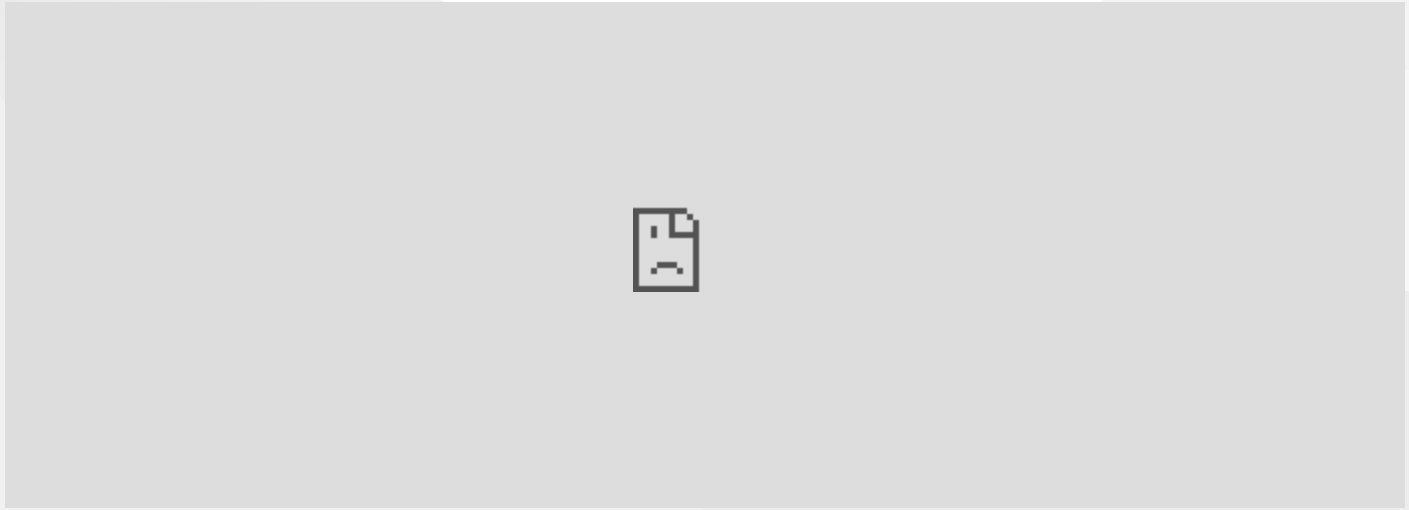
CEN206 Object-Oriented Programming

Week-12 (UML, UMPLE and Java Implementations)

Spring Semester, 2024-2025

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX



Outline

- Unified Modeling Language (UML)
  - Overview and Purpose
  - Main Diagram Types
  - Class Diagrams in Detail
  - Sequence Diagrams
  - State Diagrams
- UMPLE
  - Introduction to Model-Driven Development
  - UMPLE Language Features
  - Code Generation with UMPLE
- Implementing UML Designs in Java
  - From UML to Code
  - Best Practices

What is UML?

- **U**nified **M**odeling **L**anguage
- Standard visual modeling language for software systems
- Provides a common vocabulary for object-oriented modeling
- Developed by Grady Booch, James Rumbaugh, and Ivar Jacobson ("Three Amigos")
- Maintained by the Object Management Group (OMG)
- Current version: UML 2.5.1 (December 2017)

Purpose of UML

- Visualize system architecture and design
- Specify system structure and behavior
- Document design decisions
- Facilitate communication among stakeholders
- Guide implementation
- Generate code (with tools like UMPLE)

Structural Diagrams

- **Class diagram**: Shows classes, interfaces, and their relationships
- **Object diagram**: Depicts instances of classes at a point in time
- **Component diagram**: Shows organization of physical components
- **Deployment diagram**: Illustrates hardware topology and software deployment
- **Package diagram**: Shows logical grouping of elements
- **Composite structure diagram**: Shows internal structure of a class

Behavioral Diagrams

- **Use case diagram**: Shows functionality from user perspective
- **Activity diagram**: Depicts workflow or business process
- **State machine diagram**: Shows states and transitions
- **Sequence diagram**: Shows interactions between objects over time
- **Communication diagram**: Shows interactions focusing on links between objects

Class Diagrams

The most commonly used UML diagram, showing:

- Classes and their properties
- Relationships between classes
- Interfaces and their implementations
- Inheritance hierarchies
- Associations, dependencies, and more

Class Diagram Example

Class Notation

- Can be unidirectional or bidirectional

Aggregation

- "Has-a" relationship (weak ownership)
- Diamond at the owner's end
- Part can exist independently of the whole

Composition

- Strong form of aggregation (strong ownership)
- Filled diamond at the owner's end
- Part's lifecycle depends on the whole

Generalization/Inheritance

- "Is-a" relationship
- Triangle pointing to the parent class

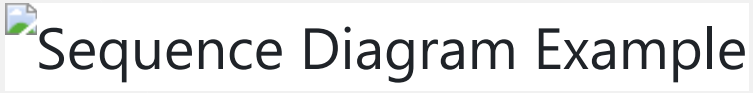
Class Relationships Example


Class Relationships Example

Sequence Diagrams

Show the sequence of interactions between objects over time:

- Objects and their lifelines
- Messages exchanged between objects
- Time ordering of interactions
- Creation and destruction of objects
- Activation and deactivation of objects


Sequence Diagram Example

- Represents an object over time
- Vertical dashed line
- May include activation bars

Message

- Communication between lifelines
- Solid arrow for synchronous calls
- Dashed arrow for asynchronous calls
- Arrow with a filled arrowhead for message returns

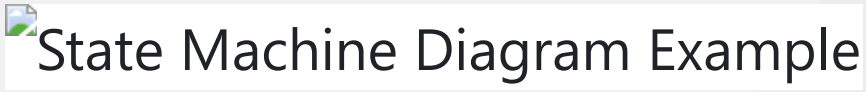
Combined Fragments

- Define conditional behavior
- Include alt (alternatives), opt (optional), loop, etc.
- Surround a group of messages

State Machine Diagrams

Show how an object responds to events based on its current state:

- States
- Transitions between states
- Events triggering transitions
- Actions performed during transitions
- Entry/exit actions
- Nested states


State Machine Diagram Example

- Combines UML with programming languages
- Embeds UML directly in code
- Generates code from models
- Supports Java, C++, PHP, and Ruby
- Open-source and web-based tools available

UMPLE Philosophy

- **Model-Code Duality**: Models and code are the same artifact
- **Incremental Adoption**: Use as much or as little as needed
- **Multiple Views**: Generate different views of the same system
- **Executable Models**: Models can be directly executed

UMPLE Tools

- **UmpleOnline**: Web-based editor and code generator
- **Umple Eclipse Plugin**: Integrates with Eclipse IDE

UMPLE Basic Syntax

```
class Student {
  // Attributes with types
  Integer id;
  String name;

  // Associations with multiplicities
  * -- 1 University;

  // State machine
  status {
    Active {
      suspend -> Suspended;
    }
    Suspended {
      reinstate -> Active;
      expel -> Expelled;
    }
    Expelled {}
  }

  // Methods (in target language)
  void registerForCourse(Course c) {
    // Implementation in target language
  }
}
```

```
  Integer age;
  Date birthDate;

  // With default values
  const String country = "Canada";
  Boolean isActive = true;

  // With constraints
  [age > 0]
  [email ~= /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/]
}
```

Associations

```
// One-to-many association
class Professor {
  // Professor can teach many courses
  1 -- * Course;
}

// Many-to-many association
class Student {
  // Students enroll in many courses
  * -- * Course;
```

UMPLE State Machines

```
class TrafficLight {
  // State machine definition
  status {
    Red {
      // Entry and exit actions
      entry / { turnOnRedLight(); }
      exit / { turnOffRedLight(); }

      // Transition with guard and action
      timer [timeInState() > 60] -> Green { resetTimer(); }
    }

    Yellow {
      entry / { turnOnYellowLight(); }
      exit / { turnOffYellowLight(); }
      timer [timeInState() > 5] -> Red;
    }

    Green {
      entry / { turnOnGreenLight(); }
      exit / { turnOffGreenLight(); }
      timer [timeInState() > 45] -> Yellow;
    }
  }

  // Methods needed by the state machine
  private void turnOnRedLight() { /* implementation */ }
  private void turnOffRedLight() { /* implementation */ }
  // Other methods...
}
```

Code Generation with UMPLE

Java Code Generation

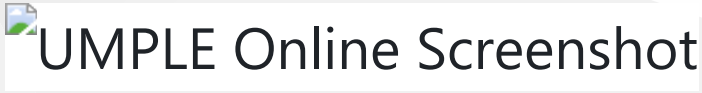
```
class Car {
  String make;
  String model;
  Integer year;

  * -- 1 Manufacturer;

  status {
    Stopped {
      startEngine -> Running;
    }
    Running {
      stopEngine -> Stopped;
      accelerate -> Accelerating;
    }
    Accelerating {
      releaseGas -> Running;
      brake -> Braking;
    }
    Braking {
      stop -> Stopped;
      releaseBreak -> Running;
    }
  }
}
```

Generated Java Code (Partial)

```
public class Car {
  //------------------------
  // MEMBER VARIABLES
  //------------------------
  private String make;
  private String model;
  private Integer year;

  // State machine variables
  public enum Status { Stopped, Running, Accelerating, Braking }
  private Status status;

  //------------------------
  // CONSTRUCTOR
  //------------------------
  public Car(String aMake, String aModel, Integer aYear, Manufacturer aManufacturer) {
    make = aMake;
    model = aModel;
    year = aYear;
    boolean didAddManufacturer = setManufacturer(aManufacturer);
    if (!didAddManufacturer) {
      throw new RuntimeException("Unable to create car due to manufacturer");
    }
    setStatus(Status.Stopped);
  }

  // State machine methods, getters, setters, etc.
}
```

UMPLE Online Demo

- UMPLE provides an online environment for model development
- Allows real-time visualization and code generation
- Can be used to demonstrate UML concepts quickly



Visit: http://try.umple.org

From UML to Java Implementation

Implementing Classes

UML Feature	Java Implementation
Class	`public class ClassName`
Abstract class	`public abstract class ClassName`
Interface	`public interface InterfaceName`
Attributes	Fields with appropriate access modifiers
Operations	Methods with appropriate signatures
Visibility	`public`, `private`, `protected`, or package-private

Inheritance/Generalization

```
// UML: Child inherits from Parent
public class Parent {
    // Parent members
}

public class Child extends Parent {
    // Child members
}
```

Implementation (Interface)

```
// UML: Class implements Interface
public interface MyInterface {
    void doSomething();
}

public class MyClass implements MyInterface {
    @Override
    public void doSomething() {
        // Implementation
```

Association

```
// UML: Class A has a reference to Class B
public class A {
    private B b; // One-to-one

    public A(B b) {
        this.b = b;
    }

    public B getB() {
        return b;
    }

    public void setB(B b) {
        this.b = b;
    }
}

// UML: Class C has many references to Class D
public class C {
    private List<D> dList; // One-to-many

    public C() {
        this.dList = new ArrayList<>();
    }

    public void addD(D d) {
        dList.add(d);
    }

    // Other methods to manage the relationship
}
```

```
    private final Part part; // The 'final' emphasizes strong ownership

    public Whole() {
        // Create the part when the whole is created
        this.part = new Part();
    }

    // No setter for part - it cannot be changed after creation
    public Part getPart() {
        return part;
    }
}
```

Aggregation

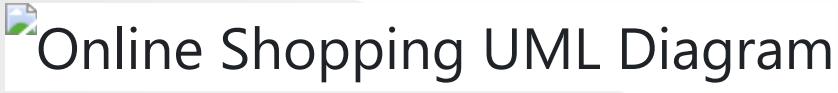
```
// UML: Class Container has Parts (aggregation)
public class Container {
    private Part part; // Not final - can exist independently

    public Container(Part part) {
        this.part = part;
    }

    // Part can be changed or set to null
```

Implementing Behavioral Elements

State Machines

```
public class Document {
    // State enumeration
    public enum State {
        DRAFT, REVIEW, APPROVED, PUBLISHED
    }

    private State currentState;

    public Document() {
        currentState = State.DRAFT;
    }

    public void submitForReview() {
        if (currentState == State.DRAFT) {
            currentState = State.REVIEW;
            System.out.println("Document submitted for review");
        } else {
            System.out.println("Cannot submit - not in DRAFT state");
        }
    }

    public void approve() {
        if (currentState == State.REVIEW) {
            currentState = State.APPROVED;
            System.out.println("Document approved");
        } else {
            System.out.println("Cannot approve - not in REVIEW state");
        }
    }

    // More state transition methods...
}
```

Case Study: Online Shopping System



UMPLE Implementation

```
class Customer {
  String name;
  String email;
  String address;

  1 -- * Order;
}

class Order {
  Date orderDate;
  Float totalAmount;

  status {
    New {
      processPayment -> PaymentProcessing;
    }
    PaymentProcessing {
      paymentSuccessful -> Confirmed;
      paymentFailed -> PaymentFailed;
    }
    PaymentFailed {
      retry -> PaymentProcessing;
      cancel -> Cancelled;
    }
    Confirmed {
      ship -> Shipped;
    }
    Shipped {
      deliver -> Delivered;
    }
    Delivered {}
    Cancelled {}
  }

  * -- * Product;
}

class Product {
  String name;
  String description;
  Float price;
  Integer stockQuantity;
}
```

Java Implementation (Partial)

```java
public class Customer {
    private String name;
    private String email;
    private String address;
    private List<Order> orders;

    public Customer(String name, String email, String address) {
        this.name = name;
        this.email = email;
        this.address = address;
        this.orders = new ArrayList<>();
    }

    public void addOrder(Order order) {
        orders.add(order);
    }

    // Getters, setters, and other methods
}

public class Order {
    private Date orderDate;
    private float totalAmount;
    private List<Product> products;
    private OrderState state;

    public Order() {
        this.orderDate = new Date();
        this.products = new ArrayList<>();
        this.state = OrderState.NEW;
    }

    // State transition methods, getters, setters, and other functionality
}
```

- Each class should have only one reason to change

2. **Encapsulate field access**
    - Use private fields with public getters/setters
3. **Prefer composition over inheritance**
    - "Has-a" is often better than "is-a"
4. **Implement interfaces for behavior**
    - Use interfaces to define contracts
5. **Use design patterns appropriately**
    - Match patterns to common problems
6. **Keep entities immutable when possible**
    - Especially for value objects
7. **Test against the model**

UMPLE vs. Other UML Tools

Feature	UMPLE	Traditional UML Tools
Code Integration	Direct integration in code	Separate models from code
Learning Curve	Moderate (like learning a language extension)	Steep (completely different paradigm)
Round-tripping	Natural (model and code are the same)	Often problematic
Version Control	Standard source control tools	May require special tools
IDE Support	Varies, good Eclipse support	Often extensive

Requirements

1. Create a UML class diagram showing:
   - `BankAccount` (abstract class)
   - `SavingsAccount` and `CheckingAccount` (concrete classes)
   - `Customer` with relationships to accounts
   - `Transaction` class related to accounts
2. Model a state machine for account status
3. Implement in Java using UMPLE
4. Generate code and test the implementation

Learning Outcomes

- Practice UML modeling
- Gain experience with UMPLE

References

- OMG Unified Modeling Language Specification: https://www.omg.org/spec/UML/
- UMPLE User Manual: https://cruise.umple.org/umple/
- Fowler, M. (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2004). The Unified Modeling Language Reference Manual. Addison-Wesley.
- UMPLE GitHub Repository: https://github.com/umple/umple

Next Week

Quiz 2 - covering UML, UMPLE, design patterns, and Java implementations.