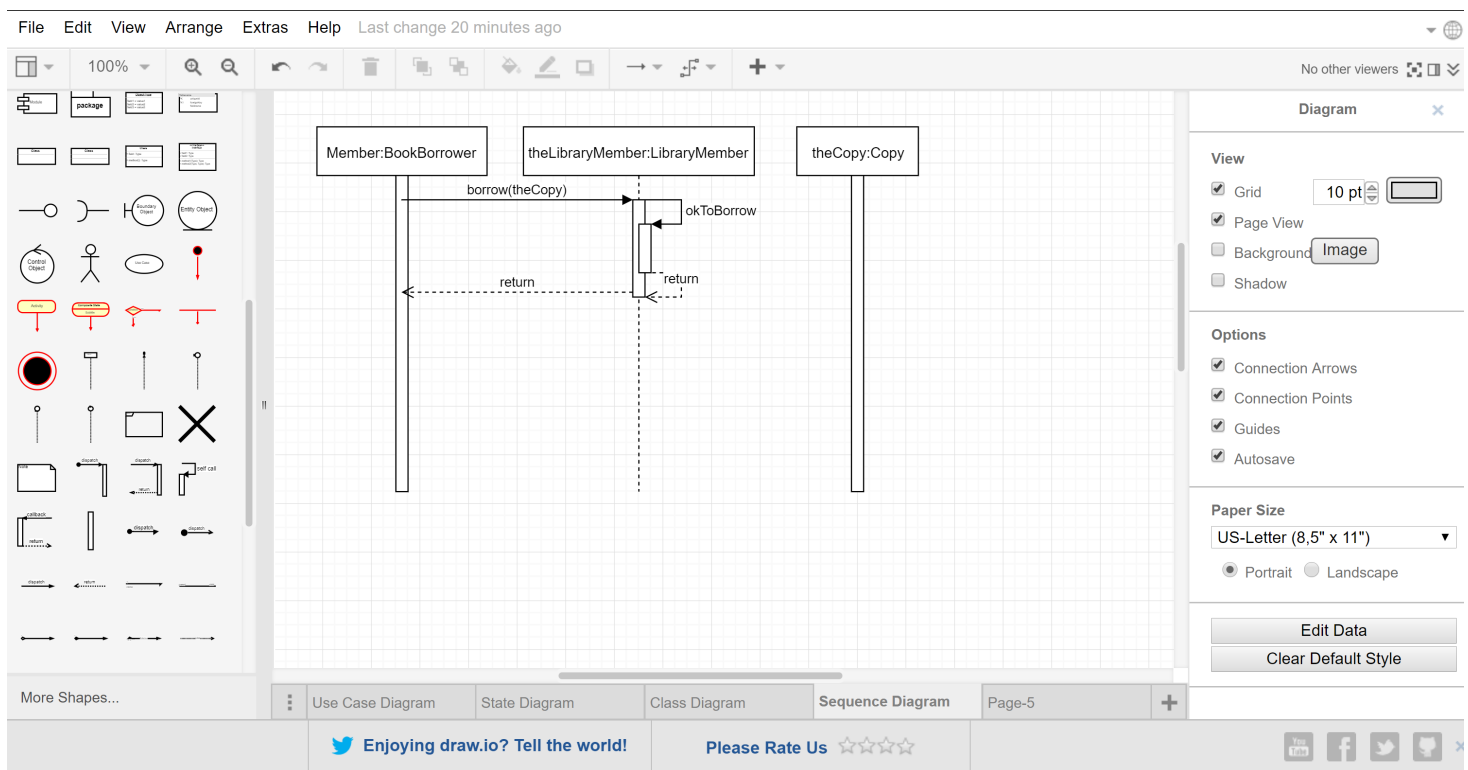
### Actors/Objects


The first step in defining the sequence diagram is to add the actors/objects that are involved in the interaction to be displayed. You can add generic objects, but in general it is best to associate the lifeline with a type that we've already defined in the model. For this, add **"Lifeline"** or **"Actor Lifeline"** that represent the type of the object or actor respectively to add to the diagram. Click or drag on this entry and it will appear in your canvas. This will create a lifeline that is associated with the selected type in your diagram. As usual, properties of this model member can be modified - in general at least a name should be provided. Also add **"Activation"** which is to indicate that the class/actor is being active or used along the length of the activation bar. Click or drag the activation bar to cover the dotted line of the lifeline. The screenshot below shows a sequence diagram that has been populated with two lifelines: on the left is an instance ("aMember") of an actor ("BookBorrow"), on the right there is an instance ("theLibraryMember") of a class ("LibraryMember").



## Messages

With the actor and object instances placed in the diagram, the next step is to depict the sequence of messages that are passed between each instance. Most often these message will represent the calling of a method, so to add these messages you'll select the **"Synchronous Invocation"** tool from the toolbar and then click or drag the instance which will be generating the message and adjust it to the instance which will be receiving it. This will create a message call and return action in the sequence diagram. As usual, you will need to add the name of the message. The snapshot below shows the sequence diagram from above updated with two messages: one from "aMember" to "theLibraryMember" requesting to borrow a specific copy, and another showing that "theLibraryMember" will send a message to itself to see if it's alright for the member to borrow the copy. The second diagram has also been updated to include a third lifeline for "theCopy", an instance of the "Copy" class, since "theCopy" is involved in the interaction between "BookBorrower" and "theLibraryMember".



**Note:** To create a message-to-self you'll need to use the **"Self-Call"** tool in the toolbar and drag to the LibraryMember lifeline. Then, use **"Dispatch"** tool in the toolbar and drag it to the bottom part of the Self-Call entry. Select the Dispatch entry and format it using  so that it could make a u-turn line.

## Exercise #4

Translate the following English statements into a representative sequence diagram which models the interaction necessary for a member to borrow a book:

Four actors/objects will be involved in a book borrowing interaction: a BookBorrower named aMember, a LibraryMember named theLibraryMember, a Copy named theCopy and a Book named theBook. aMember will request to borrow theCopy by sending a message to theLibraryMember. theLibraryMember will then check with itself to make sure it's ok for the borrowing to occur and then notify theCopy that it is going to be borrowed. theCopy will then in turn notify theBook that it has been borrowed. Once all of these nested operations are complete, aMember is now borrowing the book!

**Download the resulting diagram to a PDF file, File > Export As > PDF > Export > Download. You will turn this in at the completion of the tutorial.**

## What to Submit

Name the generated PDF files from all of the four exercises appropriately, include your name as LASTNAME\_FIRSTNAME, and submit them [here](#).

<https://utexas.instructure.com/courses/1266665/assignments/4876009>

