Seminar Session 01. Software Requirements Diagram (using Enterprise Architect)

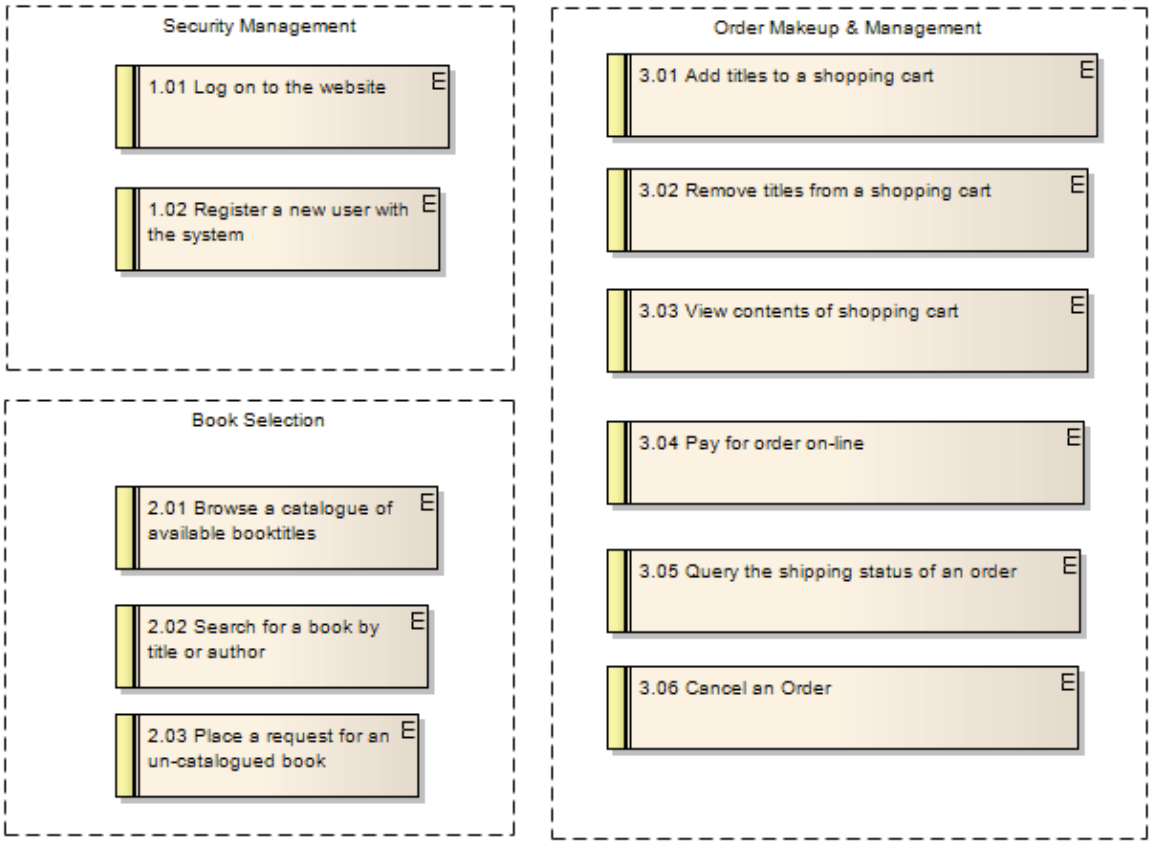
Introduction

Requirements in **software** describe the *functionalities* of the system that is going to be developed. They define a property that a system must be able to perform and convey what users actually expect from the final product. The process of gathering the software requirements from the client, analyze and document them is known as **requirement engineering**.

A **Software Requirements diagram** describe a system's features as a visual model. It also represents how requirements are related to each other and to other elements in the model (Use Cases or Business Rules, for instance).

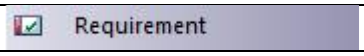
Objective

In this session, you will learn how to design a **Software Requirements diagram** using **Enterprise Architect**. You'll learn how to add requirements to a diagram and group them according to their functionality based on a problem description analysis.

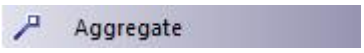


Software Requirements Diagrams in Enterprise Architect

You can include the following elements in this type of diagrams:

Element	Description
	**Packages** organize project contents. When added onto a diagram they can be used for structural or relational depictions.
	A **Requirement** captures the details of a system requirement.
	A **Feature** is a small function or characteristic expressed in client-valued terms as a satisfaction of a requirement.
	An **Object** is a specific instance of a Class at run time

Moreover, you can depict relationships between requirements and other elements by using connectors such as the following:

Connector	Description
	An **Aggregation** connector is a type of association that shows that an element contains or is composed of other elements.
	A **Generalization** is used to indicate inheritance.
	An **Association** implies that two model elements have a relationship. It is usually implemented as an instance variable in one or both Classes.
	An **Implements** connector represents that the source object realizes its destination object

Problem Description

A library database needs to store information pertaining to:

- Its customers
- Its workers
- The physical locations of its branches,
- And the media stored in those locations (two media types are considered: books and videos).

The library must keep track of the status of each media item: its location, status, descriptive attributes, and cost for losses and late returns. Books will be identified by their ISBN, while movies by their title and year. In order to allow multiple copies of the same book or video, each media item will have a unique ID number.

Customers will provide their name, address, phone number, and date of birth when signing up for a library card. They will then be assigned a unique user name and ID number, plus a temporary password that will have to be changed.

Checkout operations will require a library card, as will requests to put media on hold. Each library card will have its own fines, but active fines on any of a customer's cards will prevent the customer from using the library's services.

The library will have branches in various physical locations. Branches will be identified by name, and each branch will have an address and a phone number associated with it. Additionally, a library branch will store media and have employees.

Employees will work at a specific branch of the library. They receive a paycheck, but they can also have library cards; therefore, the same information that is collected about customers should be collected about employees.

Functions for customers (users):

- Log in
- Search for media based on one or more of the following criteria:
    - type (book, video, or both)
    - title
    - author or director
    - year
- Access their own account information:

    - Card number(s)
    - Fines
    - Media currently checked out
    - Media on hold
- Put media on hold
- Pay fines for lost or late items
- Update personal information:
    - Phone numbers
    - Addresses
    - Passwords

Functions for librarians (employees) are the same as the functions for customers plus the following:

- Add customers
- Add library cards and assign them to customers
- Check out media
- Manage and transfer media that is currently on hold
- Handle returns
- Modify customers' fines
- Add media to the database
- Remove media from the database
- Receive payments from customers and update the customers' fines
- View all customer information except passwords

Requirements Analysis

In general, requirements can be identified as either **Functional** or **Non-Functional**:

- Functional Requirements: They describe behavior: what should/must a software system do? Example: A customer can search for media elements by type, title, author, director or year.
- Non-Functional Requirements: They define constraints, standards or quality attributes: how will the system accomplish its requirements? Example: Media search results should be retrieved in less than 7 seconds.

Based on this, let's identify the **Functional Requirements** of the current problem and group them:

- Accessing the system:
    - Customers and librarians can log into the system.

- Media management:
    - Customers and librarians can search for media.
    - Librarians can add media to the database
    - Librarians can remove media to the database

- Account information:
    - Customers and librarians can access their own account information.
    - Librarians can add customers into the system
    - Librarians can access customers information

- Personal information:
    - Customers and librarians can update their personal info.

- Library cards:
    - Librarians can create library cards
    - Librarians can assign a library card to a customer

- Reservations:
    - Customers and librarians can put media on hold.
    - Librarians can check out media
    - Librarians can manage currently-on-hold media
    - Librarians can transfer currently-on-hold media
    - Customers and librarians can pay fines for lost/late items
    - Librarians can handle returns
    - Librarians can modify customers' fines
    - Librarians can receive payments from customers

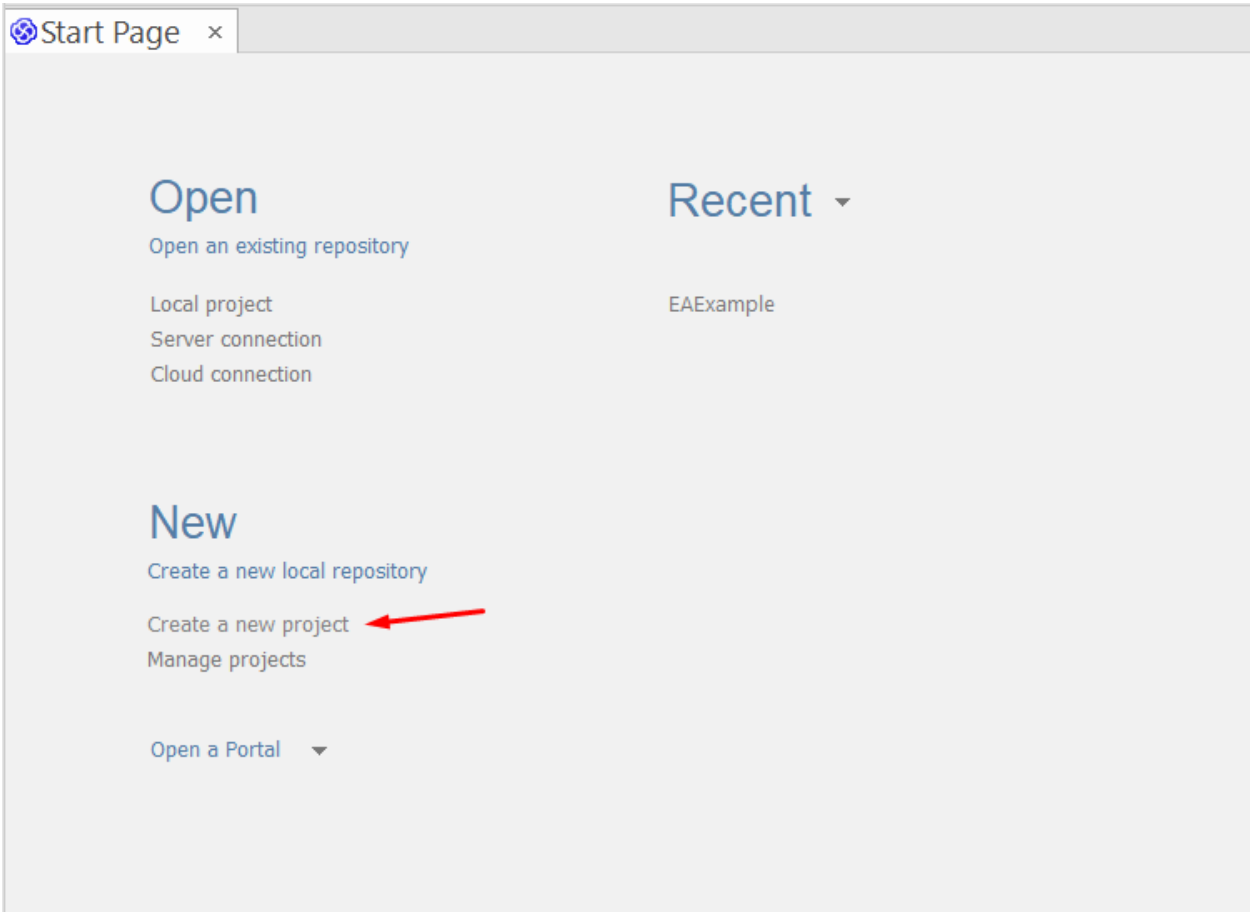
And think about the following **Non-Functional Requirements**:

- The system can be accessed by any user from the Internet using a web browser.
- The web system is available 24/7 (all day, all week).
- Displayed information is refreshed every minute.
- Displaying search results must take no longer than 5 seconds.
- Customers are assigned an initial, random login password after created. They must update it after the first successful login.
- The system is user-friendly.
- The website is available in English and Czech languages.

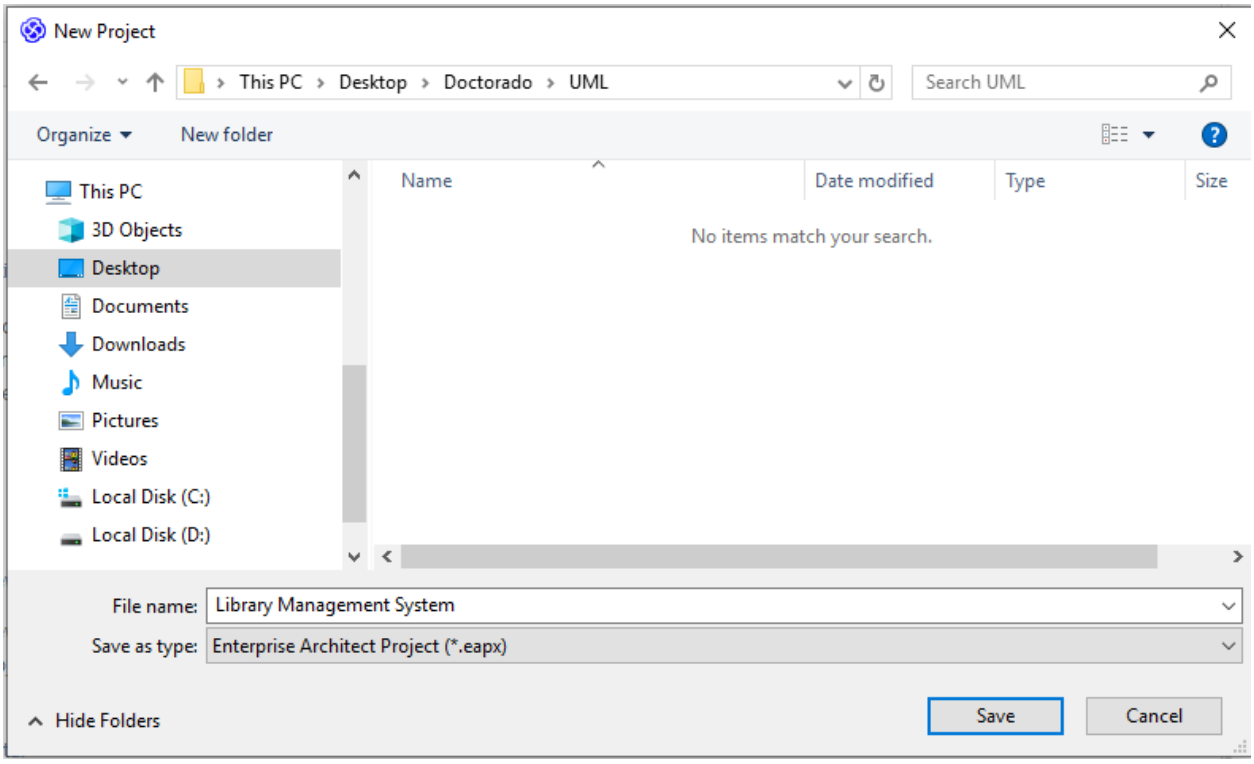
Modeling requirements

Step 1. Open Enterprise Architect:

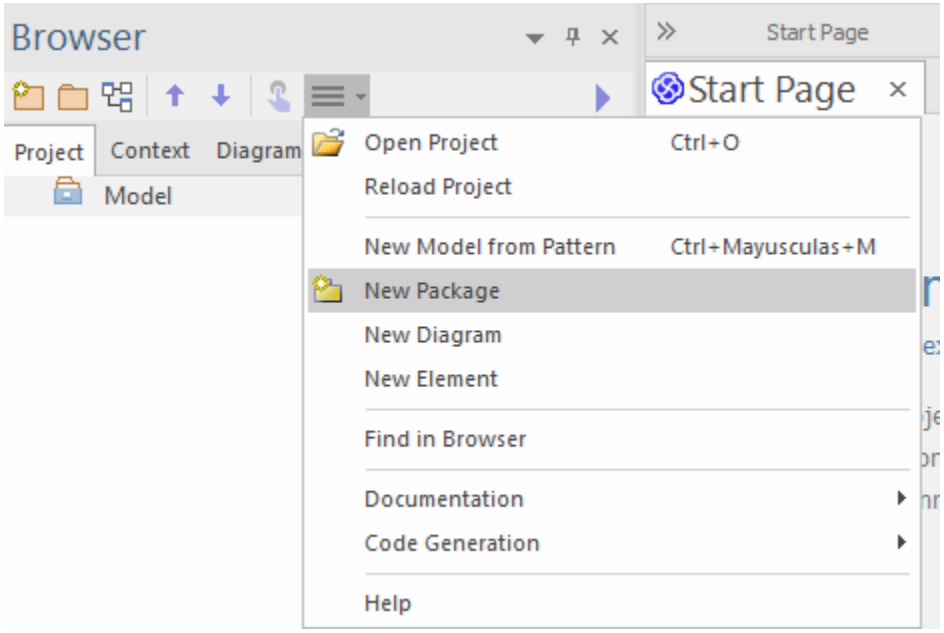
Create a **new project**:



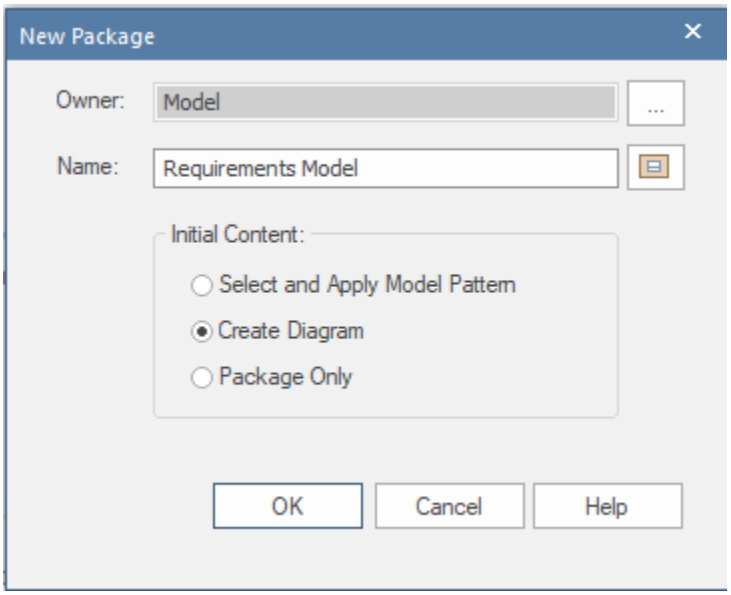
The name of the project is **Library Management System**



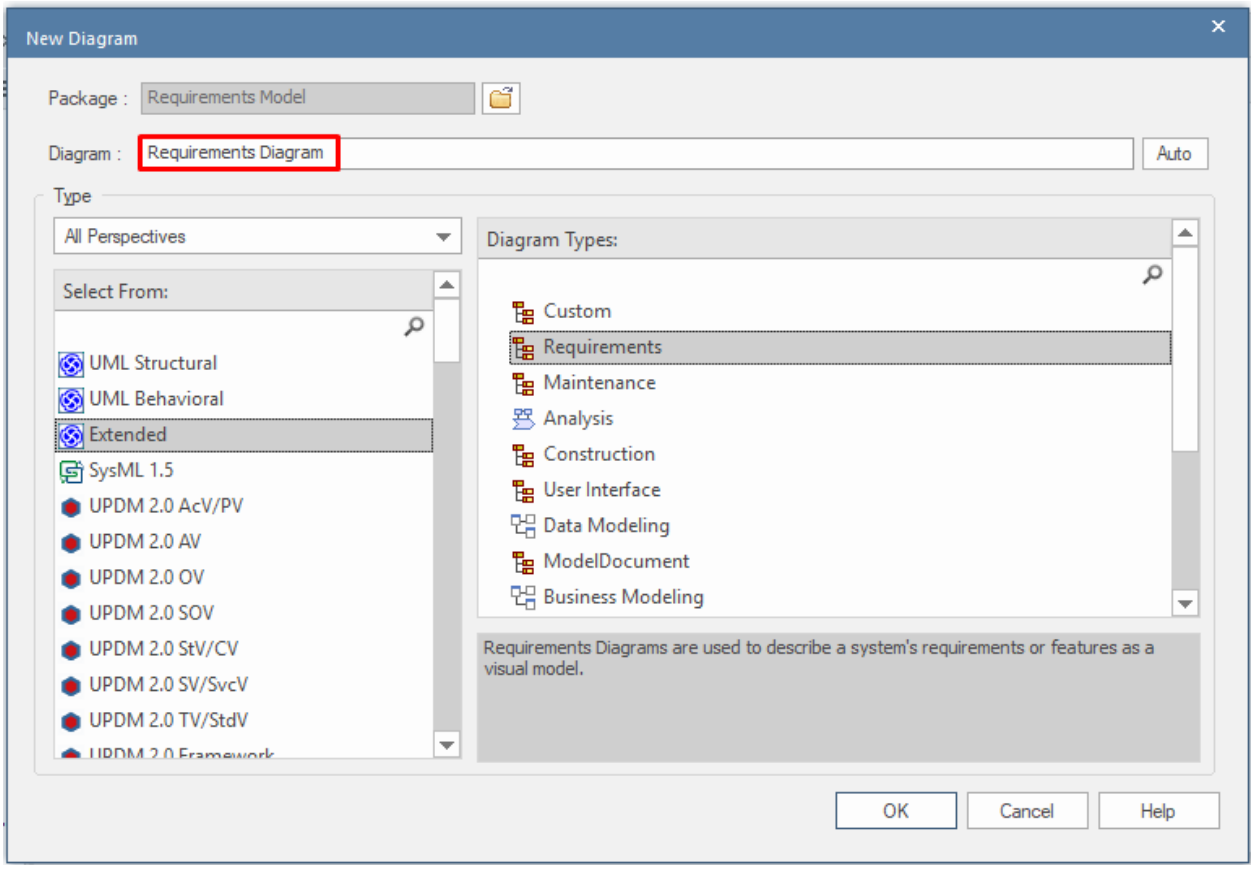
Step 2. On the left **Browser** panel, click on the icon and select **New Package**:



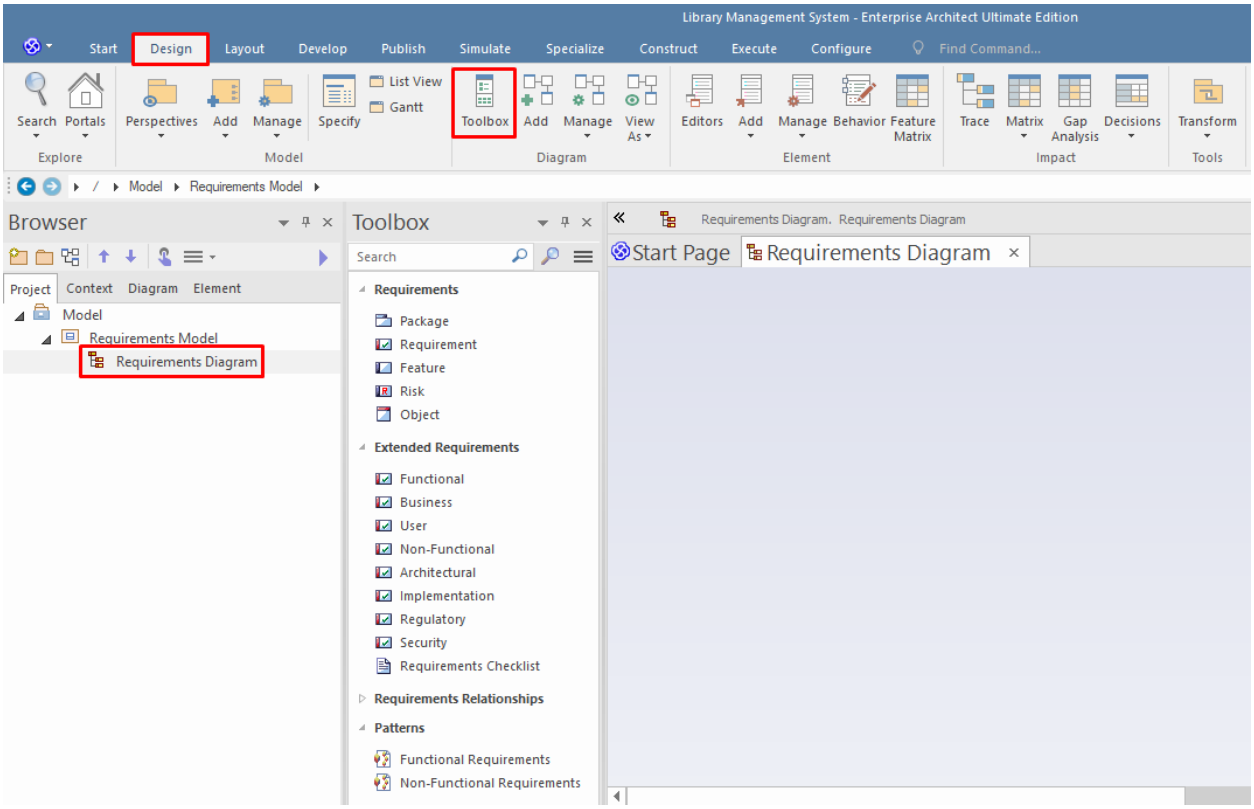
The name of this new element is **Requirements Model**. Choose **Create Diagram** and click on **OK**.



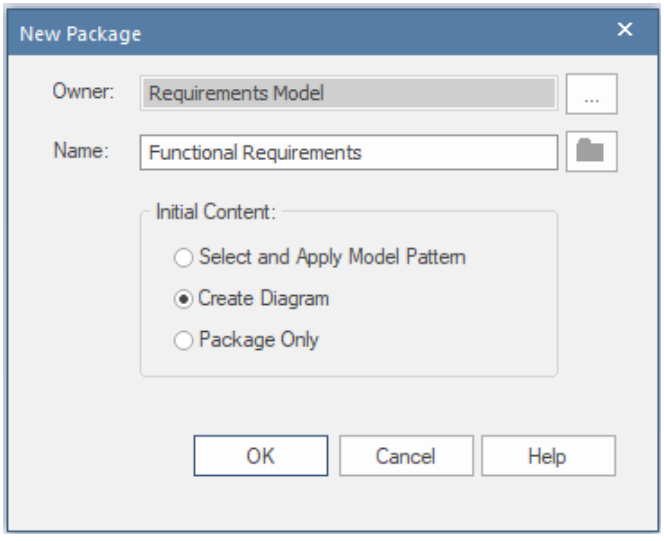
Now look for the **Requirements** diagram, which can be found under the **Extended** category. Also, change the Diagram name to **Requirements Diagram**.



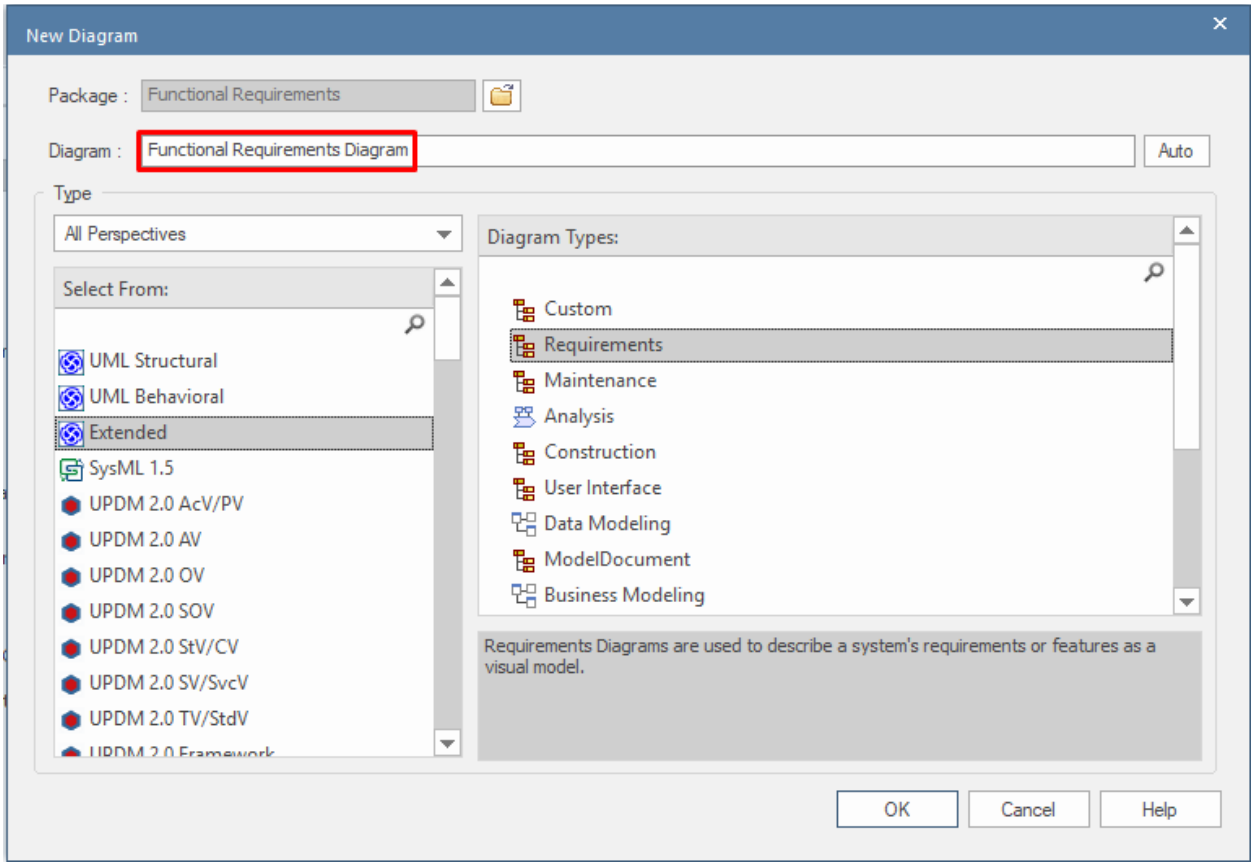
Step 3. Double click the **Requirements Diagram** from the left panel. Under **Design**, click on **Toolbox:**



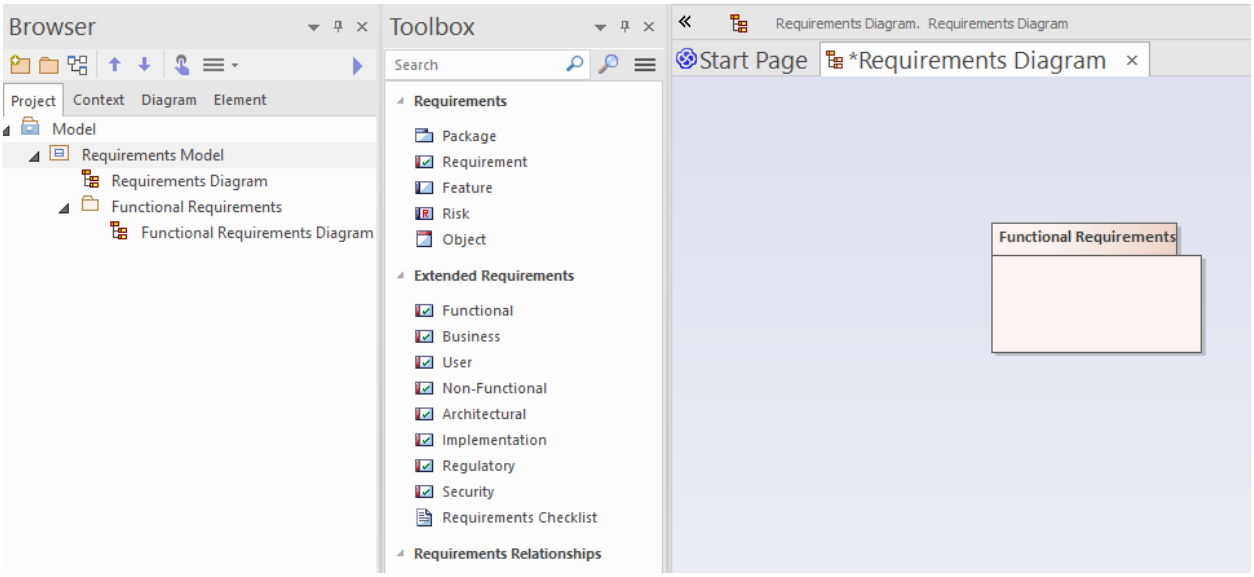
Next, drag the **Package** element (from the toolbox) and drop it on the Requirements Diagram work area (the blank section on the right). The following familiar window appears. Name the new package **Functional Requirements** and select **Create Diagram**.



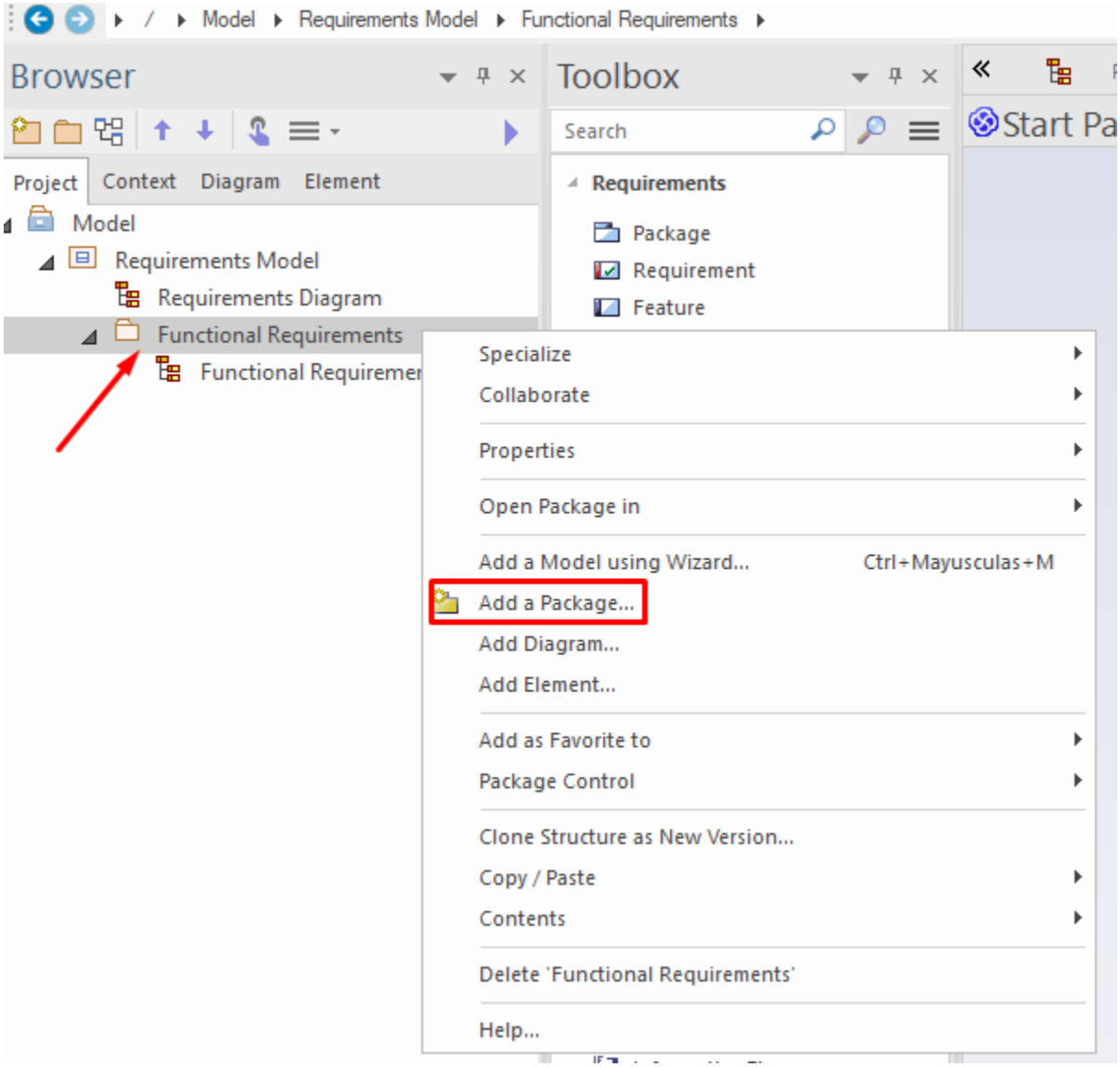
Again, select a **Requirements Diagram** and name it **Functional Requirements Diagram**:



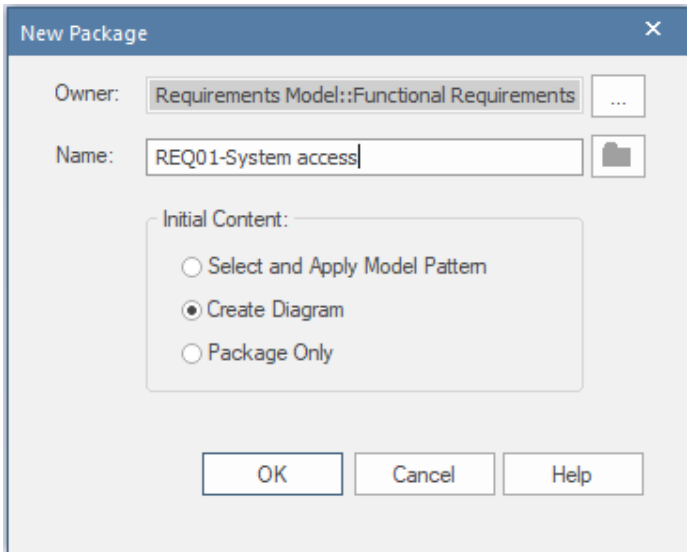
This is the current situation so far:



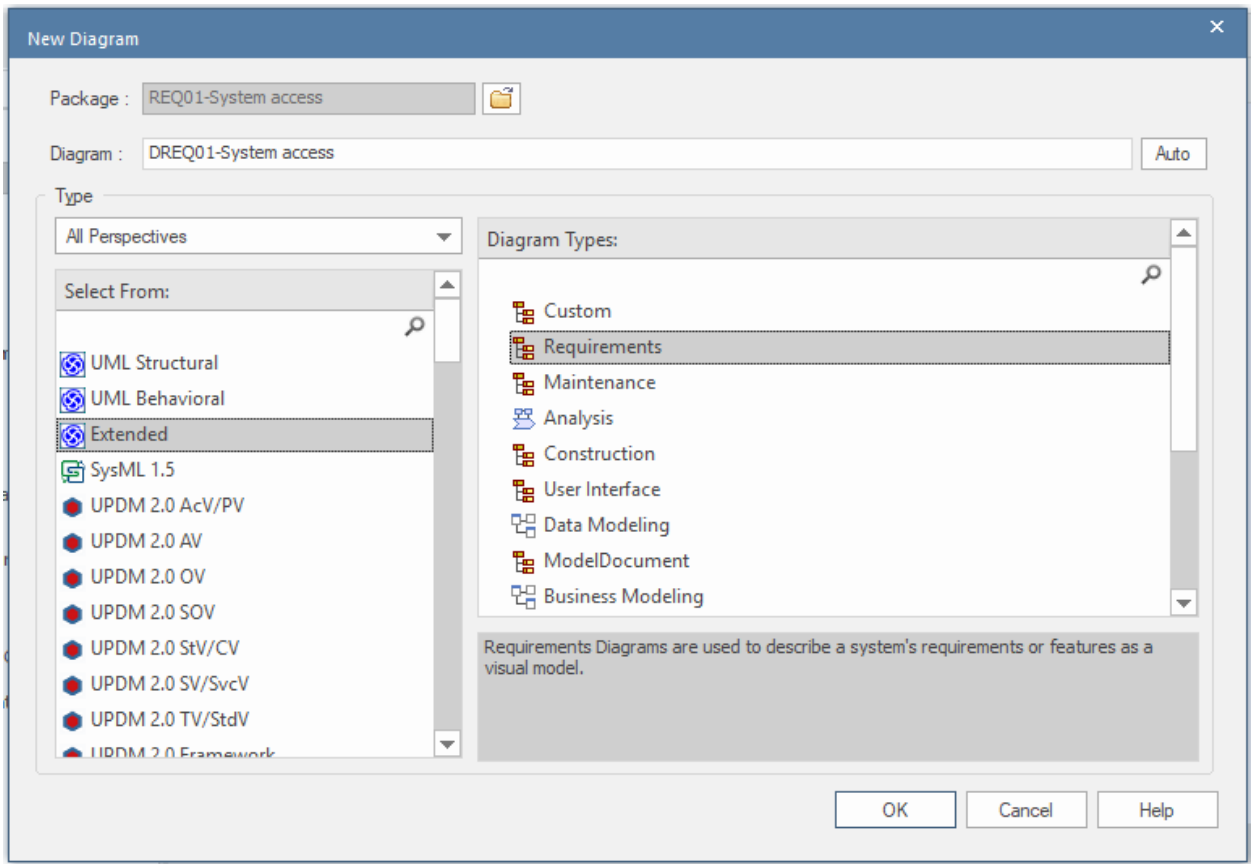
Step 4. Let's add a package in a different way this time. Right click the **Functional Requirements** package (on the left panel) and select **Add a Package…**



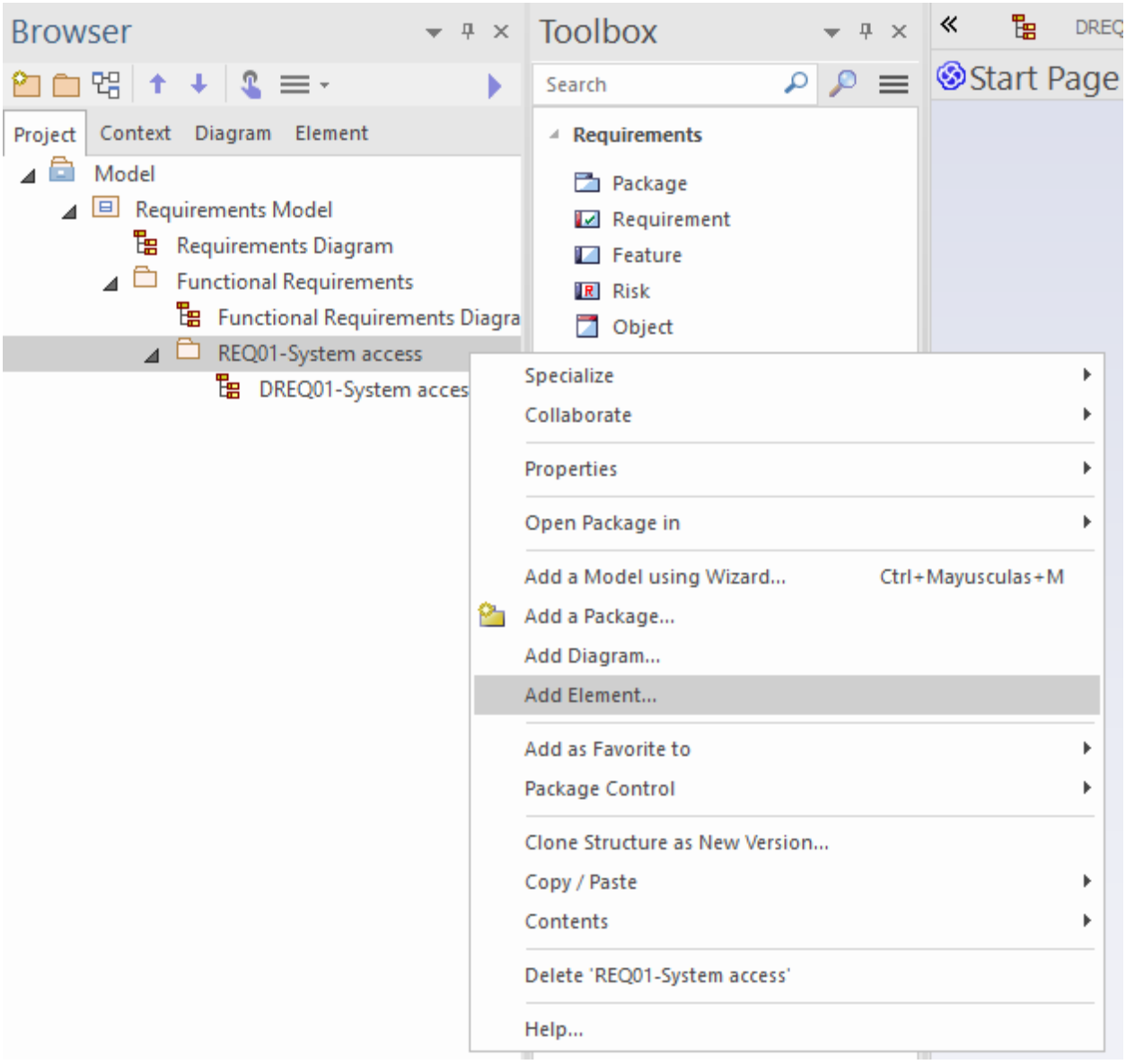
Name it **REQ01-System access** and select **Create Diagram**:



The name of this new element will be **DREQ01-System access.** Don't forget to select **Requirements** from the **Extended** category

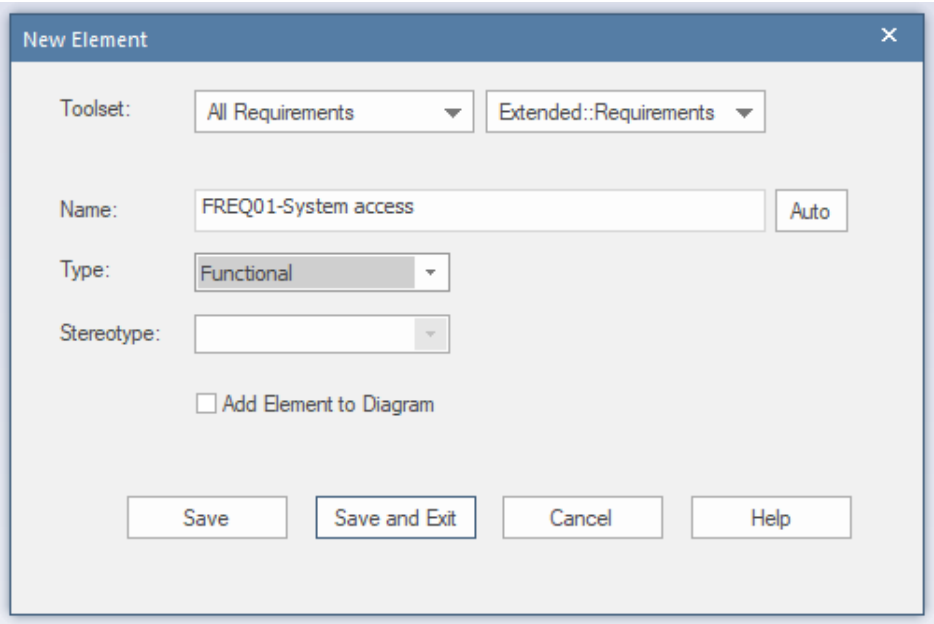


Step 5. Right click the **DRE01-System access** package (on the left panel) and select **Add Element…**

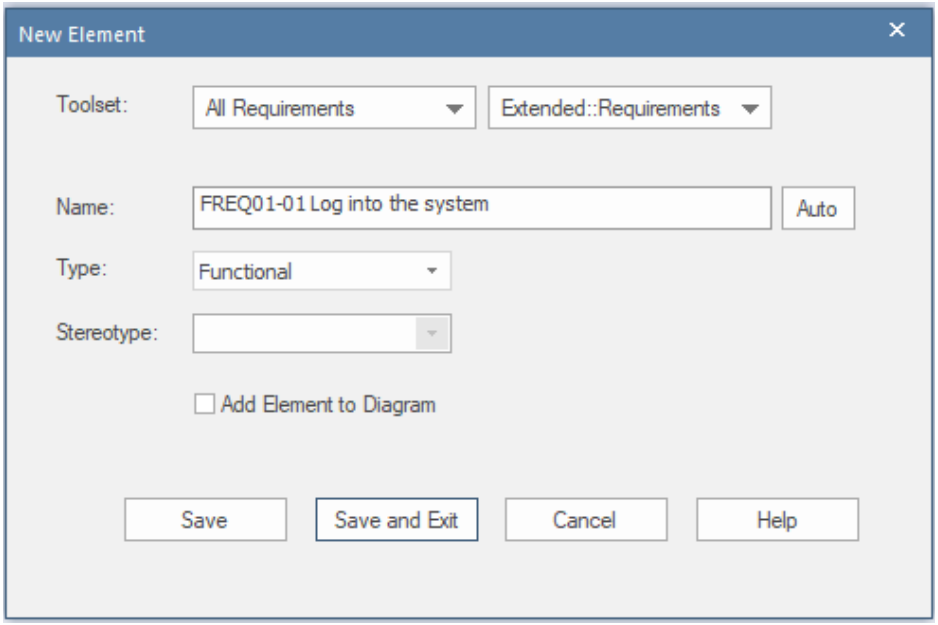


On the next window:

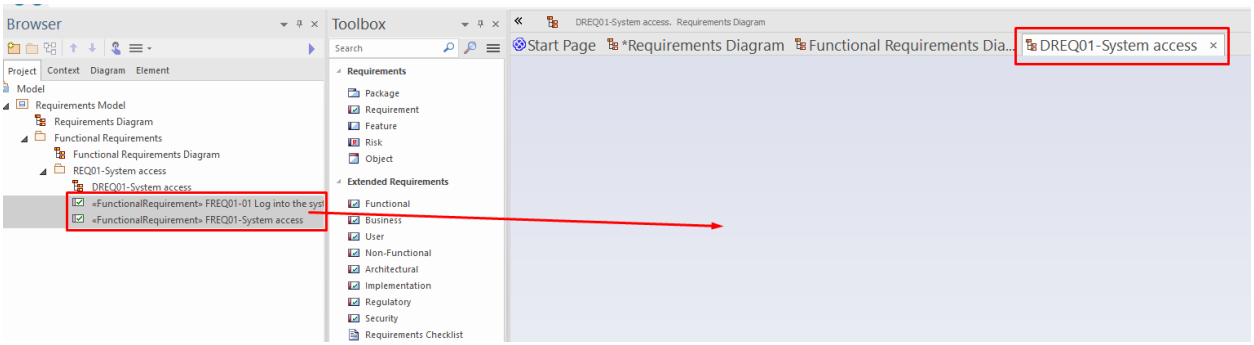
- In Toolset, select **All Requirements → Extended Requirements.**
- Name it **FREQ01-System access**.
- Select **Functional** Type.
- Click on **Save**



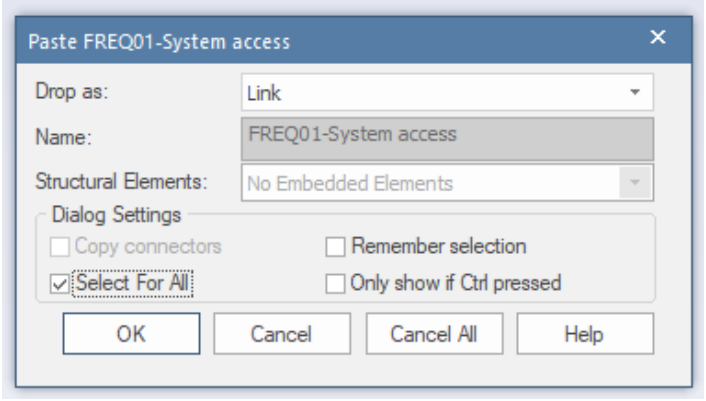
You'll be prompted to add another element. It is similar to the previous one, except that the name is **FREQ01-01 Log into the system**. This time, click on **Save and Exit**.



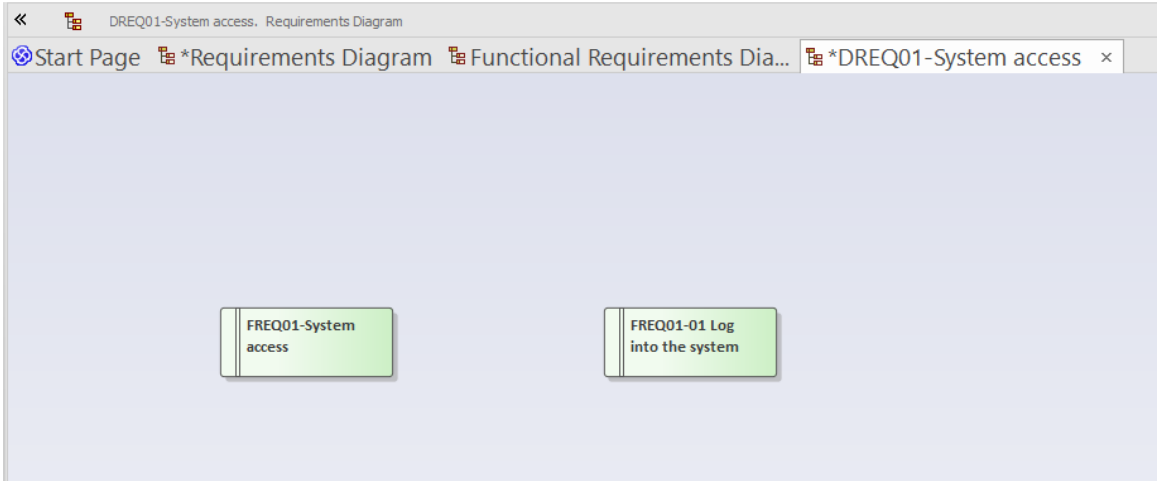
You'll notice that both new elements have been added to the left panel. Select both and drag and drop them to the **DREQ01-System access** diagram on the right.



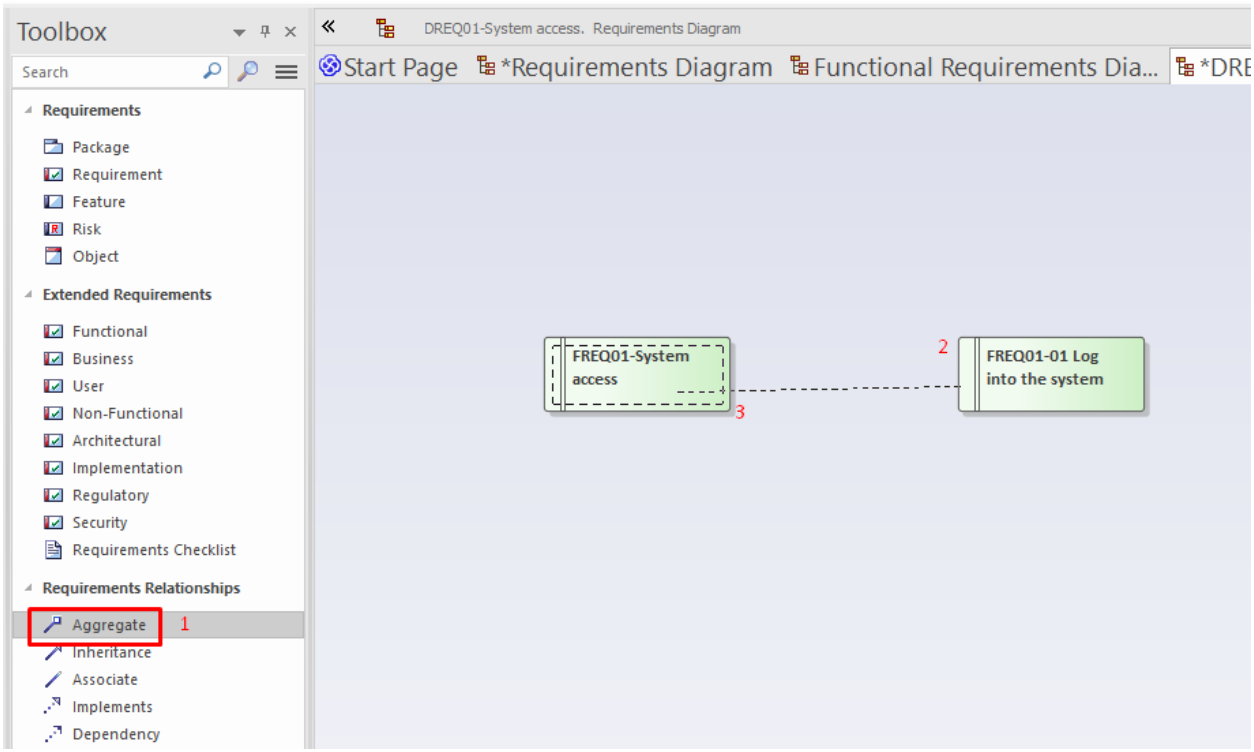
On the next window, select **Link,** then check the **Select For All** option and click on **OK.**



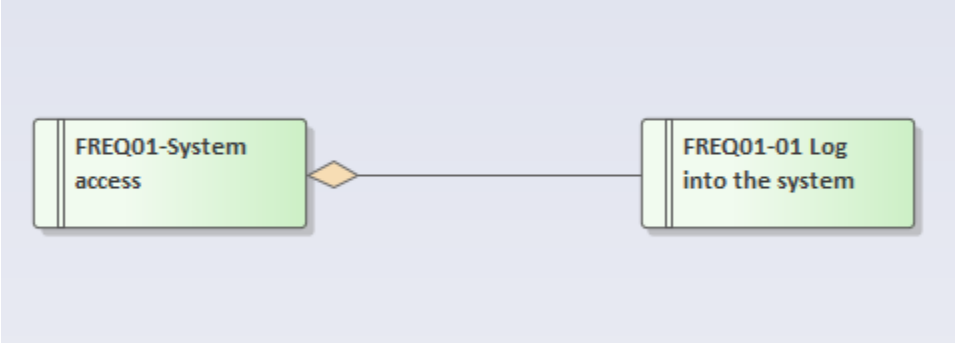
Both requirements will be added to the diagram. Arrange them as indicated in the next image:



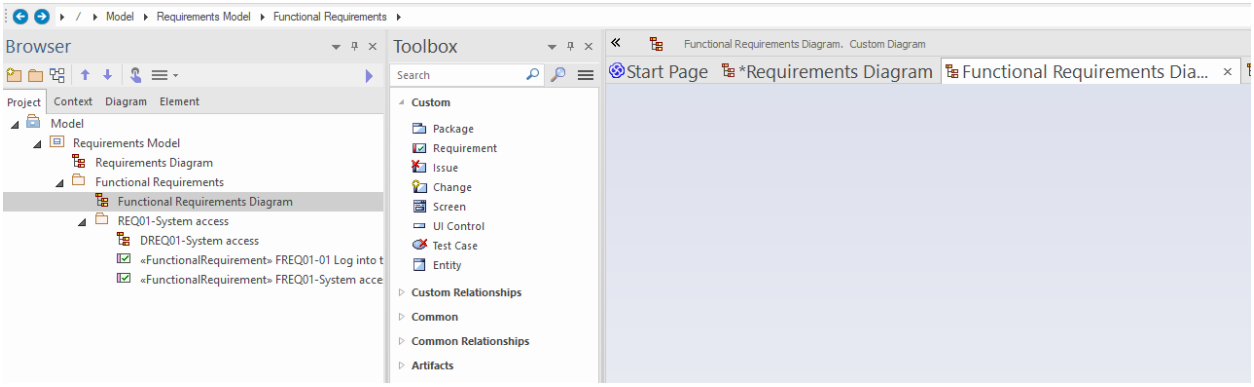
Step 6. Under the Toolbox, click on the **Aggregate** relationship. Then click on the **FREQ01-01 Log into the system** requirement and drop the arrow to the **FREQ01-System access**:



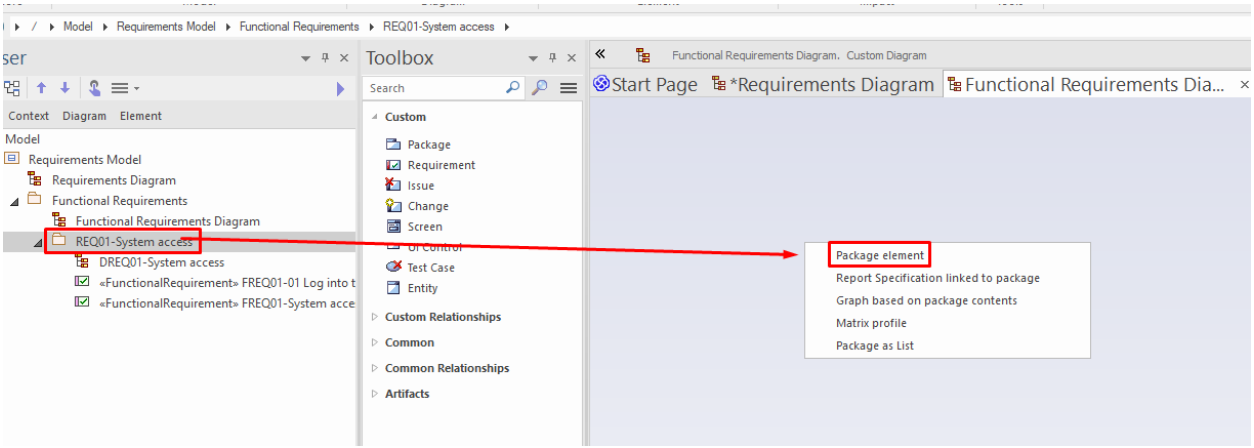
This is the desired output, which means that the FREQ01 element is the parent:



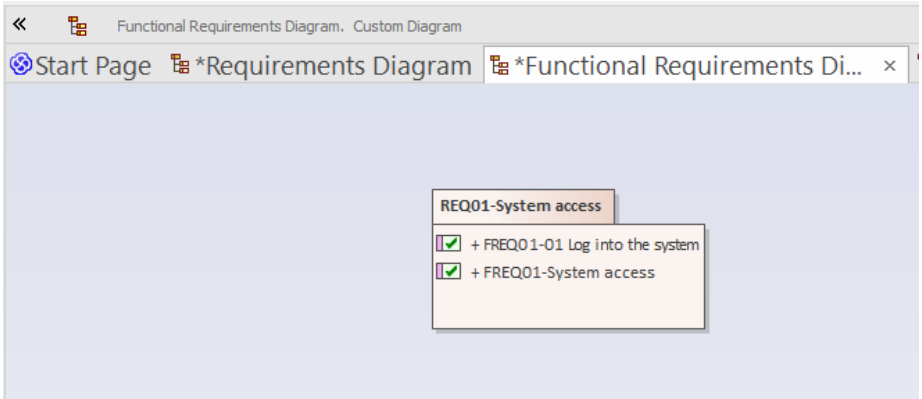
Step 7. On the left panel, double click the **Functional Requirements Diagram** element. At this moment, the workspace on the right should be empty:



Now, from the left section drag the **REQ01-System access package** and drop it on the right panel. A new box will appear. Select **Package element**:

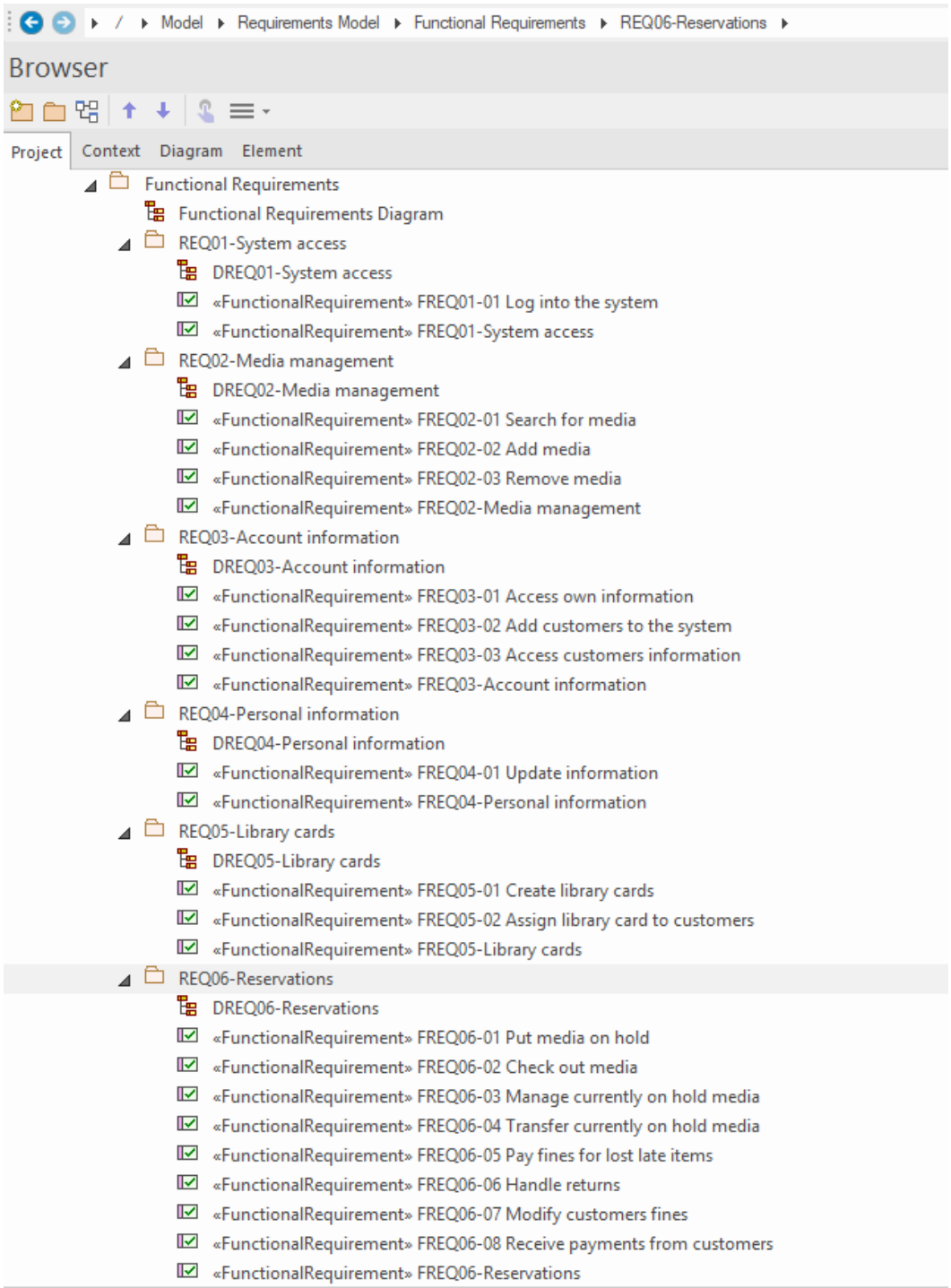


The element (along with all the requirements) will be added to the diagram:

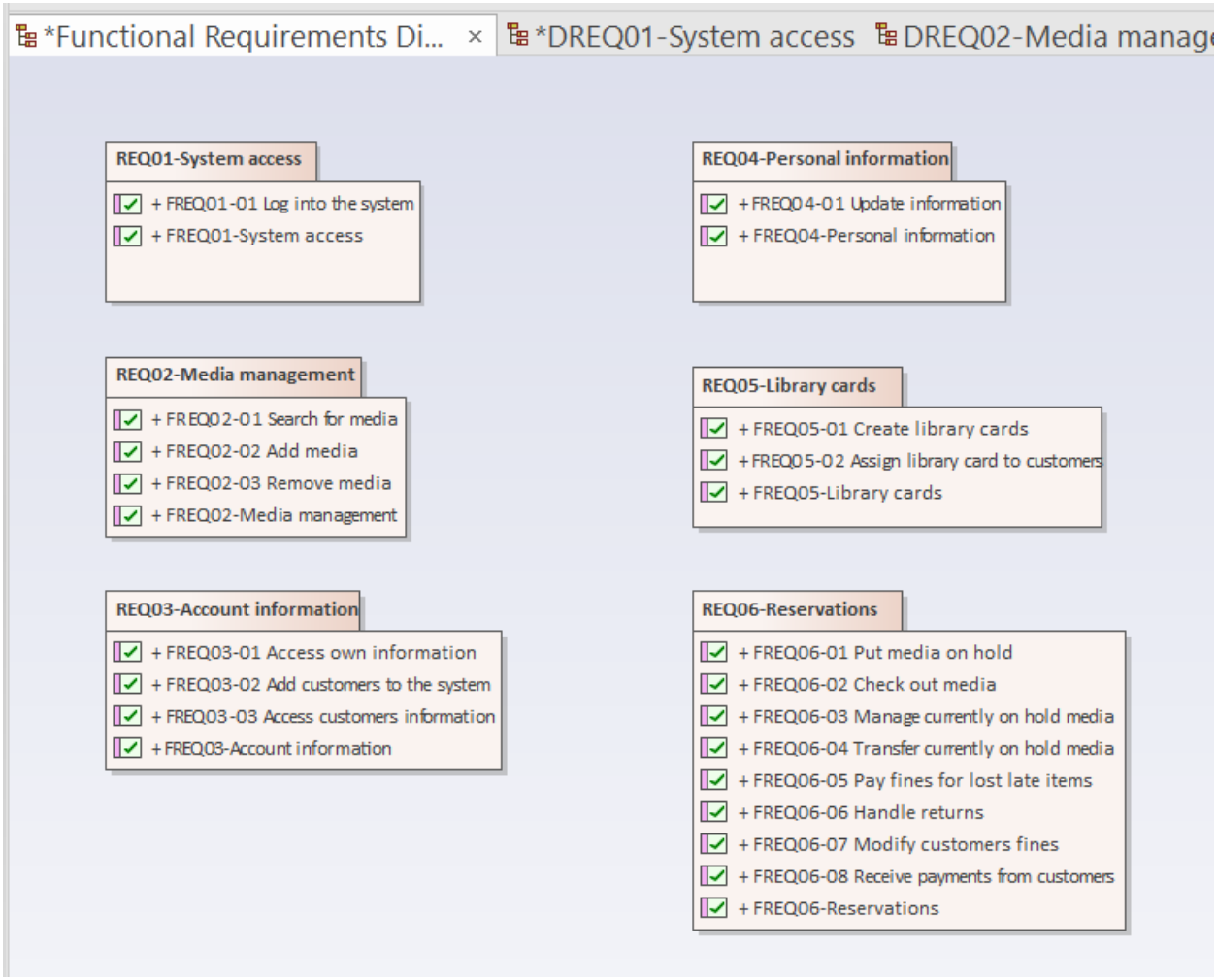


Step 8. Repeat **Steps 4-7** to add the rest of the **functional requirements**. The following images depict the expected result:

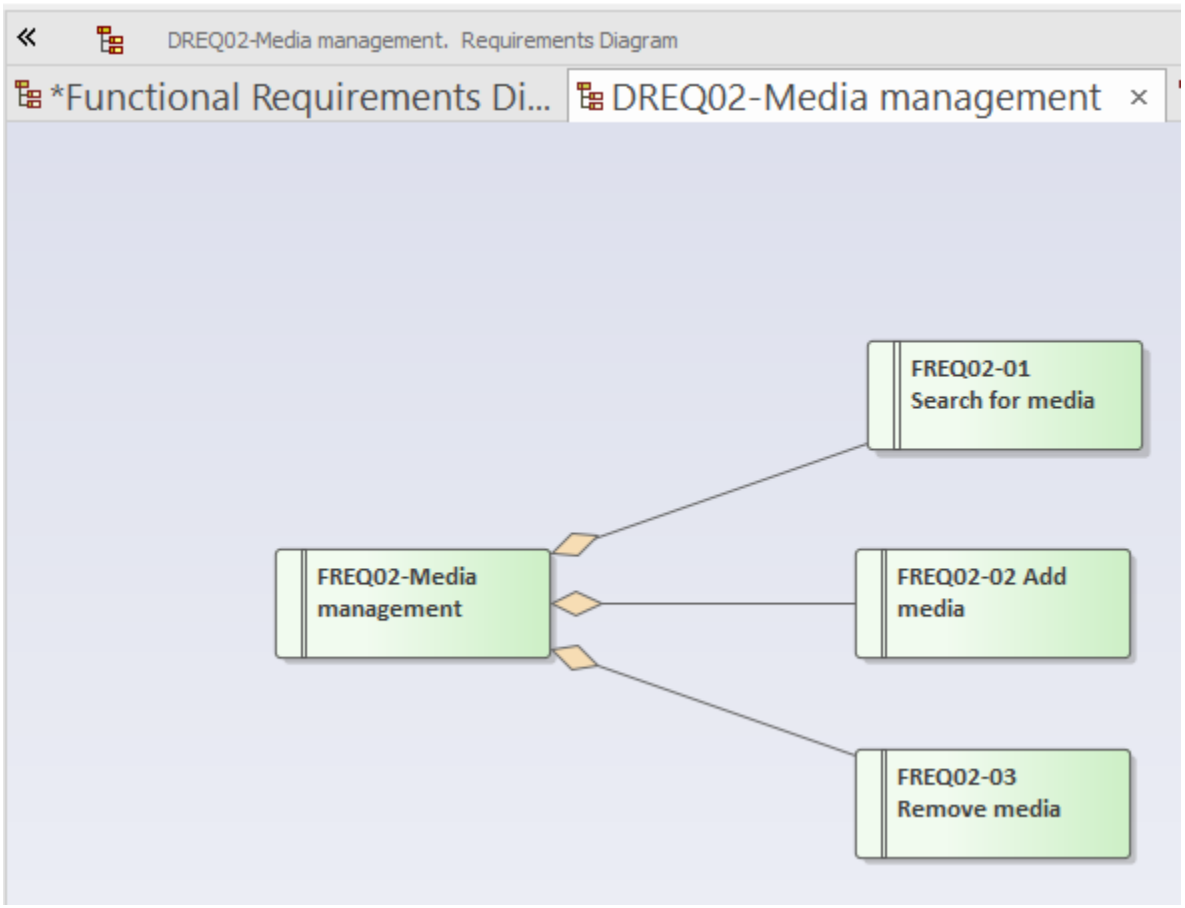
a) Functional Requirements Package:



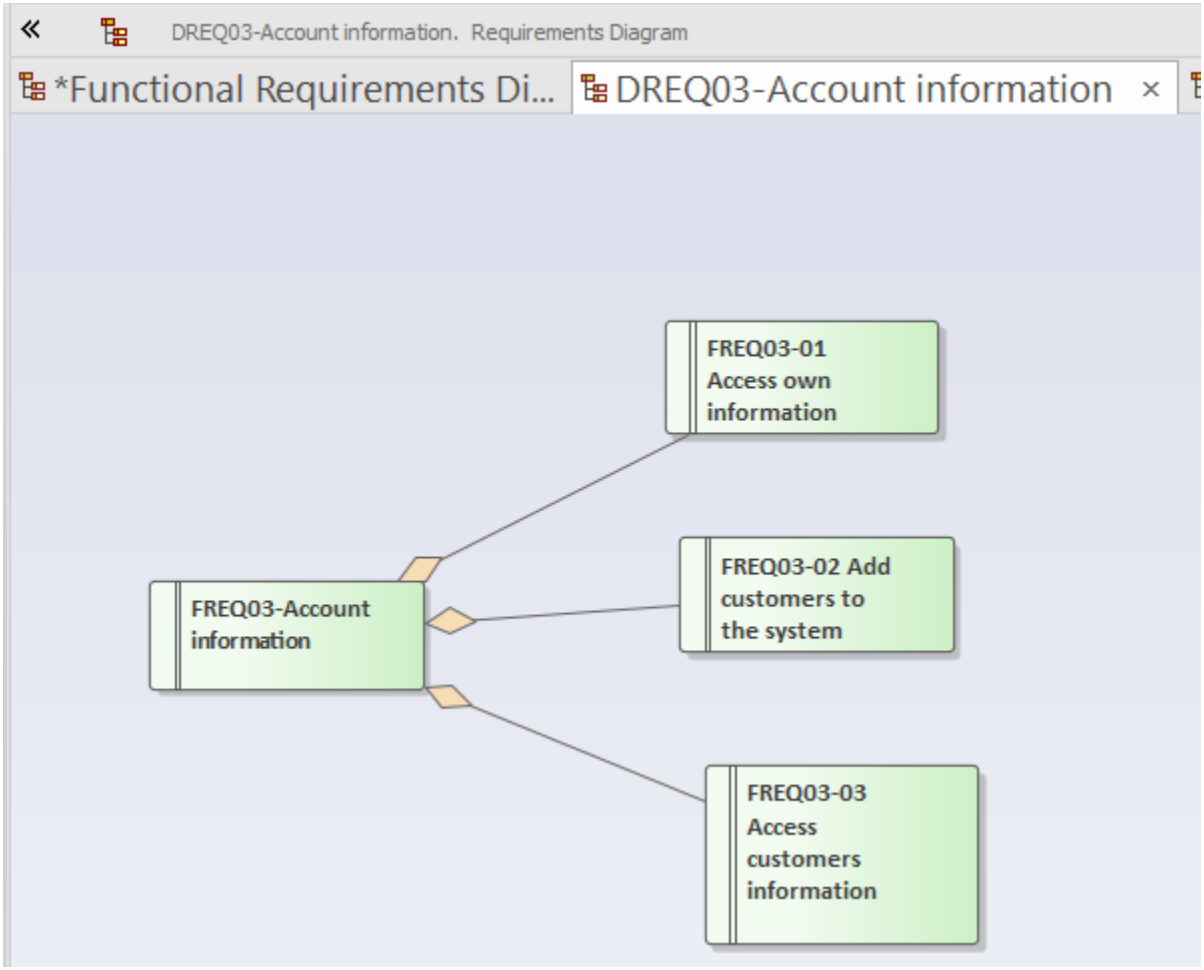
b) Functional Requirements Diagram:



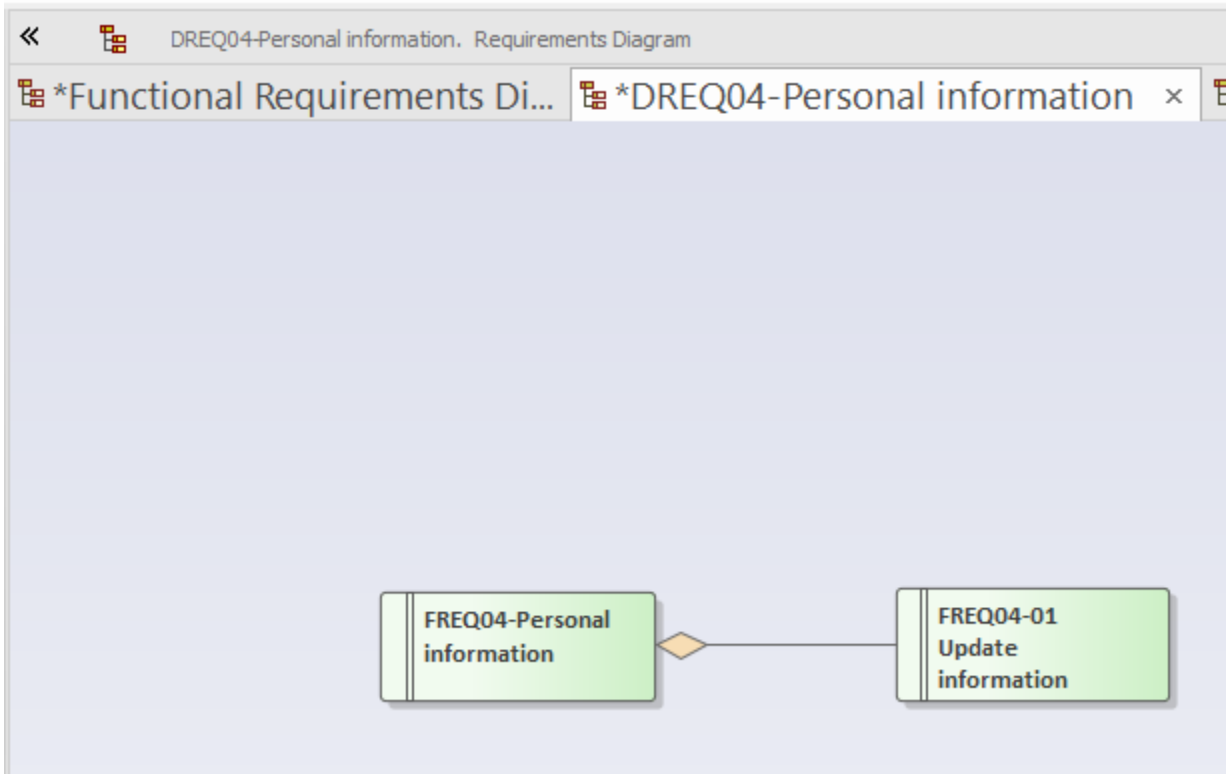
c) DREQ02-Media management diagram



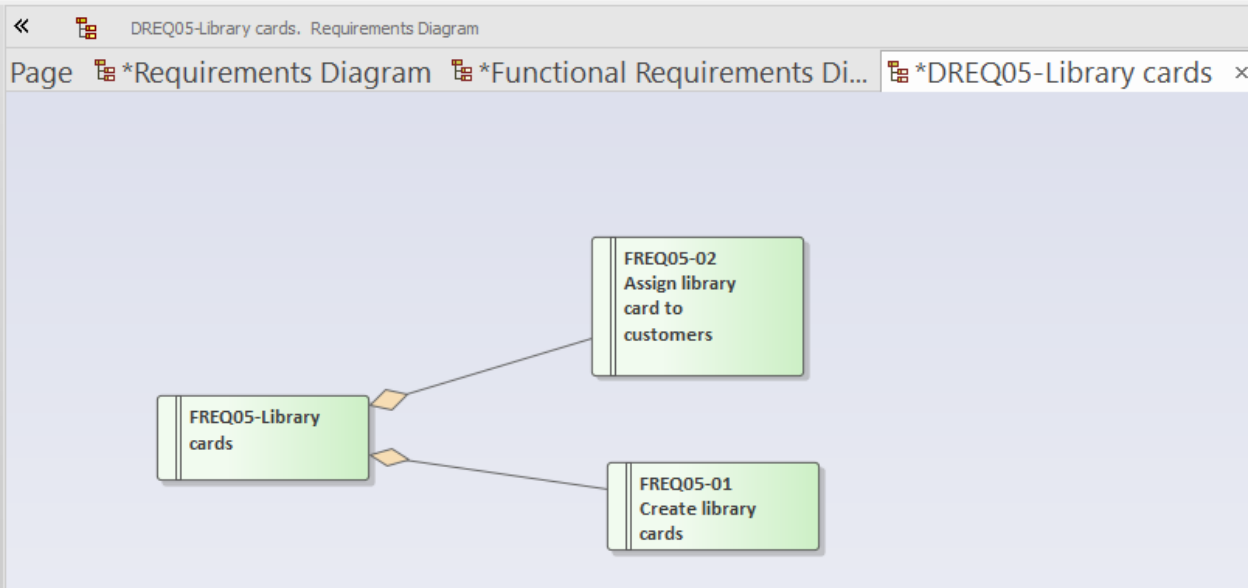
d) DREQ03-Account information diagram



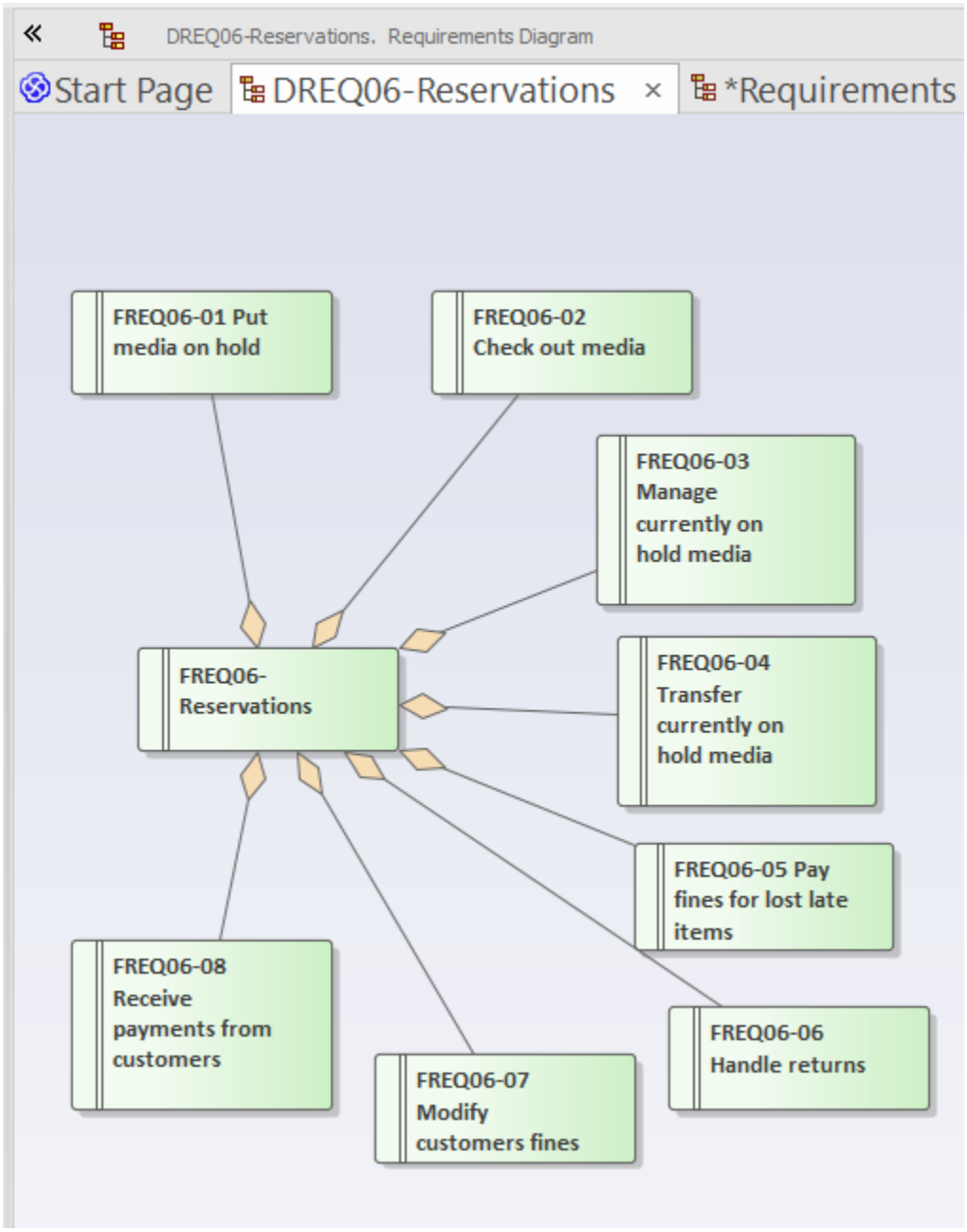
e) DREQ04-Personal information diagram



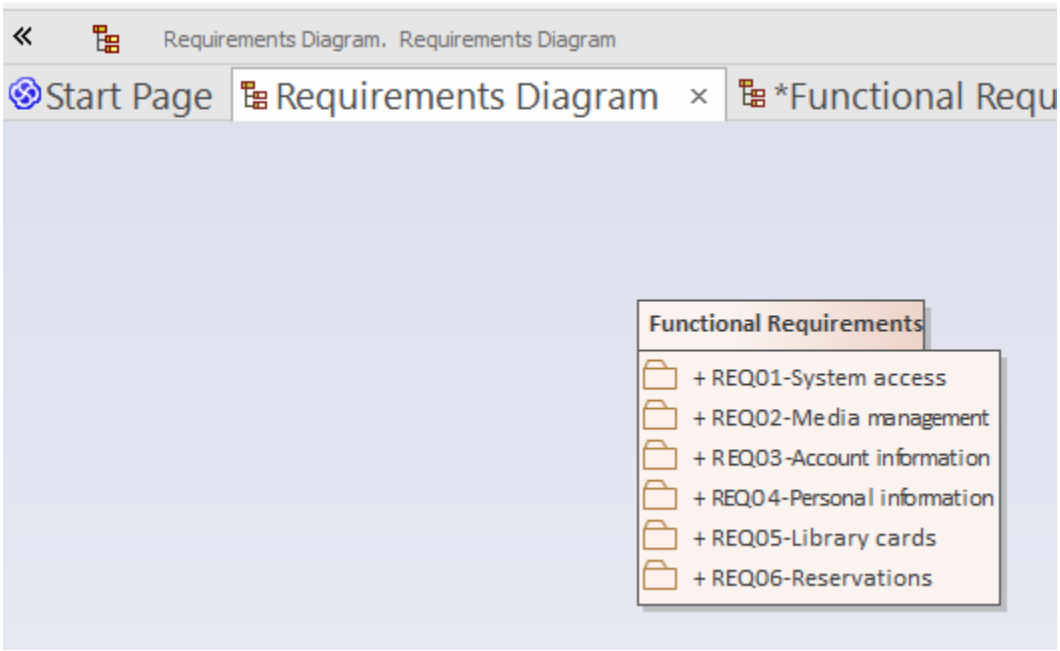
f) DREQ05-Library cards



g) DREQ06-Reservations

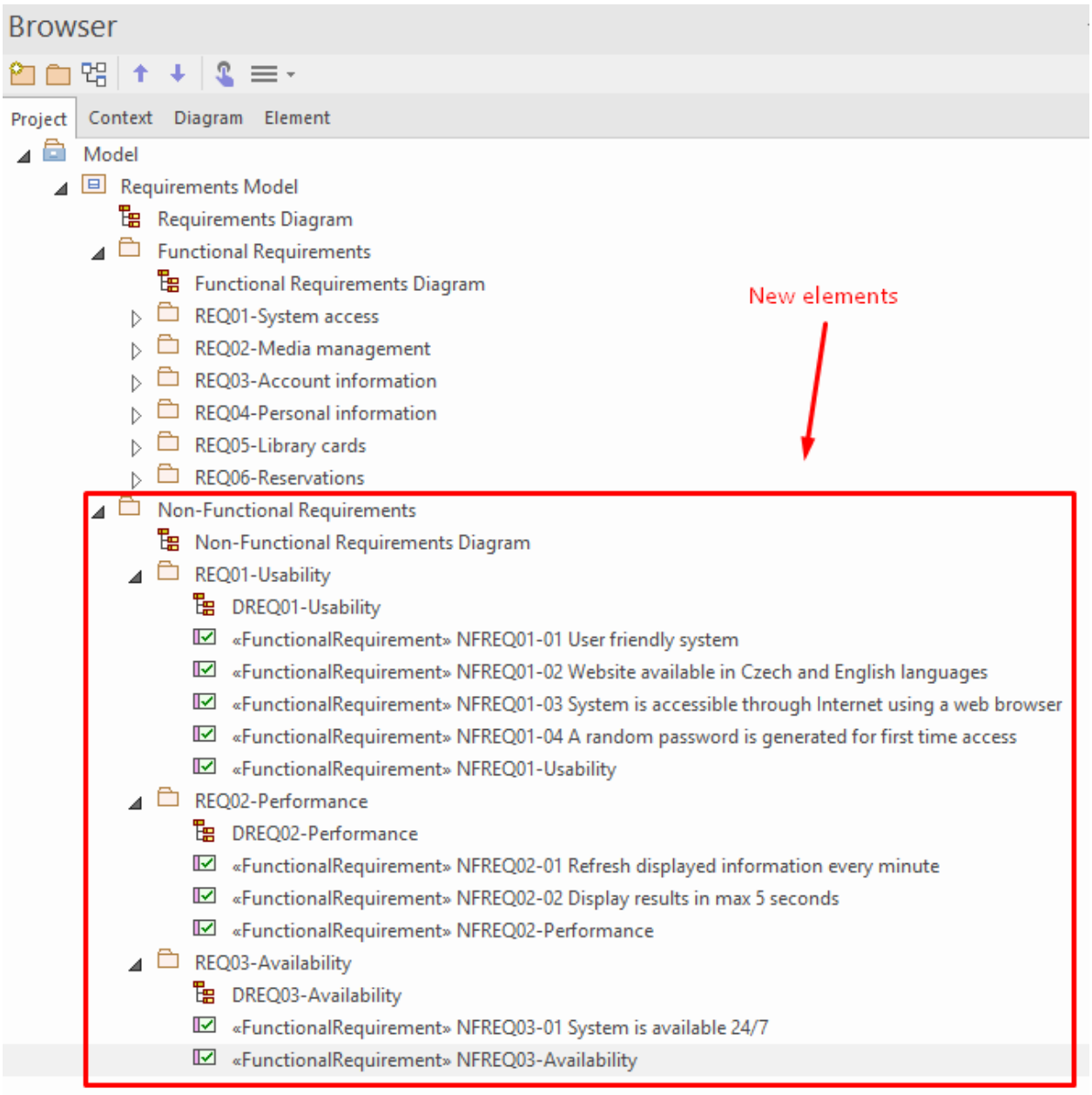


Step 9. If you check the main **Requirements Diagram** (under Requirements Model on the left), you'll see that it has been updated with all the elements that have been created:

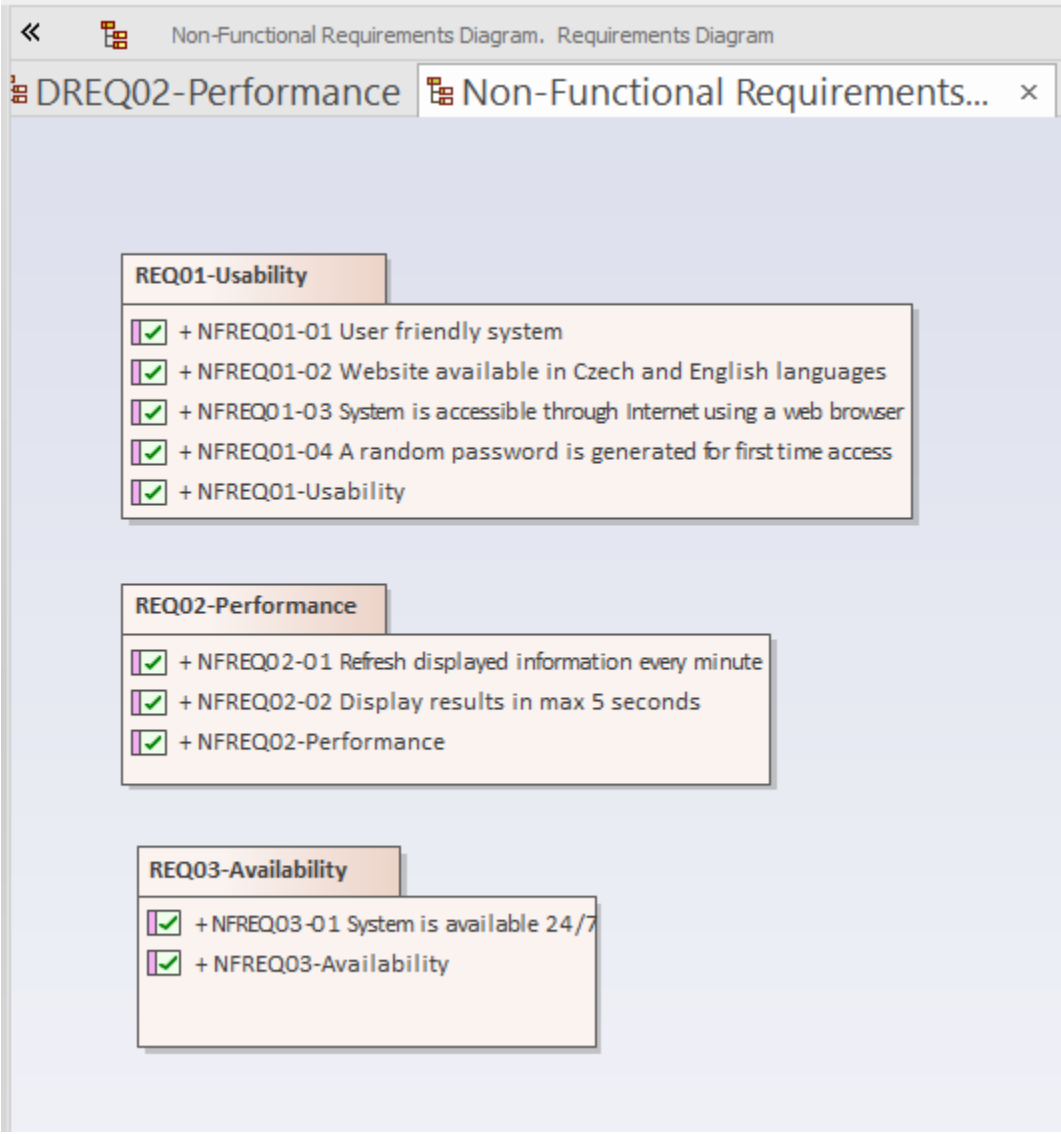


Step 10. Repeat **steps 3-9** to include the Non-Functional Requirements into the project. The following images show the expected output.

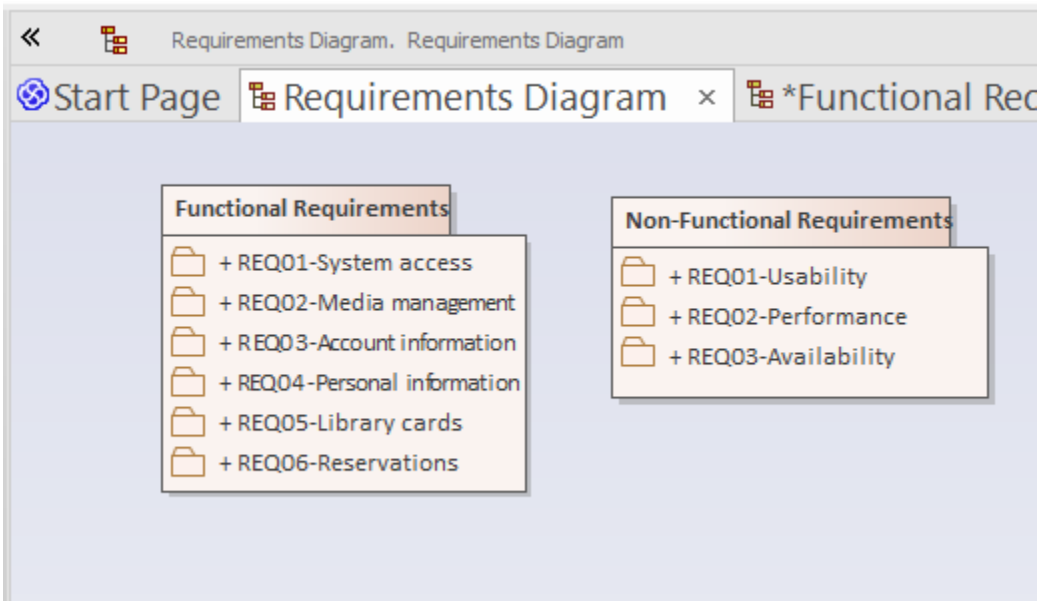
a) Requirements Model



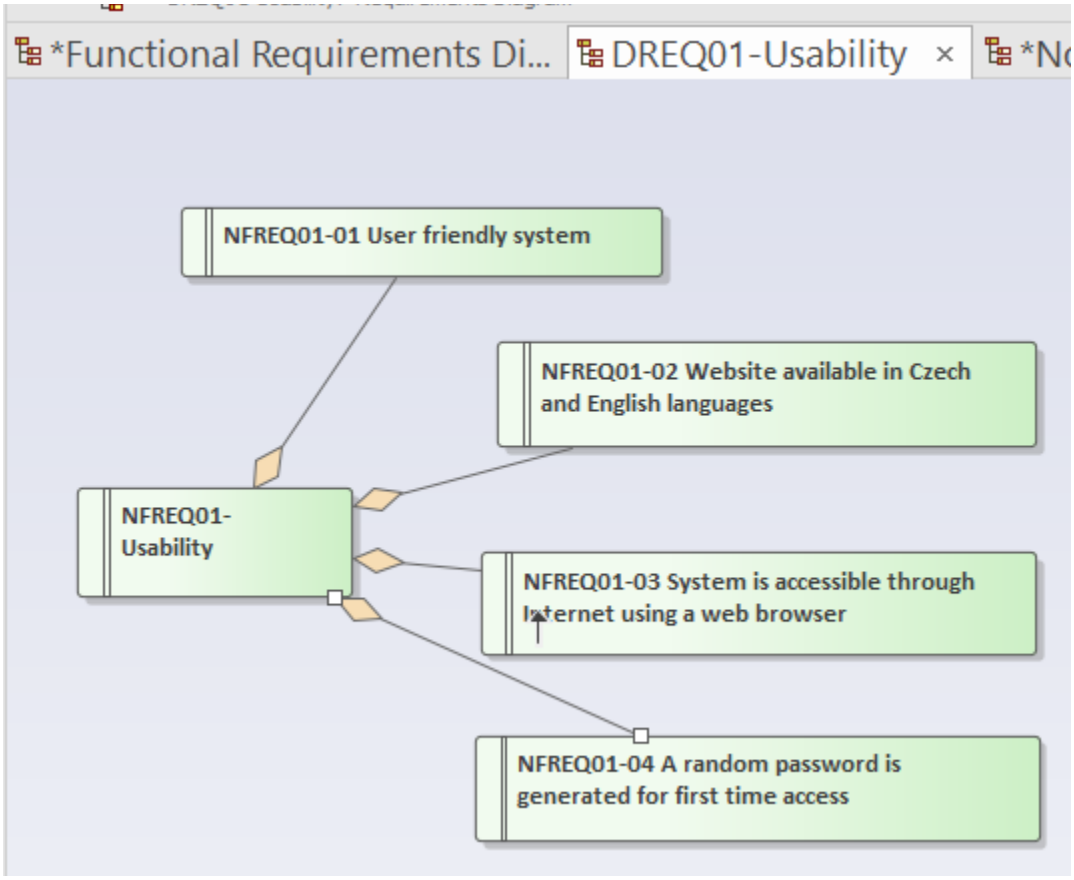
b) Non-Functional Requirements Diagram



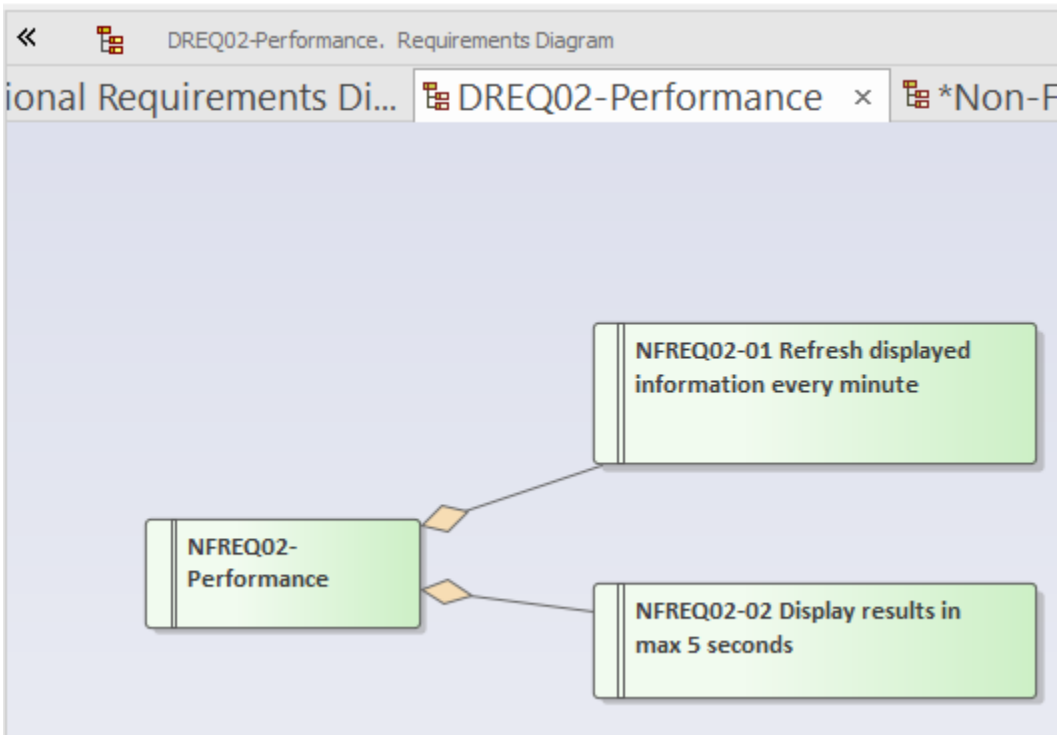
c) Requirements Diagram:



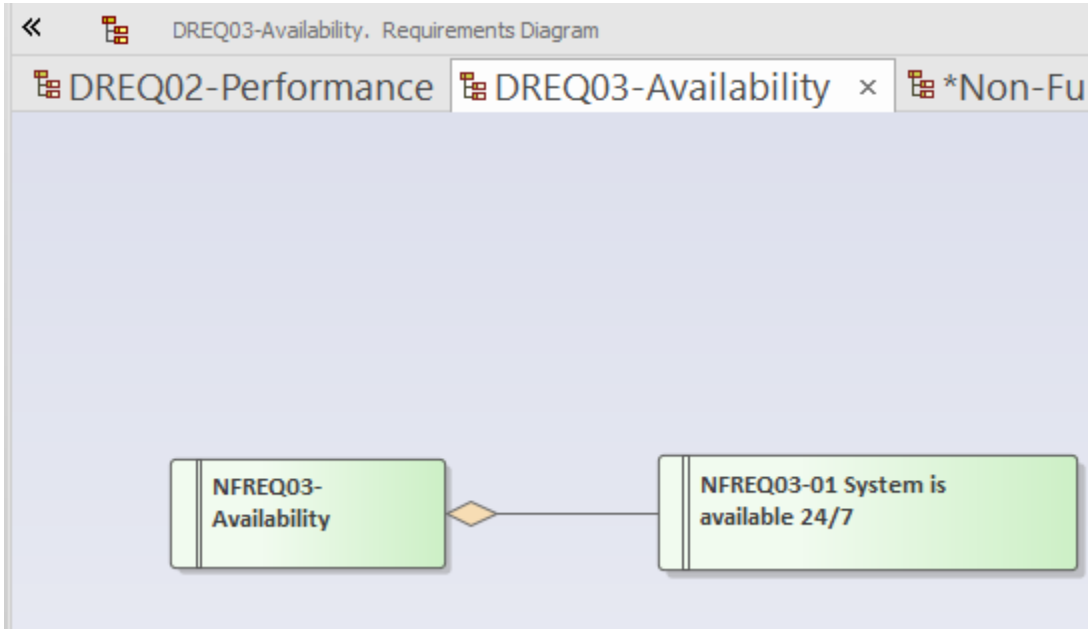
d) DREQ01-Usability:



e) DREQ02-Performance:



f) DREQ03-Availability:



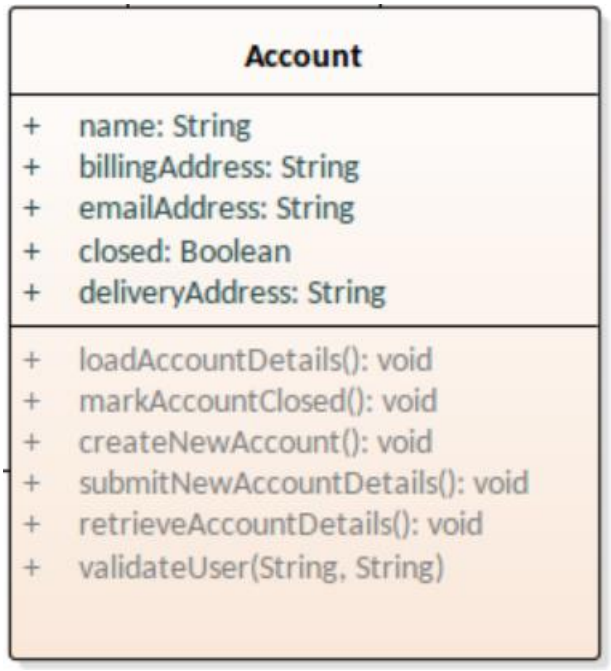
Learn more:

- What are Requirements?
- Requirements Diagram in Enterprise Architect
- Requirements Management with Enterprise Architect
- Requirements Diagram Examples
- Functional and Non-Functional Requirements
- Model Requirements in EA
- Requirement Models in EA

Seminar Session 02. Class Diagram (using Enterprise Architect)

Introduction

A **class** is an element that defines the *attributes and behaviors* that an object is able to generate. The **behavior** is described by the possible messages the class is able to understand, along with *operations* that are appropriate for each message. Classes may also have definitions of constraints, tagged values and stereotypes.



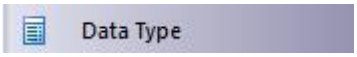
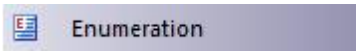
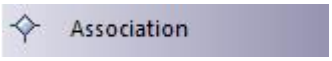
The **Class diagram** captures the logical structure of the system - the Classes - and things that make up the model. It is a static model, describing what exists and what attributes and behavior it has, rather than how something is done. On a Class diagram you can illustrate relationships between Classes and Interfaces using **Generalizations**, **Aggregations** and **Associations**, which are valuable in reflecting inheritance, composition or usage, and connections respectively.

Objective

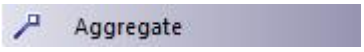
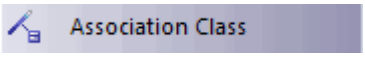
In this session, you will learn how to design a **Class diagram** using **Enterprise Architect**. You'll learn how to add classes to a diagram and add relationships between them depending on problem description analysis.

Class Diagrams in Enterprise Architect

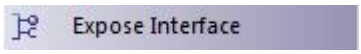
You can include the following elements in this type of diagrams:

Element	Description
	A **Class** is a representation of a type of object that reflects the structure and behavior of such objects within the system.
	An **Interface** is a specification of behavior (or contract) that implementers agree to meet.
	A **Data Type** is a specific kind of classifier, similar to a Class except that a Data Type cannot own sub Data Types, and instances of a Data Type are identified only by their value.
	An **Enumeration** is a data type, whose instances can be any of a number of user-defined enumeration literals.
	A **Primitive** element identifies a predefined data type, without any relevant substructure (that is, it has no parts in the context of UML).
	A **Signal** is a specification of Send request instances communicated between objects, typically in a Class or Package diagram.
	An n-Ary **Association** element is used to model complex relationships between three or more elements, typically in a Class diagram.

Moreover, you can depict relationships between requirements and other elements by using connectors such as the following:

Connector	Description
	An **Association** implies that two model elements have a relationship, usually implemented as an instance variable in one or both Classes.
	A **Generalization** is used to indicate inheritance.
	A **Composition** is used to depict an element that is made up of smaller components, typically in a Class or Package diagram.
	An **Aggregation** connector is a type of association that shows that an element contains or is composed of other elements.
	An **Association Class** is a UML construct that enables an Association to have attributes and operations (features).
	A source object implements or **Realizes** its destination object.
	You create a **Template Binding** connector between a binding Class and a parameterized Class.

Finally, it is important to mention that there are also Composite elements in this diagram:

Connector	Description
	**Parts** are run-time instances of Classes or Interfaces.
	**Ports** define the interaction between a classifier and its environment.
	The **Expose Interface** element is a graphical method of depicting the required or supplied interfaces of a Component, Class or Part, in a Class, Component or Composite Structure diagram.
	An **Assembly connector** bridges a component's required interface (Component1) with the provided interface of another component (Component2), typically in a Component diagram.
	**Connectors** illustrate communication links between Parts to fulfill the structure's purpose, typically in a Class or Composite Structure diagram.
	A **Delegate** connector defines the internal assembly of a component's external Ports and Interfaces, on a Class diagram or Component diagram.

Problem Description

A library database needs to store information pertaining to:

- Its customers
- Its workers
- The physical locations of its branches,
- And the media stored in those locations (two media types are considered: books and videos).

The library must keep track of the status of each media item: its location, status, descriptive attributes, and cost for losses and late returns. Books will be identified by their ISBN, while movies by their title and year. In order to allow multiple copies of the same book or video, each media item will have a unique ID number.

Customers will provide their name, address, phone number, and date of birth when signing up for a library card. They will then be assigned a unique user name and ID number, plus a temporary password that will have to be changed.

Checkout operations will require a library card, as will requests to put media on hold. Each library card will have its own fines, but active fines on any of a customer's cards will prevent the customer from using the library's services.

The library will have branches in various physical locations. Branches will be identified by name, and each branch will have an address and a phone number associated with it. Additionally, a library branch will store media and have employees.

Employees will work at a specific branch of the library. They receive a paycheck, but they can also have library cards; therefore, the same information that is collected about customers should be collected about employees.

Functions for customers (users):

- Log in
- Search for media based on one or more of the following criteria:
    - type (book, video, or both)
    - title
    - author or director
    - year
- Access their own account information:

    - Card number(s)
    - Fines
    - Media currently checked out
    - Media on hold
- Put media on hold
- Pay fines for lost or late items
- Update personal information:
    - Phone numbers
    - Addresses
    - Passwords

Functions for librarians (employees) are the same as the functions for customers plus the following:

- Add customers
- Add library cards and assign them to customers
- Check out media
- Manage and transfer media that is currently on hold
- Handle returns
- Modify customers' fines
- Add media to the database
- Remove media from the database
- Receive payments from customers and update the customers' fines
- View all customer information except passwords

Classes Analysis

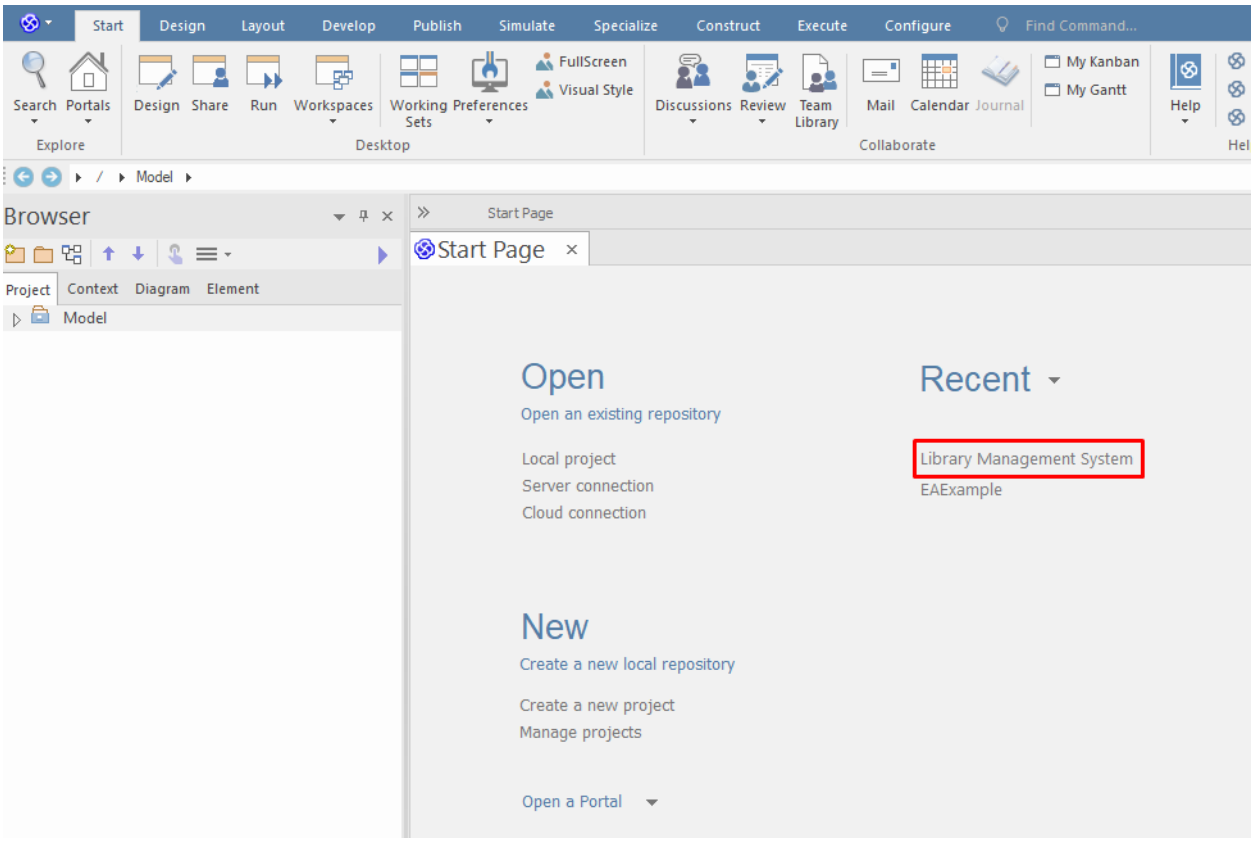
Classes can be identified as elements that can “do something” or can “be used” in the system. From the problem description, you can consider the following elements:

- System Users, which can be either Customers or Employees
- Branches (Locations)
- Media, there are two types: Books and Videos
- Library cards
- Checkout operation

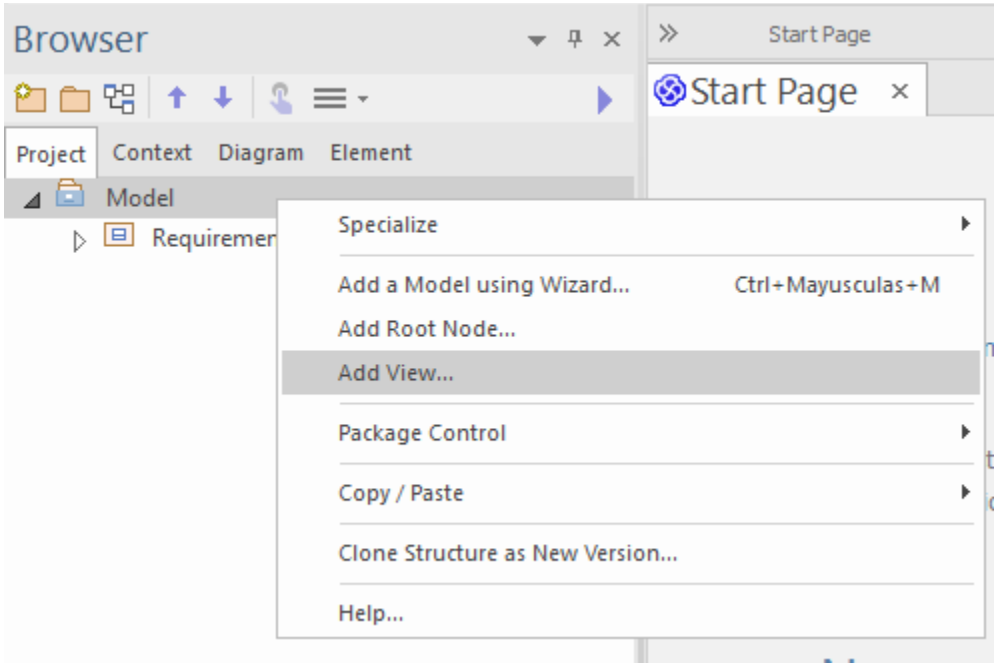
Each element is relevant since they contain attributes, for instance a customer has name, address, phone number, date of birth, user name, ID number, password. Moreover, some actions can be done with a customer object: a new user can be added into the system, a user can sign into the system, or update their personal information, to name a few actions.

Modeling classes in Enterprise Architect

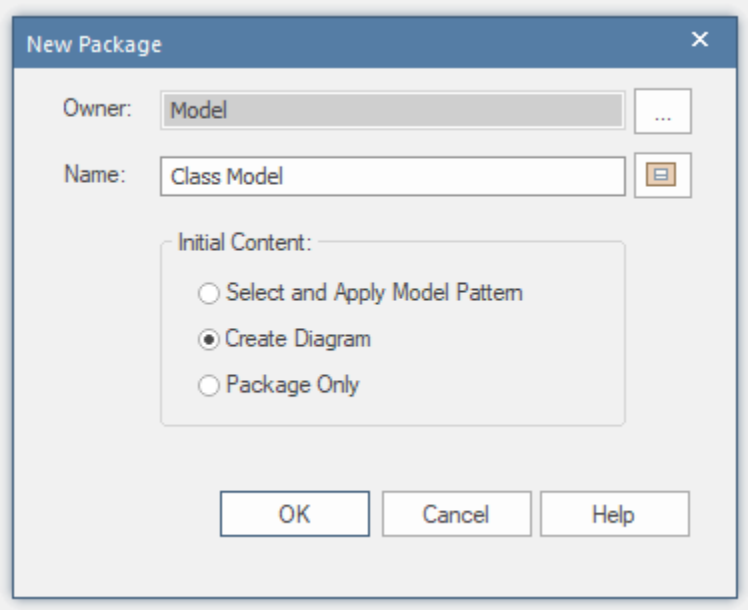
Step 1. Open the **Library Management System** project that was created in the previous session:



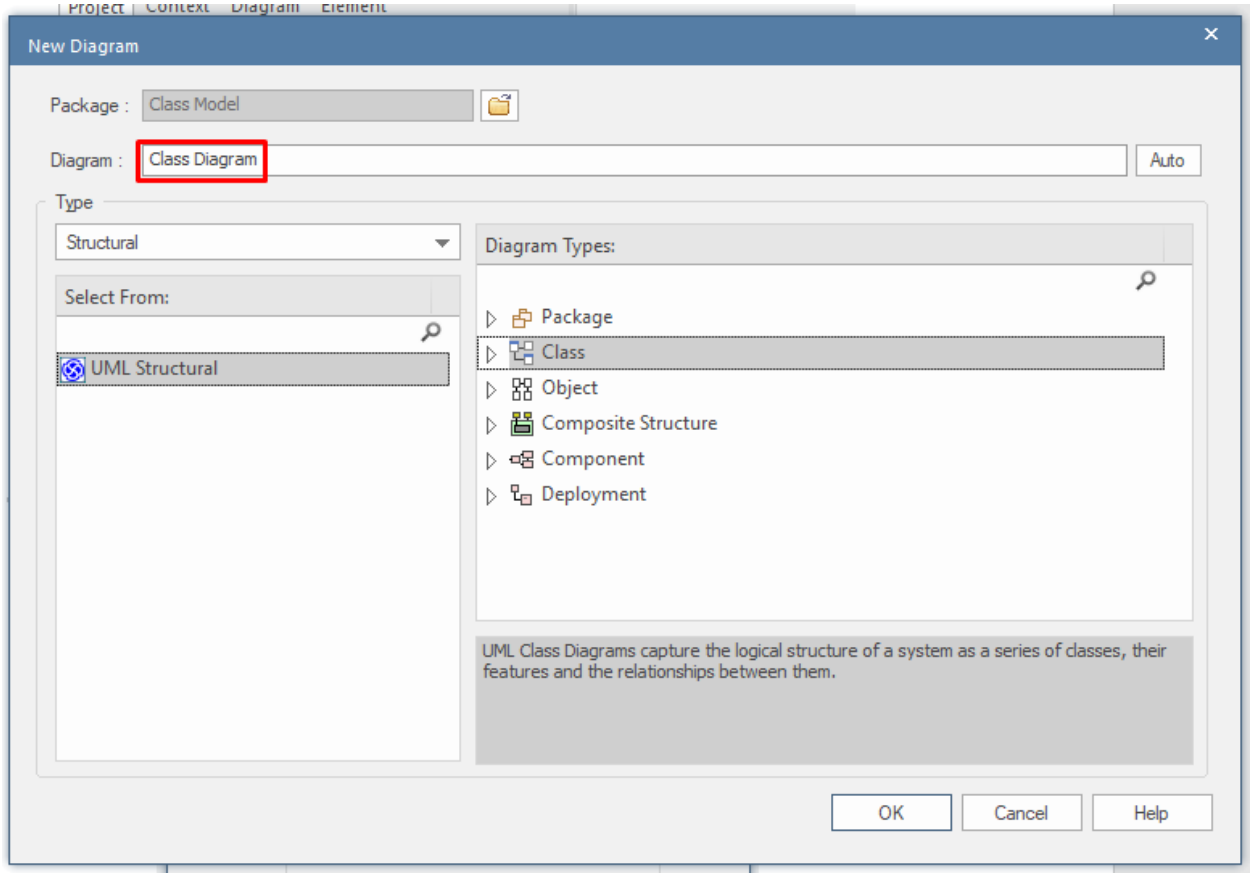
Step 2. Right click on Model and select Add View…



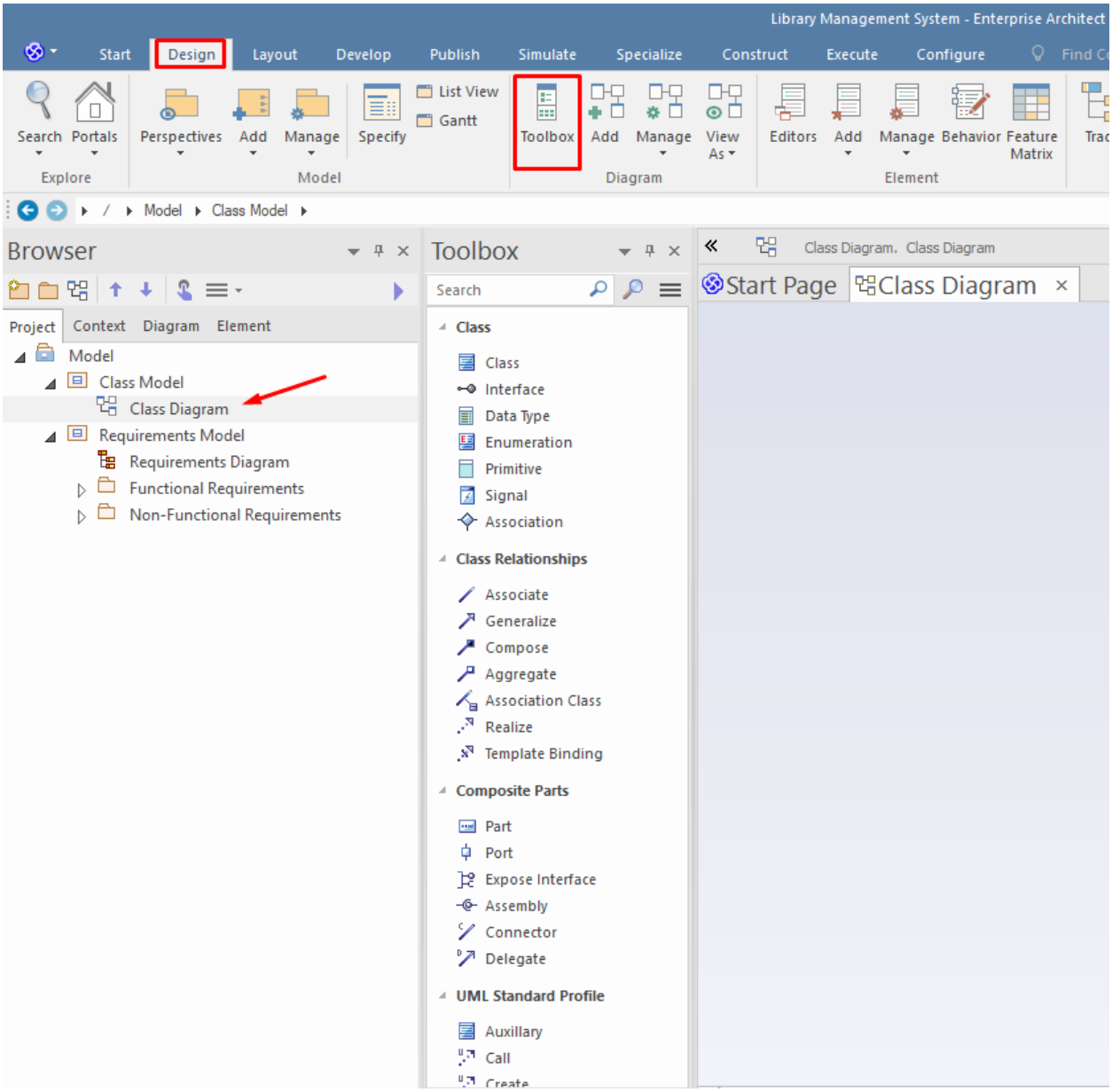
Type **Class Model** and select **Create Diagram:**



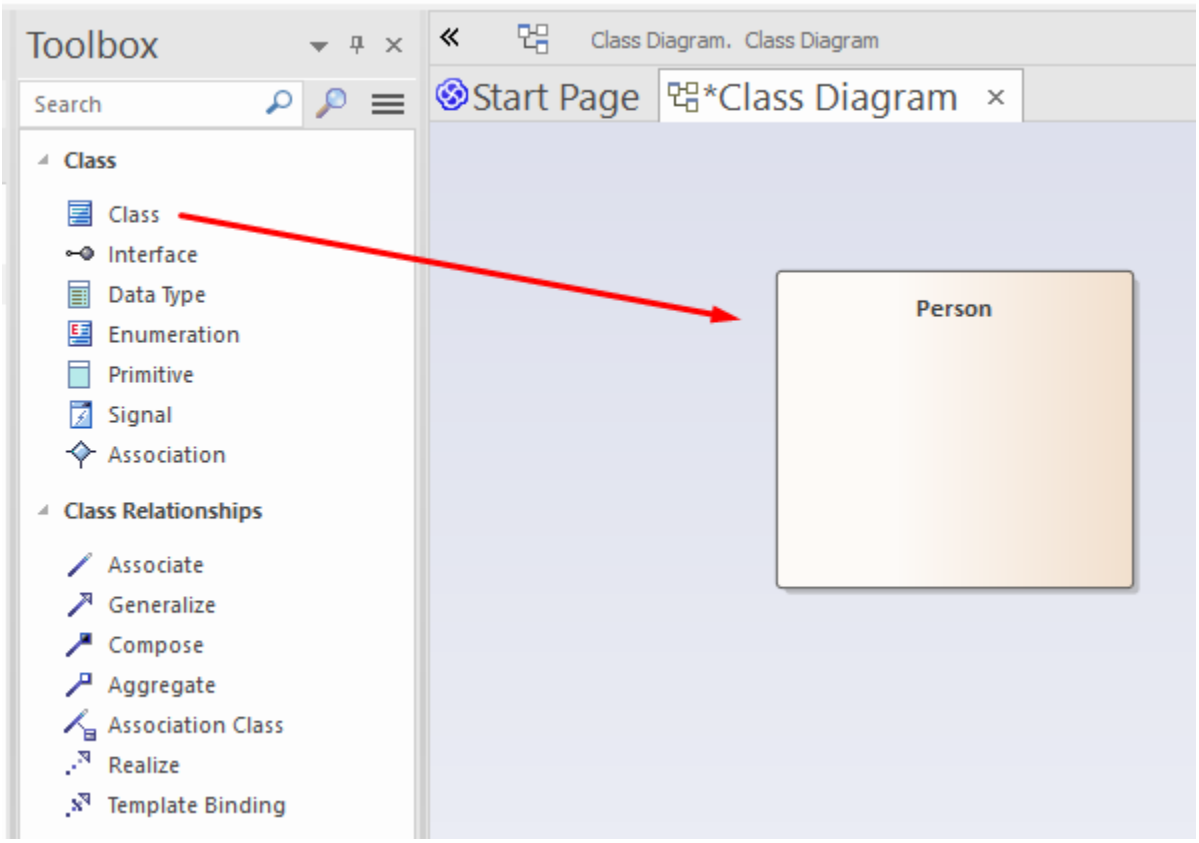
The name of the new element is **Class Diagram**. Select **Class** Diagram Type under UML Structural:



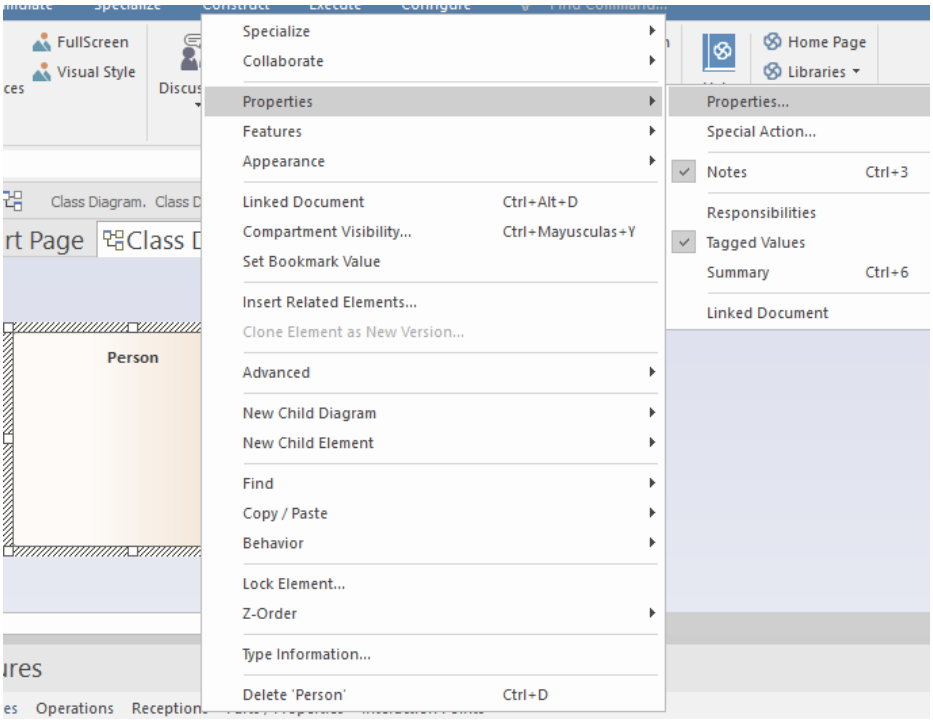
Step 3. Double click the **Class Diagram** from the left panel. Under **Design**, click on **Toolbox:**



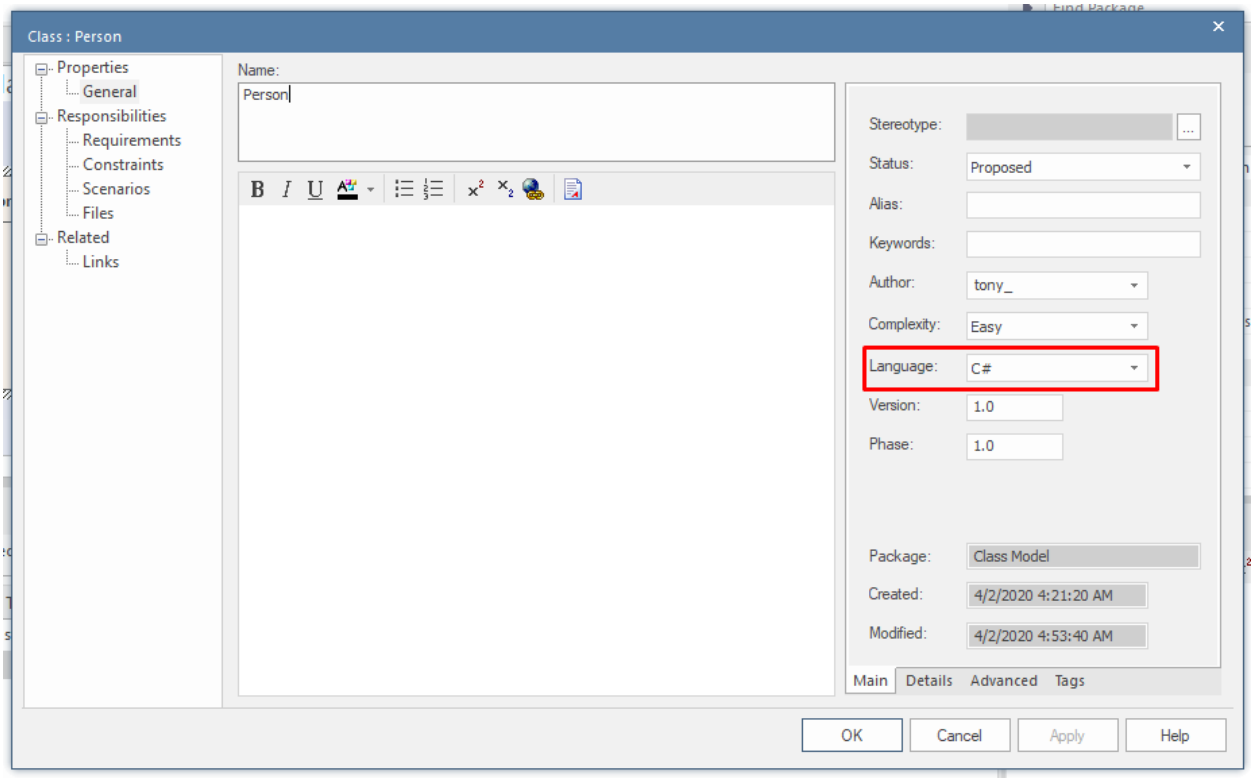
Drag and drop the **Class** element from the toolbox into the diagram. Rename it **Person**.



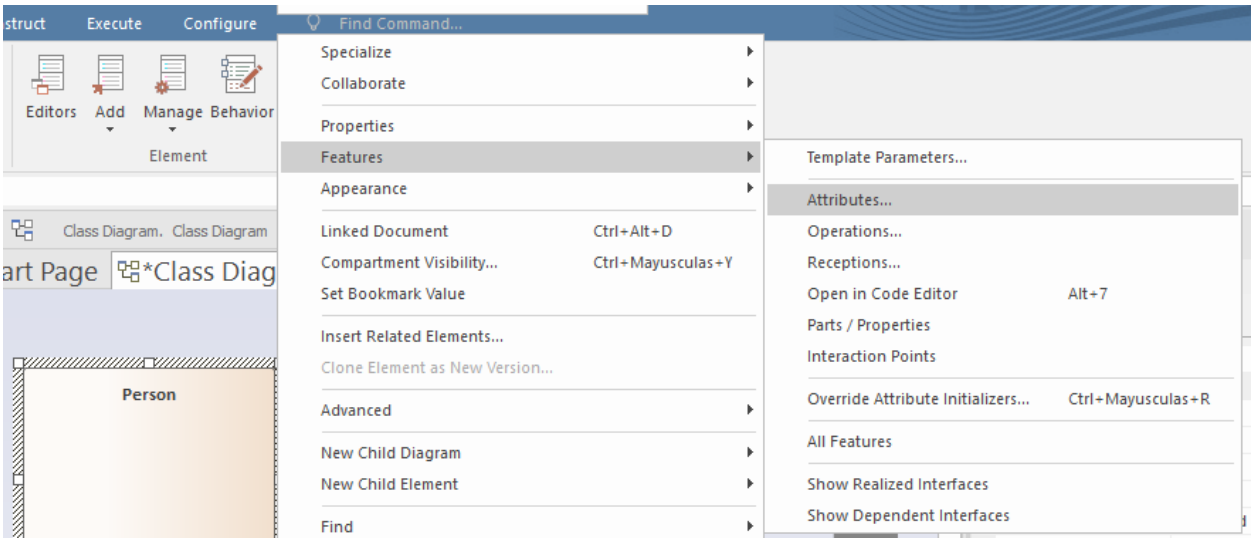
Right-click on this new element and select Properties → Properties:



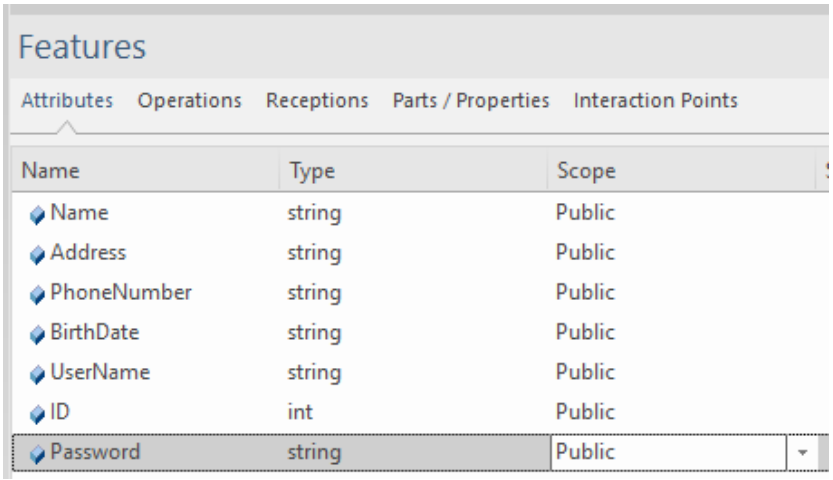
Change the language to C#.



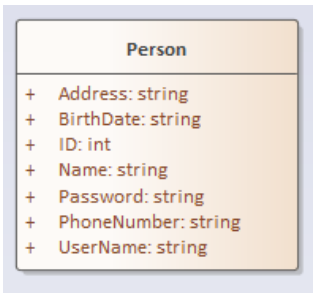
Right-click on it again and this time select **Features**, then **Attributes**:



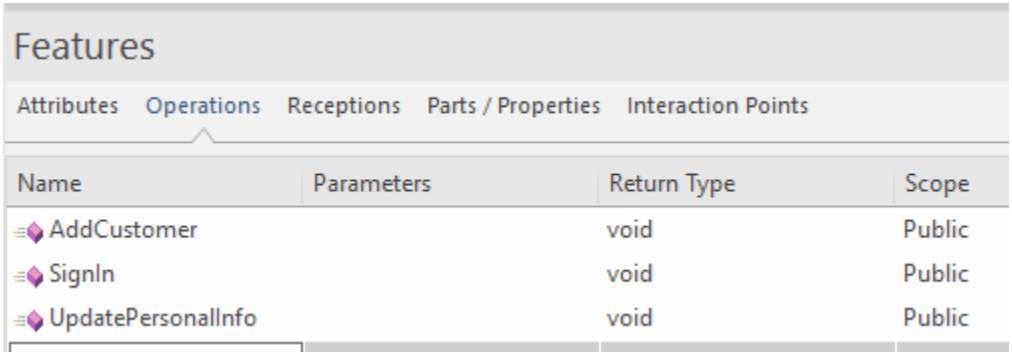
Add the following information:



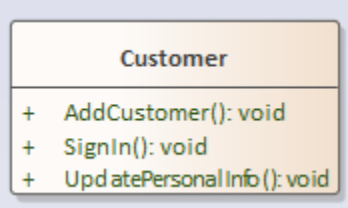
The result is:



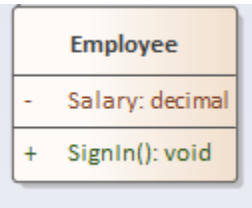
Step 4. Add another class: **Customer**, then right-click on it and select Features → Operations:



The result is:

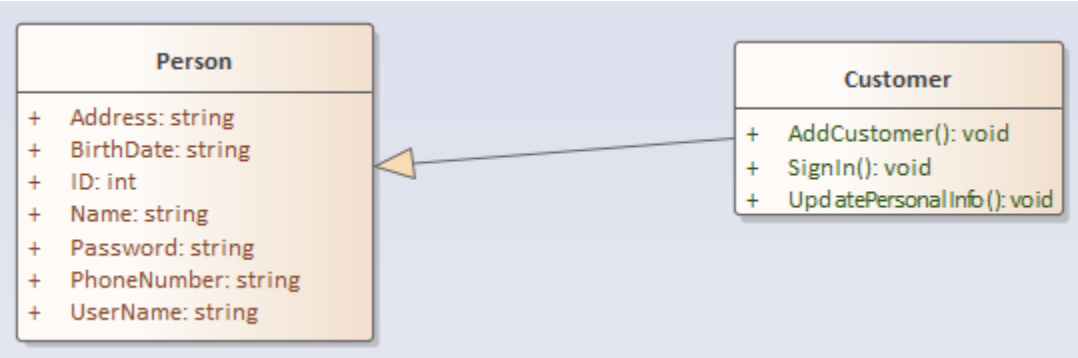


Add another class: **Employee**. This one has both Attributes and Operations:

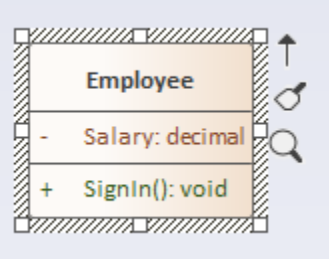


Step 5. Now, from the toolbox, click on the **Class Relationship** type **Generalize**.

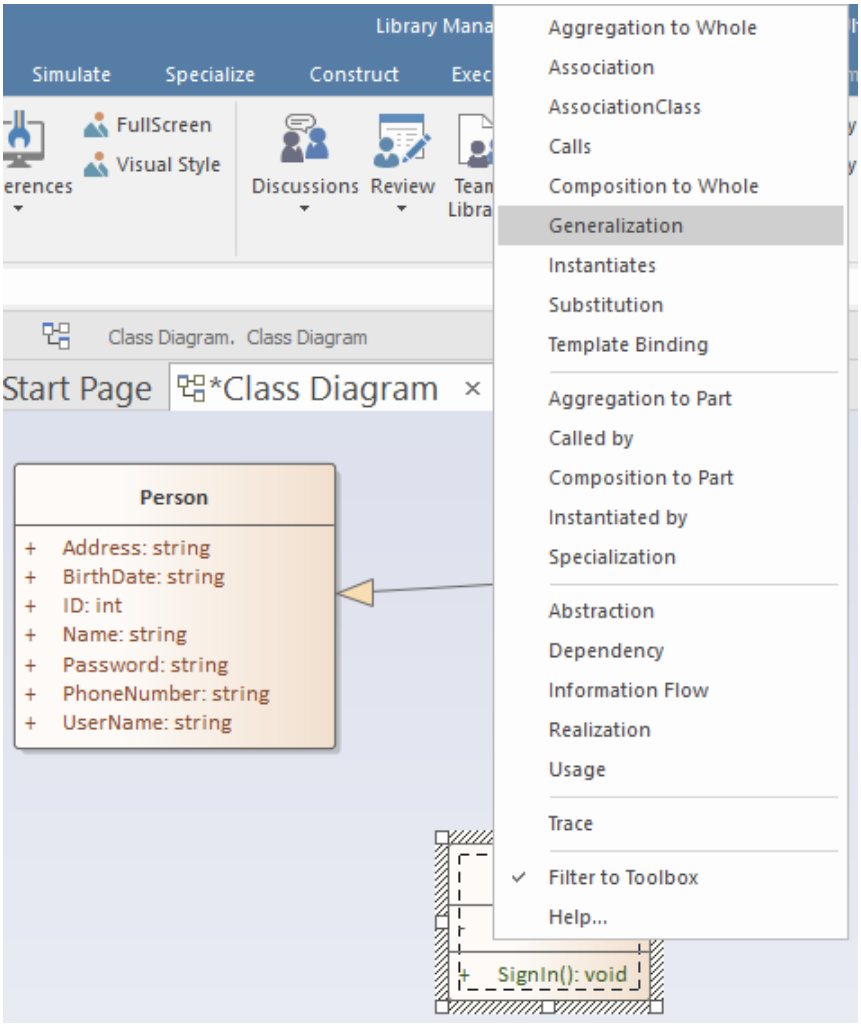
Then click on Customer class and pull the line to Person class:



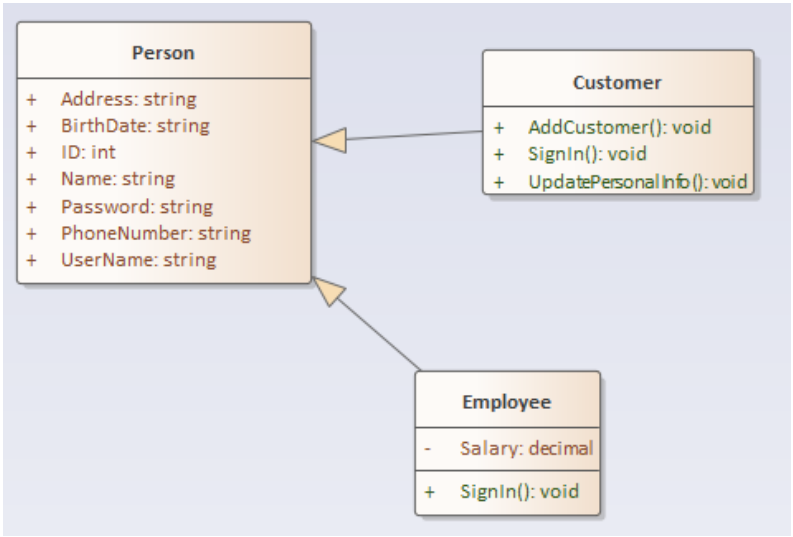
You can achieve the same functionality firstly by clicking on the Employee class, then on the top arrow:



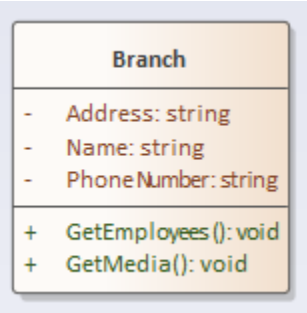
Now pull it to Person class and select Generalization:


Library Mana
Simulate
Specialize
Construct
Exec
FullScreen
Visual Style
erences
Discussions
Review
Team
Libra
Class Diagram. Class Diagram
Start Page
*Class Diagram
Person
+ Address: string
+ BirthDate: string
+ ID: int
+ Name: string
+ Password: string
+ PhoneNumber: string
+ UserName: string
+ SignIn(): void
Aggregation to Whole
Association
AssociationClass
Calls
Composition to Whole
Generalization
Instantiates
Substitution
Template Binding
Aggregation to Part
Called by
Composition to Part
Instantiated by
Specialization
Abstraction
Dependency
Information Flow
Realization
Usage
Trace
Filter to Toolbox
Help...

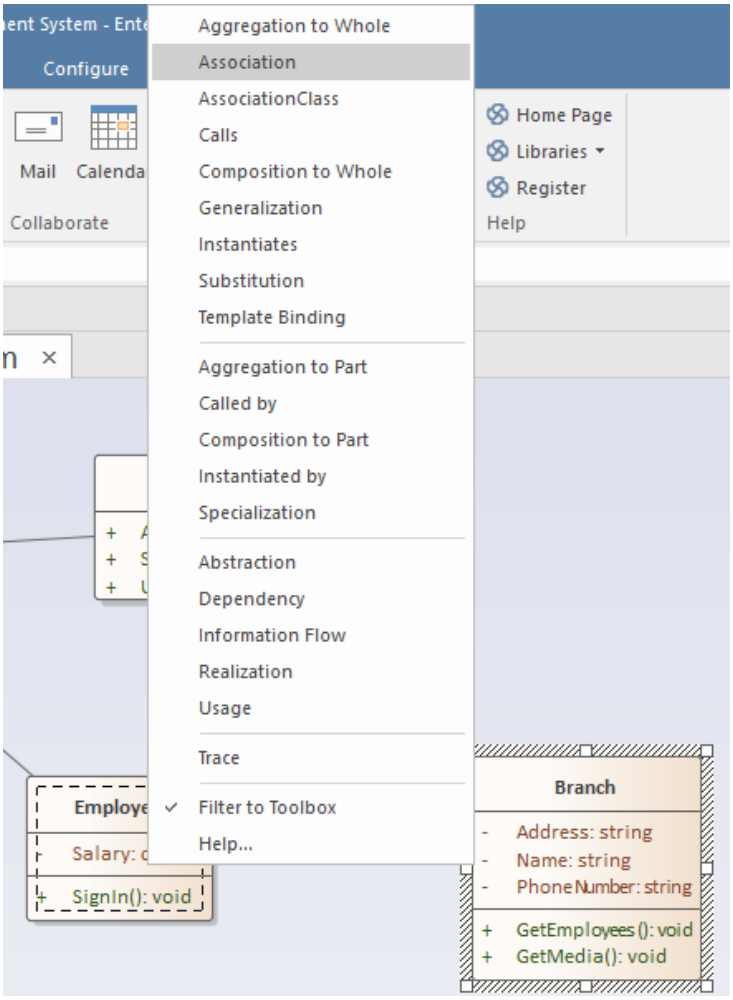
Result:


Person
+ Address: string
+ BirthDate: string
+ ID: int
+ Name: string
+ Password: string
+ PhoneNumber: string
+ UserName: string
Customer
+ AddCustomer(): void
+ SignIn(): void
+ UpdatePersonalInfo(): void
Employee
- Salary: decimal
+ SignIn(): void

Step 6. Add the **Branch** class:

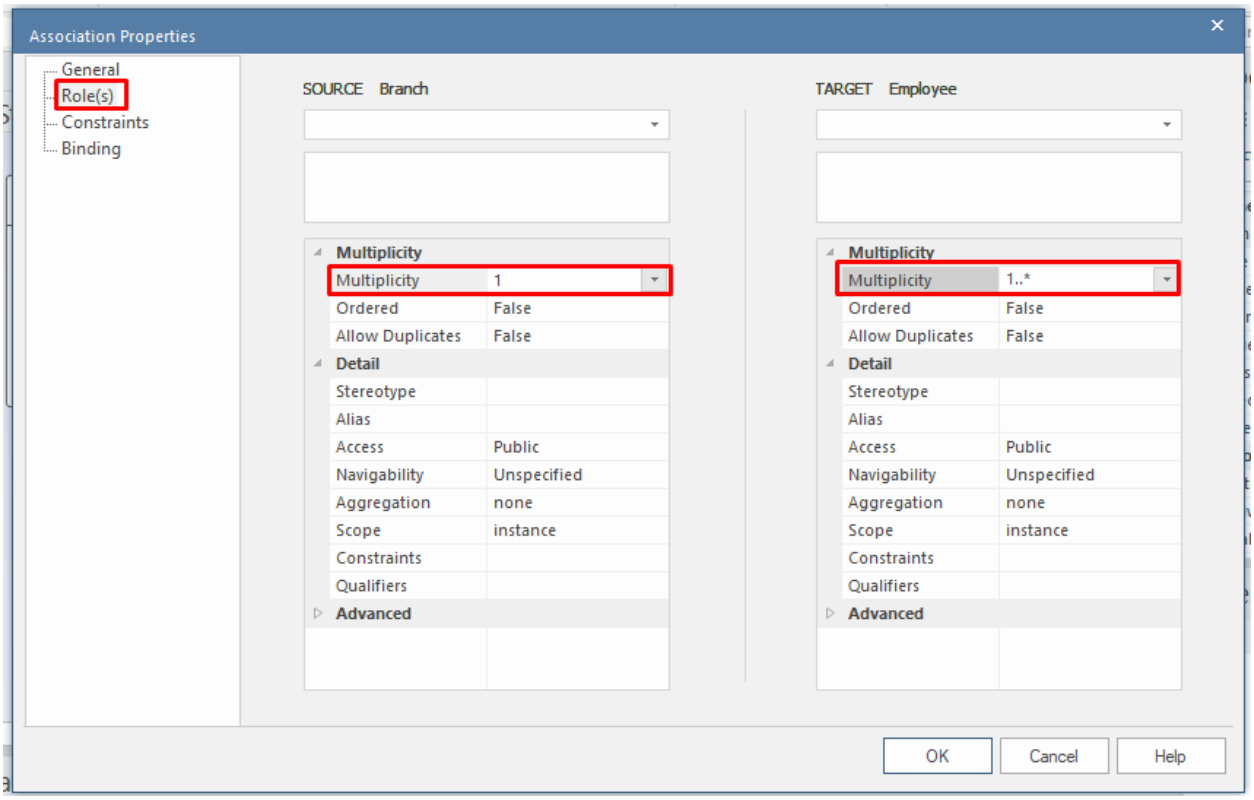


Connect it to the **Employees** class, this time use an **Association** relationship

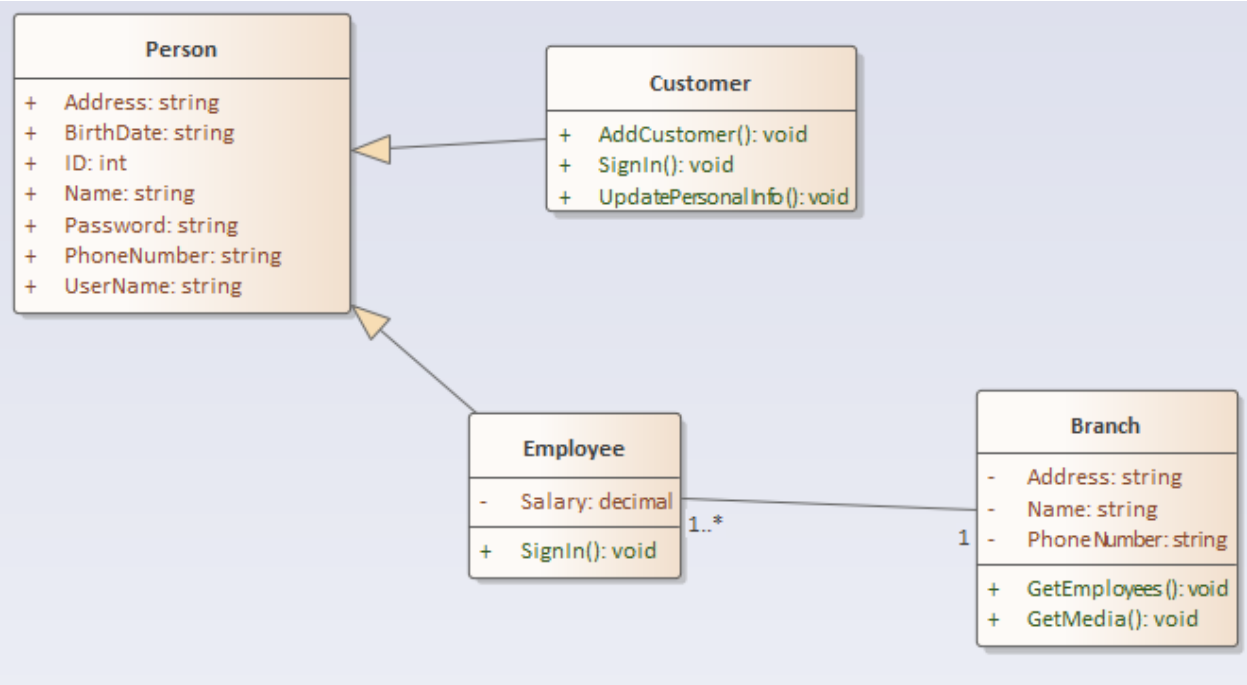


Result:

Double click the line connecting both classes (the relationship) and update



Result:



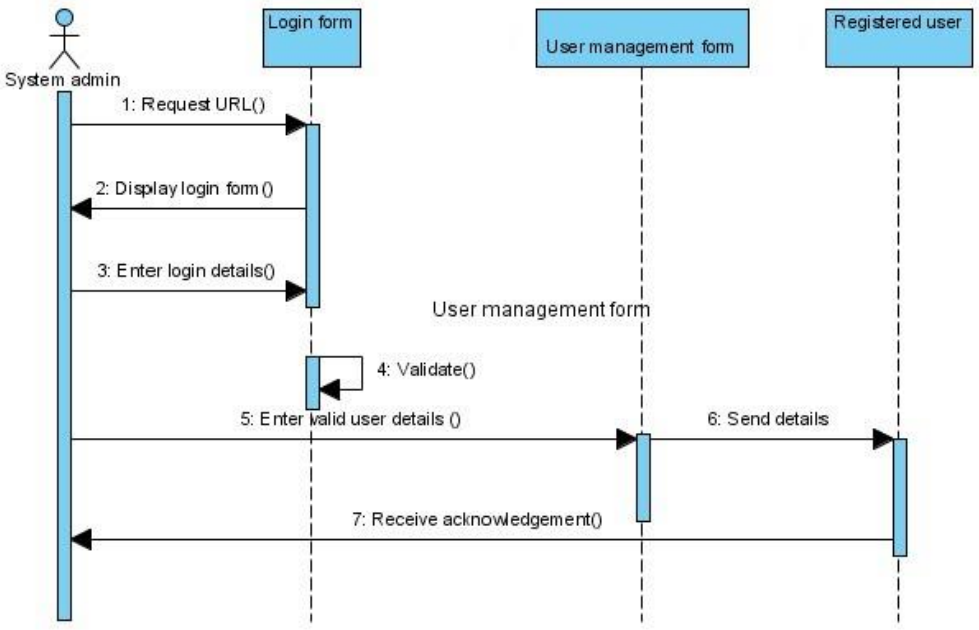
Which reads **“An Employee exists in 1 branch, while one branch can have 1 or more employees.”**

Seminar Session 04. Sequence Diagram (using Enterprise Architect)

Introduction

A **sequence diagram** is a visual interaction that details how operations are carried out. They capture the interaction between objects in the context of a collaboration. Sequence Diagrams are time-focused and show the order of the interaction visually by using the vertical axis of the diagram to represent time, what messages are sent and when. You can use it to:

- Depict workflow, Message passing and how elements in general cooperate over time to achieve a result.
- Capture the flow of information and responsibility throughout the system, early in analysis; Messages between elements eventually become method calls in the Class model.
- Make explanatory models for Use Case scenarios; by creating a Sequence diagram with an Actor and elements involved in the Use Case, you can model the sequence of steps the user and the system undertake to complete the required tasks.



Sequence elements are arranged in a **horizontal sequence**, with **Messages** passing back and forward between elements. **Messages on a Sequence diagram** can be of several types and can also be configured to reflect the operations and properties of the source and target elements.

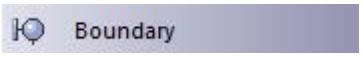
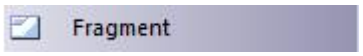
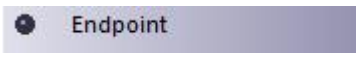
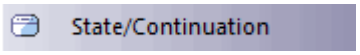
An **Actor** element can be used to represent the user initiating the flow of events. Stereotyped elements, such as **Boundary, Control** and **Entity**, can be used to illustrate **screens**, **controllers** and **database items**, respectively. Each element has a dashed stem called a **Lifeline**, where that element exists and potentially takes part in the interactions

Objective

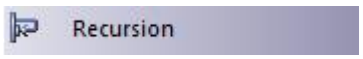
In this session, you will learn how to design a **Sequence diagram** using **Enterprise Architect**. You'll learn how to add interactions between actors, objects in use cases, depending on problem description analysis.

Sequence Diagrams in Enterprise Architect

You can include the following elements in this type of diagrams:

Element	Description
	An **Actor** is a user of the system: a human user, a machine, or even another system or subsystem in the model.
	A **Lifeline** represents a distinct connectable element and is an individual participant in an interaction.
	**Boundary** elements are used in analysis to capture user interactions, screen flows and element interactions.
	A **Control** organizes and schedules other activities and elements.
	An **Entity** is a stereotyped Object that models a store or persistence mechanism that captures the information or knowledge in a system.
	A Fragment element can represents iterations or alternative processes in a Sequence diagram.
	An Endpoint is used in Interaction diagrams to reflect a lost or found Message in sequence.
	A Diagram Gate is a simple graphical way to indicate the point at which messages can be transmitted into and out of interaction fragments.
	The State/Continuation element serves two different purposes for Sequence diagrams, as State Invariants and Continuations.
	You can use an Interaction element to insert an Interaction diagram as a child of a Class element.

Moreover, you can use connectors such as the following:

Connector	Description
	A Message indicates a flow of information or transition of control between elements.
	A Self-Message reflects a new process or method invoked within the calling lifeline's operation.
	A Recursion is a type of Message used in Sequence diagrams to indicate a recursive function.
	A Call is a type of Message connector that extends the level of activation from the previous Message.

Problem Description

A library database needs to store information pertaining to:

- Its customers
- Its workers
- The physical locations of its branches,
- And the media stored in those locations (two media types are considered: books and videos).

The library must keep track of the status of each media item: its location, status, descriptive attributes, and cost for losses and late returns. Books will be identified by their ISBN, while movies by their title and year. In order to allow multiple copies of the same book or video, each media item will have a unique ID number.

Customers will provide their name, address, phone number, and date of birth when signing up for a library card. They will then be assigned a unique user name and ID number, plus a temporary password that will have to be changed.

Checkout operations will require a library card, as will requests to put media on hold. Each library card will have its own fines, but active fines on any of a customer's cards will prevent the customer from using the library's services.

The library will have branches in various physical locations. Branches will be identified by name, and each branch will have an address and a phone number associated with it. Additionally, a library branch will store media and have employees.

Employees will work at a specific branch of the library. They receive a paycheck, but they can also have library cards; therefore, the same information that is collected about customers should be collected about employees.

Functions for customers (users):

- Log in
- Search for media based on one or more of the following criteria:
    - type (book, video, or both)
    - title
    - author or director
    - year
- Access their own account information:

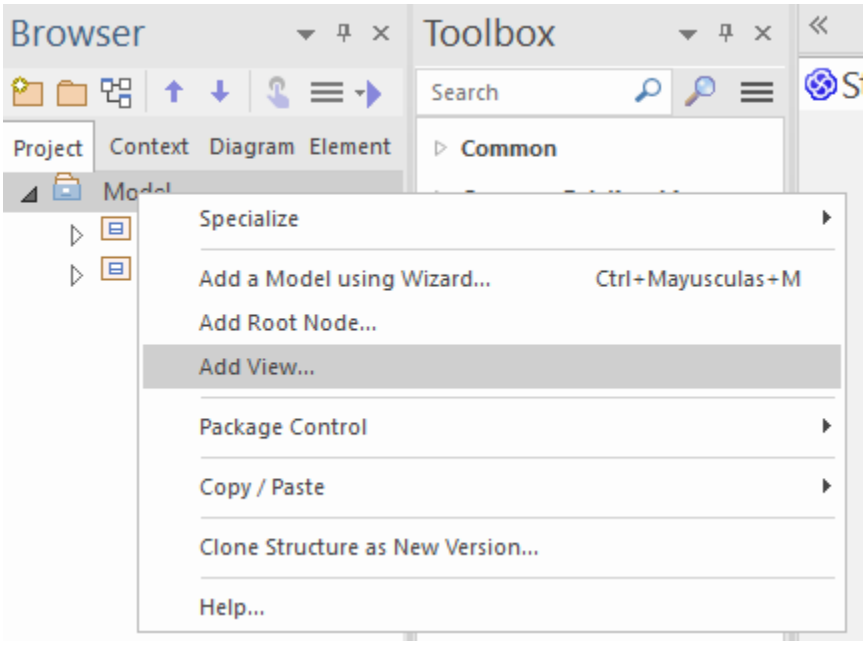
    - Card number(s)
    - Fines
    - Media currently checked out
    - Media on hold
- Put media on hold
- Pay fines for lost or late items
- Update personal information:
    - Phone numbers
    - Addresses
    - Passwords

Functions for librarians (employees) are the same as the functions for customers plus the following:

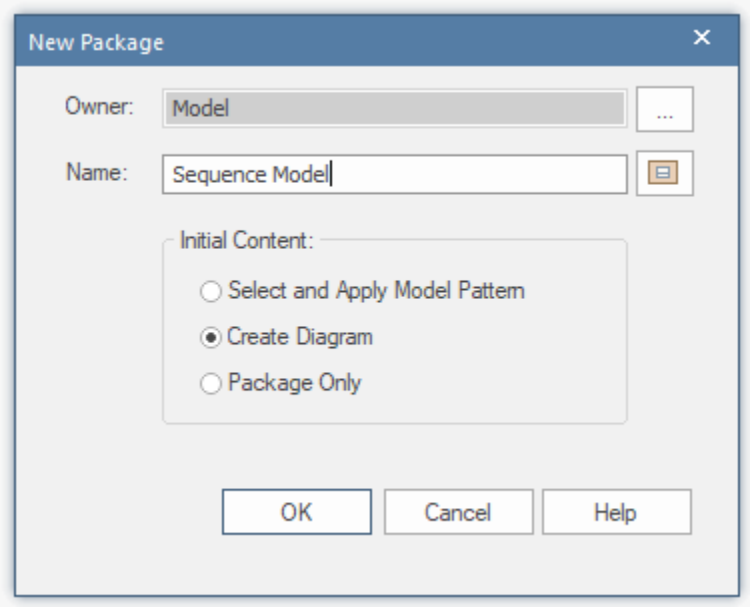
- Add customers
- Add library cards and assign them to customers
- Check out media
- Manage and transfer media that is currently on hold
- Handle returns
- Modify customers' fines
- Add media to the database
- Remove media from the database
- Receive payments from customers and update the customers' fines
- View all customer information except passwords

Analysis

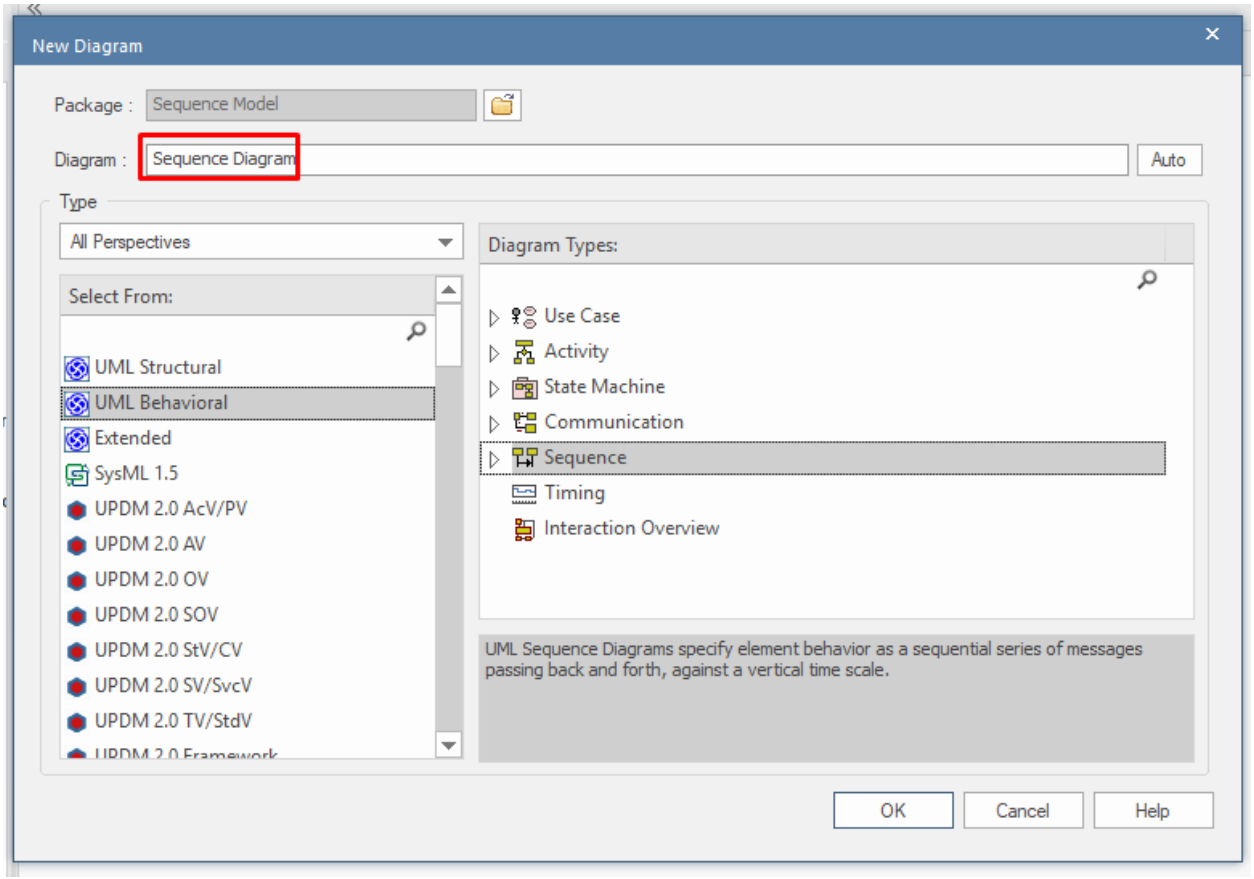
Step 1. Add a **View** in your **Model**



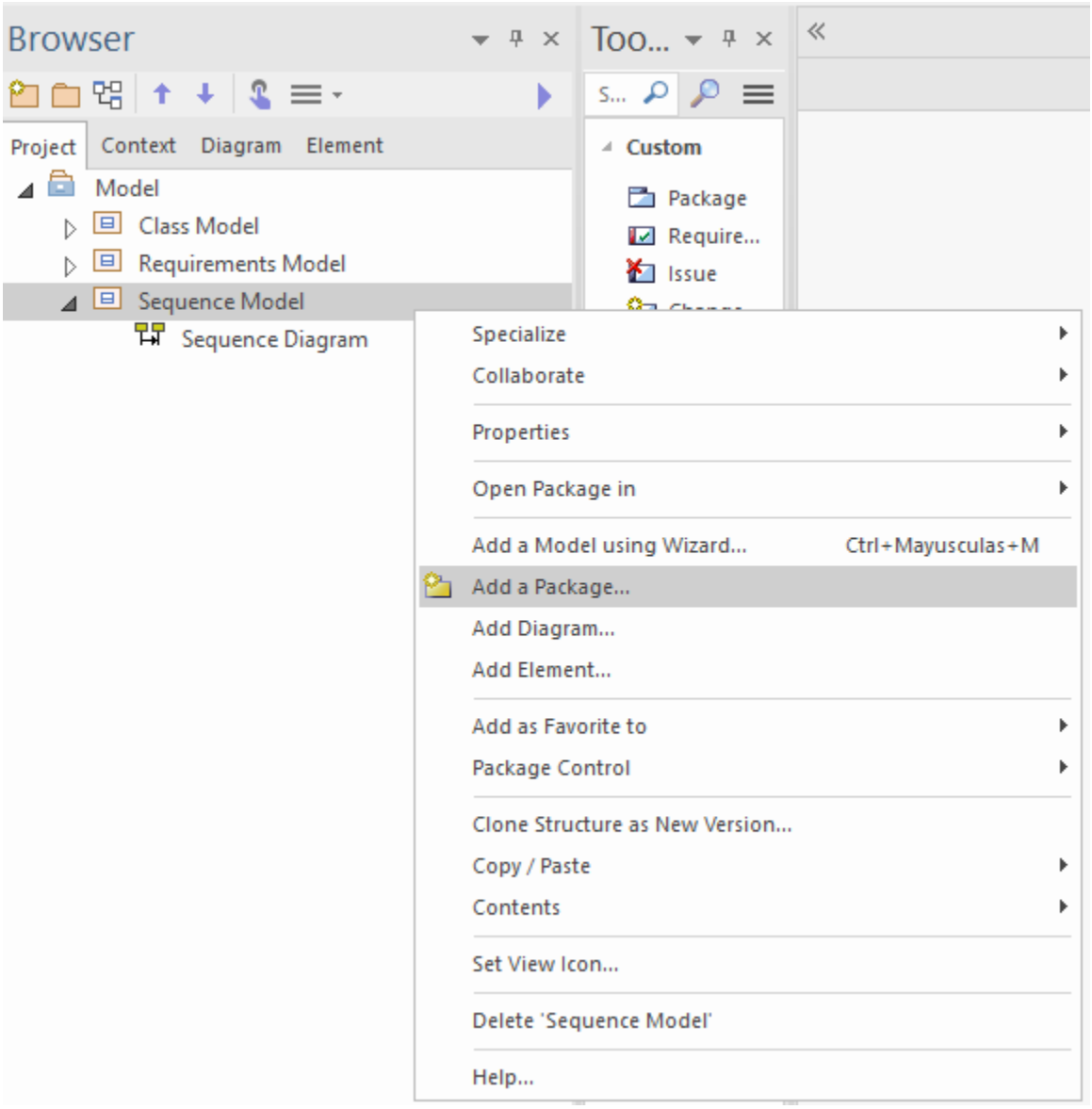
Name it **Sequence Model**, and select **Create Diagram** option



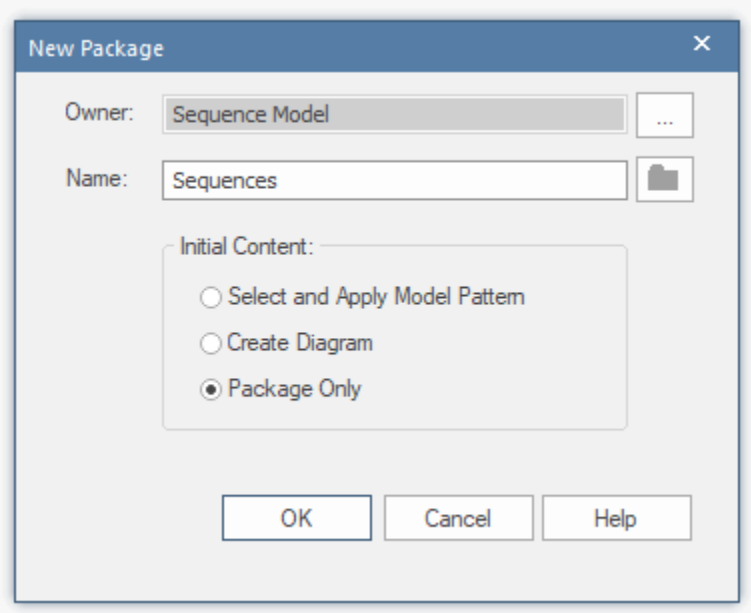
The name of the diagram is **Sequence Diagram**, and it's a **Sequence diagram** type (from **UML Behavioral** category)



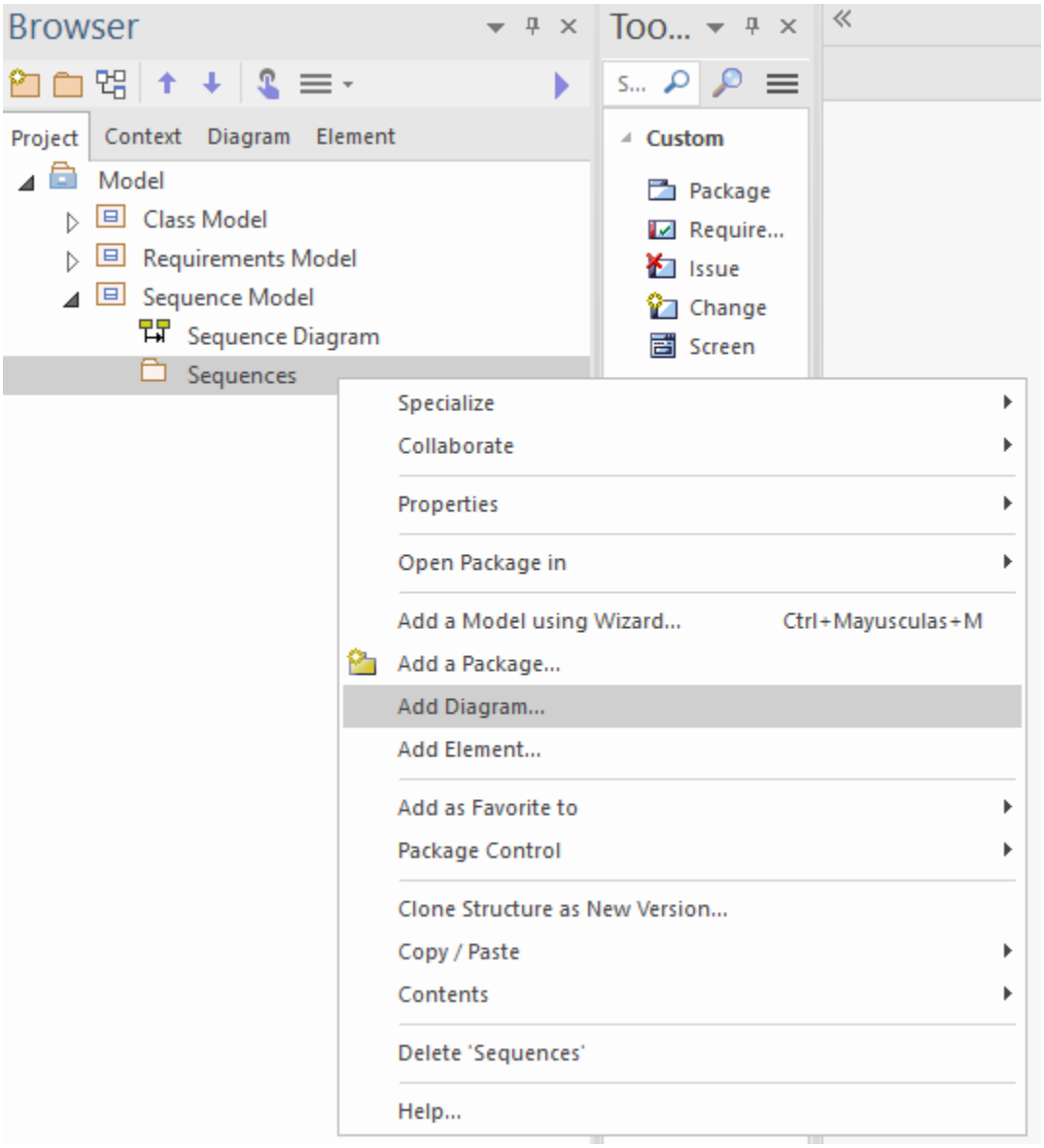
Step 2. Add a **Package** inside the **Sequence Model**



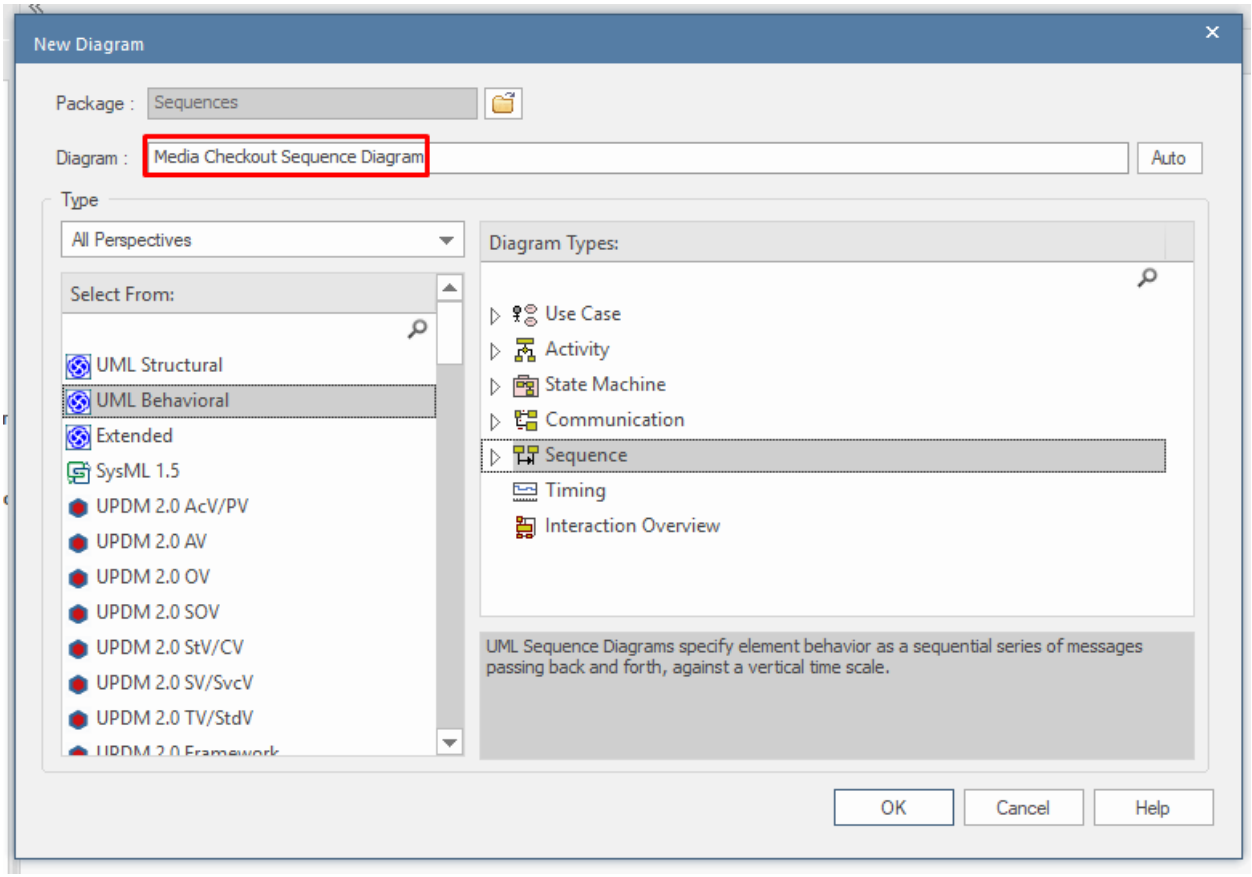
The name is **Sequences** and selecting **Package Only**



Step 3. Let's add a **Sequence Diagram** for each Use Case that you identify:

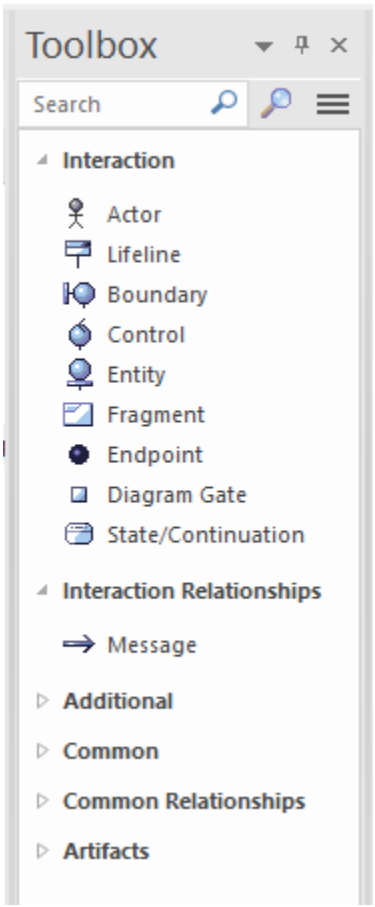


The first one is **Media Checkout Sequence Diagram:**

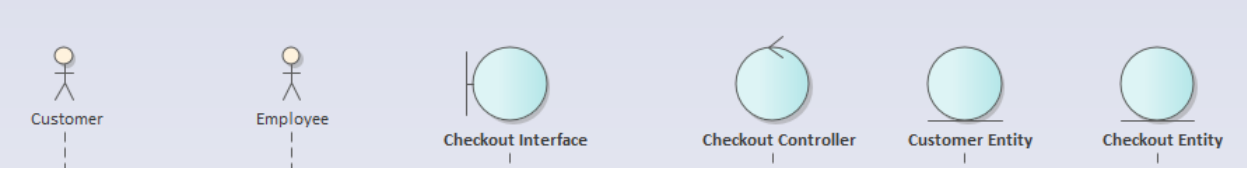


Step 4. Use the **toolbox** and add the following elements:

- **Customer: Actor**
- **Employee: Actor**
- **Checkout Interface: Boundary**
- **Checkout Controller: Control**
- **Customer Entity: Entity**
- **Checkout Entity: Entity**

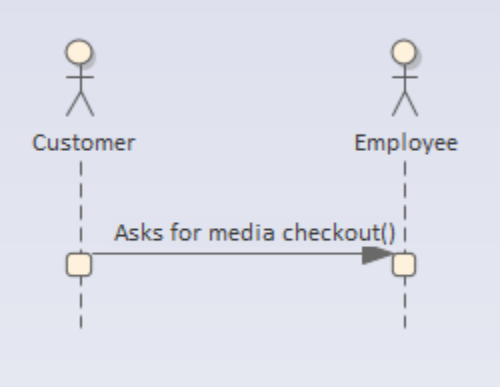


Expected output:

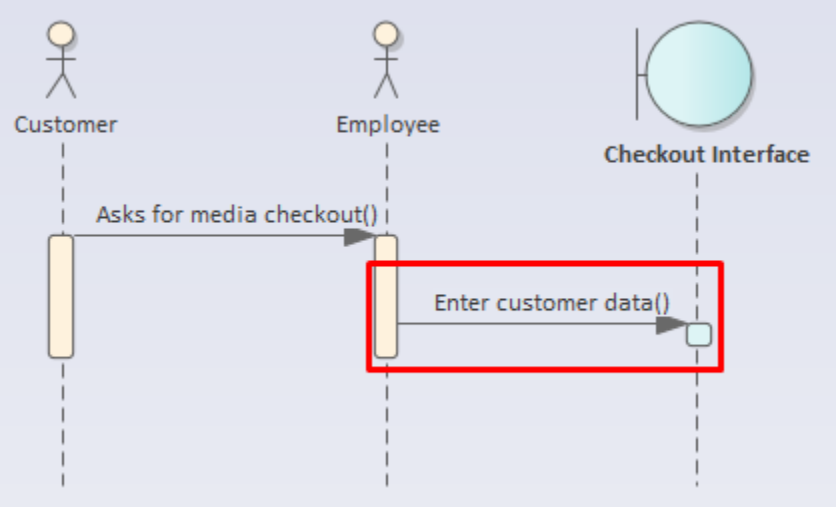


Step 5. Now let's start adding messages between elements in the Sequence Diagram:

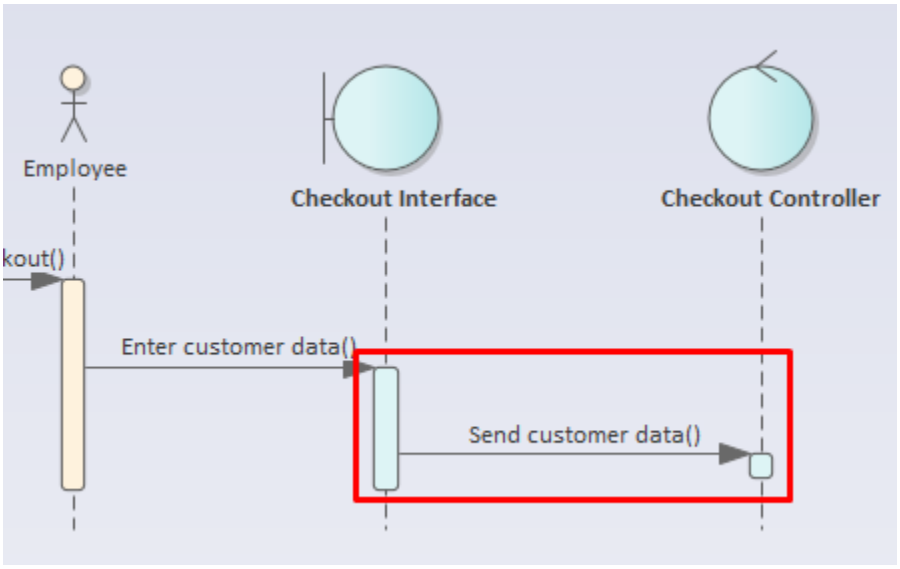
Using the toolbox, add a **message** from the **Customer** actor to the **Employee** actor. Double click it and under the message property, type "**Asks for media checkout**"



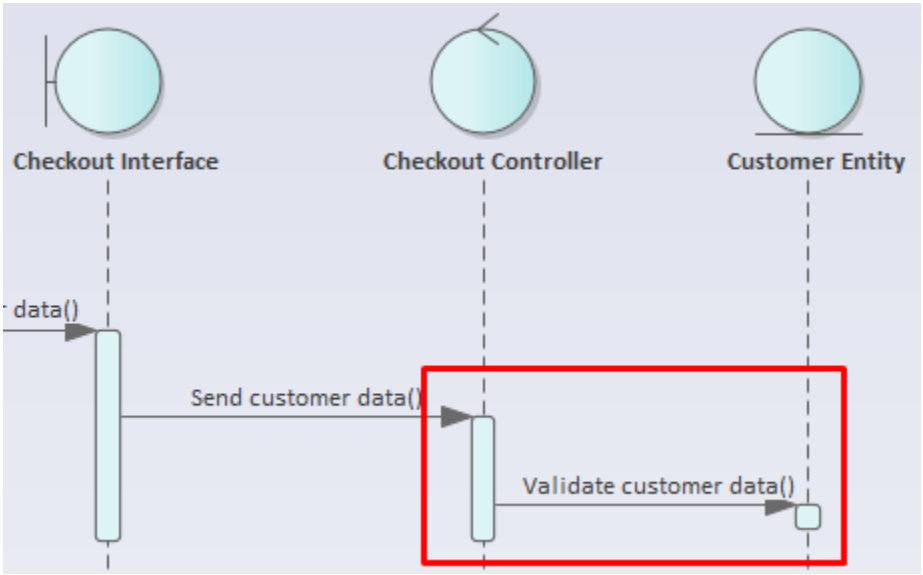
Next, add another message from the **Employee** actor to the **Checkout Interface** boundary with the **Enter customer data** value.



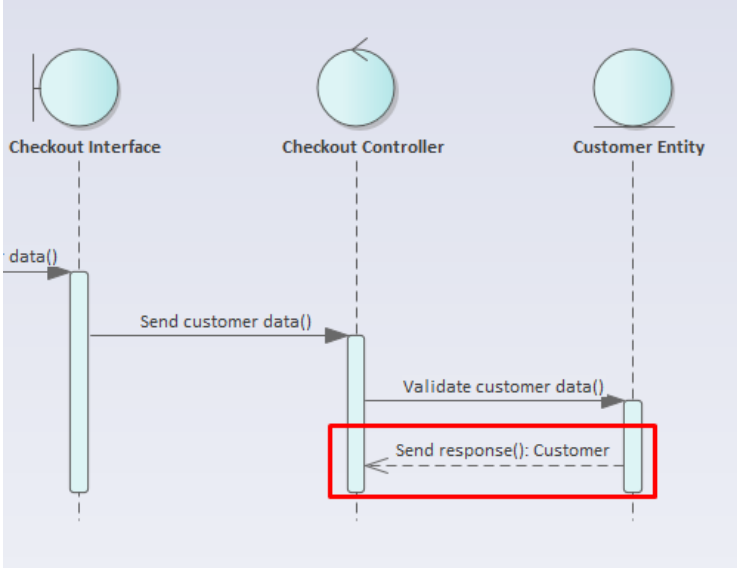
The third message is **Send Customer data** and goes from the **Checkout Interface** boundary to the **Checkout Controller** control.

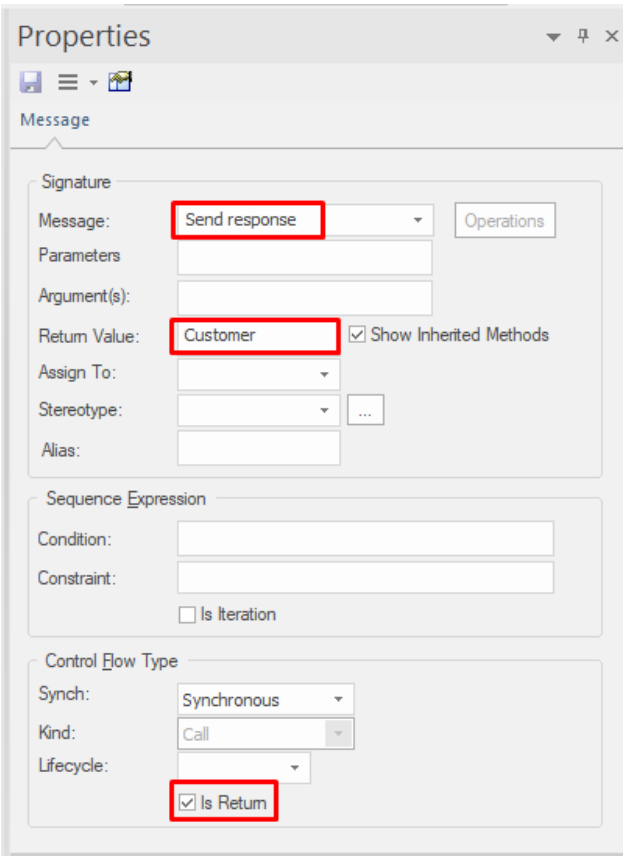


Next up is the **Validate customer data** message, which goes from **Checkout Controller** control to **Customer Entity** entity.

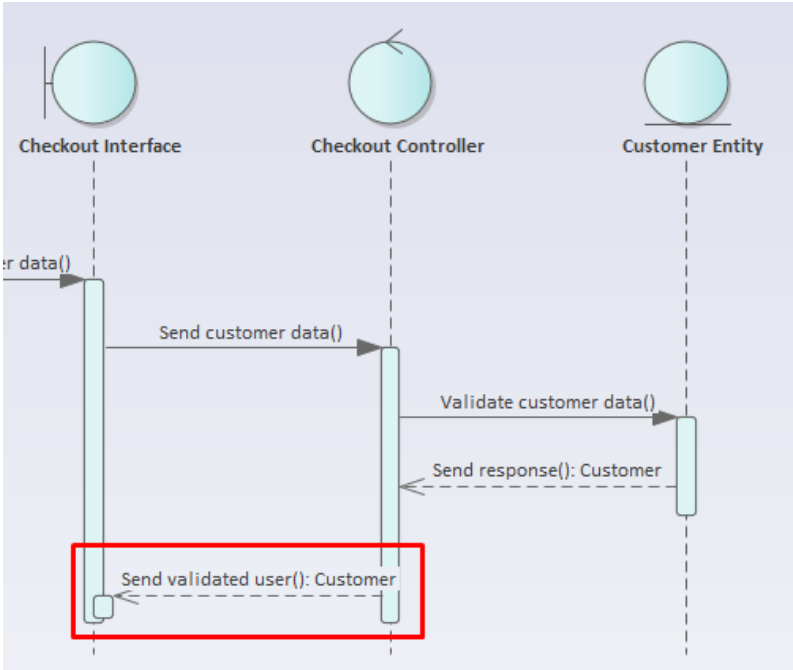


Step 6. Now add another message, this time from the Customer Entity entity to the Checkout Controller control. The message value is Send response, with a Customer return value and the Is Return property activated (true):

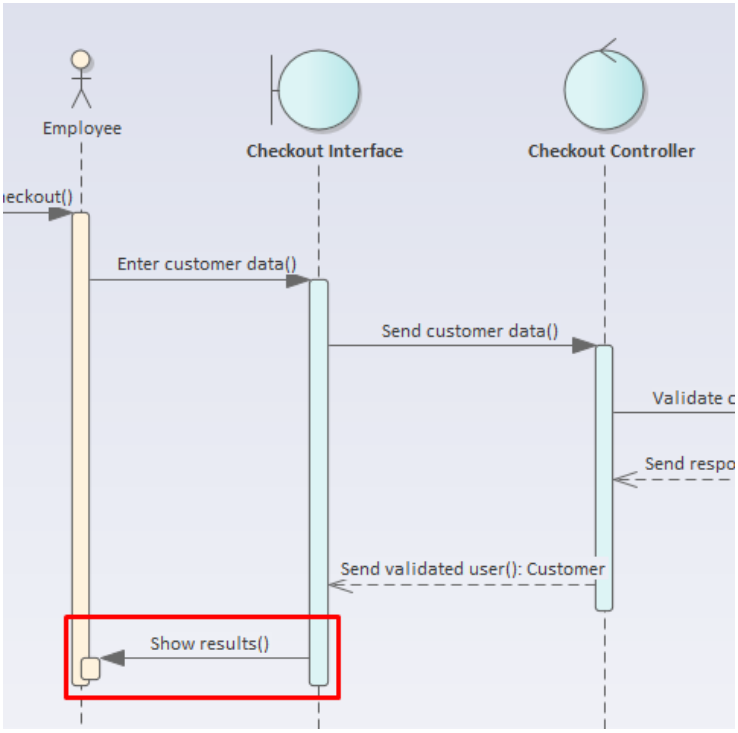




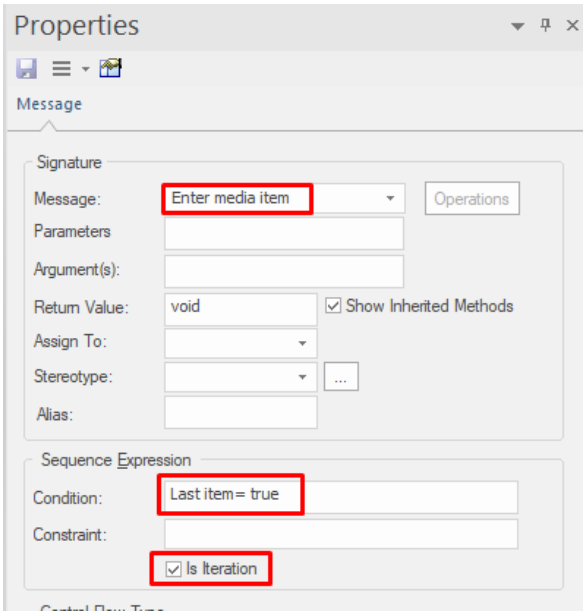
Another **return** message is **Send validated user**, again with **Customer** return value, and goes from **Checkout Controller** control to **Checkout Interface** boundary.



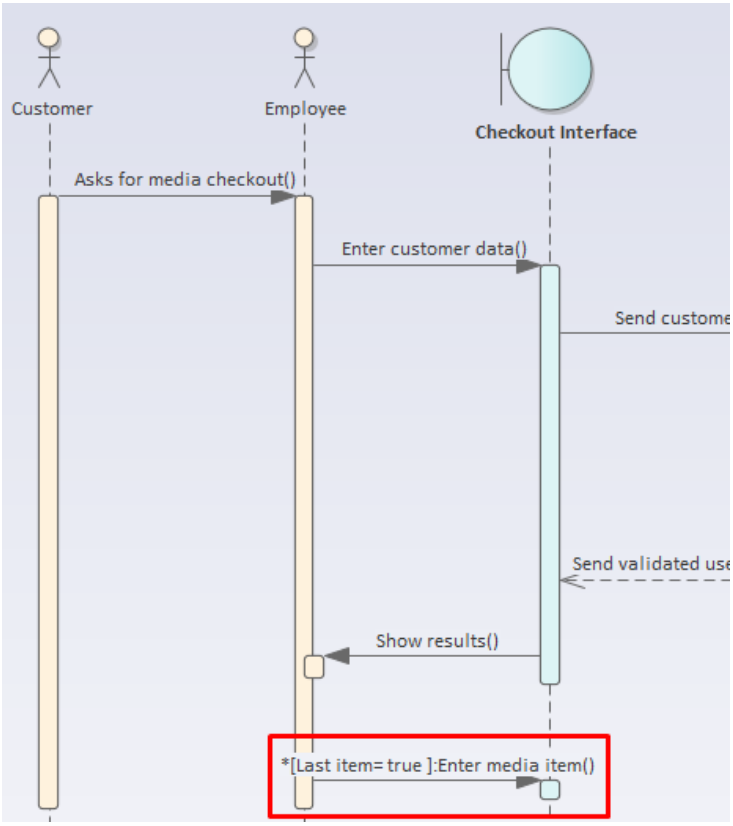
The **Checkout Interface** boundary sends a **Show results** message to the **Employee** actor:



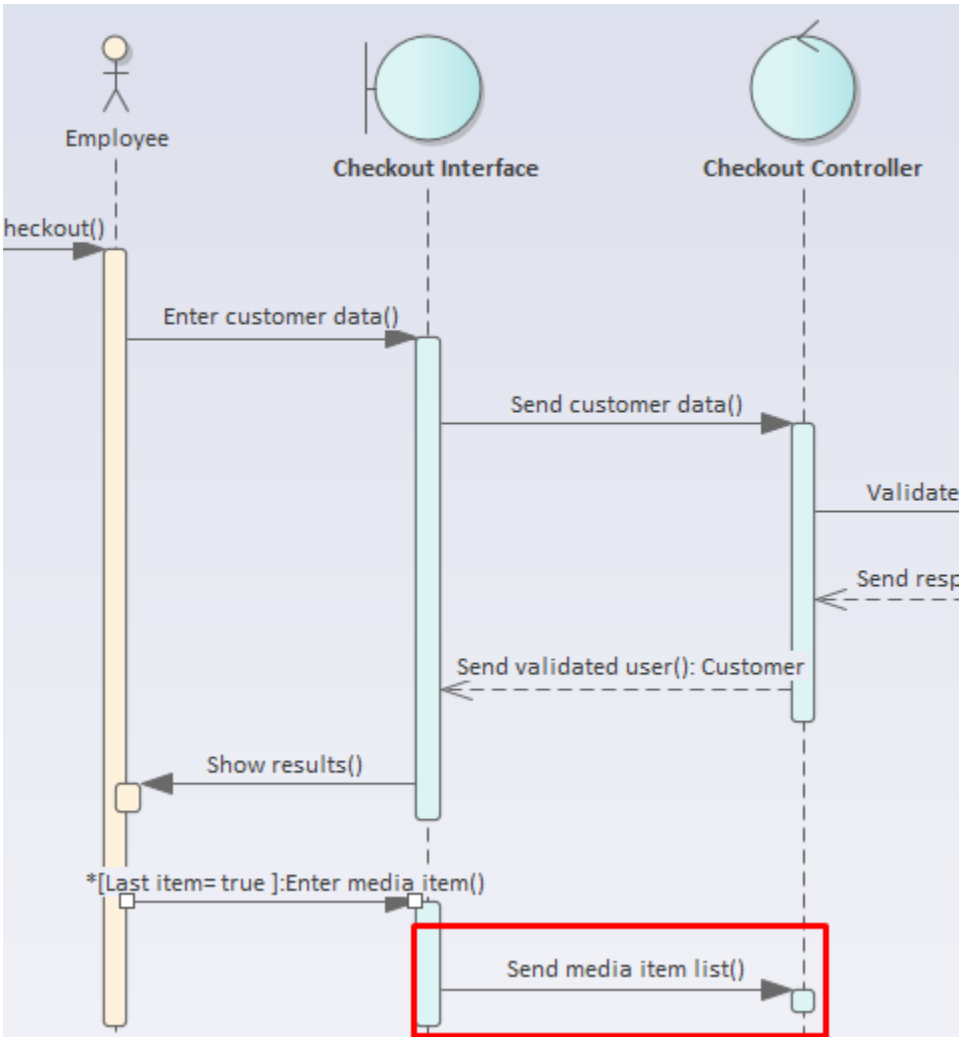
Step 7. A message can be sent several times until certain condition is met. That's the concept of iteration. Add a new message from the **Employee** actor to **Checkout Interface** boundary with the following properties:



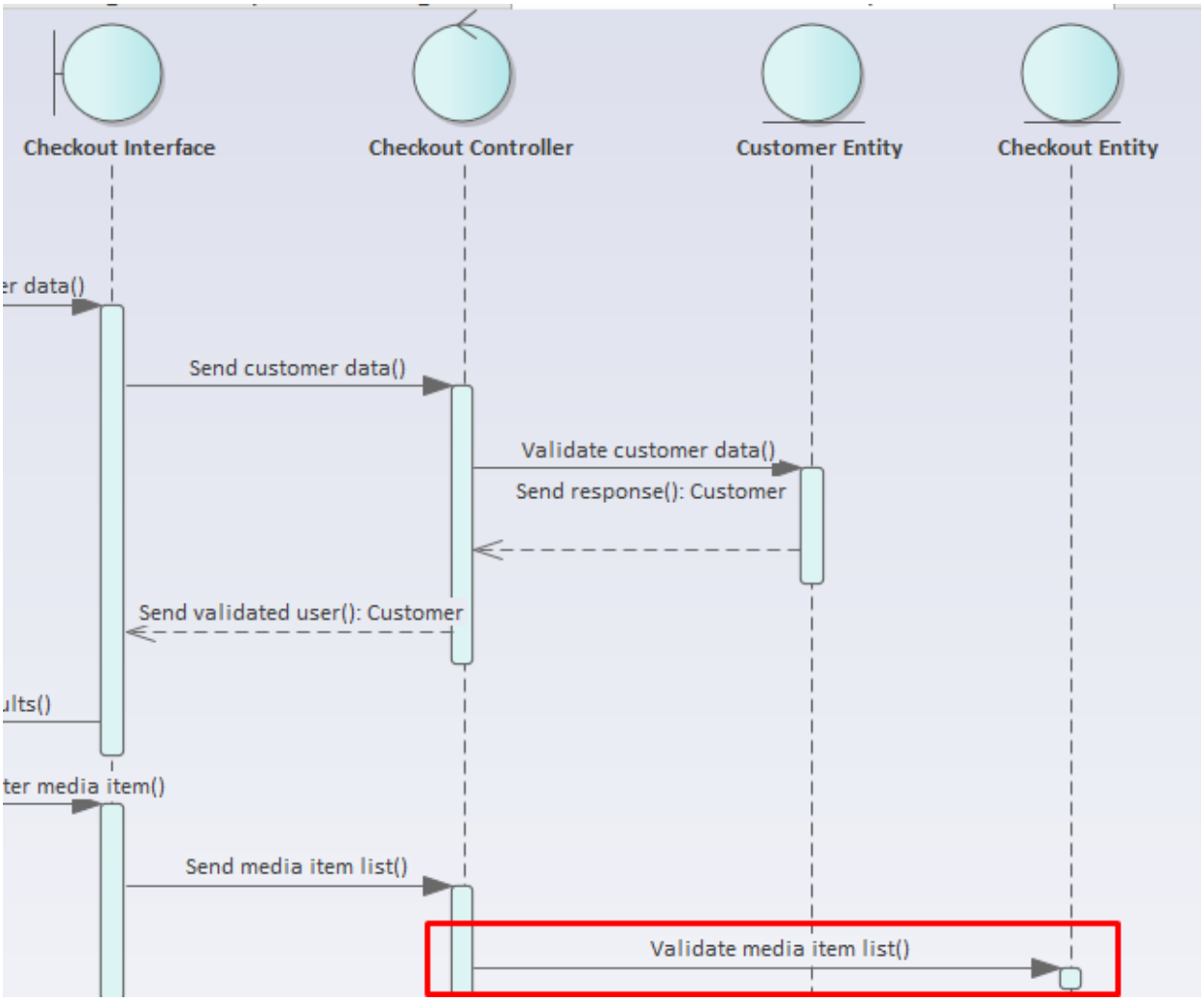
Expected result:



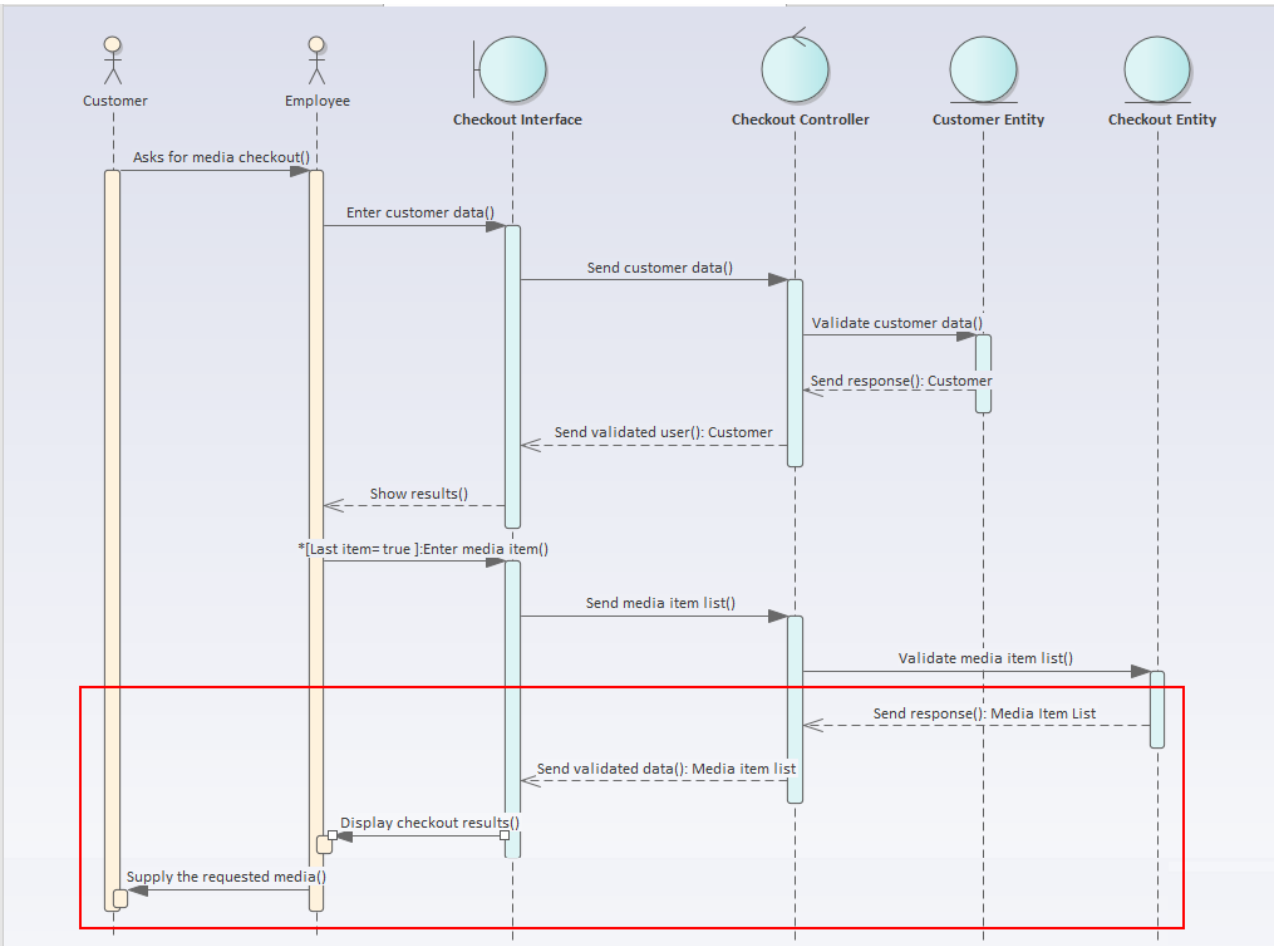
From **Checkout Interface** boundary, add a **Send media item list** message to **Checkout Controller** control:



And from the **Checkout Controller** control, add a **Validate media item list** message to the **Checkout Entity** entity.



Step 8. The rest of the messages deal with the validation of the media item list, in similar way that the Customer data was done. Check the diagram:



New elements added:

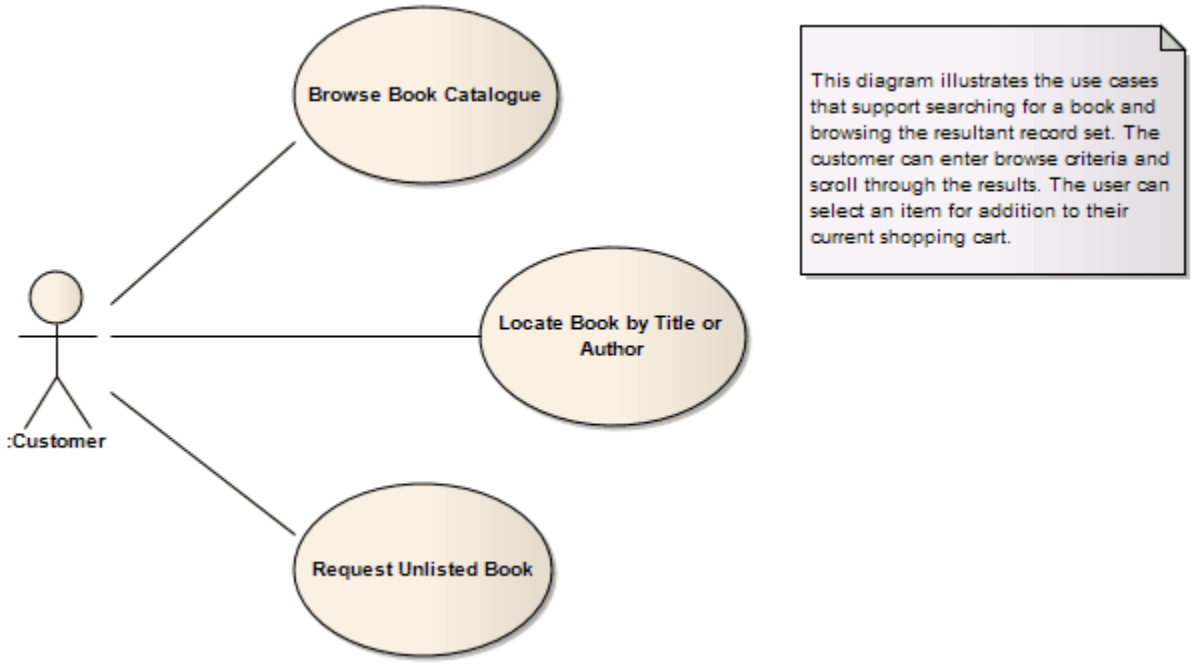
Message	From	To	Attributes
Send response	Checkout Entity	Checkout Controller	Return Value: Media Item List Is Return: ✓
Send validated data	Checkout Controller	Checkout Interface	Return Value: Media Item List Is Return: ✓
Display checkout results	Checkout Interface	Employee	
Supply the requested media	Employee	Customer	

Seminar Session 02. Use Case Diagram (using Enterprise Architect)

Introduction

Use Case diagrams capture Use Cases and the relationships between Actors and the subject (system). You can use them to:

- Describe the functional requirements of the system
- Describe the manner in which outside things (Actors) interact at the system boundary
- Describe the response of the system



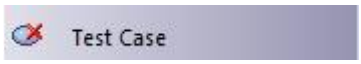
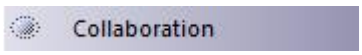
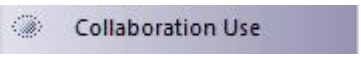
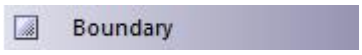
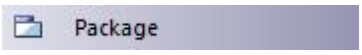
Objective

In this session, you will learn how to design a **Use case diagram** using **Enterprise Architect**. You'll learn how to add interactions between actors and the system based on problem description analysis.

Sequence Diagrams in Enterprise Architect

You can include the following elements in this type of diagrams:

Element	Description
	An Actor is a user of the system; user can mean a human user, a machine, or even another system or subsystem in the model.
	A Use Case is a UML modeling element that describes how a user of the proposed system interacts with the system to perform a discrete unit of work.

	A Test Case is a stereotyped Use Case element which enables you to give greater visibility to tests.
	A Collaboration defines a set of cooperating roles and their connectors.
	A Collaboration Use element allows for a Pattern defined by a Collaboration to applied to a specific situation.
	A System Boundary element is a non-UML element used to define conceptual boundaries.
	Packages are used to organize your project contents, but when added onto a diagram they can be use for structural or relational depictions.

Moreover, you can use connectors such as the following:

Connector	Description
	A Use relationship indicates that one element requires another to perform some interaction.
	An Association implies that two model elements have a relationship, usually implemented as an instance variable in one or both Classes.
	A Generalization is used to indicate inheritance.
	An Include connection indicates that the source element includes the functionality of the target element.
	An Extend connector is used to indicate that an element extends the behavior of another.
	A Realizes connector represents that the source object implements or Realizes its destination object.
	An Invokes connector indicates that source object, at some point, causes the destination object to happen.
	A Precedes connector indicates that the source object must be completed before the destination object can begin.

Problem Description

A library database needs to store information pertaining to:

- Its customers
- Its workers
- The physical locations of its branches,
- And the media stored in those locations (two media types are considered: books and videos).

The library must keep track of the status of each media item: its location, status, descriptive attributes, and cost for losses and late returns. Books will be identified by their ISBN, while movies by their title and year. In order to allow multiple copies of the same book or video, each media item will have a unique ID number.

Customers will provide their name, address, phone number, and date of birth when signing up for a library card. They will then be assigned a unique user name and ID number, plus a temporary password that will have to be changed.

Checkout operations will require a library card, as will requests to put media on hold. Each library card will have its own fines, but active fines on any of a customer's cards will prevent the customer from using the library's services.

The library will have branches in various physical locations. Branches will be identified by name, and each branch will have an address and a phone number associated with it. Additionally, a library branch will store media and have employees.

Employees will work at a specific branch of the library. They receive a paycheck, but they can also have library cards; therefore, the same information that is collected about customers should be collected about employees.

Functions for customers (users):

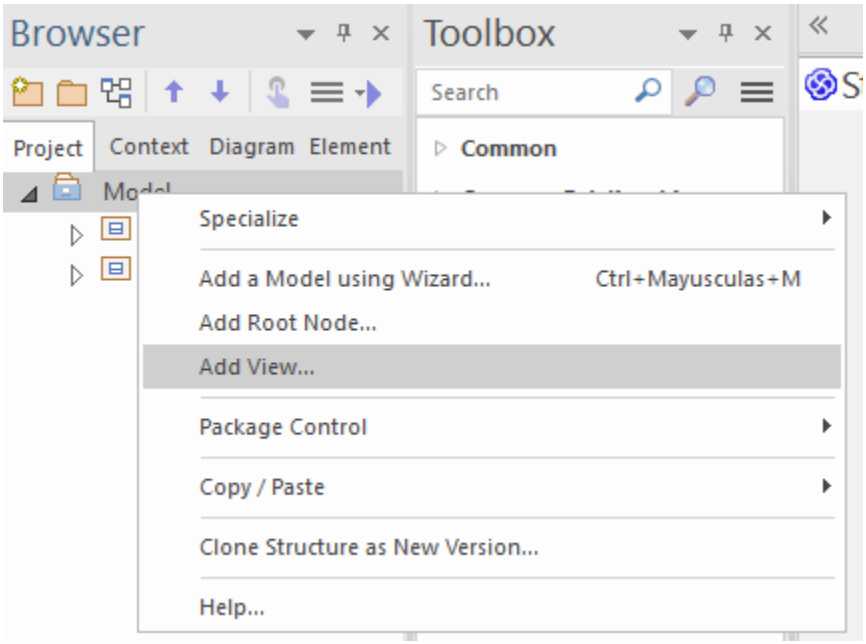
- Log in
- Search for media based on one or more of the following criteria:
    - type (book, video, or both)
    - title
    - author or director
    - year
- Access their own account information:

    - Card number(s)
    - Fines
    - Media currently checked out
    - Media on hold
- Put media on hold
- Pay fines for lost or late items
- Update personal information:
    - Phone numbers
    - Addresses
    - Passwords

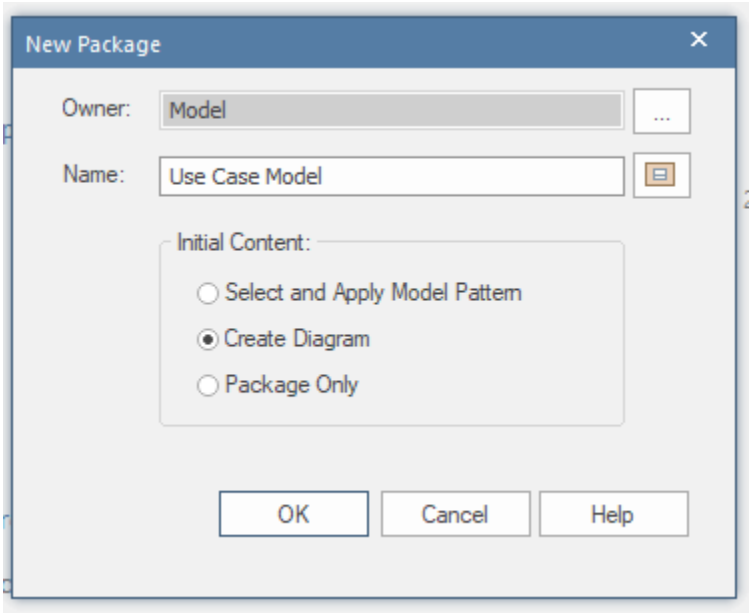
Functions for librarians (employees) are the same as the functions for customers plus the following:

- Add customers
- Add library cards and assign them to customers
- Check out media
- Manage and transfer media that is currently on hold
- Handle returns
- Modify customers' fines
- Add media to the database
- Remove media from the database
- Receive payments from customers and update the customers' fines
- View all customer information except passwords

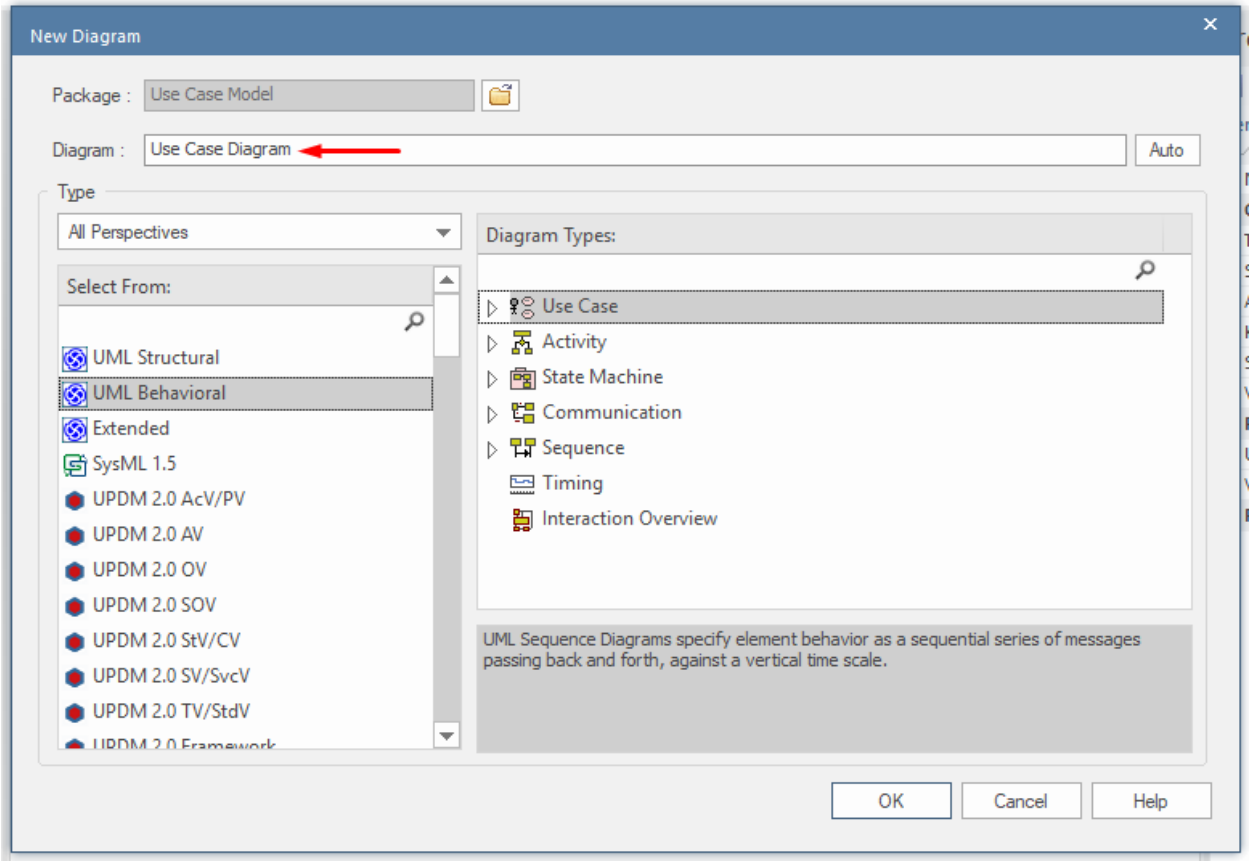
Step 1. Add a **View** in your **Model**



Name it Use Case Model, and select **Create Diagram** option



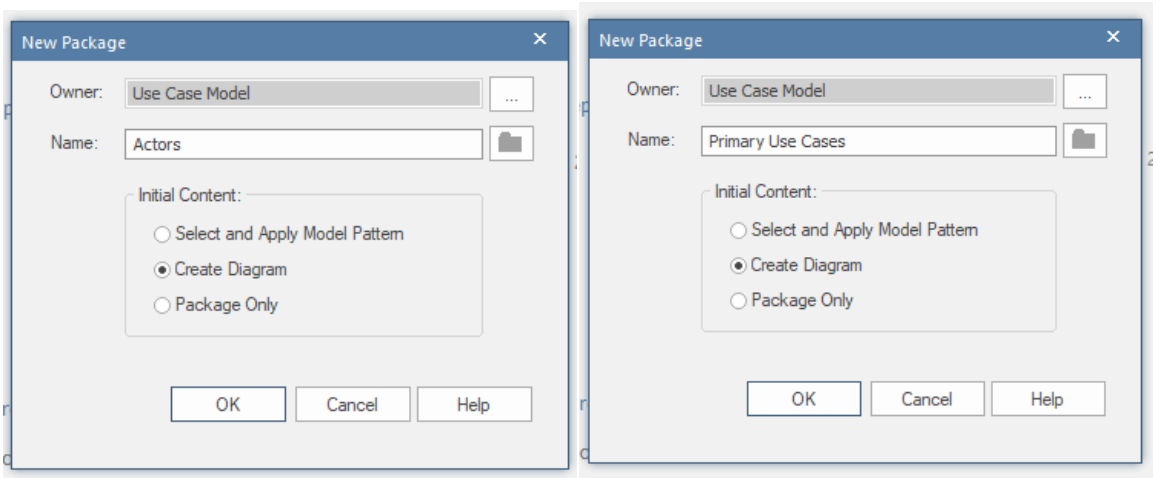
The name of the diagram is **Use Case Diagram**, and it's a **Use case diagram** type (from **UML Behavioral** category)

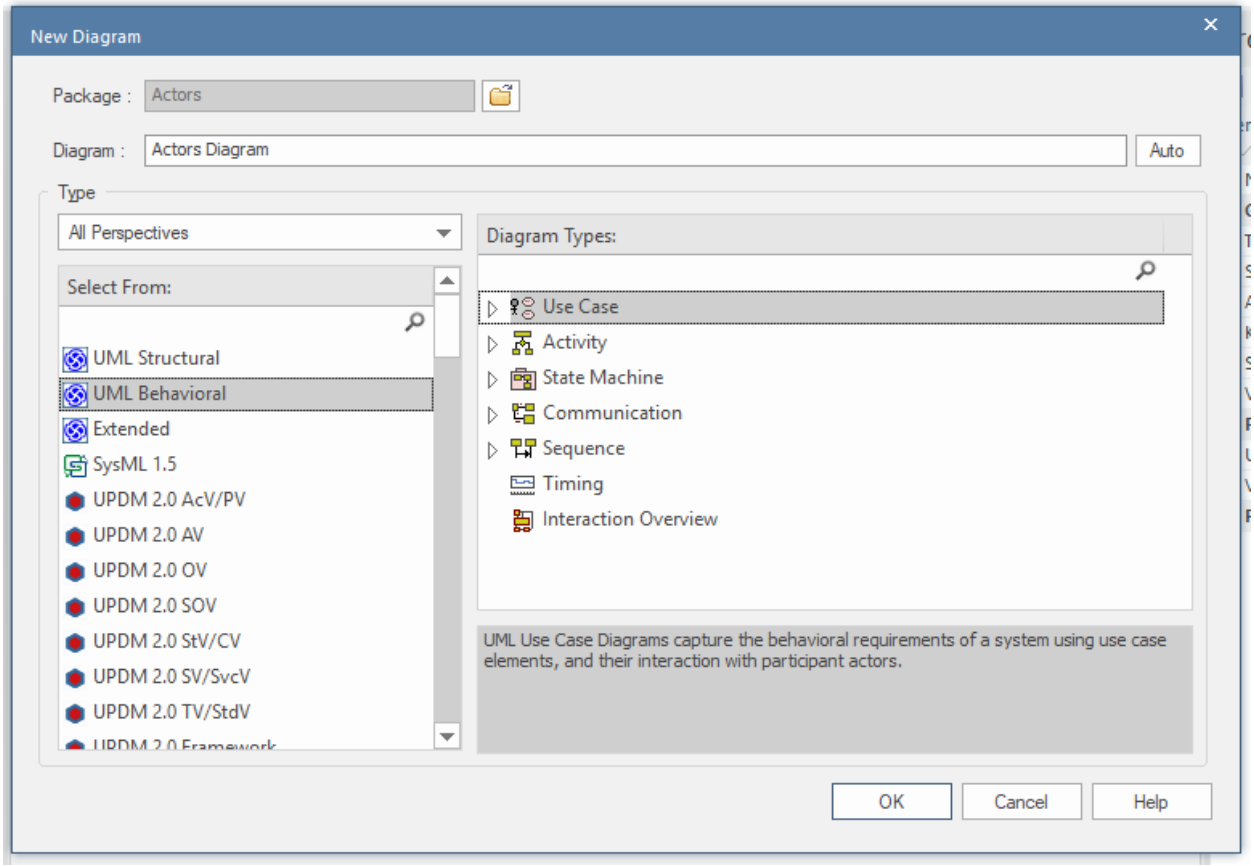


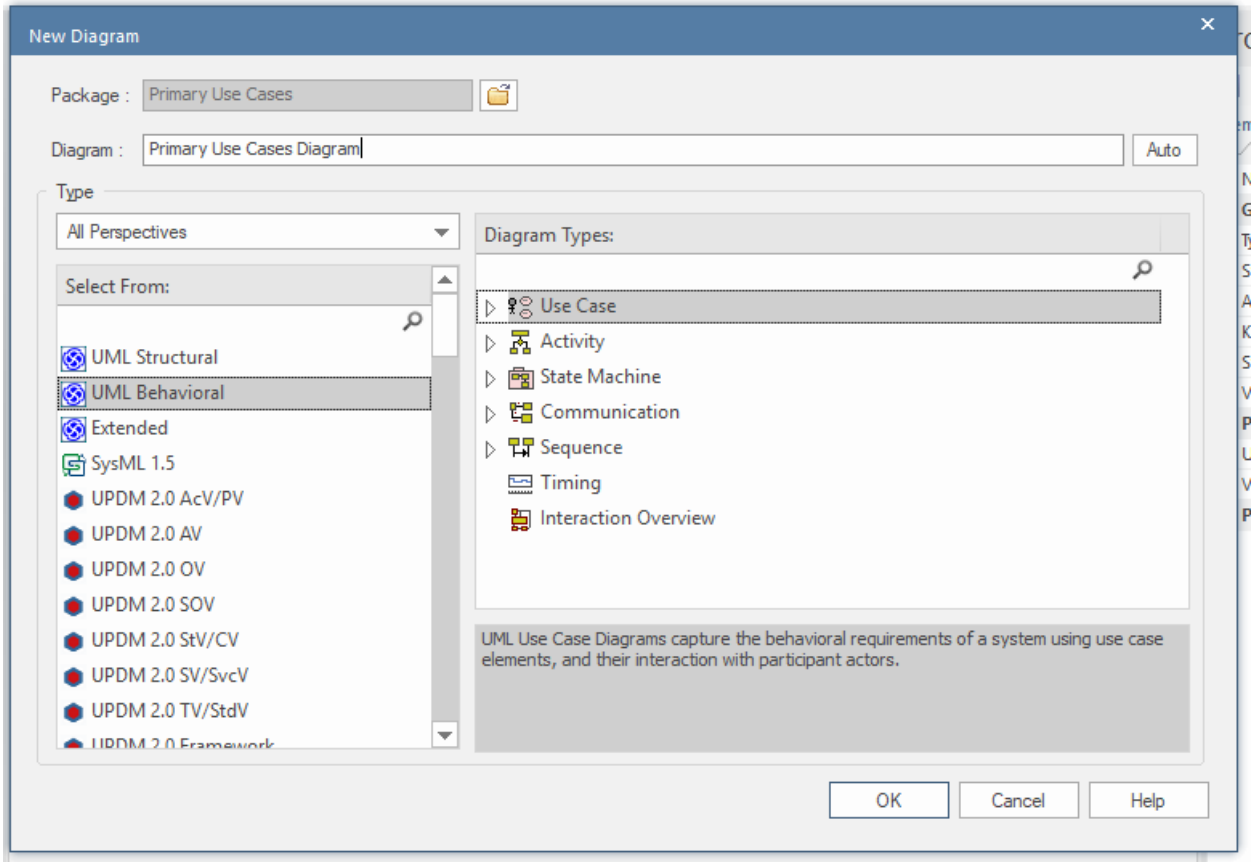
Step 2. Add two **Packages** inside the **Use Case Model:**

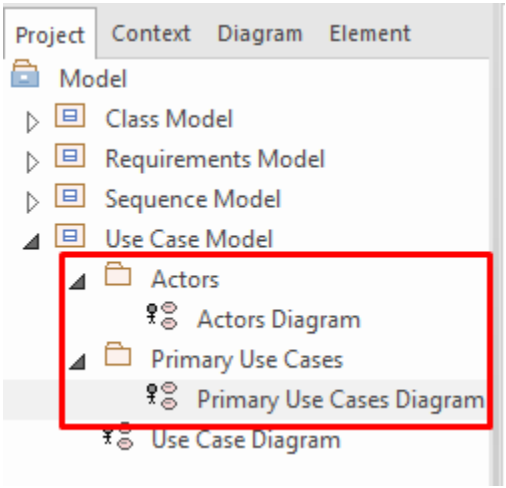
- **Actors**
- **Primary Use Cases**

Each one contains a Use Case Diagram:



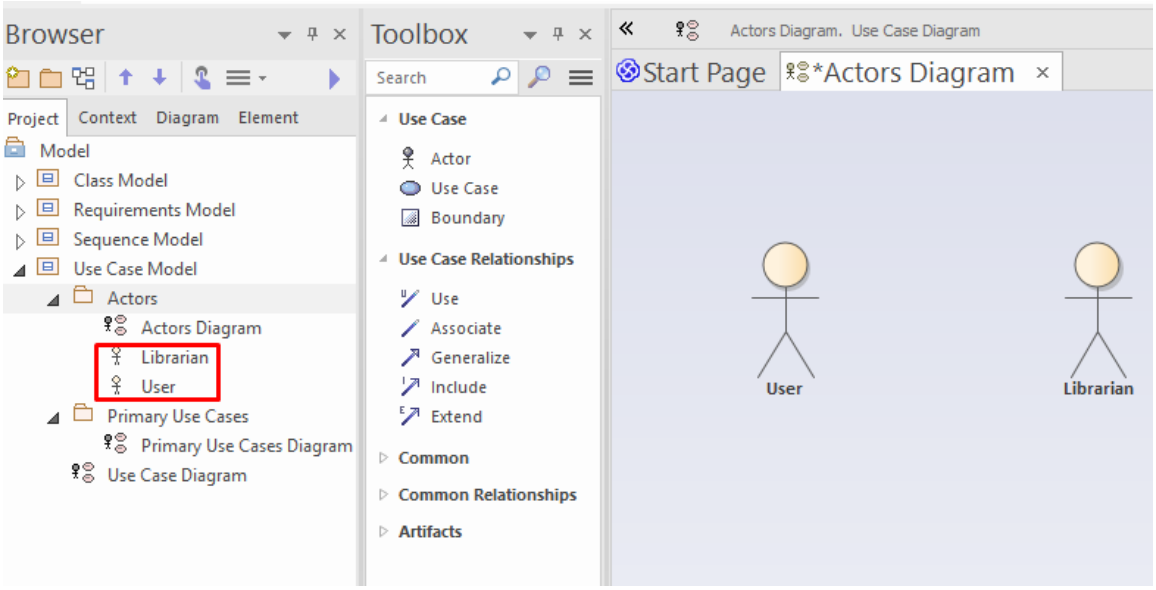

New Diagram
Package :
Actors
Diagram :
Actors Diagram
Auto
Type
All Perspectives
Select From:
UML Structural
UML Behavioral
Extended
SysML 1.5
UPDM 2.0 AcV/PV
UPDM 2.0 AV
UPDM 2.0 OV
UPDM 2.0 SOV
UPDM 2.0 StV/CV
UPDM 2.0 SV/SvcV
UPDM 2.0 TV/StdV
Diagram Types:
Use Case
Activity
State Machine
Communication
Sequence
Timing
Interaction Overview
UML Use Case Diagrams capture the behavioral requirements of a system using use case elements, and their interaction with participant actors.
OK
Cancel
Help


New Diagram
Package :
Primary Use Cases
Diagram :
Primary Use Cases Diagram
Auto
Type
All Perspectives
Select From:
UML Structural
UML Behavioral
Extended
SysML 1.5
UPDM 2.0 AcV/PV
UPDM 2.0 AV
UPDM 2.0 OV
UPDM 2.0 SOV
UPDM 2.0 StV/CV
UPDM 2.0 SV/SvcV
UPDM 2.0 TV/StdV
Diagram Types:
Use Case
Activity
State Machine
Communication
Sequence
Timing
Interaction Overview
UML Use Case Diagrams capture the behavioral requirements of a system using use case elements, and their interaction with participant actors.
OK
Cancel
Help

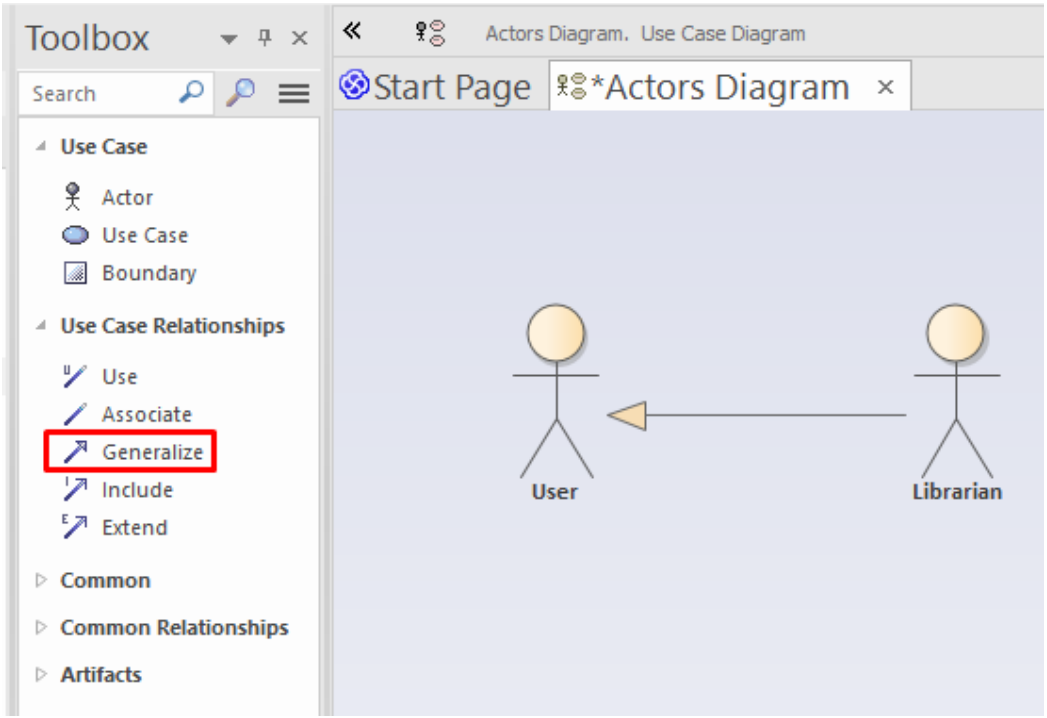

Project
Context
Diagram
Element
Model
Class Model
Requirements Model
Sequence Model
Use Case Model
Actors
Actors Diagram
Primary Use Cases
Primary Use Cases Diagram
Use Case Diagram

Step 3. In the Actors Diagram, add a couple of actors:

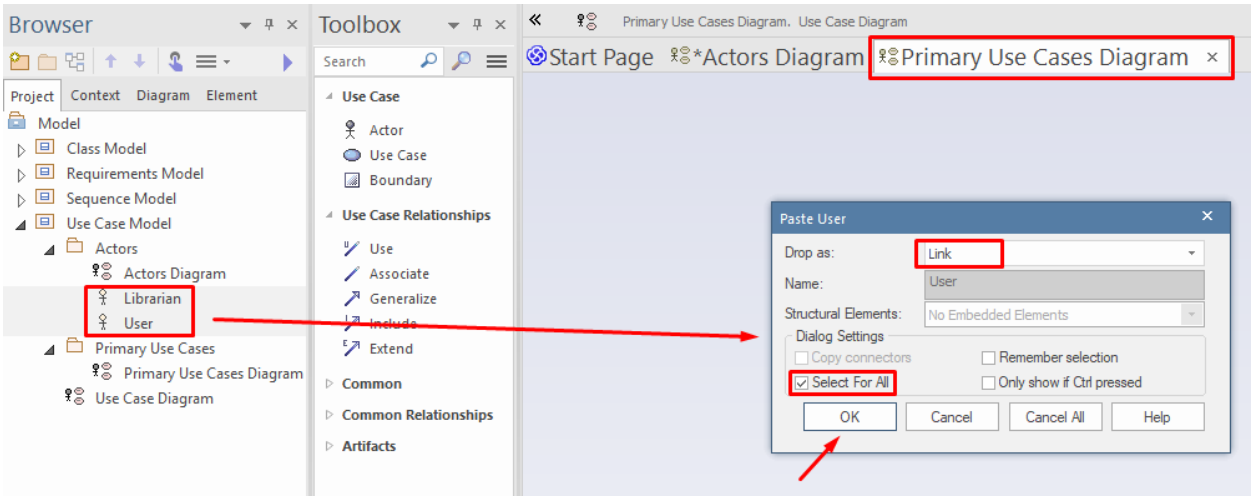
- User
- Librarian



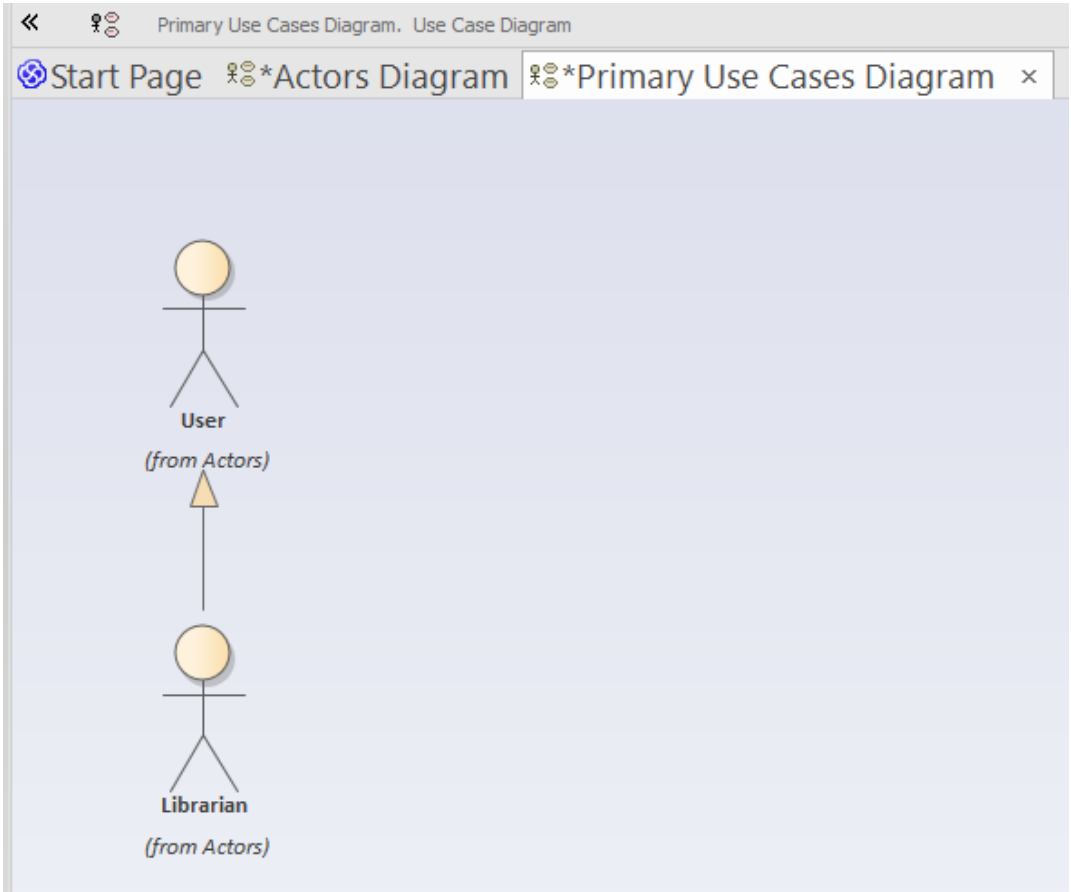
Add a Generalization relationship from the Librarian to the User actor



Now select both actors from the Browser section and add them as Link to the Primary Use Cases Diagram:



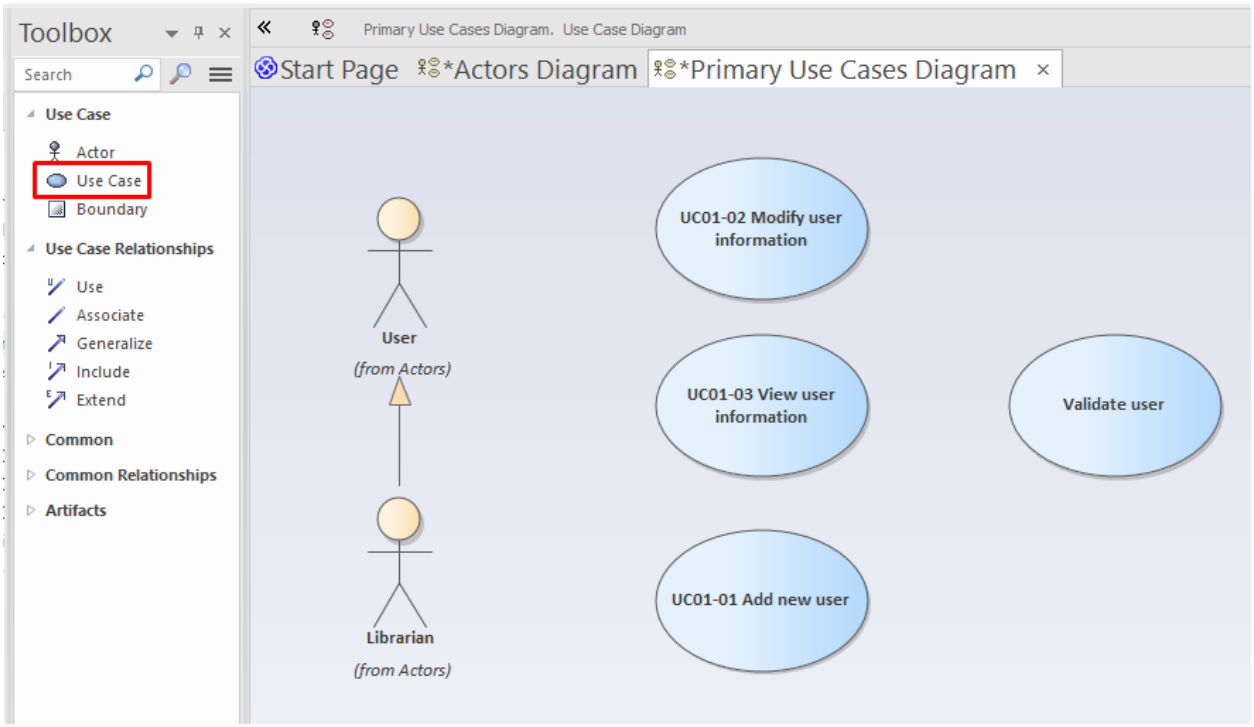
Expected output



Step 4. Add four Use Cases to the Primary Use Cases Diagram:

- UC01-01 Add new user
- UC01-02 Modify user information
- UC01-03 View user information
- UC Validate user

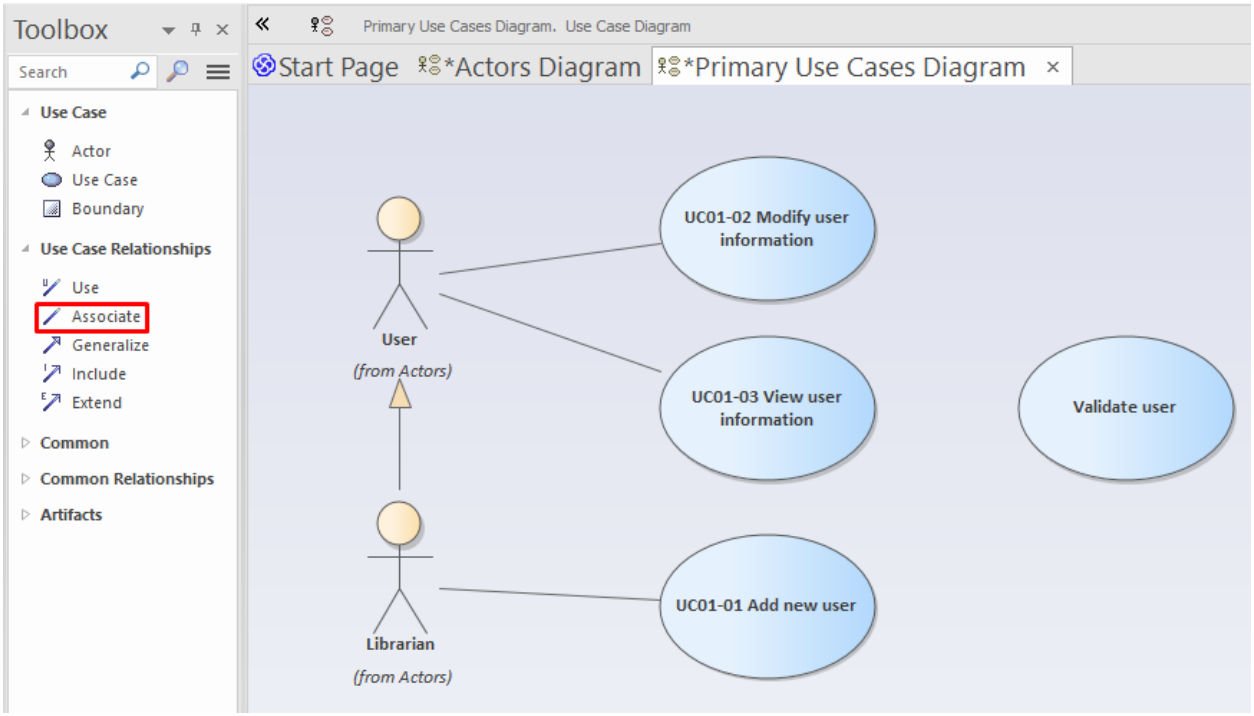
Expected output:



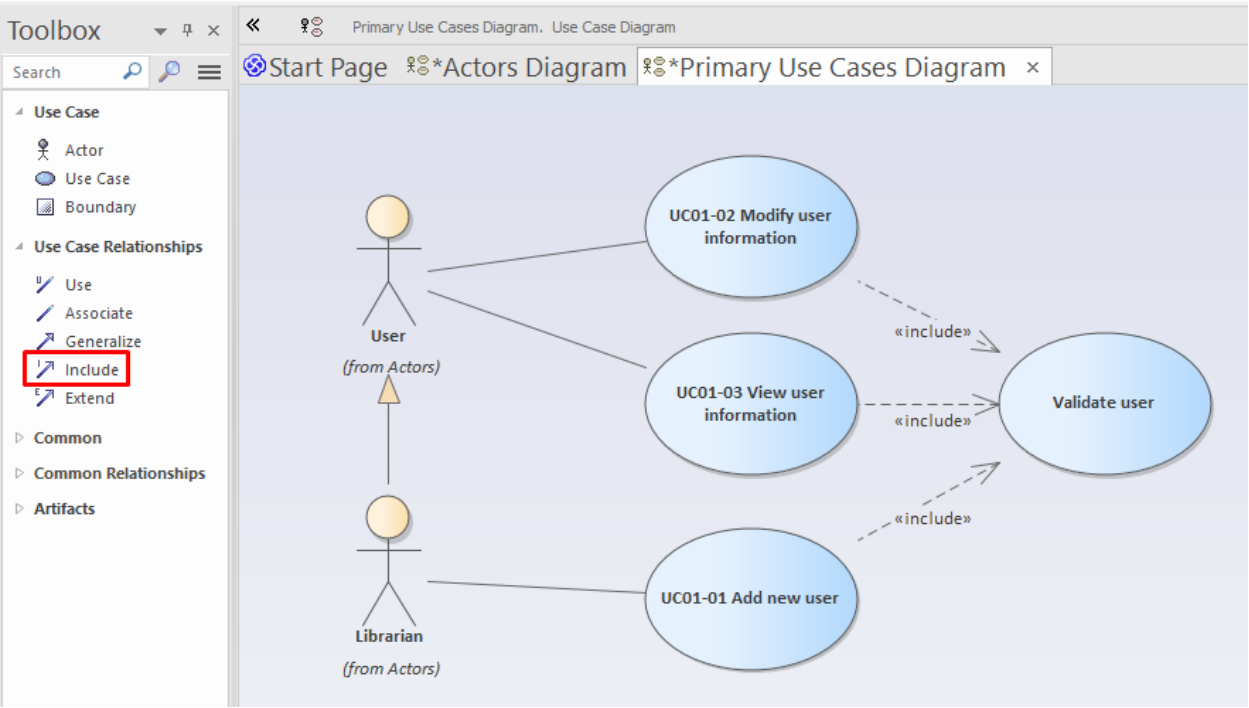
Add an Association relationship between the actors and use cases:

- User → UC01-02 Modify user information
- User → UC01-03 View user information
- Actor → UC01-01 Add new user

Expected output



Now add an Include relationship from each of the three use cases to the Validate user use case



Step 5. Add a Boundary around the use cases. Name it User Management System