

Lesson-6:Software Design Fundamentals UML Class Diagrams

CS 438
Ali Aburas PhD

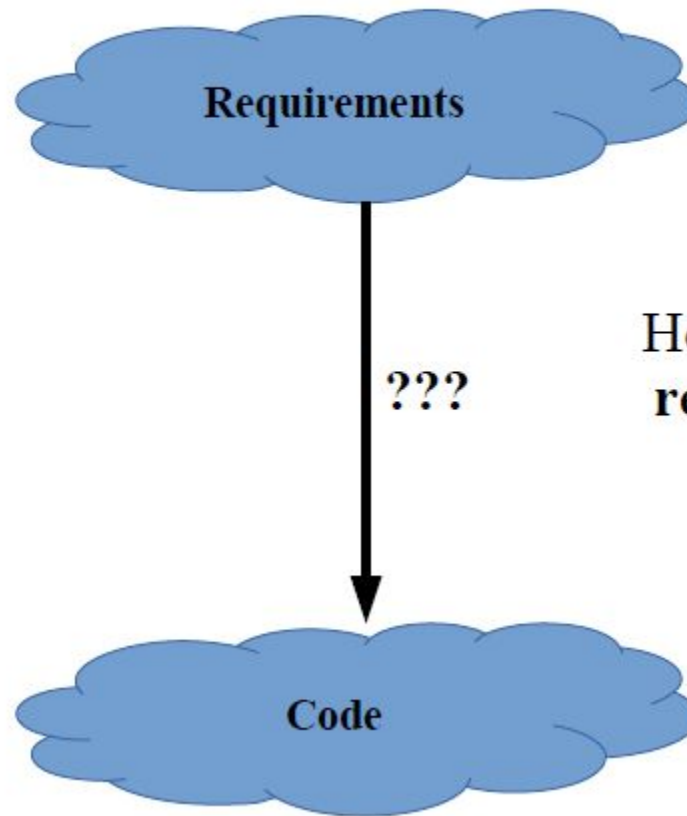
Today's Goals

- **Define design**
- Introduce the design process
- **What is UML?**
- **Define design**
- **What is UML Class Diagram?**
- **What is UML Sequence Diagram?**



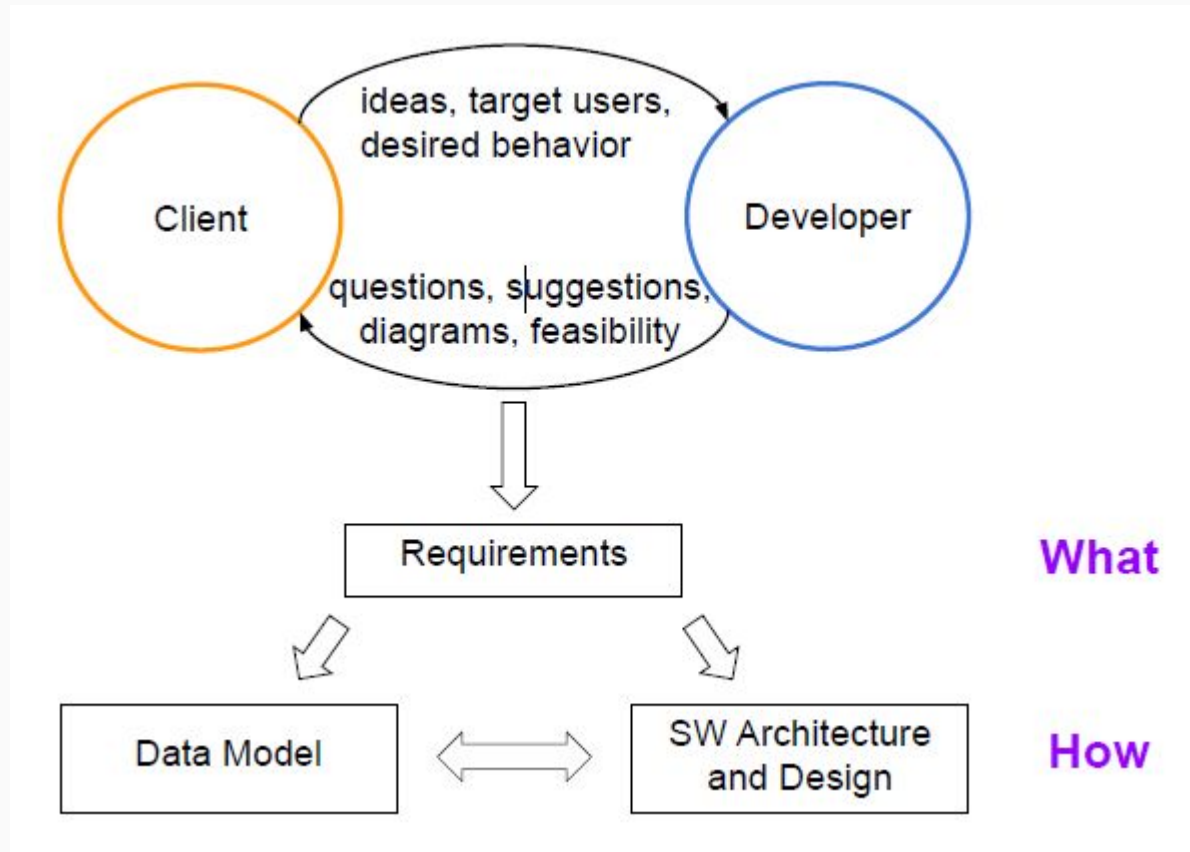
The Problem

- Here are the requirements. Go **build** it!



How do we get from the **requirements** to **code**?

From Requirements to System Design



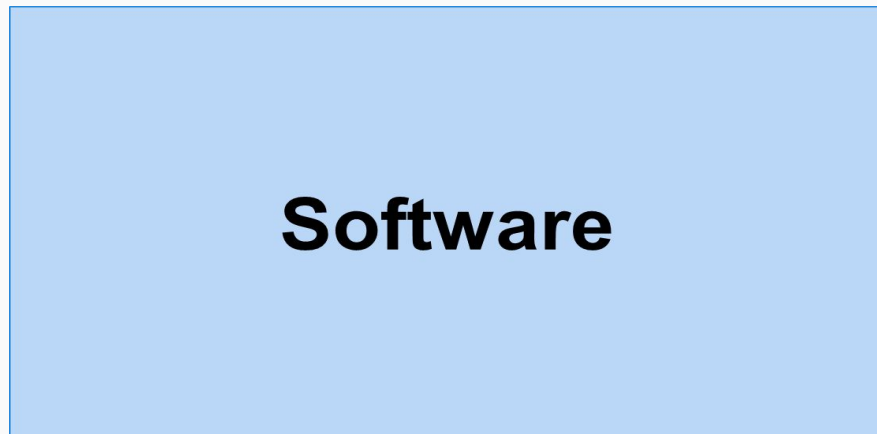
What is Design?

- Design is the creative process of transforming a problem into a solution.
- In our case, transforming a requirements specification into a detailed description of the software to be implemented.
- **Requirements** - *what* we're going to build.
- **Design** - *how* to build it.
 - A description of the structure of the solution

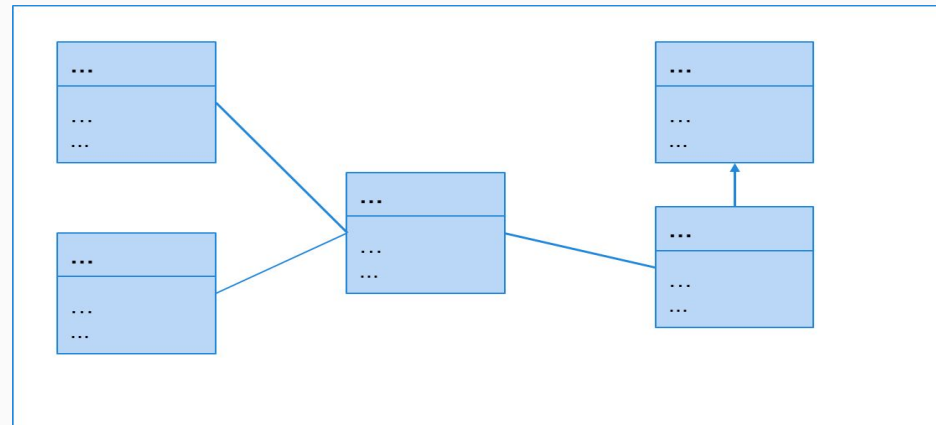
What is Design?

- Design is the process of defining the structure of the software.
 - What units make up the codebase?
 - How do those units connect to perform the required functions?

Design is the process of going from this:



... to this:



Stages of Design

Three repeating stages:

- **Problem Understanding**
 - Look at the problem from different angles to discover what needs the design needs to capture.
- **Identify Solutions**
 - Evaluate possible solutions and choose the most appropriate in terms of available resources.
- **Describe and Document Chosen Solution**
 - Use graphical, formal, or other descriptive notations to describe the components of the design.

Stages of Design

Design is performed at multiple levels of granularity:

- **Architecture**
 - How is the system structured into *subsystems*?
 - How do those subsystems work together?
- **Unit**
 - What *units* make up these subsystems?
 - How do these units work together?
- **Low-Level**
 - What algorithms will be employed?
 - What data structures will be used?

Design Activities

**Requirements
Specification**

**Architectural
Design**

**Interface
Design**

**Component
Design**

Data Design

**Algorithm
Design**

System
architecture
(high-level
breakdown of
system)

How subsystems
interact with each
other and how
external users
and systems
interact with your
system

A listing of the
individual classes
within your
system

The format of
data that is
produced and
consumed by
your system

Algorithms used
to implement
system
functionality.

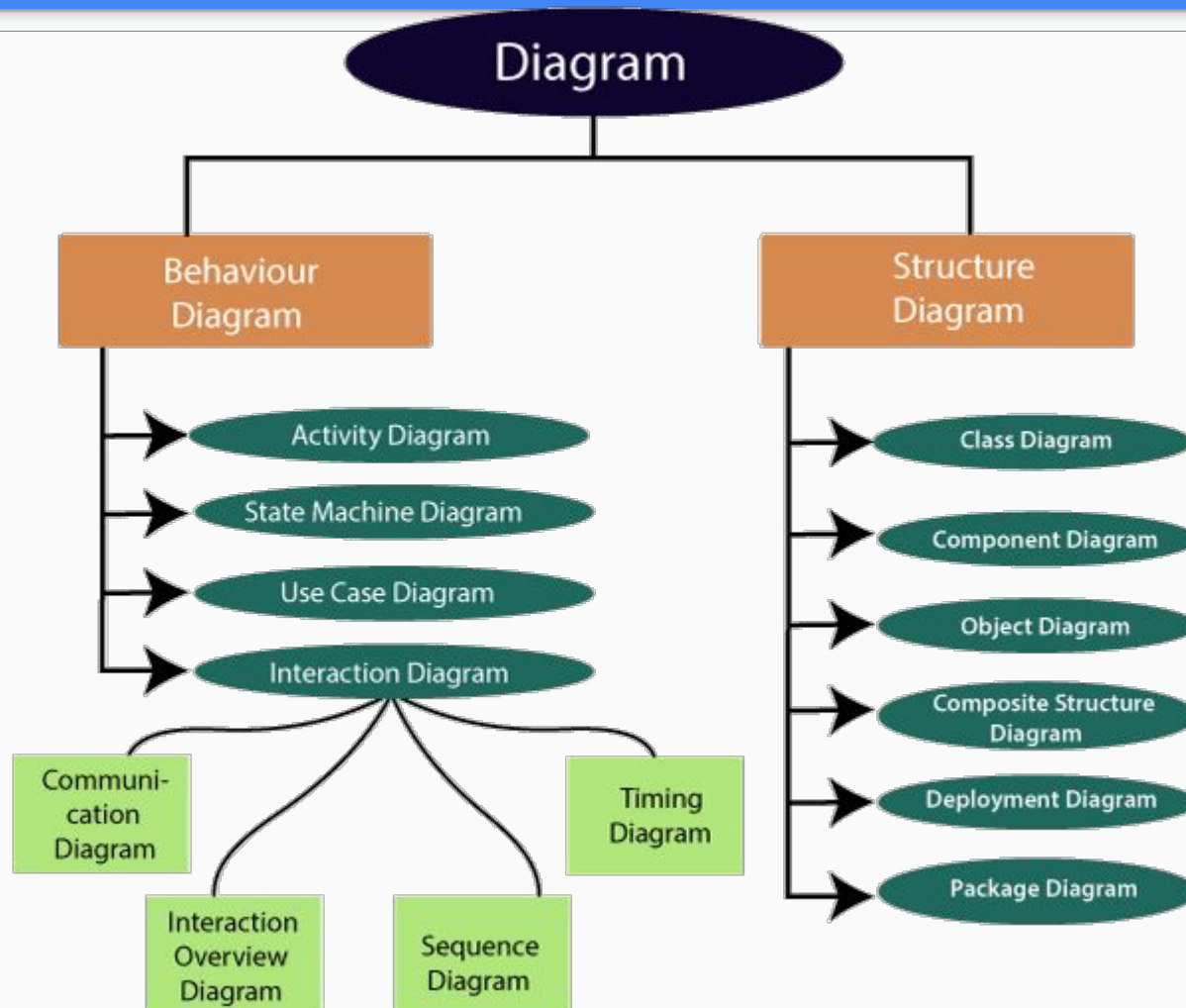
Models for Requirements Engineering and Design

- **Use models/diagrams to analyze and specify requirements.**
 1. We build models/diagrams so that we can better understand the system we are developing.
 2. The models/diagrams provide a bridge between the client's understanding and the developers.
 3. We build models/diagrams of complex system because we cannot comprehend such system in its entirety
 4. We use models/diagrams to *communicate, visualize, or analyze* something, and that something has some sort of structure.
 5. We use models of the existing system during requirements gathering to help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses.

Unified Modeling Language (UML)

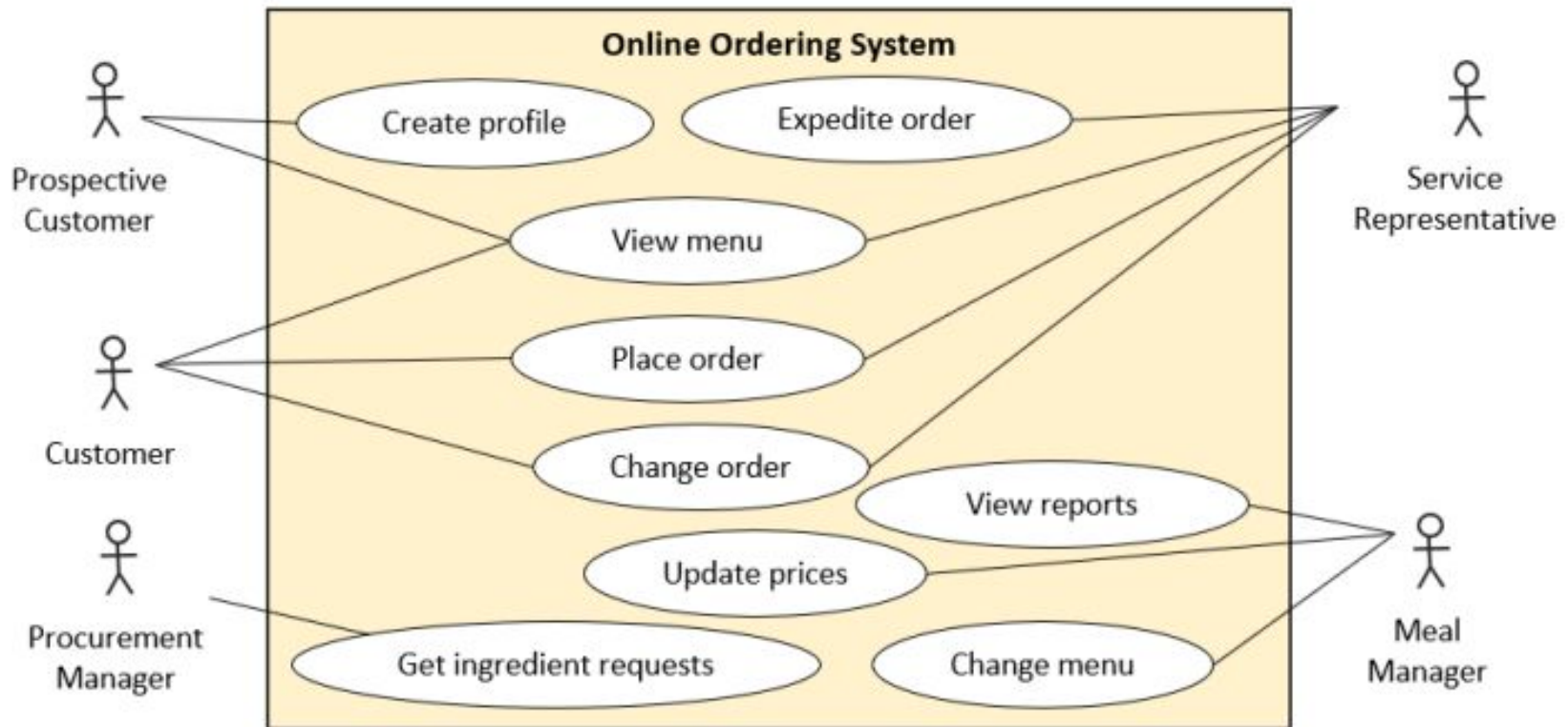
- Modeling Language (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system.
 - Drawing Tools: [ArgoUML](#), Visio (Microsoft), [UMLet](#), [Violet](#), [Dia](#), IBM Rational Rose
- UML Models: System development focuses on three different models of the system:
 1. **Structural diagrams** depict a static view or structure of a system. It is widely used in the documentation of software architecture.
 - It embraces class diagrams, composite structure diagrams, component diagrams, deployment diagrams, object diagrams, and package diagrams.
 - It presents an outline for the system. It stresses the elements to be present that are to be modeled.
 2. **Behavioral diagrams** describes a dynamic view of a system or the behavior of a system, which describes the functioning of the system.
 - It includes use case diagrams, state diagrams, and activity diagrams.
 - It defines the interaction within the system.
 3. **Interaction Diagrams** :shows how objects interact with each other and how the data flows within them.
 - It consists of communication, interaction overview, sequence, and timing diagrams.

UML-Diagrams



Do we need all these diagrams?

Use Case Diagrams and Templates

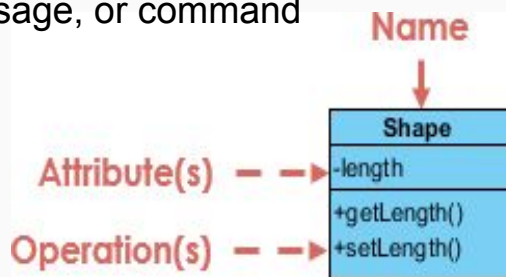


| | | |
|------------------------------|--|---|
| Use case name: | <i>Ship items.</i> | |
| Scenario: | Ship items for a new sale. | |
| Triggering event: | Shipping is notified of a new sale to be shipped. | |
| Brief description: | Shipping retrieves sale details, finds each item and records it is shipped, records which items are not available, and sends shipment. | |
| Actors: | Shipping clerk. | |
| Related use cases | None. | |
| Stakeholders: | Sales, Marketing, Shipping, warehouse manager. | |
| Preconditions: | Customer and address must exist. Sale must exist. Sale items must exist. | |
| Postconditions: | Shipment is created and associated with shipper. Shipped sale items are updated as shipped and associated with the shipment. Unshipped items are marked as on back order. Shipping label is verified and produced. | |
| Flow of activities: | Actor | System |
| | 1. Shipping requests sale and sale item information. 2. Shipping assigns shipper. 3. For each available item, shipping records item is shipped. 4. For each unavailable item, shipping records back order. 5. Shipping requests shipping label supplying package size and weight. | 1.1 System looks up sale and returns customer, address, sale, and sales item information. 2.1 System creates shipment and associates it with the shipper. 3.1 System updates sale item as shipped and associates it with shipment. 4.1 System updates sale item as on back order. 5.1 System produces shipping label for shipment. 5.2 System records shipment cost. |
| Exception conditions: | 2.1 Shipper is not available to that location, so select another. 3.1 If order item is damaged, get new item and updated item quantity. 3.1 If item bar code isn't scanning, shipping must enter bar code manually. 5.1 If printing label isn't printing correctly, the label must be addressed manually. | |

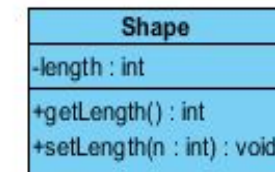
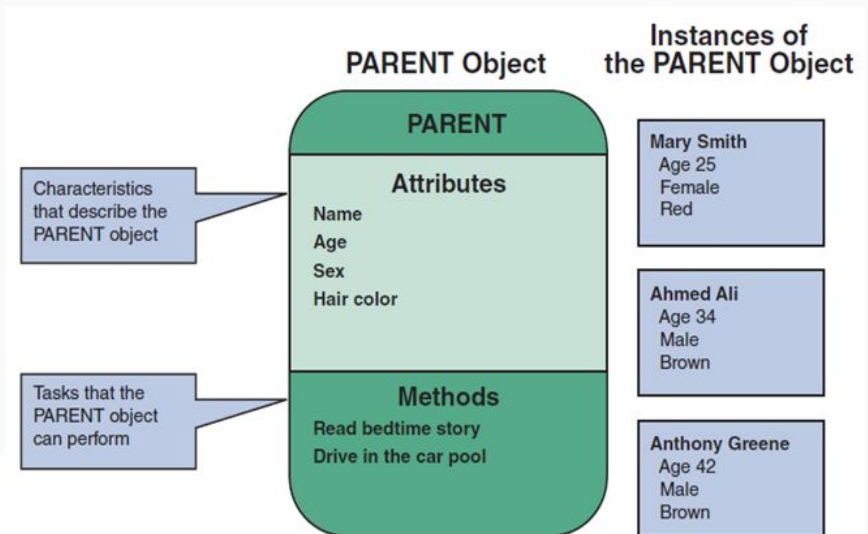
| | | |
|------------------------------|--|--|
| Use case name: | <i>Create customer account.</i> | |
| Scenario: | Create online customer account. | |
| Triggering event: | New customer wants to set up account online. | |
| Brief description: | Online customer creates customer account by entering basic information and then following up with one or more addresses and a credit or debit card. | |
| Actors: | Customer. | |
| Related use cases: | Might be invoked by the <i>Check out shopping cart</i> use case. | |
| Stakeholders: | Accounting, Marketing, Sales. | |
| Preconditions: | Customer account subsystem must be available. Credit/debit authorization services must be available. | |
| Postconditions: | Customer must be created and saved. One or more Addresses must be created and saved. Credit/debit card information must be validated. Account must be created and saved. Address and Account must be associated with Customer. | |
| Flow of activities: | Actor | System |
| | 1. Customer indicates desire to create customer account and enters basic customer information. 2. Customer enters one or more addresses. 3. Customer enters credit/debit card information. | 1.1 System creates a new customer. 1.2 System prompts for customer addresses. 2.1 System creates addresses. 2.2 System prompts for credit/debit card. 3.1 System creates account. 3.2 System verifies authorization for credit/debit card. 3.3 System associates customer, address, and account. 3.4 System returns valid customer account details. |
| Exception conditions: | 1.1 Basic customer data are incomplete. 2.1 The address isn't valid. 3.2 Credit/debit information isn't valid. | |

UML Class Diagram

- **UML Class Diagram** represents a static structural view of the system and it describes the classes and their structure, and their relationships among classes in the system
- **Class** is a description of a set of objects that share the same attributes, methods/operations, and relationships.
- **Elements of UML class**
 1. **Attributes**: describe the characteristics of an object. **Attributes** of an object are defined during the system development process
 2. **Methods**: tasks or functions that the object performs when it receives a message, or command



Class without signature



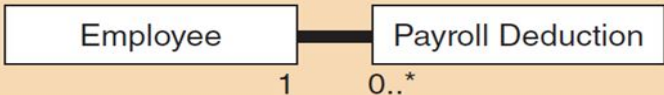
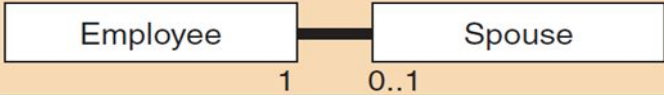
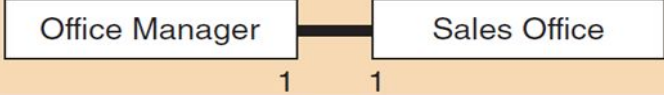

Class with signature

Relationships between classes

- Association:

- **Associations** are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.
- **Cardinality**: UML notations that indicate the nature of the relationship between instances of one class and instances of another class.

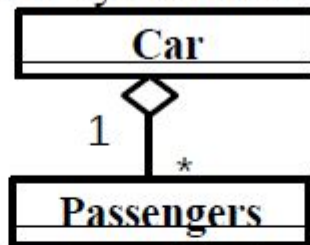
Checking, Savings, and Credit Accounts are generalized by **Account**

| UML Notation | Nature of the Relationship | Example | | Description |
|--------------|----------------------------|--|--|--|
| 0..* | Zero or many |  | | An employee can have no payroll deductions or many deductions. |
| 0..1 | Zero or one |  | | An employee can have no spouse or one spouse. |
| 1 | One and only one |  | | An office manager manages one and only one office. |
| 1..* | One or many |  | | One order can include one or many items ordered. |

- Association: a usage relationship

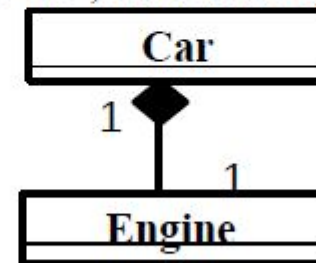
- **aggregation**: “is part of” and it is symbolized by a clear white diamond

- Cars may have passengers



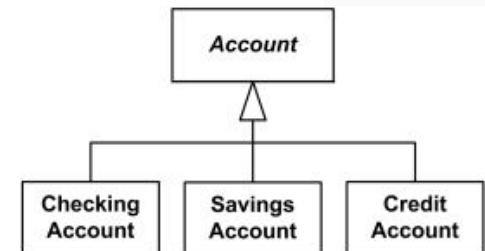
- **composition**: “is entirely made of”, and it is symbolized by a black diamond

- Every car has an engine

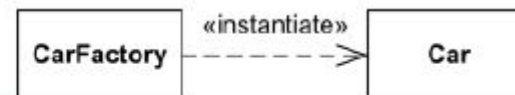


- **generalization** “x is a y” and it is symbolized by triangle shape

- Checking, Savings, and Credit Accounts are generalized by **Account**



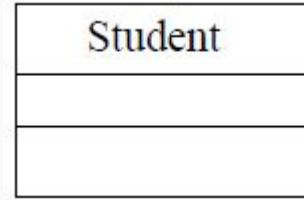
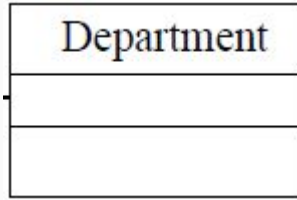
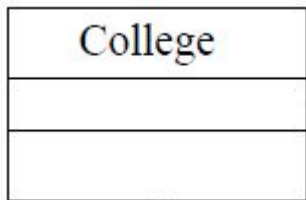
- **Dependencies**: “x uses y” and it is represented it with a dashed directed line



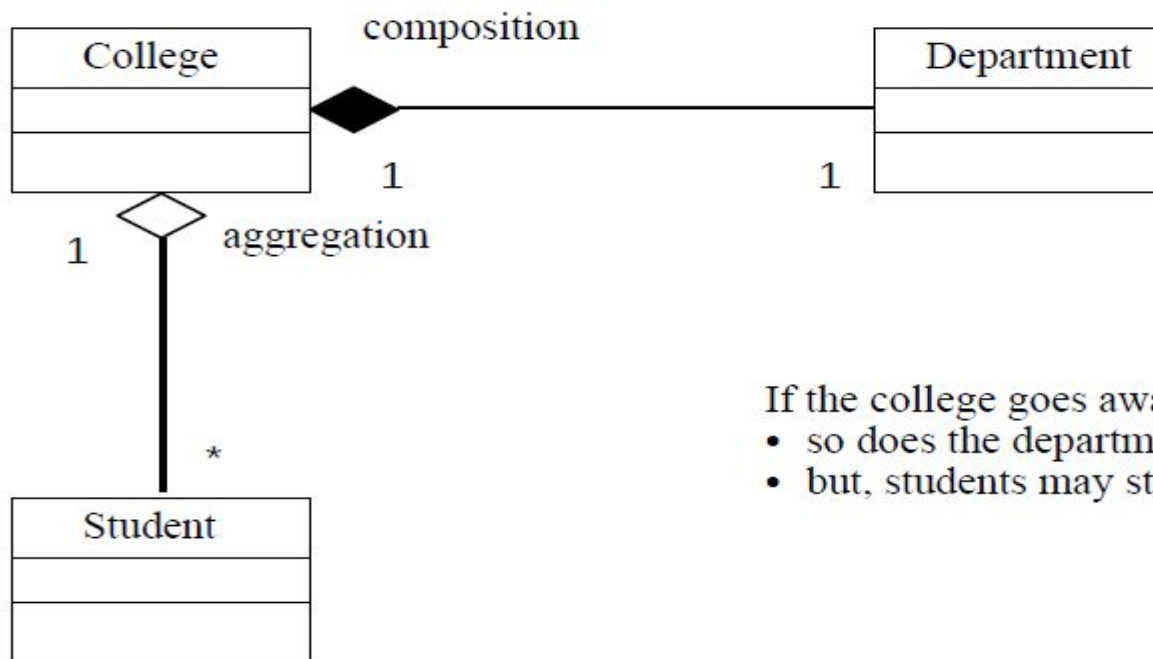
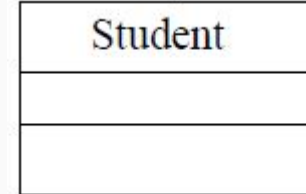
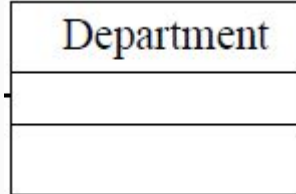
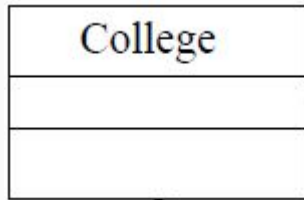
- CarFactory class depends on the Car class.

Composition/aggregation example

- Identify associations and aggregations for the following classes?



Composition/aggregation example



If the college goes away

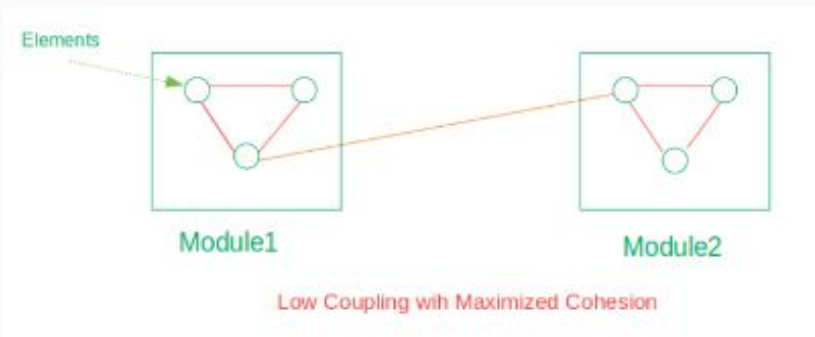
- so does the department → **composition**
- but, students may still exist → **aggregation**

Design Attributes

- The design should be understandable and facilitate change and evolution.
- In service of those two goals, there are some qualities that we should strive for in our design:
 - **Simplicity**: make something understandable
 - **Modularity**: we can break the system down into logically-grouped components that work largely independently of each other
 - Low Coupling
 - High Cohesion
 - Information Hiding
 - Data Encapsulation
 - **Other “abilities”**: can the system be adapted, can we maintain traceability to the requirements
 - Adaptability
 - Traceability

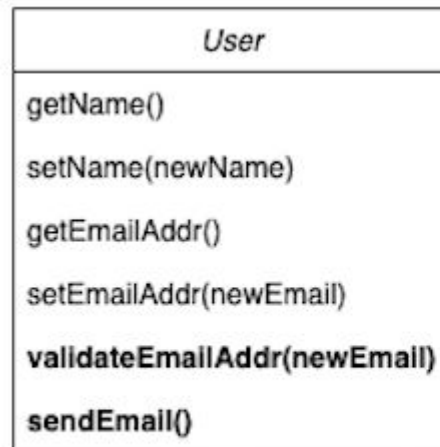
Modularity

- A complex system must be broken down into smaller modules (**modularity**)
- **Modularity** is the degree to which a system's components may be separated.
- **Three goals of modularity:**
 1. **Decomposability:** Break the system down into understandable modules.
 2. **Composability:** Construct a system from smaller pieces.
 3. **Ease of Understanding:** The system will change, we must understand it.
- **Modularity Properties**
 1. **Cohesion** = The degree to which modules are compatible.
 2. **Coupling** = The degree of interdependence between modules.
- We want **high** cohesion and **low** coupling.

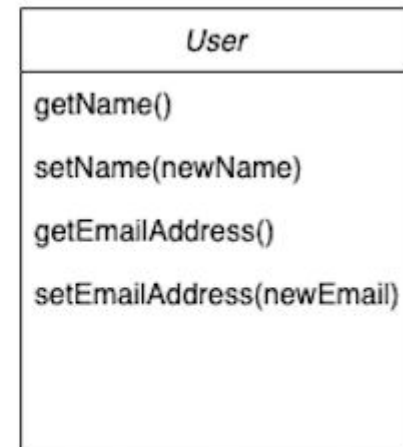


Cohesion

- The degree to which modules are compatible. A measure of how well a component “fits together”.
- A component should implement a single logical entity or feature of the software.
- A high level of cohesion is a desirable design attribute because changes are localized to a single, cohesive component.
- **High** relationships between the operations in a component improve clarity and understanding



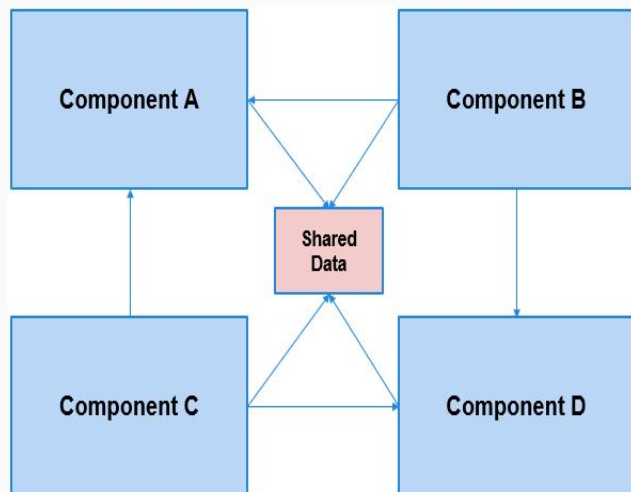
Low Cohesion



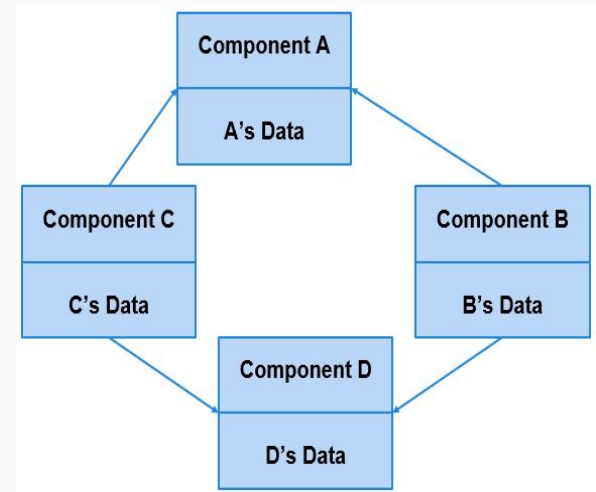
High Cohesion

Coupling

- The degree of interdependence between modules. A measure of the strength of the interconnections between components.
 - Is code from another class called often?
 - How much data is passed during those calls?
- Coupling (loose/low vs. tight/high)
 - If two components are strongly coupled, it is hard to modify one without modifying the other.
 - The more tightly coupled two components are, the harder it is to work with them separately
 - Loose coupling means component changes are unlikely to affect other components.
- Loose coupling can be achieved by storing local data in objects and communicating solely by passing data through component's parameters.

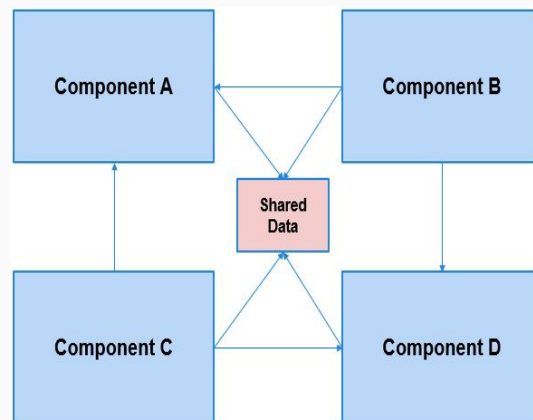


Coupling (Tight VS Loose)



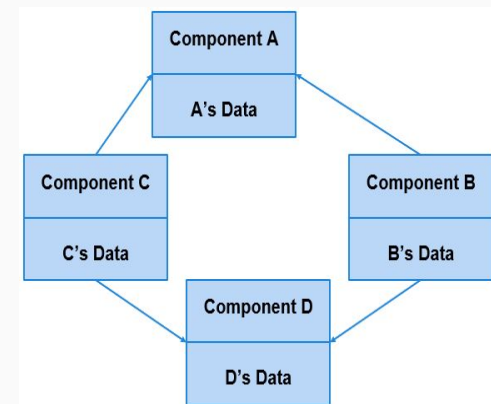
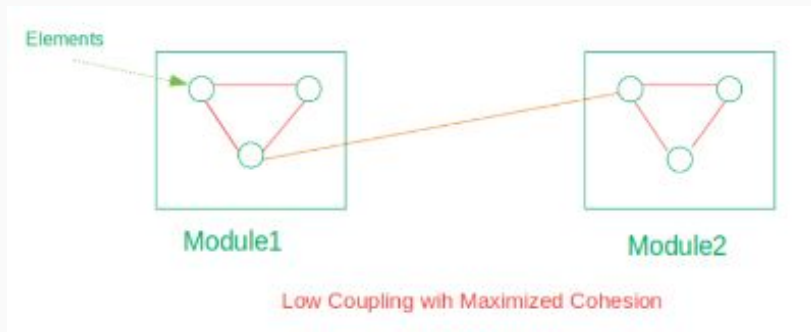
Tight/High Coupling

- If we have tight coupling, we have several components that depend on each other. Commonly, this emerges in two ways
 - either they call functionality of each other - especially if those calls are bidirectional.
 - Or, they all share the same data.
- If you have high/tight coupling, you usually also have low cohesion - elements aren't well grouped, and as a result, one component needs to use too many functions or work with too much data from another component.
 - That dependence is likely to result in more bugs, and more severe bugs, because problems ripple through the system, causing other components to also fail.
 - Fixing bugs requires fixing all of the dependent components instead of just one.
 - It makes it harder to understand how the system works.
 - It makes reuse much more difficult because you can't strip independent units of code out. It's just a pain in the head.



Loose/Low Coupling

- If we want to achieve **loose coupling**
 - we want to ensure that each component stores most of the data that it needs locally. When that component performs operations, it will only manipulate the local data.
 - We want to streamline connections between components, only calling the others if absolutely necessary. If there are problems, they are contained - as much as possible - to this component.
- We can inspect each part of the system in isolation from the rest, inspect both its operations and data, and find the problem unit.
- Most importantly, bug fixes only need to be applied to this single broken component.
- Cohesion plays a role here too - if we have high cohesion, related elements are well grouped, then we are also likely to have looser coupling
 - we don't need to constantly call elements from other components - connections are minimized, and system efficiency will be higher.



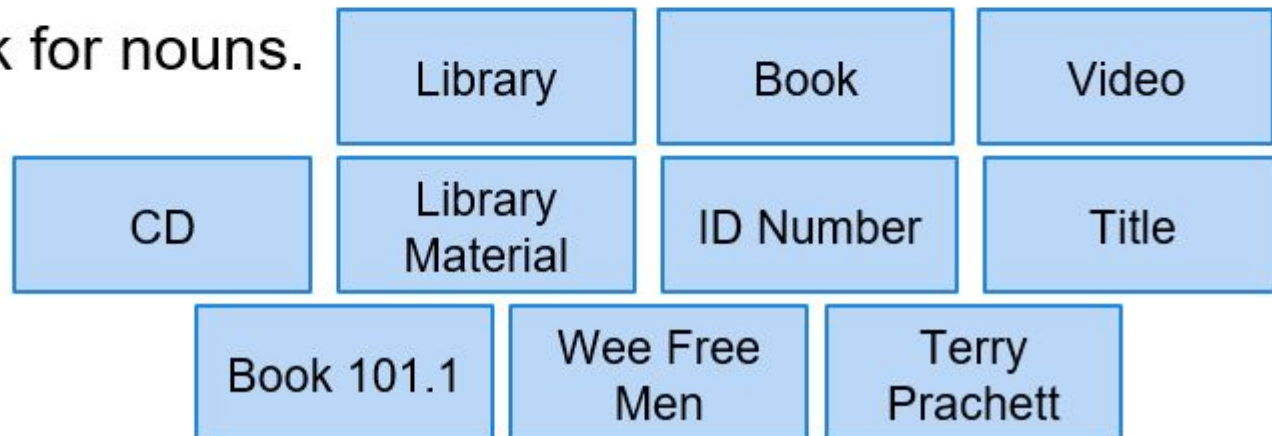
UML Class Diagrams (Example)

A library has books, videos, and CDs that it loans to its users. All library material has an ID number and a title. Book 101.1 is The Wee Free Men by Terry Prachett.

An Approach for Class Diagram (Objects) Modeling

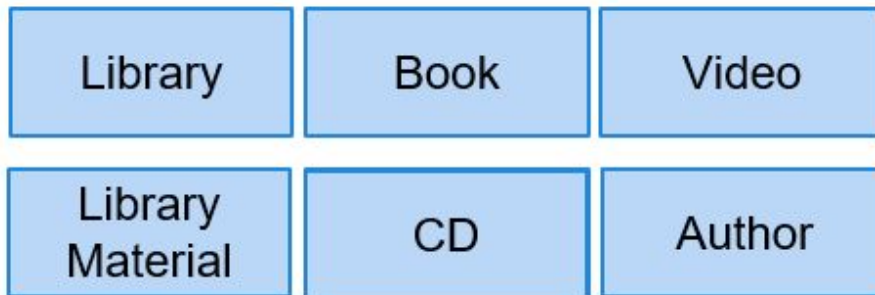
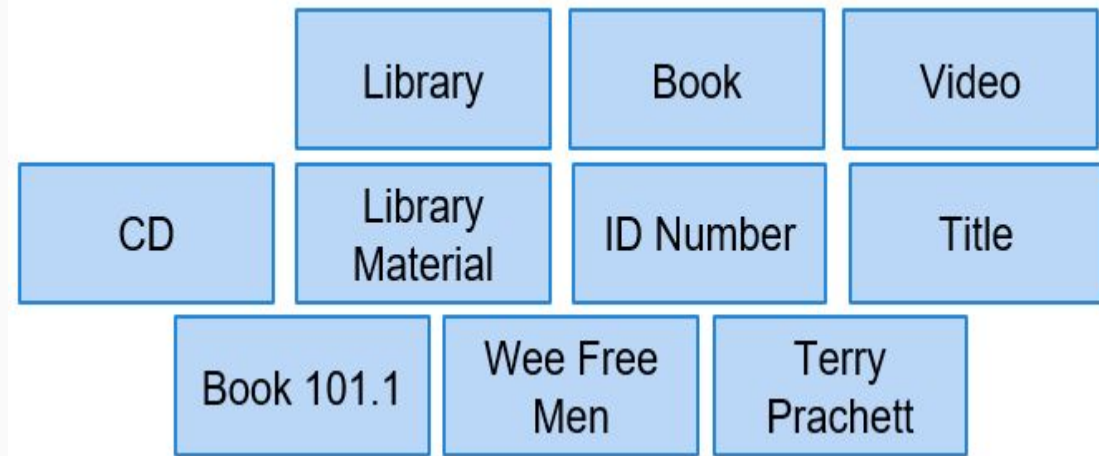
- Start with a problem statement.
 - High-level requirements
- Identify potential objects.
 - Look for nouns.

A library has books, videos, and CDs that it loans to its users. All library material has an ID number and a title. Book 101.1 is The Wee Free Men by Terry Prachett.



UML Class Diagrams

- Refine and remove bad classes
 - Redundant, vague, or irrelevant.
 - Abstract objects to classes.
- Prepare data dictionary
 - Describe each class and its purpose.



Library Material: An abstract class representing a generic library item that can be checked out. Has an ID and title.

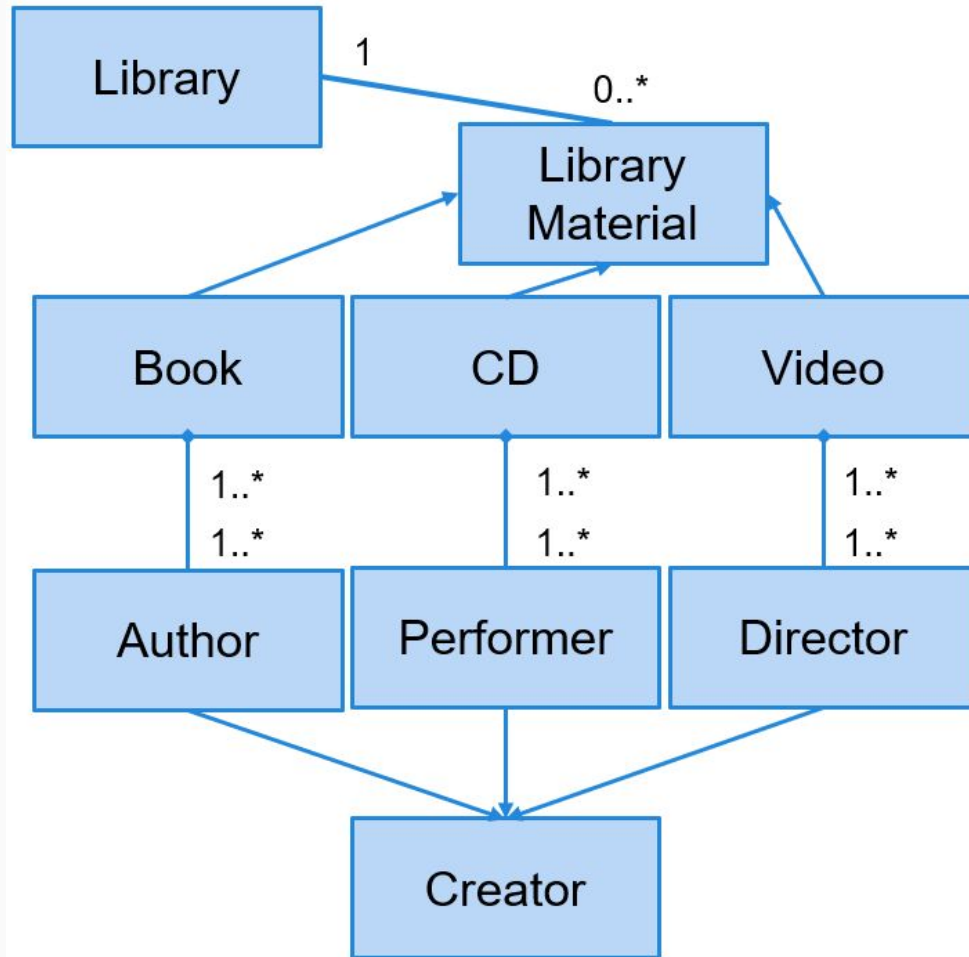
Book: A class representing a book that can be checked out. Has an author in addition to inherited attributes ID and title.

UML Class Diagrams

- Identify associations and aggregations.
- Identify the attributes and operations of classes.
- Organize and simplify using inheritance.

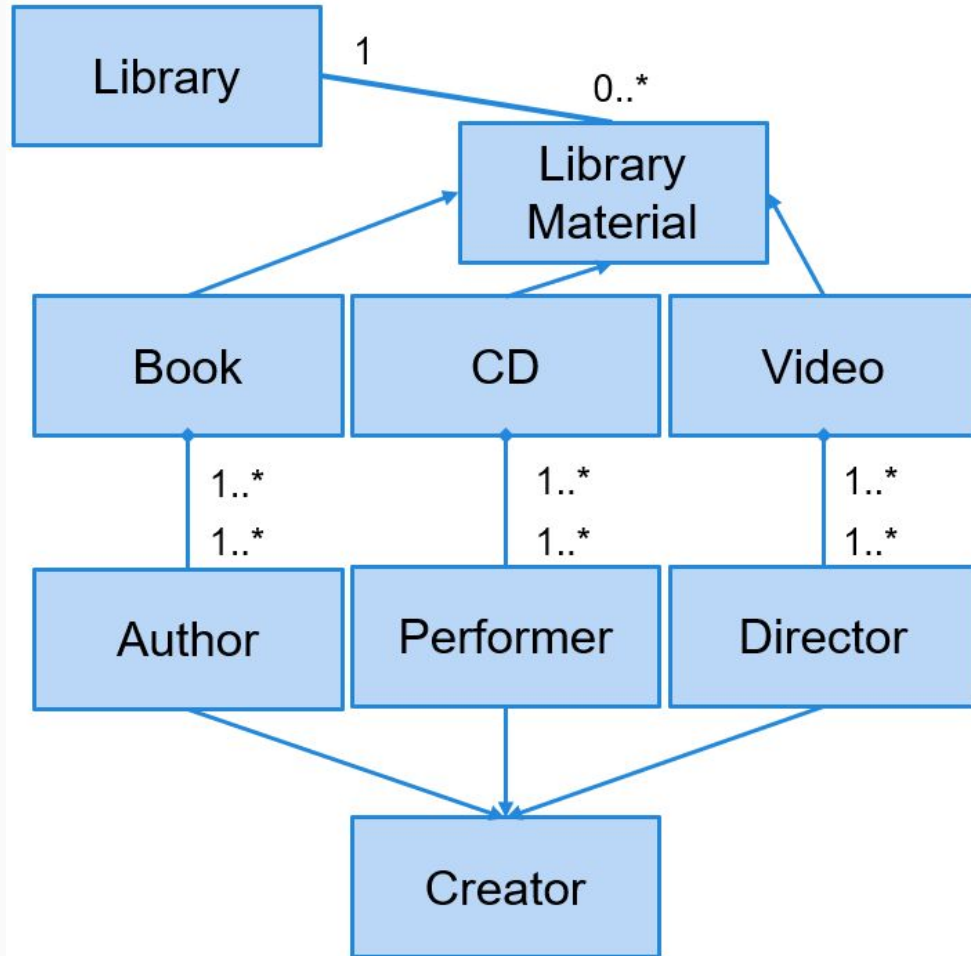
General guidelines:

- Responsibilities should be evenly distributed between classes.
- Information related to a responsibility should be stored in the class responsible for that service.
 - Those are the attributes.



What about Attributes and Operations?

- Now, we have a class diagram, but we're missing some information.
- What are some attributes we can add to these classes to make this system work?
- What are the responsibilities of the class? What does the class do? Those are its operations.



Define Attributes and Operations

- What are the *responsibilities* of the class?
 - Use tools such as data dictionaries to define responsibilities of a class - what services must they perform or allow others to perform.
 - Classes were nouns, now look for **verbs**.
- General guidelines:
 - Responsibilities should be evenly distributed between classes.
 - Information related to a responsibility should be stored in the class responsible for that service.
 - Those are the attributes.

Identify Associations

Classes fulfill responsibilities in two ways:

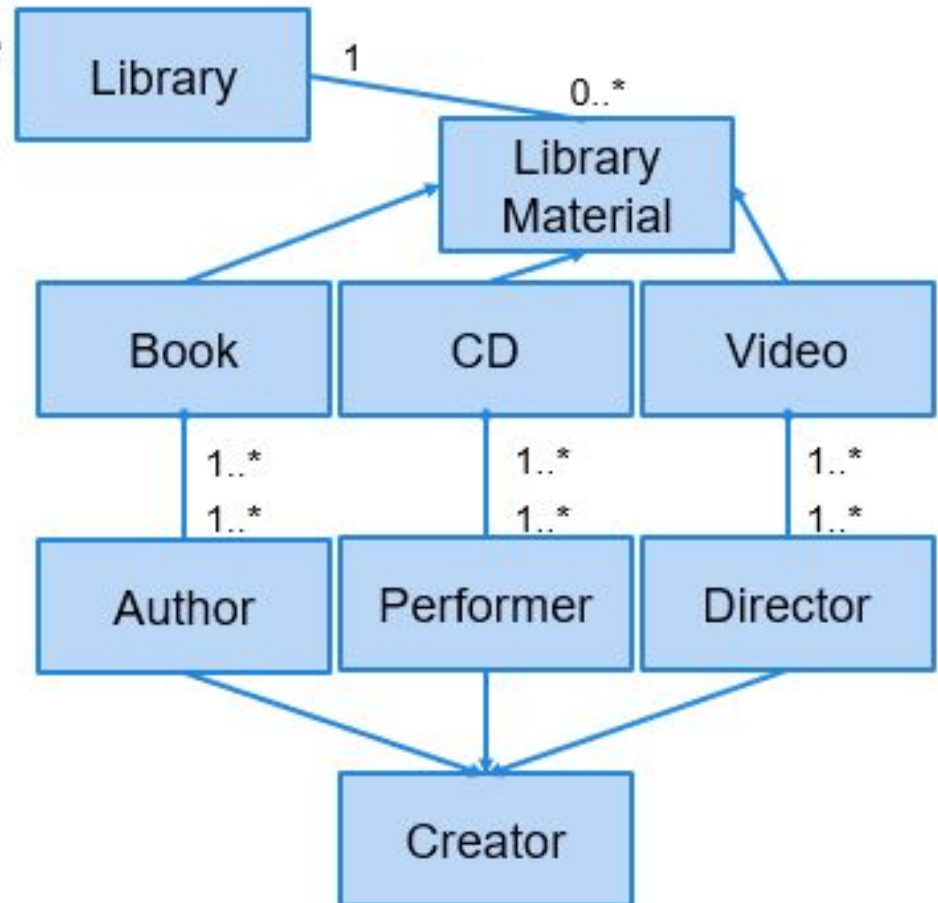
- It can use its own methods to modify its own attributes.
- It can collaborate with other classes.

If a class cannot fulfill its responsibilities alone, identify and document the associations.

- is-part-of (aggregation)
- has-knowledge-of (association)
- depends-upon (association)

UML Class Diagrams

- Iterate and refine the model.
 - You will almost always go through multiple iterations of a design.
- Group classes into subsystems.
 - Which classes can combine to form an independent grouping?



Iterate the Model

Keep on **iterating** until you, your customers, and your engineers are happy with the design.

A software with less coupled and high cohesive module design should always be preferred. A software design breaks down into multiple modules where each module solves a particular problem.

These modules are structured in a proper hierarchy. **Each and every module must be implemented in a way that they are having less dependency with other modules and the elements within that module should be functionally related together.**