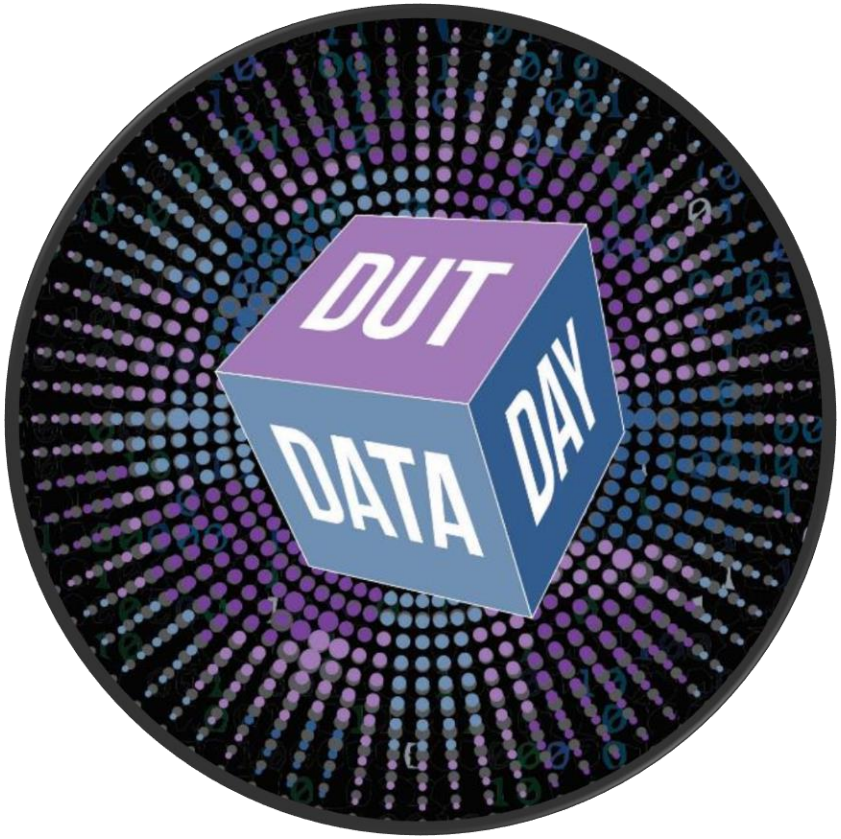


DUT

DURBAN UNIVERSITY OF TECHNOLOGY

INYUVESI YASETHEKWINI YEZOBUCHWEPHESHE

ENVISION2030

FINANCE AND INFORMATION MANAGEMENT/ INFORMATION TECHONOLOGY

BUSINESS ANALYSIS IIB (BANP202)

By: PT SIMELANE

INTRODUCTION TO UNIFIED MODELING LANGUAGE (CHAPTER 3)

GROUND RULES

1. PLEASE ALL CELLPHONES OFF OR PUT THEM ON SILENT DURING LECTURES
2. DON'T BE LATE FOR CLASS
4. DON'T MAKE NOISE IN CLASS
6. TAKE NOTES AND ASK QUESTIONS

ABOUT BANP202

ABOUT BANP202

Aim/Purpose:

- To provide students with Business Analysis tools and methodologies to solve business related problems

LEARNING OUTCOMES

- Unified Modelling Language
- A Brief Timeline for OO and UML
- UML Building Blocks
- Modelling Elements
- Modelling Elements and Diagrams
- UML Diagrams
- Modelling Perspectives
- UML Modelling Questions

UNIFIED MODELING LANGUAGE

- UML is an object-oriented modeling language (or more precisely, a collection of modeling languages) that is
  - expressive
  - semi-formal (UML 2.0 added much more formality)
  - capable of supporting incremental development
    - Elements can be hidden
    - Certain elements can be left incomplete
    - Inconsistencies can exist
  - process independent
    - UML can be used with a variety software development process models
  - Customizable and extensible

A BRIEF TIMELINE FOR OO AND UML

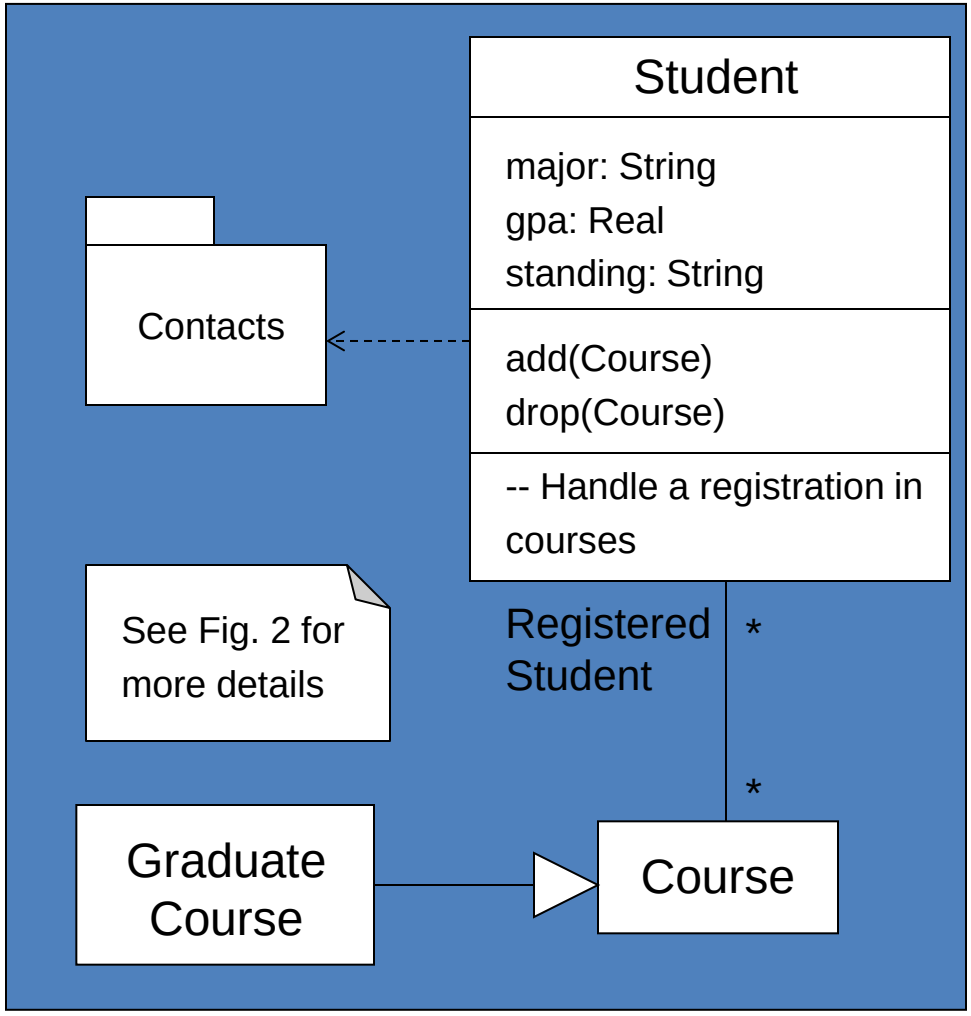
- 60's
  - Birth of initial OO ideas
- 70's
  - Nurturing of OO ideas
  - Introduction of a few more OO Programming Languages (OOPLs)
- 80's
  - Maturing of fundamental OO concepts
  - Emergence of more OOPL's
  - OOPL's gain widespread use

A BRIEF TIMELINE FOR OO AND UML

- 90's
  - The Method Wars
  - Efforts to unify concepts
  - Introduction and standardization of UML
  - Emergence of next-generation ideas, like Patterns
- Current
  - Widespread use of UML
  - Widespread use Full-Life-Cycle development tools

UML BUILDING BLOCKS

- Modeling Elements
  - Structural
  - Behavioral
  - Organizational
  - Annotational
- Diagrams that communicate ideas using the modeling elements
- Views



MODELING ELEMENTS

The following table describes the common types of model elements.

Type of model element	Description
Structural model elements	These elements model the static parts of a system. Some examples include classifiers such as actors, classes, components, information items, and nodes.
Behavioral model elements	These elements model the dynamic parts of a system. Typically, you find behavioral model elements in state machine and interaction diagrams. Some examples include activities, decisions, messages, objects, and states.
Organizational model elements	These elements group model elements into logical sets. A package is an example of an organizational model element.
Annotational model elements	These elements provide comments and descriptions. Notes® and constraints are examples of annotational model elements.

MODELING ELEMENTS

Modeling Elements are building blocks for constructing conceptual descriptions of systems

- Definition and Scope
  - Use Cases
  - Automation Boundaries
- Structural
  - Objects
  - Classes
  - Relations
  - Interfaces
  - Components
  - Nodes
- Extension
  - Templates
  - Stereotypes

- Behavioral Things
  - Messages
  - States
  - Transitions
  - Events
- Organizational Things
  - Packages
  - Views
- Annotation
  - Comments
  - Specifications

MODELING ELEMENTS AND DIAGRAMS

- Diagrams represent chunks of information that need to be communicated as part of a conceptual description.
  - It usually requires many diagrams to describe a system
  - Each diagram should focus on a single thought or a small set of tightly related thoughts
- Diagrams are like paragraphs in a section of well-structured text

UML DIAGRAMS

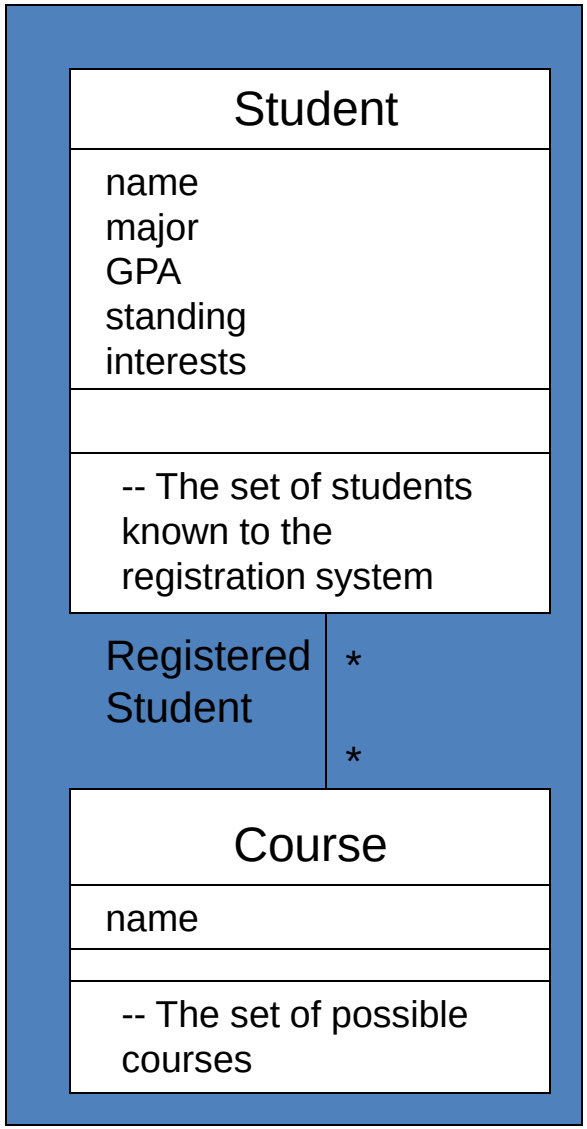
- **Use Case Diagrams**
- Class Diagrams
- Object Diagrams
- Interaction Diagrams
  - Sequence Diagrams
  - Communication Diagrams
- State Charts (enhanced State Machines)
- Component Diagrams
- Deployment Diagrams

MODELING PERSPECTIVES

- Analysis – for understanding
  - The objects represented in the models are real-world objects
  - Models focus on problem-domains concepts
  - They describe systems as they are
- Specification – for scoping and planning
  - The models include both real-world and software objects
  - The models show automation boundaries
  - The models describe what the system is to become
- Implementation – for designing / building
  - The objects in the models are mostly software objects
  - The models focus on solution-oriented concepts
  - The models describe what the software system is or will be

ANALYSIS PERSPECTIVE

- Classes are sets of objects
- Classes may include attributes and operations, but more importantly their intents are defined by responsibilities
- Relationships are set of links between objects
- Components relate to the problem domain



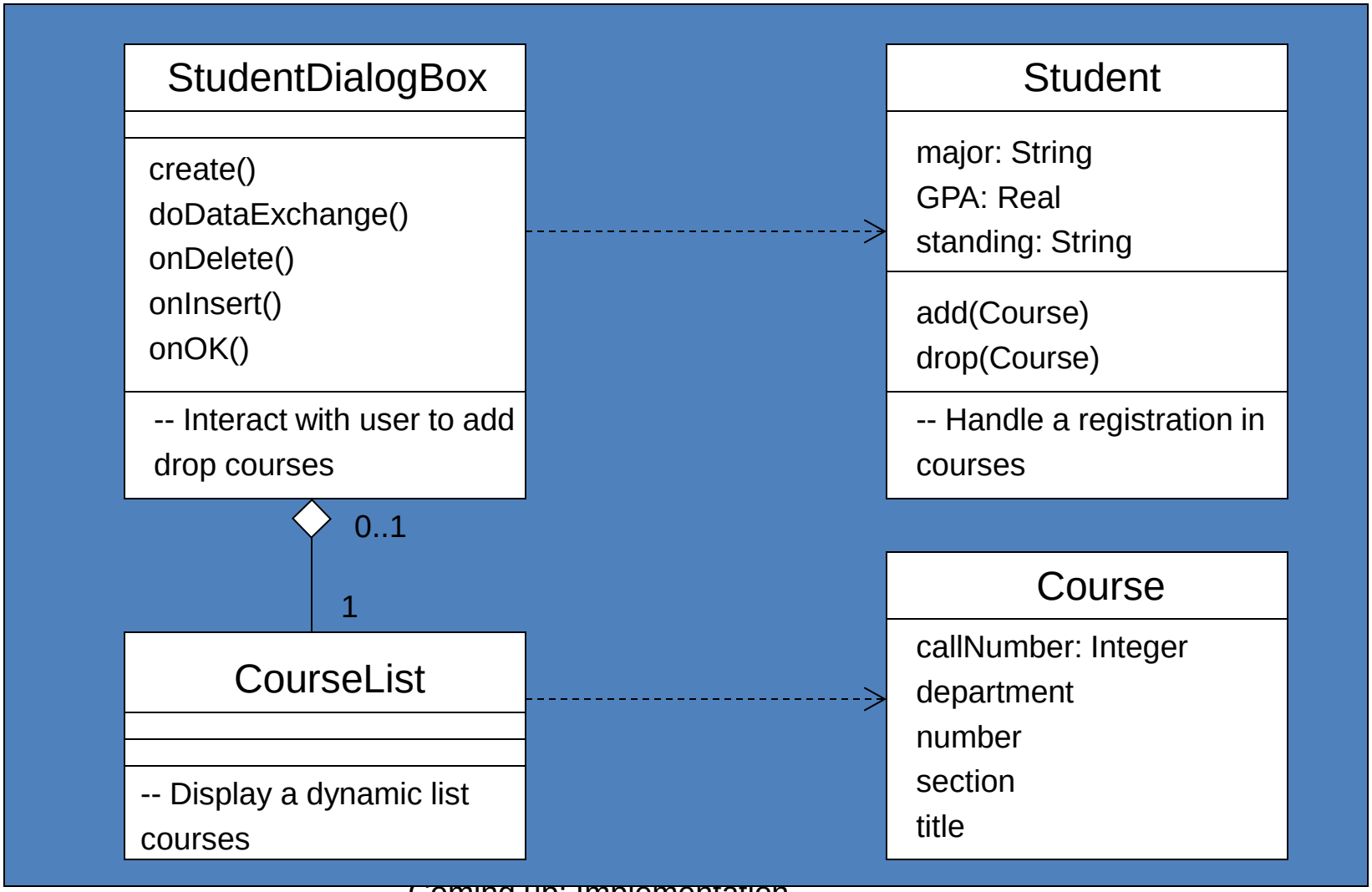
SPECIFICATION PERSPECTIVE

$Student^R$
name major GPA standing interests
-- The set of students known to the registration system

- Classes define abstraction boundaries and encapsulations for software objects

$Student^S$
name: String major: String GPA: real standing: Scode
add(Course) drop(Course)
-- Software representation of students; support registration in courses

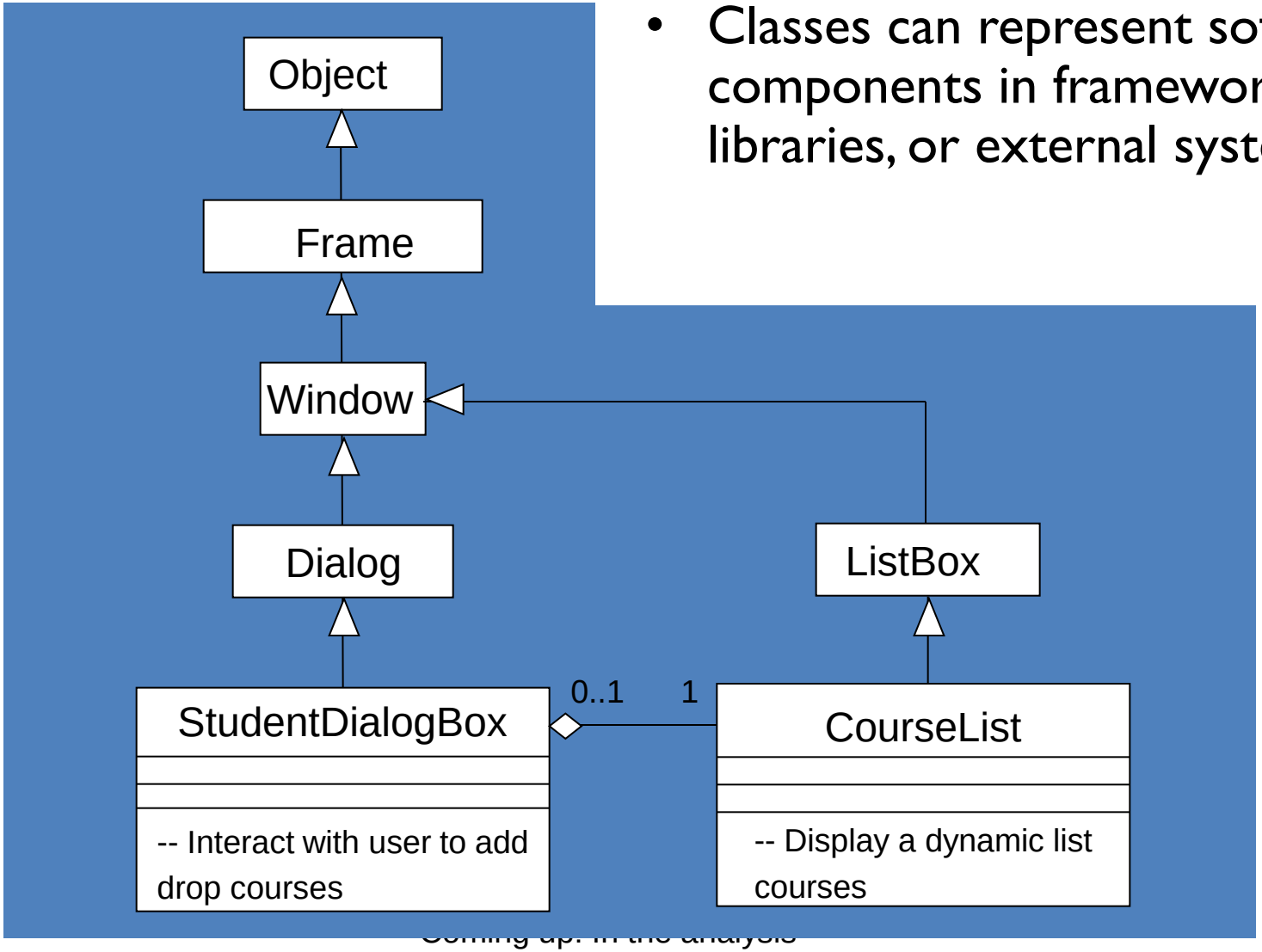
IMPLEMENTATION PERSPECTIVE



Coming up: Implementation Perspective

IMPLEMENTATION PERSPECTIVE

- Classes can represent software components in frameworks, libraries, or external systems



perspective will I need a loop

In the analysis perspective will I need a loop counter?

- A. Yes if you have a loop
- B. No
- C. It depends (be able to say on what if you choose this ☺)

SOME INTERESTING UML NODELING QUESTIONS

- How do we discover objects or classes?
- When should we focus on problem-domain objects, solution-domain objects, or environment objects?
- How can we keep the different perspectives straight?
- Should each perspective be captured by a different model or can they all be managed in one model?
- How much detail should you put in a diagram, a view, or a model?

MORE UML MODELING QUESTIONS

- How should you distribute responsibilities among classes?
  - What happens when classes get too big, i.e. inadequate distribution?
  - What happens when classes are too small, i.e. too fine of grain distribution?
  - What happens when there are a lot of dependency relationships between classes, i.e., inappropriate or ad hoc distribution?

THANK YOU