Object-Oriented Analysis & Design

Object-Oriented Programming

Author: Eng. Carlos Andrés Sierra, M.Sc.
cavirguezs@udistrital.edu.co

Lecturer Professor
Department of Computer Engineering
School of Engineering
Universidad Distrital Francisco José de Caldas

2025-III

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

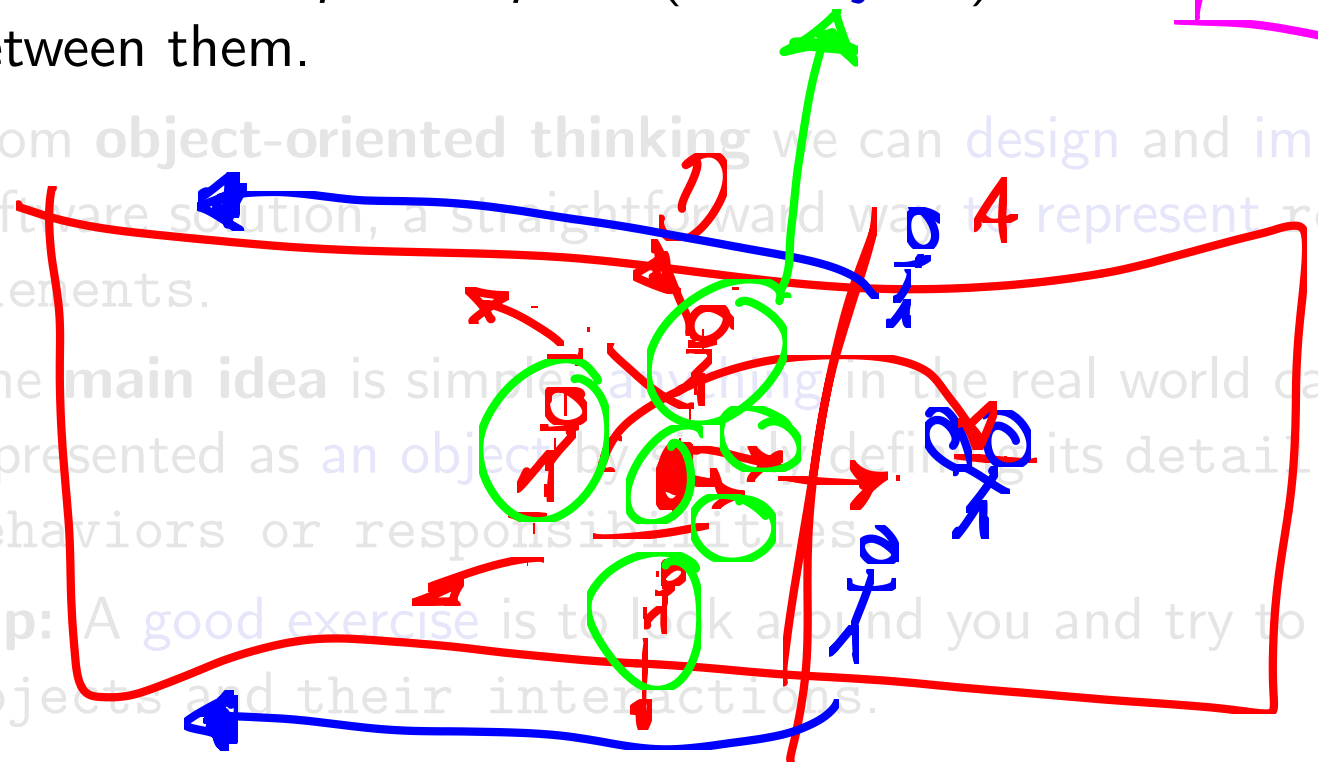
Object-Oriented

- **Object-oriented** is a way of thinking about problems. It is not just a way of programming.
- **Object-oriented thinking** involves analyzing a problem, breaking it down into *component parts* (i.e., objects) and the interactions between them.
- From **object-oriented thinking** we can design and implement a software solution, a straightforward way to represent `real-world elements`.
- The **main idea** is simple: anything in the real world can be represented as an object by simply defining its `details` and `behaviors or responsibilities`.
- **Tip:** A good exercise is to look around you and try to identify `objects and their interactions`.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Object-Oriented

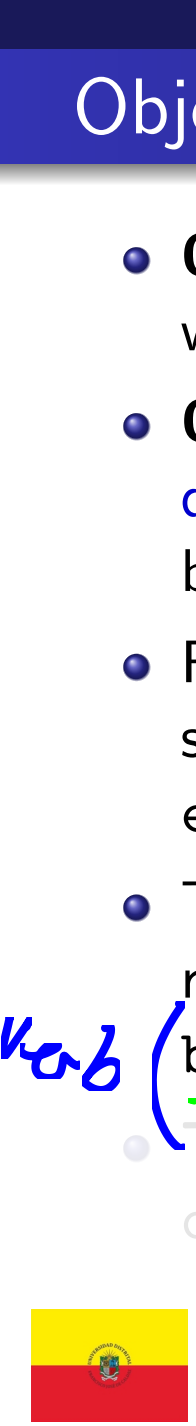
- **Object-oriented** is a way of thinking about problems. It is not just a way of programming.
- **Object-oriented thinking** involves analyzing a problem, breaking it down into *component parts* (i.e., objects) and the interactions between them.



- From **object-oriented thinking** we can design and implement a software solution, a straightforward way to represent real-world elements.
- The **main idea** is simple: anything in the real world can be represented as an object by only defining its details and behaviors or responsibilities.
- **Tip:** A good exercise is to look around you and try to identify objects and their interactions.

Object-Oriented

- **Object-oriented** is a way of thinking about problems. It is not just a way of programming.
- **Object-oriented thinking** involves analyzing a problem, breaking it down into *component parts* (i.e., objects) and the interactions between them.
- From **object-oriented thinking** we can design and implement a software solution, a straightforward way to represent `real-world elements`.
- The **main idea** is simple: anything in the real world can be represented as an object by simply defining its `details and behaviors or responsibilities`.
- **Tip:** A good exercise is to look around you and try to identify objects and their interactions.



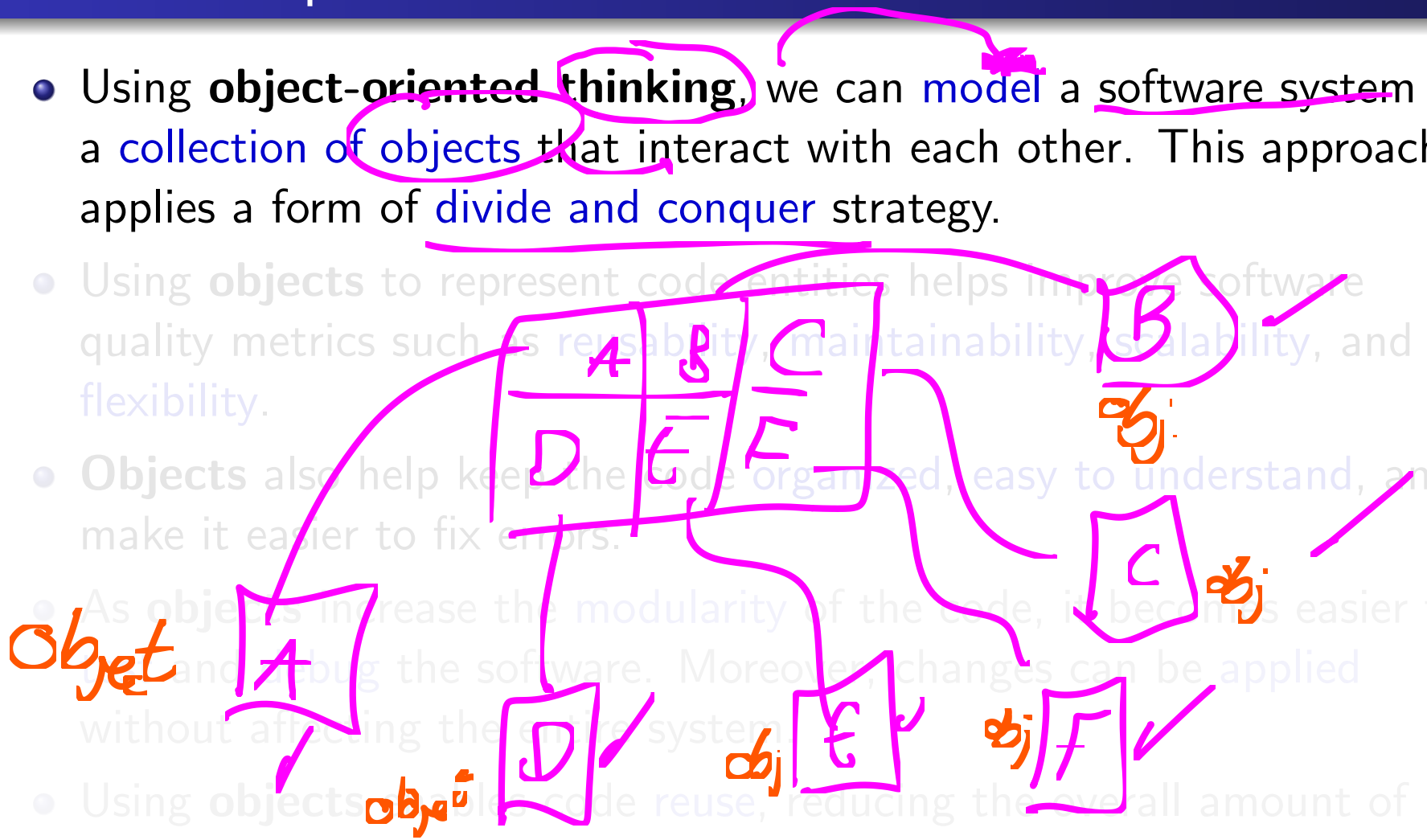
Object-Oriented

- **Object-oriented** is a way of thinking about problems. It is not just a way of programming.
- **Object-oriented thinking** involves analyzing a problem, breaking it down into *component parts* (i.e., objects) and the interactions between them.
- From **object-oriented thinking** we can design and implement a software solution, a straightforward way to represent `real-world elements`.
- The **main idea** is simple: anything in the real world can be represented as an object by simply defining its `details` and `behaviors or responsibilities`.
- **Tip:** A good exercise is to look around you and try to identify `objects` and `their interactions`.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Software Implications

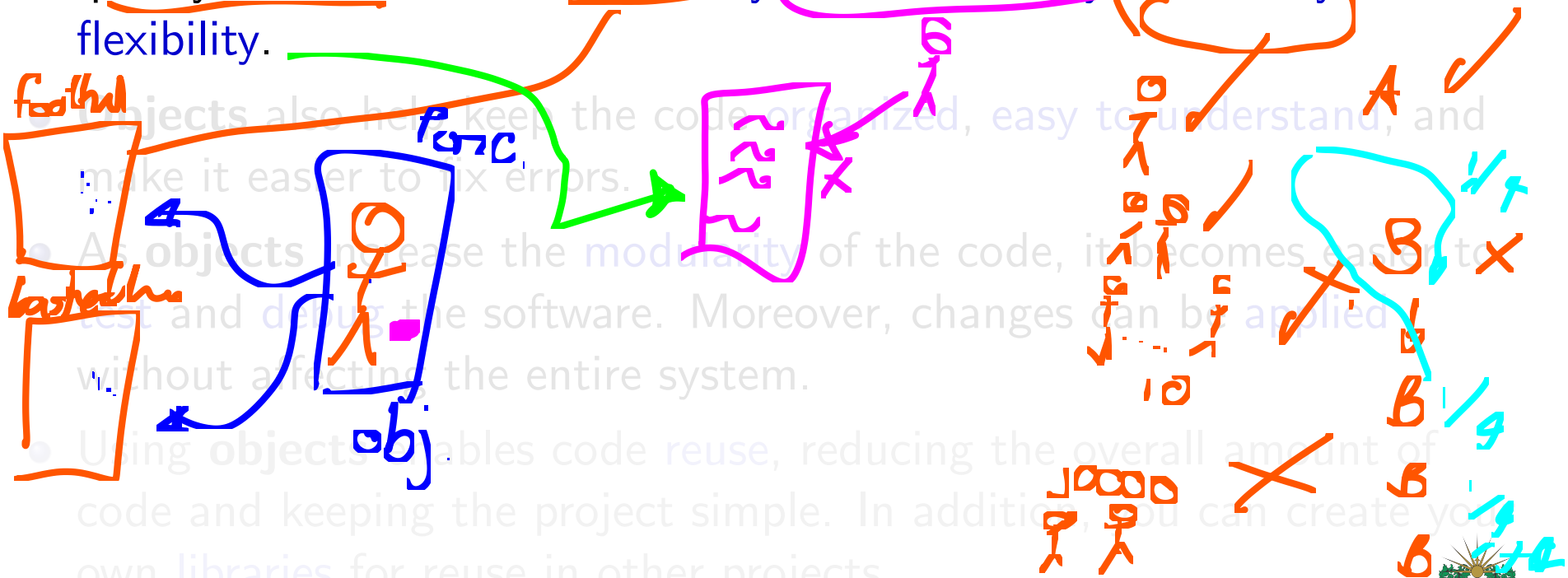
- Using **object-oriented thinking**, we can model a software system as a collection of objects that interact with each other. This approach applies a form of divide and conquer strategy.
- Using **objects** to represent code entities helps improve software quality metrics such as reusability, maintainability, scalability, and flexibility.
- **Objects** also help keep the code organized, easy to understand, and make it easier to fix errors.
- As **objects** increase the modularity of the code, it becomes easier to expand and debug the software. Moreover, changes can be applied without affecting the entire system.
- Using **objects** fosters code reuse, reducing the overall amount of code and keeping the project simple. In addition, you can create your own libraries for reuse in other projects.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Software Implications

- Using **object-oriented thinking**, we can model a software system as a collection of objects that interact with each other. This approach applies a form of divide and conquer strategy.
- Using **objects** to represent code entities helps improve software quality metrics such as reusability, maintainability, scalability, and flexibility.



- **Objects** also help keep the code organized, easy to understand, and make it easier to fix errors.
- As **objects** increase the modularity of the code, it becomes easier to test and debug the software. Moreover, changes can be applied without affecting the entire system.
- Using **objects** enables code reuse, reducing the overall amount of code and keeping the project simple. In addition, you can create your own libraries for reuse in other projects.

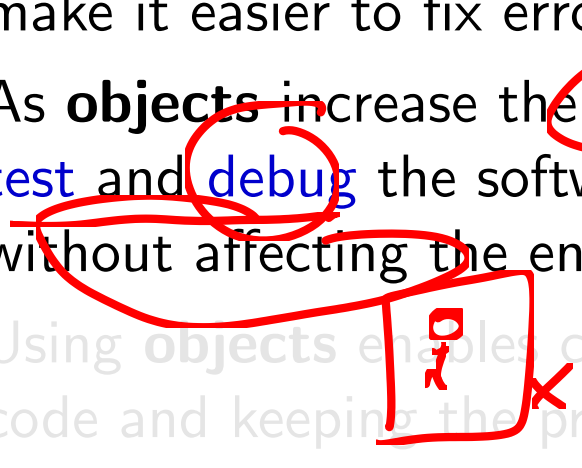
Software Implications

- Using **object-oriented thinking**, we can model a software system as a collection of objects that interact with each other. This approach applies a form of divide and conquer strategy.
- Using **objects** to represent code entities helps improve software quality metrics such as reusability, maintainability, scalability, and flexibility.
- **Objects** also help keep the code organized, easy to understand, and make it easier to fix errors.
- As **objects** increase the modularity of the code, it becomes easier to test and debug the software. Moreover, changes can be applied without affecting the entire system.
- Using **objects** enables code reuse, reducing the overall amount of code and keeping the project simple. In addition, you can create your own libraries for reuse in other projects.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Software Implications

- Using **object-oriented thinking**, we can model a software system as a collection of objects that interact with each other. This approach applies a form of divide and conquer strategy.
- Using **objects** to represent code entities helps improve software quality metrics such as reusability, maintainability, scalability, and flexibility.
- **Objects** also help keep the code organized, easy to understand, and make it easier to fix errors.
- As **objects** increase the modularity of the code, it becomes easier to test and debug the software. Moreover, changes can be applied without affecting the entire system.
- Using **objects** enables code reuse, reducing the overall amount of code and keeping the project simple. In addition, you can create your own libraries for reuse in other projects.



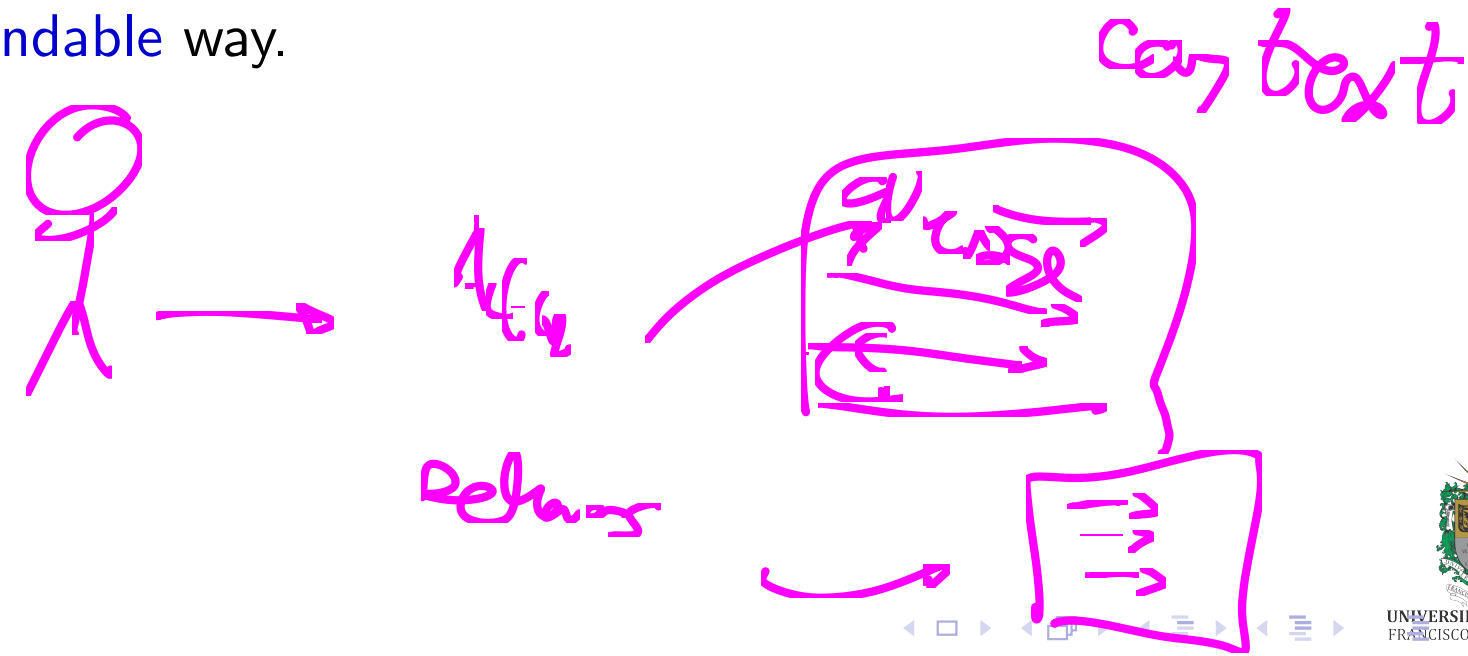
Software Implications

- Using **object-oriented thinking**, we can model a software system as a collection of objects that interact with each other. This approach applies a form of divide and conquer strategy.
- Using **objects** to represent code entities helps improve software quality metrics such as reusability, maintainability, scalability, and flexibility.
- **Objects** also help keep the code organized, easy to understand, and make it easier to fix errors.
- As **objects** increase the modularity of the code, it becomes easier to test and debug the software. Moreover, changes can be applied without affecting the entire system.
- Using **objects** enables code reuse, reducing the overall amount of code and keeping the project simple. In addition, you can create your own libraries for reuse in other projects.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

What is abstraction?

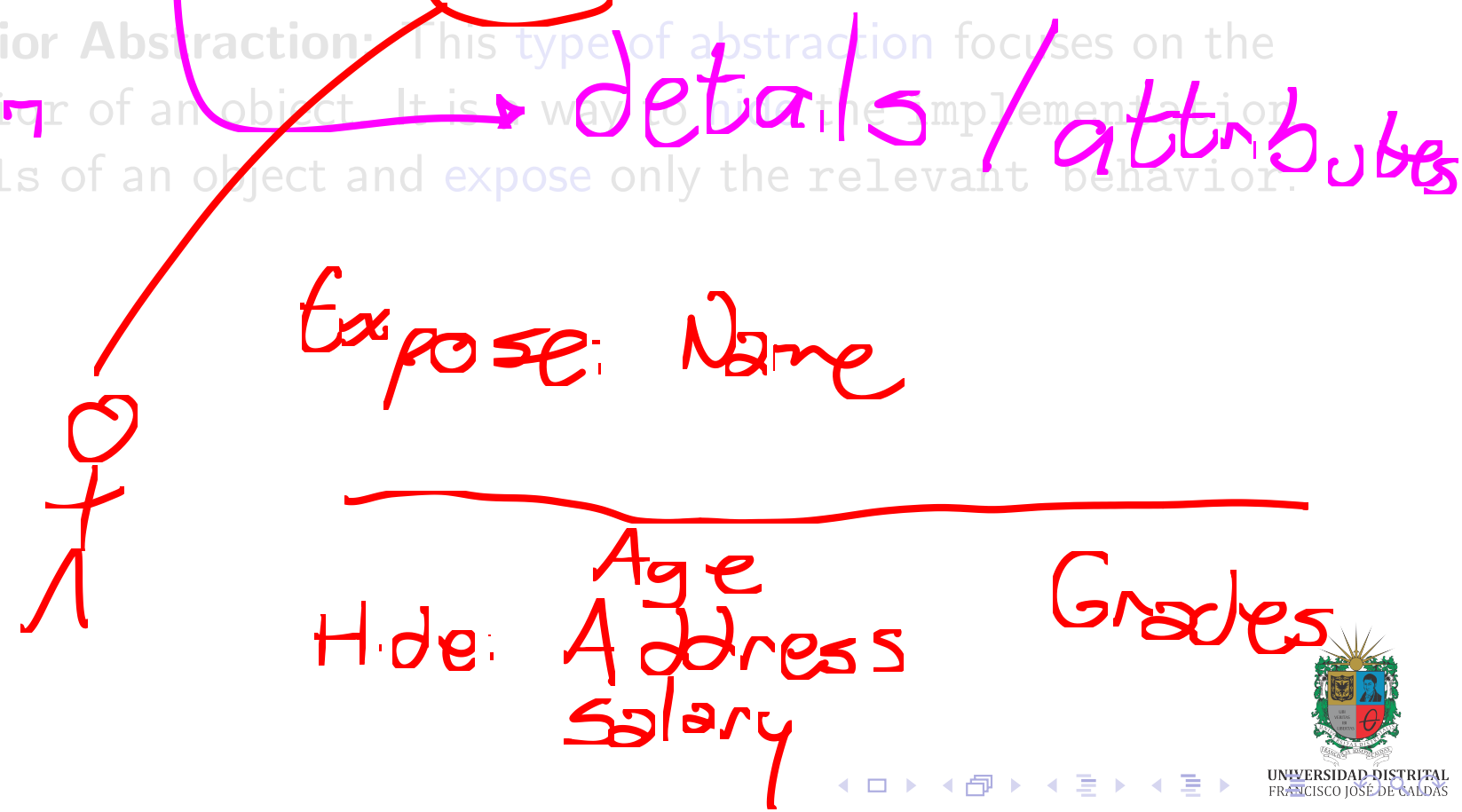
- **Abstraction** is the process of filtering out the `characteristics` of an object that we are interested in, and ignoring the rest.
- **Abstraction** is a way to simplify the complexity of the real world by focusing on the `relevant` parts.
- **Abstraction** is a way to represent the `essential` features of an object, hiding the `unnecessary` details.
- **Abstraction** is a way to model the `real world` in a simple and understandable way.



Abstraction Schemas

There are two types of **abstraction schemas**:

- **Data Abstraction:** This type of abstraction focuses on the `data` that an object contains. It is a way to hide the `implementation details` of an object and expose only the `relevant data`.
- **Behavior Abstraction:** This type of abstraction focuses on the `behavior` of an object. It is a way to hide the `implementation details` of an object and expose only the `relevant behavior`.

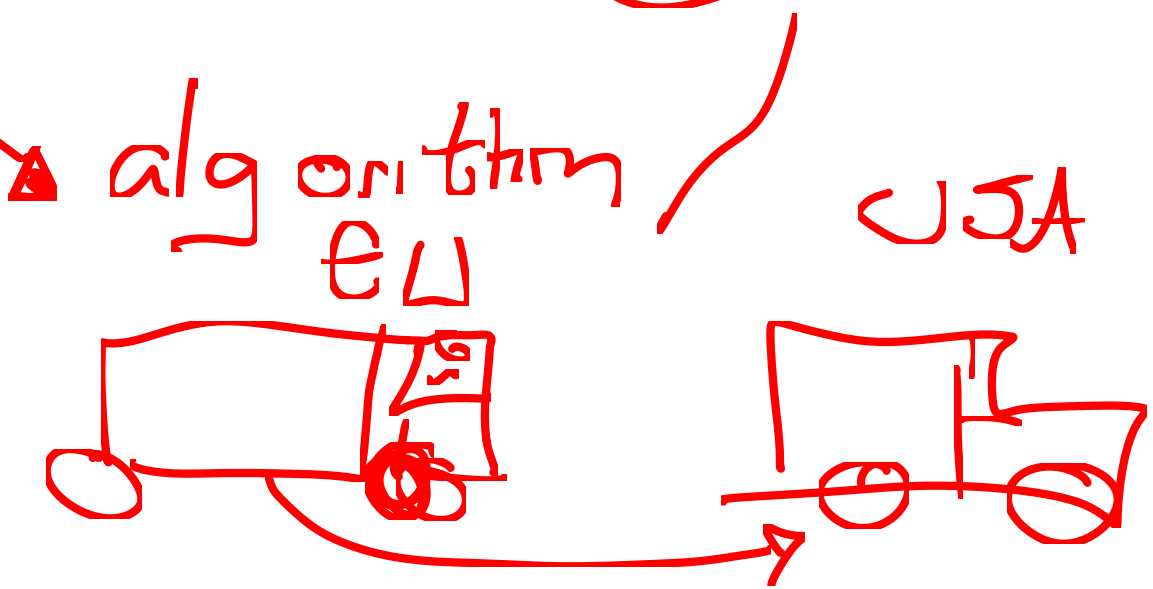


UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Abstraction Schemas

There are two types of **abstraction schemas**:

- **Data Abstraction:** This type of abstraction focuses on the `data` that an object contains. It is a way to hide the `implementation details` of an object and expose only the `relevant data`.
- **Behavior Abstraction:** This type of abstraction focuses on the `behavior` of an object. It is a way to hide the `implementation details` of an object and expose only the `relevant behavior`.

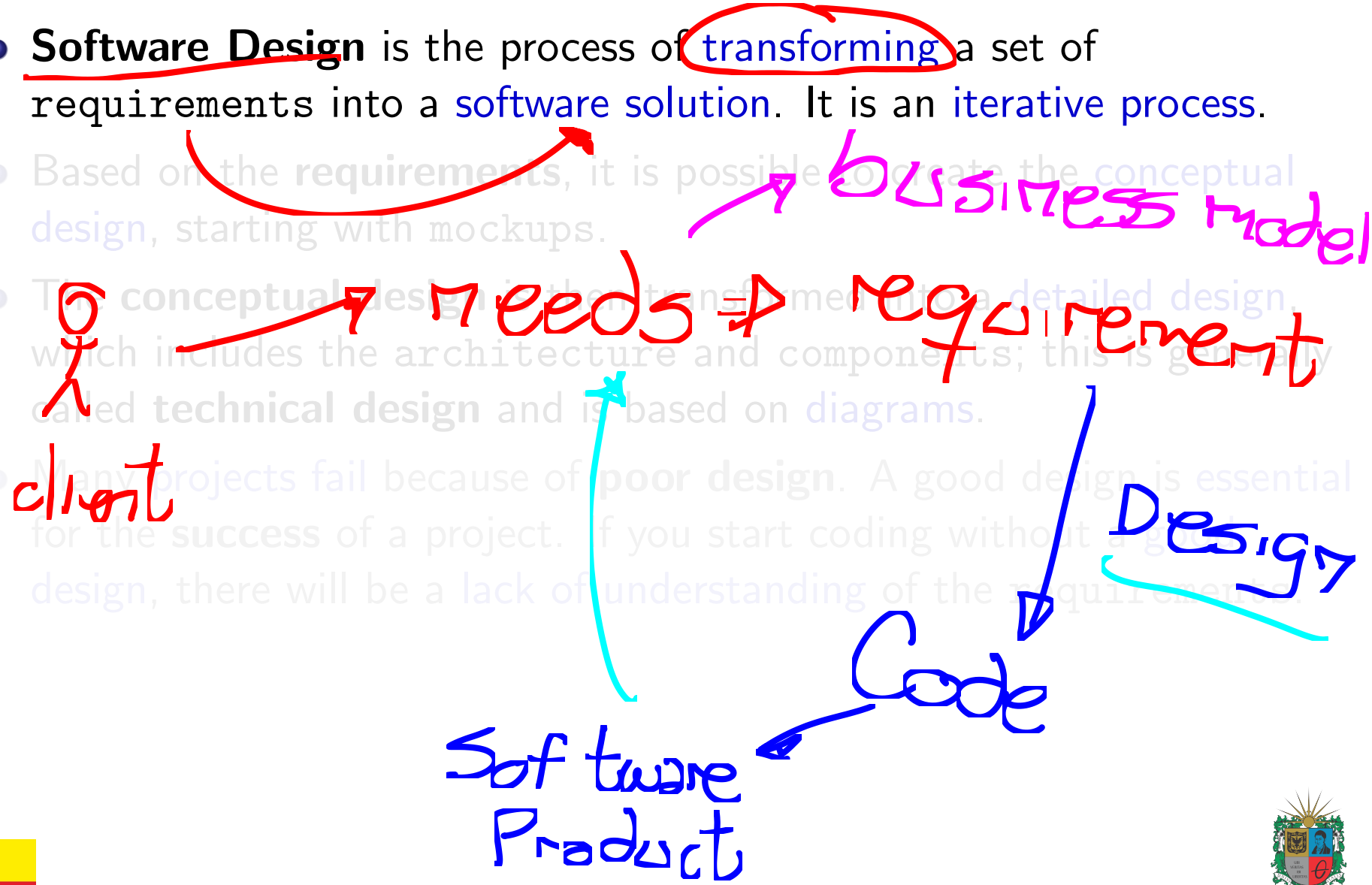


Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Software Design

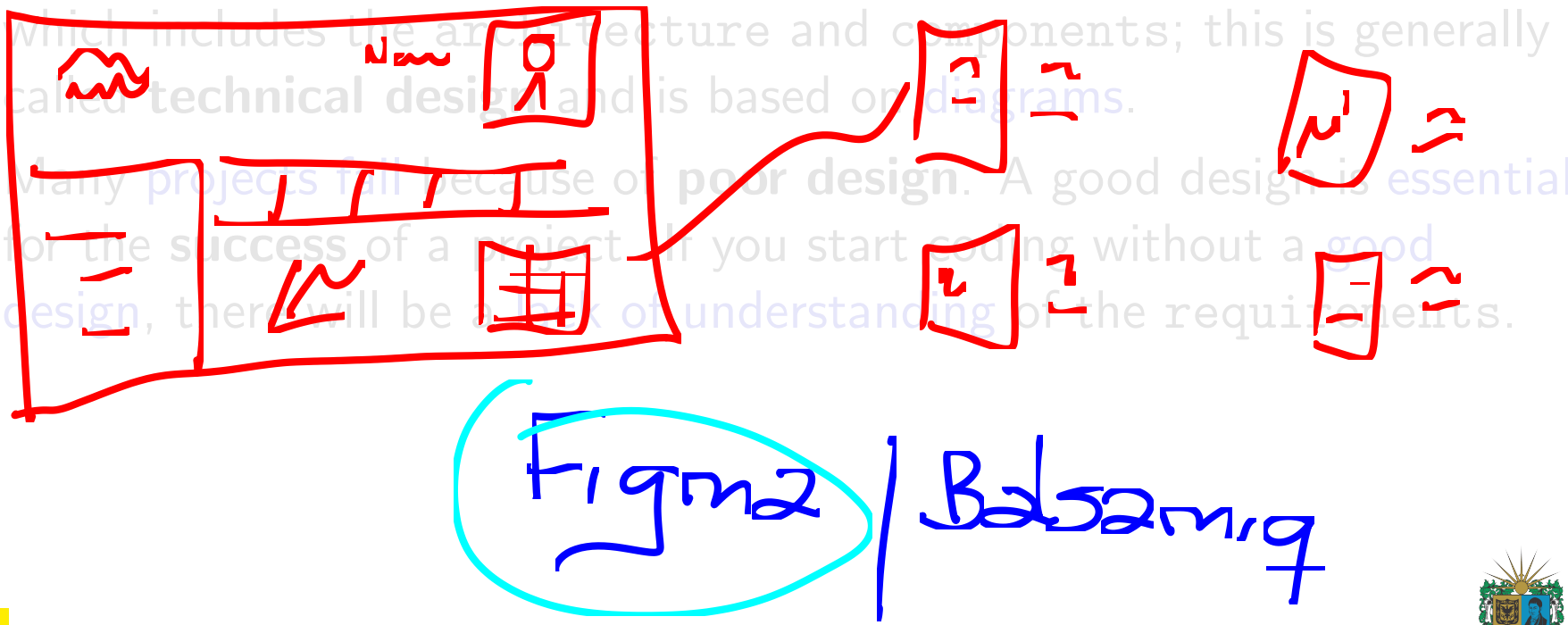
- **Software Design** is the process of transforming a set of `requirements` into a software solution. It is an iterative process.
- Based on the **requirements**, it is possible to create the conceptual design, starting with `mockups`.
- The **conceptual design** is then transformed into a detailed design, which includes the `architecture` and `components`; this is generally called **technical design** and is based on diagrams.
- Many projects fail because of **poor design**. A good design is essential for the **success** of a project. If you start coding without a good design, there will be a lack of understanding of the requirements.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

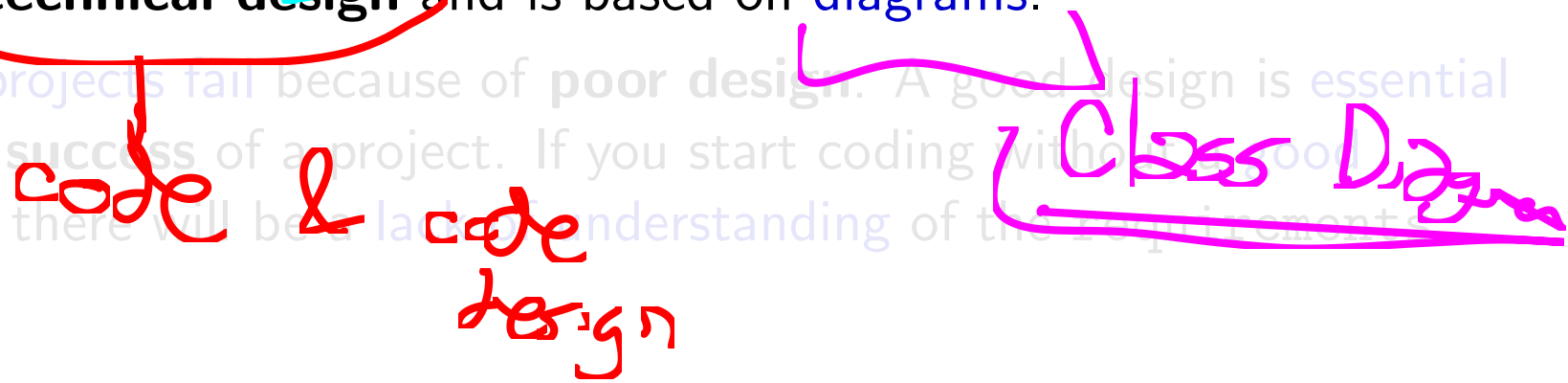
Software Design

- **Software Design** is the process of transforming a set of `requirements` into a software solution. It is an iterative process.
- Based on the **requirements**, it is possible to create the conceptual design, starting with `mockups`.
- The **conceptual design** is then transformed into a detailed design, which includes the architecture and components; this is generally called **technical design** and is based on diagrams.
- Many projects fail because of **poor design**. A good design is essential for the **success** of a project. If you start coding without a good design, there will be a lack of understanding of the requirements.



Software Design

- **Software Design** is the process of transforming a set of `requirements` into a software solution. It is an iterative process.
- Based on the **requirements**, it is possible to create the conceptual design, starting with `mockups`.
- The **conceptual design** is then transformed into a detailed design, which includes the `architecture` and `components`; this is generally called **technical design** and is based on diagrams.
- Many projects fail because of **poor design**. A good design is essential for the **success** of a project. If you start coding without a good design, there will be a lack of understanding of the requirements.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Software Design

- **Software Design** is the process of transforming a set of `requirements` into a software solution. It is an iterative process.
- Based on the **requirements**, it is possible to create the conceptual design, starting with `mockups`.
- The **conceptual design** is then transformed into a detailed design, which includes the `architecture` and `components`; this is generally called **technical design** and is based on diagrams.
- Many projects fail because of **poor design**. A good design is essential for the **success** of a project. If you start coding without a good design, there will be a lack of understanding of the `requirements`.

Requirements

Definition

Requirements are conditions or capabilities that must be implemented in a `software product`.

It is important to think like a software architect: consider both the structure and the behavior of the software.

- **Requirements** form the foundation of a software project. They define what the software should do and what the clients want (i.e., the `scope`).
- **Elicit Requirements** is the process of gathering (i.e., asking the right questions) and documenting the `needs` of the clients.
- **Functional Requirements** are the features that the software should have. They define what the software should do.
- **Non-Functional Requirements** are the qualities that the software should exhibit. They define how the software should operate.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Requirements

Definition

Requirements are conditions or capabilities that must be implemented in a `software product`.

It is important to think like a software architect: consider both the structure and the behavior of the software.

- **Requirements** form the foundation of a software project. They define what the software should do and what the clients want (i.e., the `scope`).
- **Elicit Requirements** is the process of gathering (i.e., asking the right questions) and documenting the `needs` of the clients.
- **Functional Requirements** are the features that the software should have. They define what the software should do.
- **Non-Functional Requirements** are the qualities that the software should exhibit. They define how the software should operate.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Requirements

Definition

Requirements are conditions or capabilities that must be implemented in a `software product`.

It is important to think like a software architect: consider both the structure and the behavior of the software.

- **Requirements** form the foundation of a software project. They define what the software should do and what the clients want (i.e., the `scope`).
- **Elicit Requirements** is the process of gathering (i.e., asking the right questions) and documenting the `needs` of the clients.
- **Functional Requirements** are the features that the software should have. They define what the software should do.
- **Non-Functional Requirements** are the qualities that the software should exhibit. They define how the software should operate.

Requirements

Definition

Requirements are conditions or capabilities that must be implemented in a `software product`.

It is important to think like a software architect: consider both the structure and the behavior of the software.

- **Requirements** form the foundation of a software project. They define what the software should do and what the clients want (i.e., the `scope`).
- **Elicit Requirements** is the process of gathering (i.e., asking the right questions) and documenting the `needs` of the clients.
- **Functional Requirements** are the features that the software should have. They define what the software should do.
- **Non-Functional Requirements** are the qualities that the software should exhibit. They define how the software should operate.

Requirements

Definition

Requirements are conditions or capabilities that must be implemented in a `software product`.

It is important to think like a software architect: consider both the structure and the behavior of the software.

- **Requirements** form the foundation of a software project. They define what the software should do and what the clients want (i.e., the `scope`).
- **Elicit Requirements** is the process of gathering (i.e., asking the right questions) and documenting the `needs` of the clients.
- **Functional Requirements** are the features that the software should have. They define what the software should do.
- **Non-Functional Requirements** are the qualities that the software should exhibit. They define how the software should operate.

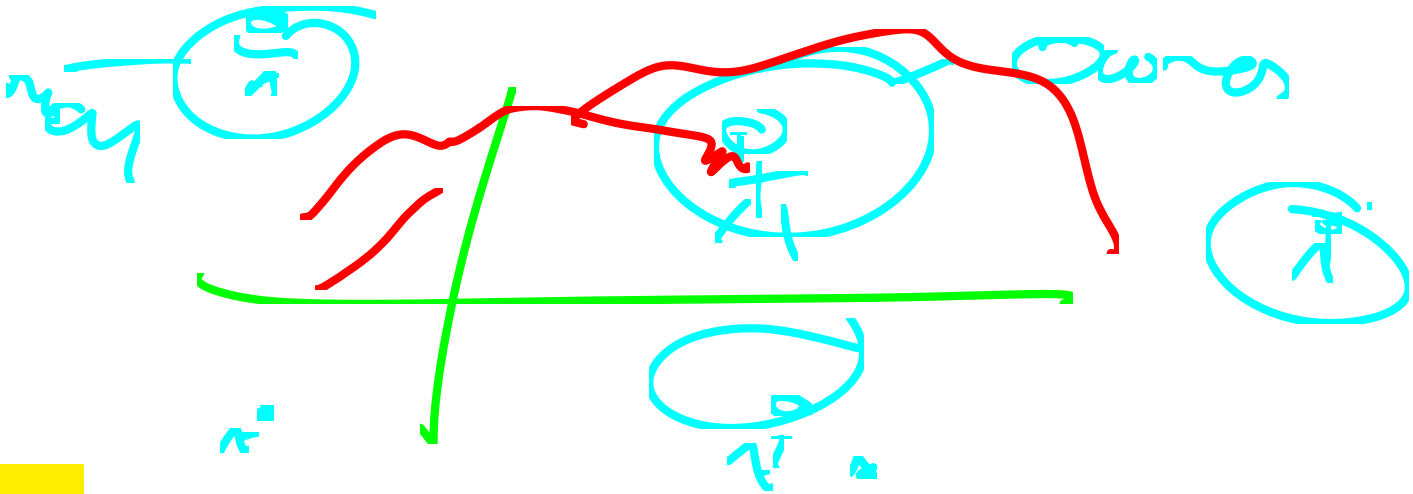
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Conceptual Design

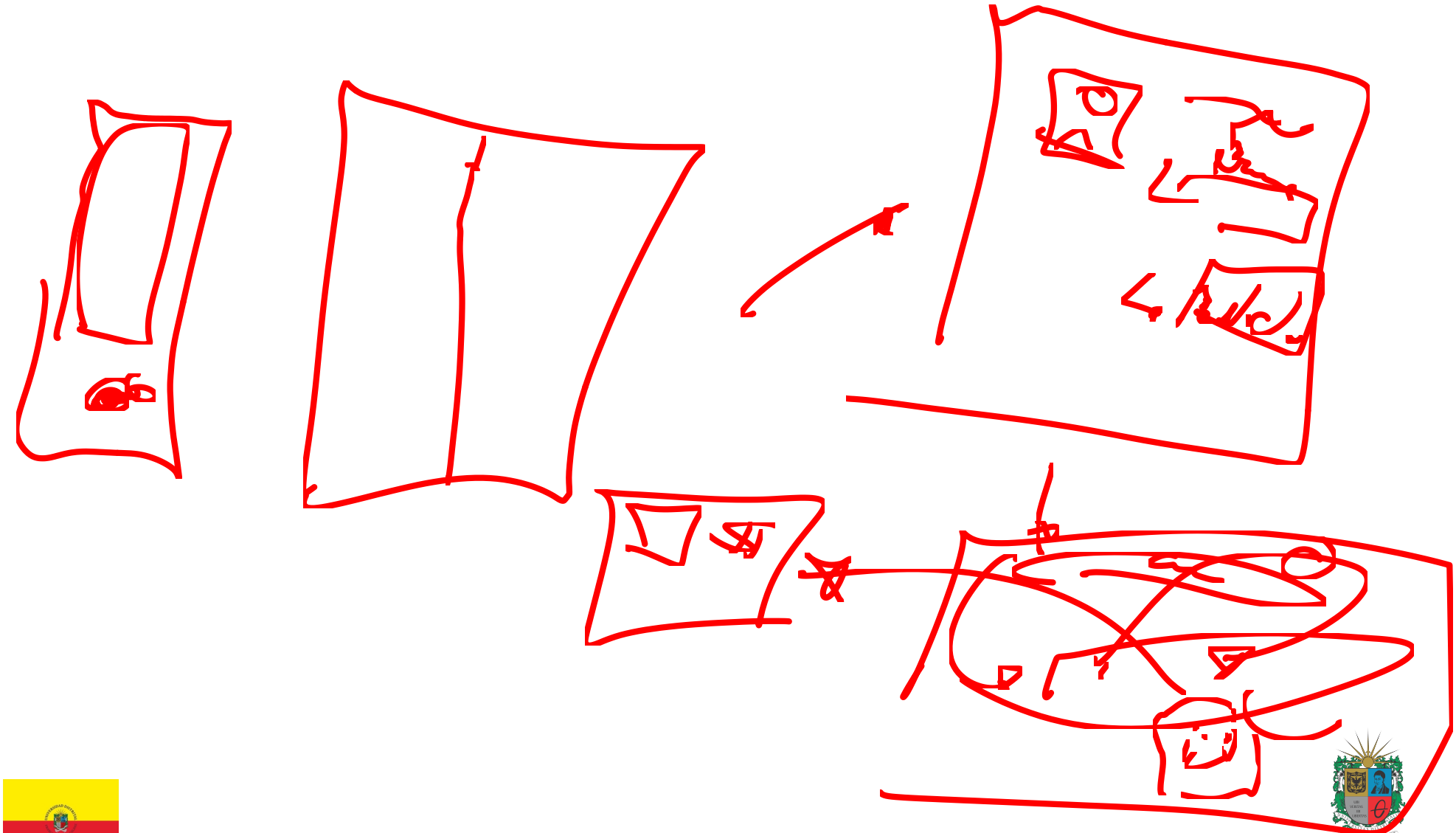
- Once the initial set of requirements are defined, the next step is to create a **conceptual design** of the software.
- **Conceptual Design** is a high-level design that defines the `structure` and `behavior` of the software. It is achieved by the recognition of the appropiate components, connections, and responsibilities.
- Conceptual Design is a visual representation of the software that helps communicate the design (`layout`, `structure`, and `flow`) to the `stakeholders` through `mockups`.

Conceptual Design

- Once the initial set of requirements are defined, the next step is to create a **conceptual design** of the software.
- **Conceptual Design** is a high-level design that defines the `structure` and `behavior` of the software. It is achieved by the recognition of the appropiate components, connections, and responsibilities.
- Conceptual Design is a visual representation of the software that helps communicate the `design` (`layout`, `structure`, and `flow`) to the `stakeholders` through `mockups`.



Mockup Example: *Cell-Phone On-Line Store*

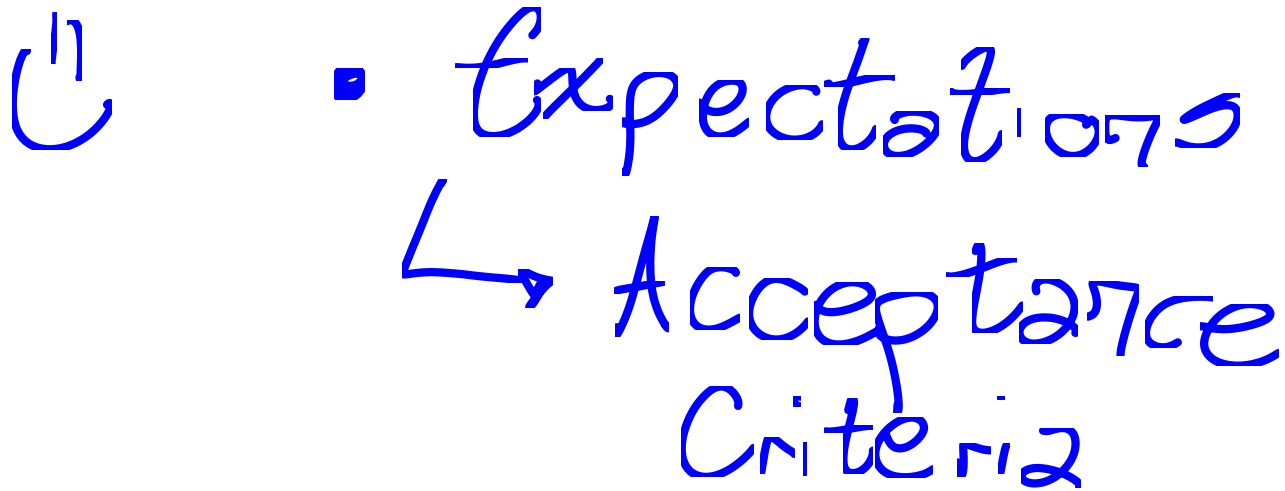


User Stories

- **User stories** are short, simple descriptions of a feature or function of a system.
- They are written from the perspective of the user and describe what the user wants to achieve.
- They are used to capture the requirements of a system in a simple and understandable way.

User Stories

- **User stories** are short, simple descriptions of a feature or function of a system.
- They are written from the perspective of the user and describe what the user wants to achieve.
- They are used to capture the requirements of a system in a simple and understandable way.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

User Stories: Format Example

User Story

Title: Watch sch. schedule	Priority:	Estimate:
User Story: As a [description of user], I want [functionality] so that [benefit].	As a student, I want watch my schedule, so that I can attend to my courses.	
Acceptance Criteria: Given [how things begin] When [action taken] Then [outcome of taking action]	Given the enrolling out dates when I do click in schedule button Then my schedule should be watching as a table	

role

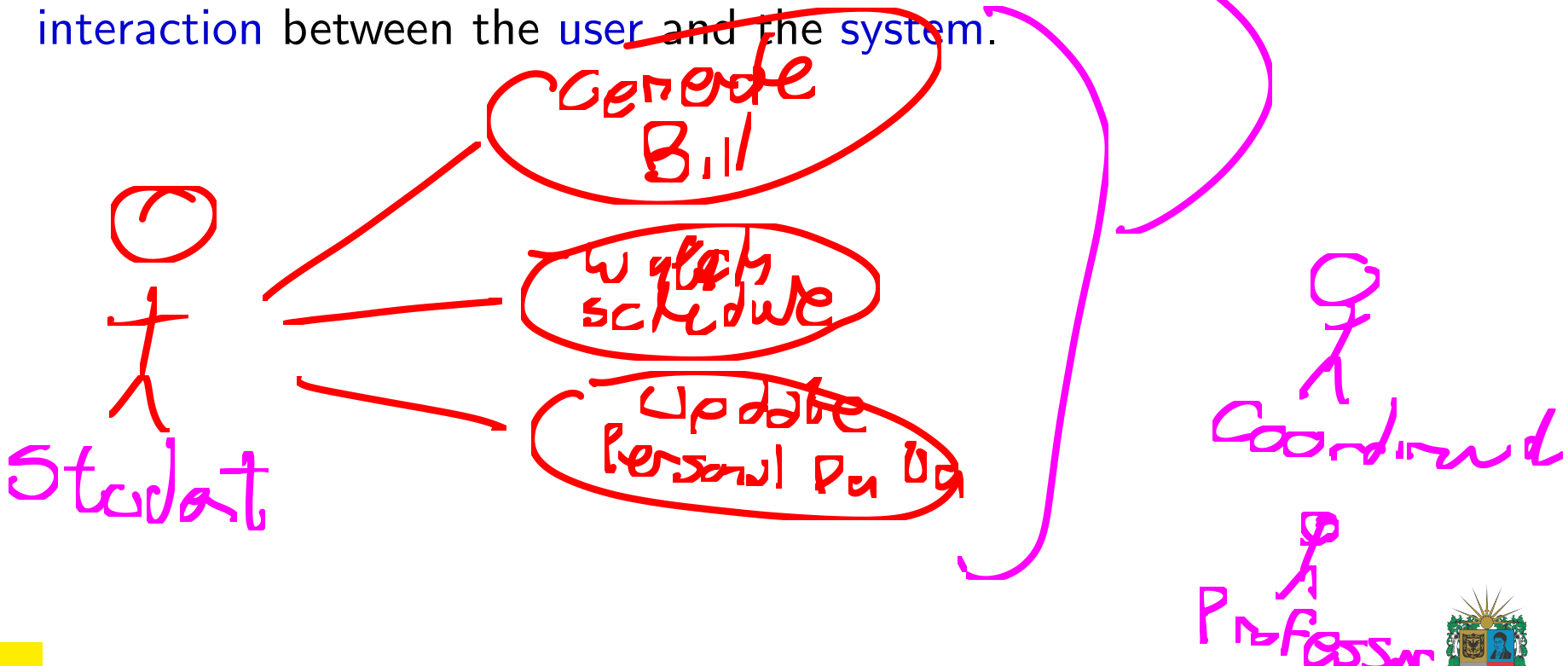
action

ProductPlan

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

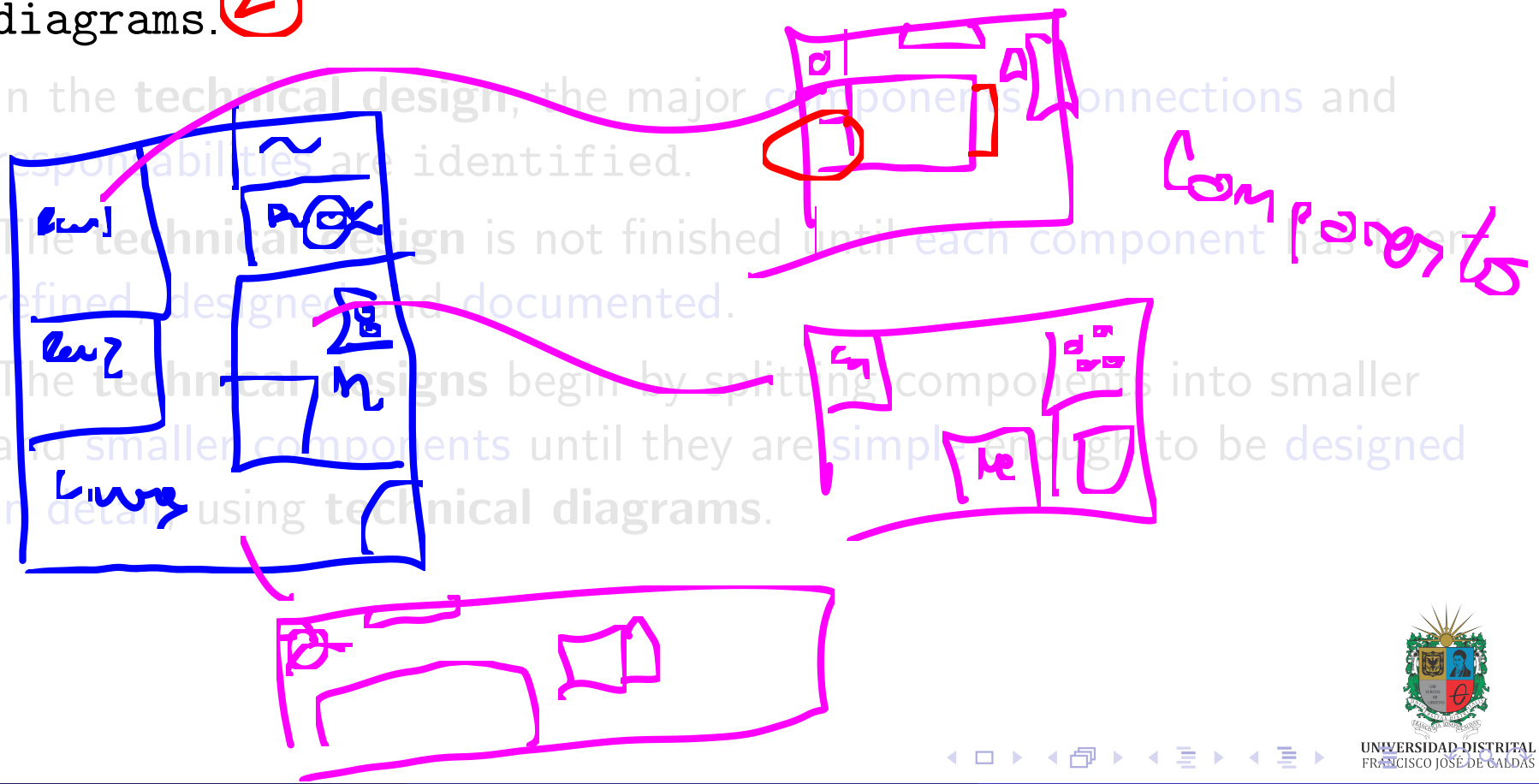
Use Cases

- **Use cases** are descriptions of how a system will be used by its users.
- They are used to capture the functional requirements of a system in a structured and detailed way.
- They are written from the perspective of the user and describe the interaction between the user and the system.



Technical Design

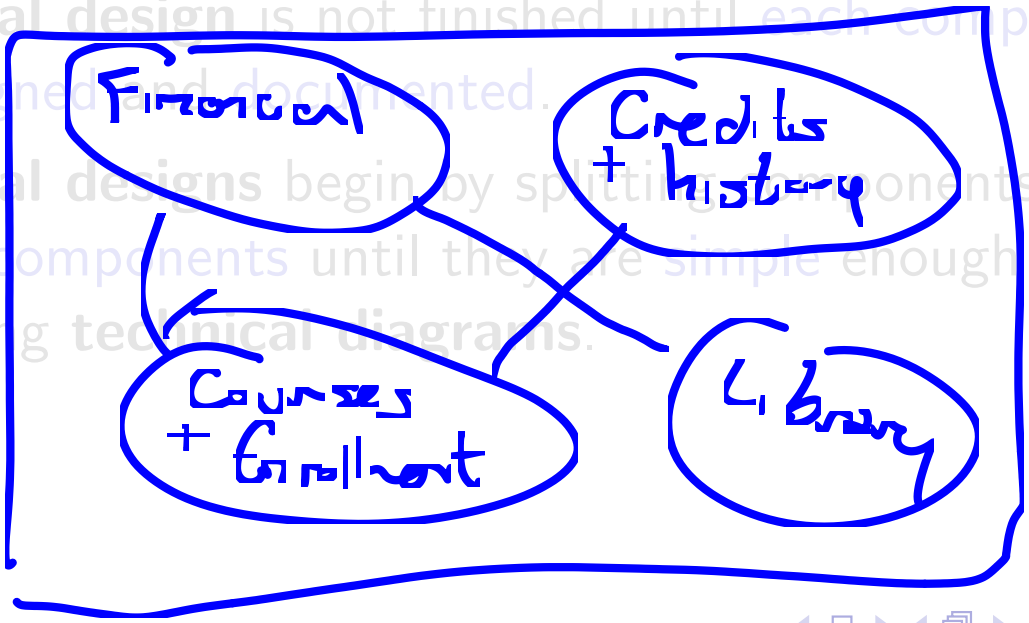
- Once the **conceptual design** is complete, the next step is to create a **technical design** of the software.
- **Technical Design** is a detailed design that defines the architecture and components of the software. It is achieved by the creation of diagrams.



- In the **technical design**, the major components, connections and responsabilities are identified.
- The **technical design** is not finished until each component has been refined, designed and documented.
- The **technical designs** begin by splitting components into smaller and smaller components until they are simple enough to be designed in detail using **technical diagrams**.

Technical Design

- Once the **conceptual design** is complete, the next step is to create a **technical design** of the software.
- **Technical Design** is a detailed design that defines the architecture and components of the `software`. It is achieved by the creation of `diagrams`.
- In the **technical design**, the major components, connections and responsabilities are `identified`.
- The **technical design** is not finished until each component has been refined, designed and documented.
- The **technical designs** begin by splitting components into smaller and smaller components until they are simple enough to be designed in detail, using **technical diagrams**.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Technical Design

- Once the **conceptual design** is complete, the next step is to create a **technical design** of the software.
- **Technical Design** is a detailed design that defines the architecture and components of the `software`. It is achieved by the creation of `diagrams`.
- In the **technical design**, the major components, connections and responsabilities are `identified`.
- The **technical design** is not finished until each component has been refined, designed and documented.
- The **technical designs** begin by splitting components into smaller and smaller components until they are simple enough to be designed in detail, using **technical diagrams**.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Compromise in Requirements and Design

- **Requirements** and **Design** are interrelated. **Requirements** are the foundation of the **design**.
- **Constant communication** and **feedback** is key to creating the right solution that satisfies the **client needs**.
- **Designs** will need to be reworked if components, connections, and the responsibilities of the **conceptual design** prove impossible to achieve in the **technical design**, or if they **fail to meet requirements**.
- **Larger systems** generally require more time to design, more time to implement, and more time to test.
- **Components** at this stage may be *refined enough* to become collections of `functions`, `classes`, or `other components`. These pieces become a more manageable problem that **developers** can individually **implement**.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Compromise in Requirements and Design

- **Requirements** and **Design** are interrelated. **Requirements** are the foundation of the **design**.
- **Constant communication** and **feedback** is key to creating the right solution that satisfies the **client needs**.
- **Designs** will need to be `reworked` if components, connections, and the responsibilities of the **conceptual design** prove impossible to achieve in the **technical design**, or if they **fail to meet requirements**.
- **Larger systems** generally require more time to design, more time to implement, and more time to test.
- **Components** at this stage may be *refined enough* to become collections of `functions`, `classes`, or `other components`. These pieces become a more manageable problem that **developers** can individually **implement**.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Compromise in Requirements and Design

- **Requirements** and **Design** are interrelated. **Requirements** are the foundation of the **design**.
- **Constant communication** and **feedback** is key to creating the right solution that satisfies the **client needs**.
- **Designs** will need to be `reworked` if components, connections, and the responsibilities of the **conceptual design** prove impossible to achieve in the **technical design**, or if they **fail to meet requirements**.
- **Larger systems** generally require more time to design, more time to implement, and more time to test.
- **Components** at this stage may be *refined enough* to become collections of `functions`, `classes`, or `other components`. These pieces become a more manageable problem that **developers** can individually **implement**.

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Trade-Offs

- **Trade-offs** are inevitable in `software design`. **Quality attributes** are often competing and contradictory.
- **Trade-offs** are *necessary* to *balance* the `competing` quality attributes of a system.
- **Trade-offs** are *made* by weighing the *importance* of each quality attribute and *deciding* which attributes are most important.
- **Trade-offs** are *made* by `compromising` on less important quality attributes in order to improve the more important attributes.
- **Trade-offs** are necessary to *create* a system that satisfies the needs of the `users` and `stakeholders`.

Security
+ auth → speed
- performance
slow

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Trade-Offs

- **Trade-offs** are inevitable in software design. **Quality attributes** are often competing and contradictory.
- **Trade-offs** are *necessary* to *balance* the competing quality attributes of a system.
- **Trade-offs** are *made* by weighing the *importance* of each quality attribute and *deciding* which attributes are most important.
- **Trade-offs** are *made* by compromising on less important quality attributes in order to improve the more important attributes.
- Trade-offs are necessary to *create* a system that satisfies the needs of the users and stakeholders.

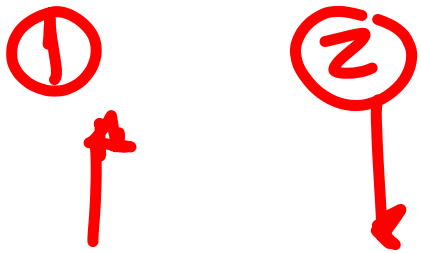
sec. 0.6

Fast 0.9

Friendly 0.2

Trade-Offs

- **Trade-offs** are inevitable in `software design`. **Quality attributes** are often competing and contradictory.
- **Trade-offs** are *necessary* to *balance* the `competing` quality attributes of a system.
- **Trade-offs** are *made* by weighing the *importance* of each quality attribute and *deciding* which attributes are most important.
- **Trade-offs** are *made* by `compromising` on less important quality attributes in order to `improve` the more important attributes.
- **Trade-offs** are necessary to *create* a system that satisfies the needs of the `users` and `stakeholders`.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Context and Consequences

- **Context** is the environment in which a system will be used. It includes the users, the stakeholders, and the constraints of the system.
- **Context** provides important information when deciding on the balance of qualities in design.
- **Consequences** are the results of the decisions that are made during the *design* of a system. They include the **trade-offs** that are made and the impact that they have on the system.
- A **good practice** is to seek other perspectives on technical designs. This can be done by *asking* other developers for their opinion, or by having a design review session.
- Another **good practice** is to perform prototyping and simulation to test the design before implementing it.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

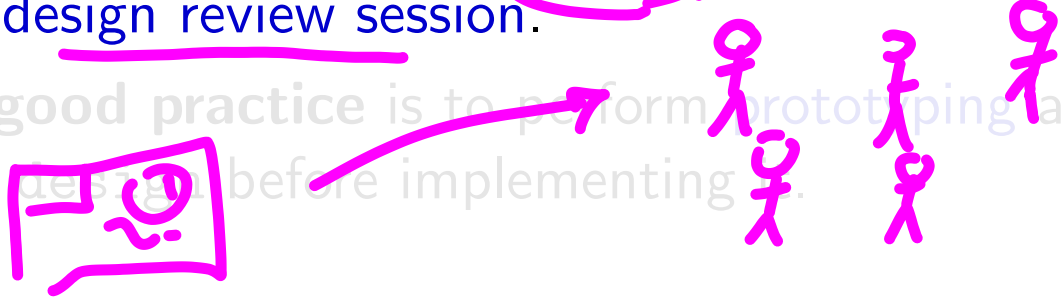
Context and Consequences

- **Context** is the environment in which a system will be used. It includes the users, the stakeholders, and the constraints of the system.
- **Context** provides important information when deciding on the balance of qualities in design.
- **Consequences** are the results of the decisions that are made during the *design* of a system. They include the **trade-offs** that are made and the impact that they have on the system.
- A **good practice** is to seek other perspectives on technical designs. This can be done by asking other developers for their opinion, or by having a design review session.
- Another **good practice** is to perform prototyping and simulation to test the design before implementing it.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Context and Consequences

- **Context** is the environment in which a system will be used. It includes the users, the stakeholders, and the constraints of the system.
- **Context** provides important information when deciding on the balance of qualities in design.
- **Consequences** are the results of the decisions that are made during the *design* of a system. They include the **trade-offs** that are made and the impact that they have on the system.
- A **good practice** is to seek other perspectives on technical designs. This can be done by *asking* other developers for their opinion, or by having a design review session.
- Another **good practice** is to perform prototyping and simulation to test the design before implementing it.



Context and Consequences

- **Context** is the environment in which a system will be used. It includes the users, the stakeholders, and the constraints of the system.
- **Context** provides important information when deciding on the balance of qualities in design.
- **Consequences** are the results of the decisions that are made during the *design* of a system. They include the **trade-offs** that are made and the impact that they have on the system.
- A **good practice** is to seek other perspectives on technical designs. This can be done by *asking* other developers for their opinion, or by having a design review session.
- Another **good practice** is to perform prototyping and simulation to test the design before implementing it.

code

(Figma)

Balsamiq

mockups

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Satisfying Qualities

- **Quality attributes** are the characteristics of a system that *determine* its quality. They are the **features** that *define* how well a system *satisfies* the needs of its `users`.
- **Quality attributes** are important because they *determine* how well a system will satisfy the needs of its `users`.
- **Quality attributes** have a strong *relationship* with the `non-functional requirements` to satisfy aspects as performance, resource usage and efficiency.
- Other **qualities** that software often satisfies in `non-functional requirements` include reusability, flexibility, and maintainability. This helps inform *how well* the code of software can evolve and allow for *future changes*.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Satisfying Qualities

- **Quality attributes** are the characteristics of a system that *determine* its quality. They are the **features** that *define* how well a system *satisfies* the needs of its `users`.
- **Quality attributes** are important because they *determine* how well a system will satisfy the needs of its `users`.
- **Quality attributes** have a strong *relationship* with the `non-functional requirements` satisfy aspects as performance, resource usage and efficiency.
- Other **qualities** that software often satisfies in `non-functional requirements` include reusability, flexibility, and maintainability. This helps inform *how well* the code of software can evolve and allow for *future changes*.

+ ⇒ global quality

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Satisfying Qualities

- **Quality attributes** are the characteristics of a system that *determine* its quality. They are the **features** that *define* how well a system *satisfies* the needs of its `users`.
- **Quality attributes** are important because they *determine* how well a system will satisfy the needs of its `users`.
- **Quality attributes** have a strong *relationship* with the `non-functional requirements` to satisfy aspects as performance, resource usage and efficiency.
- Other qualities that software often satisfies in `non-functional requirements` include reusability, flexibility, and maintainability. This helps inform how the code of software can evolve and allow for *future changes*.

Req Func.
- show schedule
- enroll course
- pay debt

Req. non-Func.
- Resp < 300 ms
- Support > 5000 users
- Enqueue re los requests

Satisfying Qualities

- **Quality attributes** are the characteristics of a system that *determine* its quality. They are the **features** that *define* how well a system *satisfies* the needs of its `users`.
- **Quality attributes** are important because they *determine* how well a system will satisfy the needs of its `users`.
- **Quality attributes** have a strong *relationship* with the `non-functional requirements` to satisfy aspects as performance, resource usage and efficiency.
- Other **qualities** that software often satisfies in `non-functional requirements` include reusability, flexibility, and maintainability. This helps inform *how well* the `code` of software can evolve and allow for *future changes*.

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Basics of Object-Oriented Design I

- **Object-oriented** has become one of the most traditional and popular **paradigms** in software development.
- It is based on the concept of **objects**, which can contain data, in the form of **fields** (often known as attributes or properties), and code, in the form of **procedures** (often known as methods).

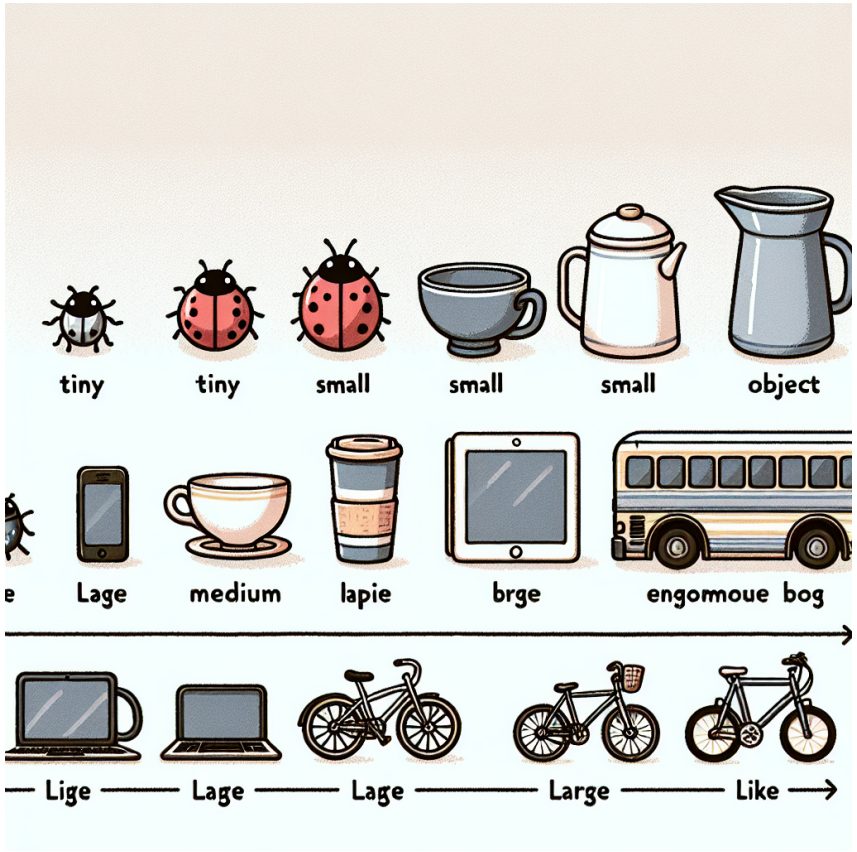


Figure: Prompt: Draw several objects sorted by size.

Basics of Object-Oriented Design I

- **Object-oriented** has become one of the most traditional and popular **paradigms** in software development.
- It is based on the concept of **objects**, which can contain data, in the form of **fields** (often known as attributes or properties), and code, in the form of **procedures** (often known as methods).

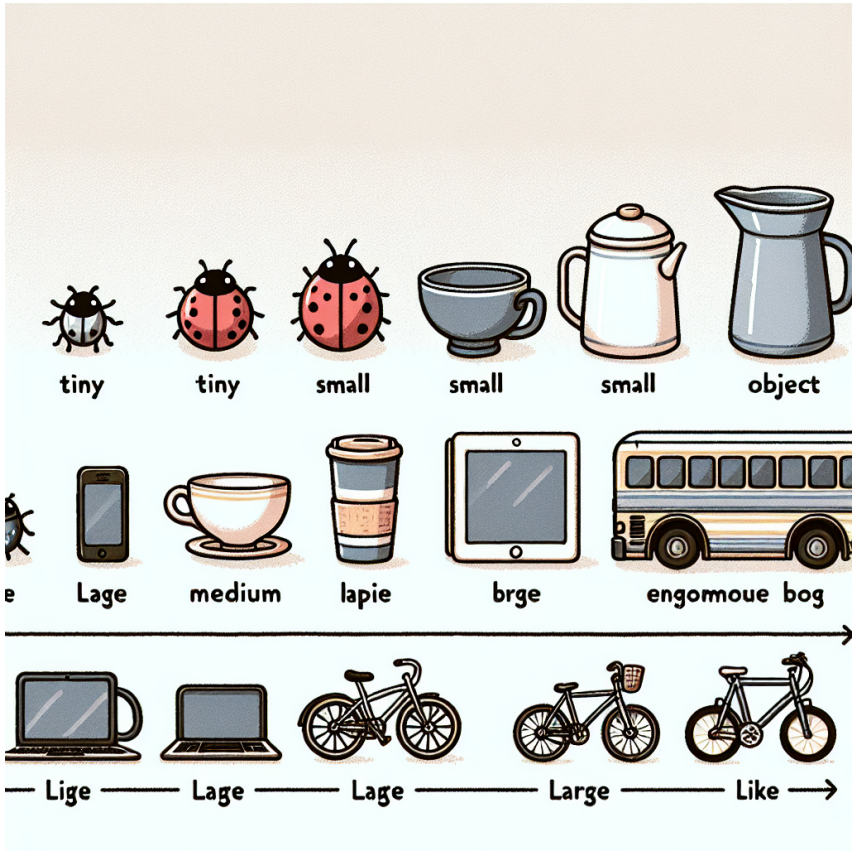


Figure: Prompt: Draw several objects sorted by size.

Basics of Object-Oriented Design II

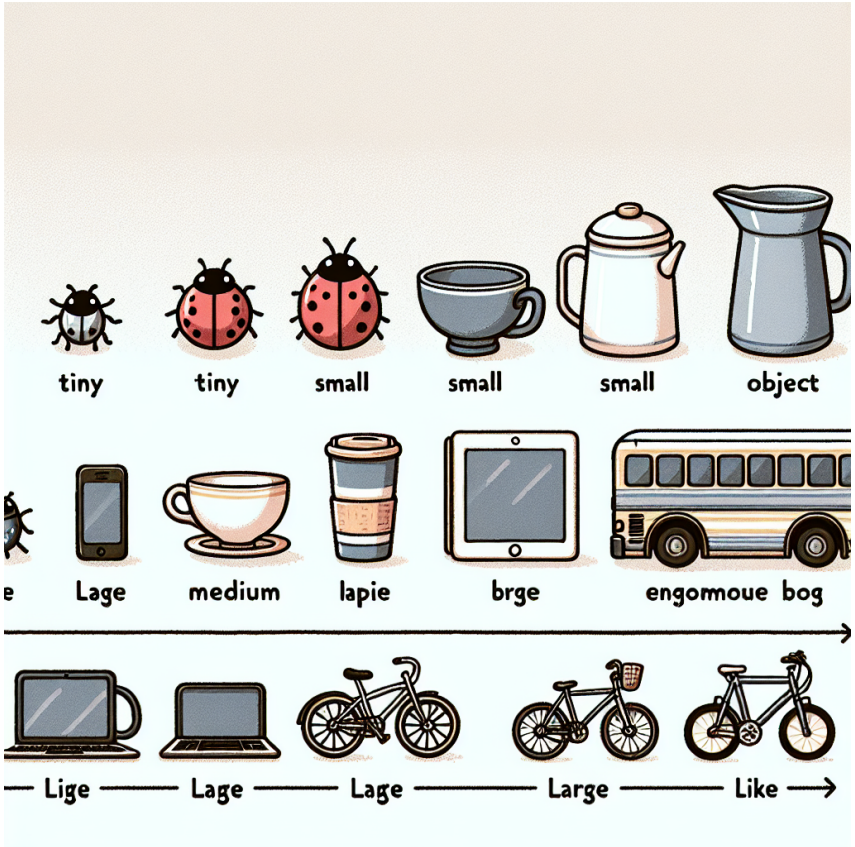
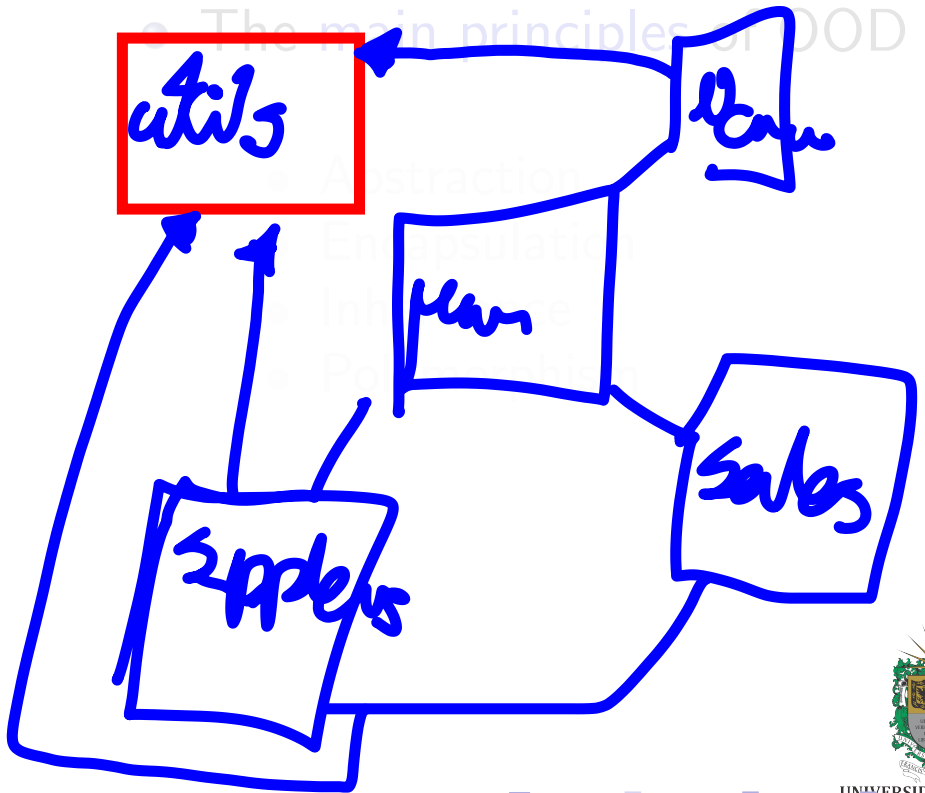


Figure: Prompt: Draw several objects sorted by size.

- The idea is to design a **system modularly**, making it `easier to maintain` and understand. Another key idea is to emphasize code reuse.
- The main principles of OOD are:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism



Basics of Object-Oriented Design II

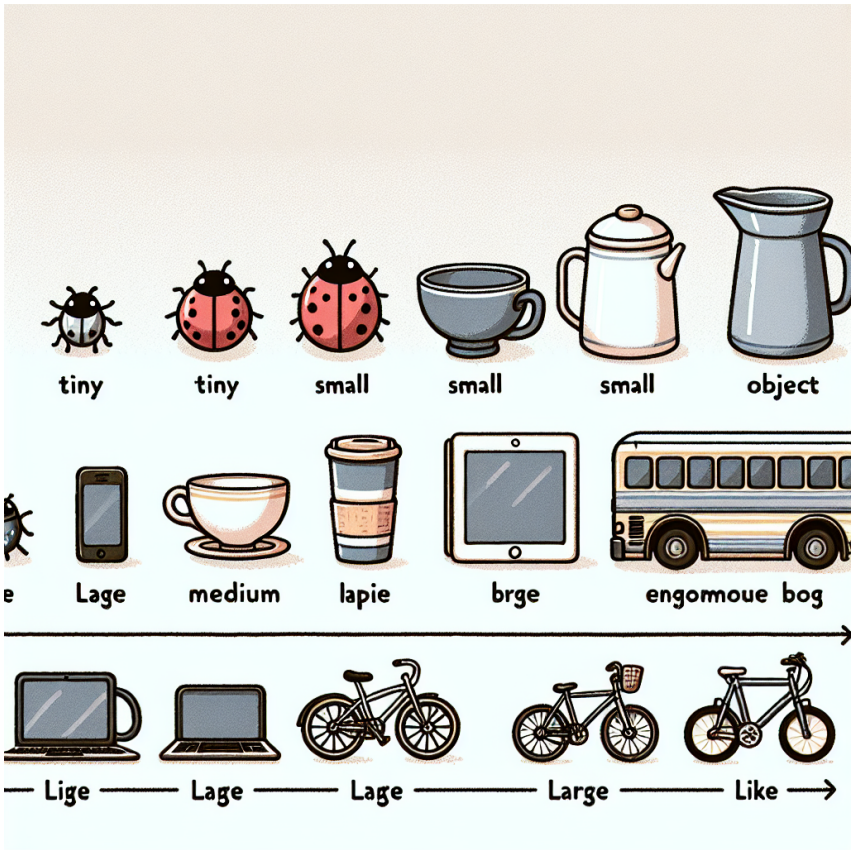
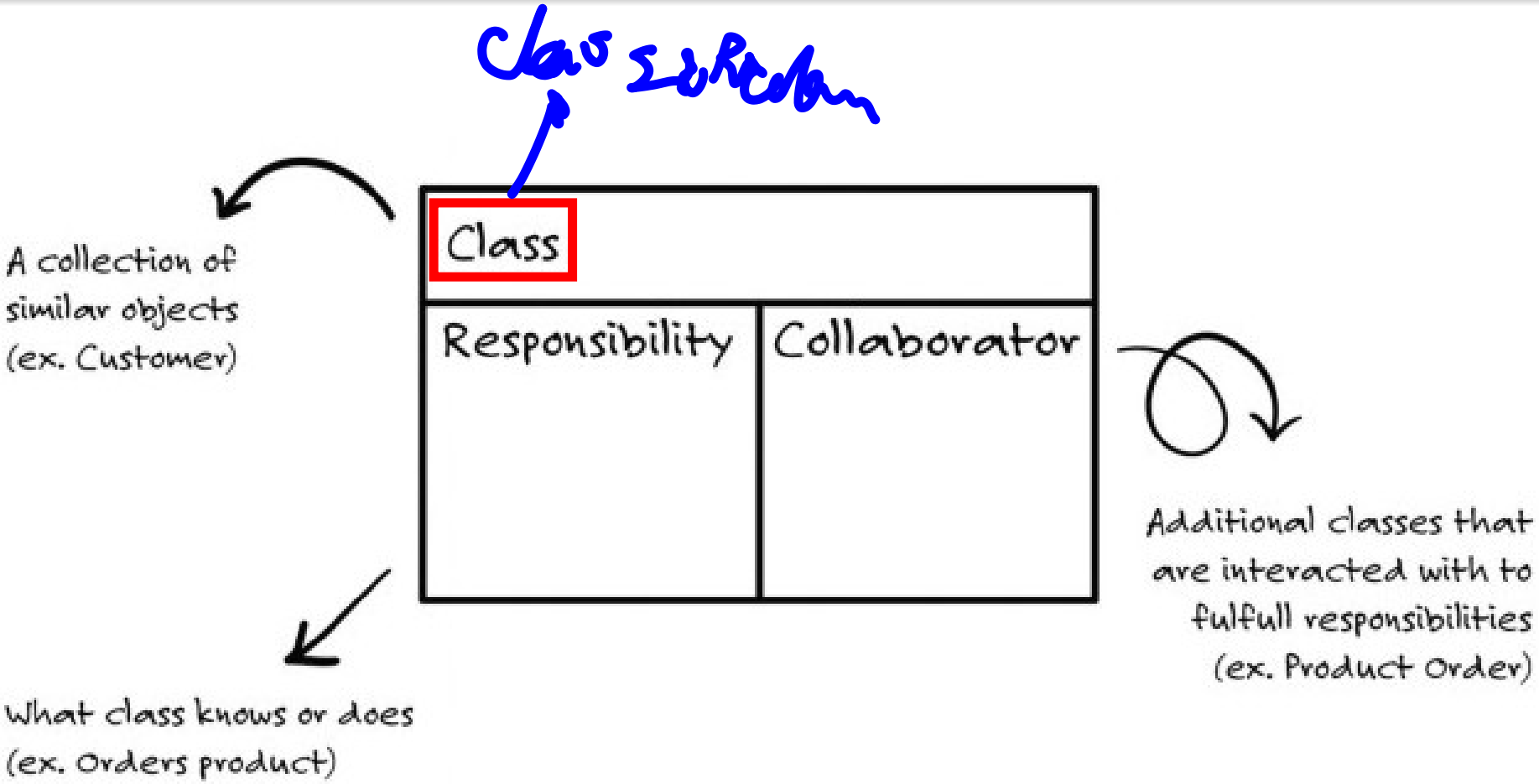


Figure: Prompt: Draw several objects sorted by size.

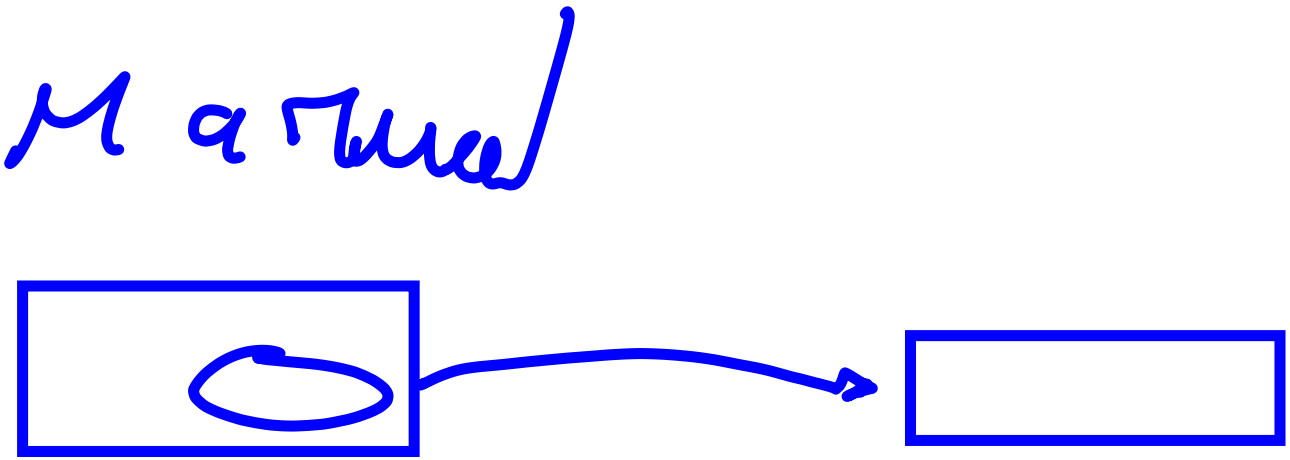
- The idea is to design a **system modularly**, making it `easier to maintain` and understand. Another key idea is to emphasize code reuse.
- The main principles of OOD are:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

CRC Cards


Class
Responsibility
Collaborator
A collection of similar objects (ex. Customer)
What class knows or does (ex. Orders product)
Additional classes that are interacted with to fulfull responsibilities (ex. Product Order)

Prototyping and Simulation

- **CRC cards** are useful tools, especially for prototyping and simulation during `conceptual design`.
- **CRC cards** are excellent tools to bring to software development team meetings. All the cards can be placed on the table to `facilitate discussion` or `simulate` how these classes work together with other classes to achieve their responsibilities.



Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/object-oriented-programming*

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS