MC322 - Object Oriented Programming

Lesson 9.1

Building Objects

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Object Relationships: Inheritance vs. Composition

Inheritance:

- Represents a hierarchical relationship between a parent and a child class.
- Conceptually results in a single class incorporating the behaviors and attributes of the entire inheritance hierarchy.
- An **Employee** is also a **Person**, inheriting all the attributes and methods of the **Person** class.
- An **Employee** object does not simply interact with a **Person** object; it is a **Person**.

Composition:

- Involves using one or more classes to build another class.
- Represents interactions between distinct objects.
- Enables class reuse by delegating behavior to other objects, allowing flexibility.

Composition Relationships

Is-a vs. Has-a Relationships

Composition (Has-a):

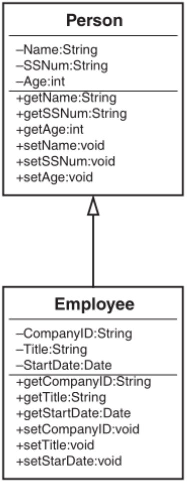
- Represents a part-of-whole relationship.
- The relationship is expressed as **has-a**, meaning that one object is part of another.
- For example, a car **has-a** steering wheel (Figure 9.2).

Inheritance (Is-a):

- Represents a hierarchical relationship between classes.
- The relationship is expressed as **is-a**, meaning that one class is a type of another.
- For instance, an **Employee** is a **Person**.

Is-a and Has-a

Forgive my grammar: I will consistently use "has-a engine," even though "has an engine" is grammatically correct. I do this to state the rules simply as "is-a" and "has-a."



An inheritance relationship.

Benefits of Composition

Simplicity and Abstraction:

- Composition builds systems by combining simpler parts.
- People can only handle about seven chunks of data in short-term memory, so abstraction makes it easier to manage complex concepts.
- Instead of listing all car components (steering wheel, tires, engine), we simply refer to the whole concept as ”car.”

Interchangeability and Reuse:

- Composition enables parts to be interchangeable, increasing flexibility.
- Interchangeable parts in software development lead to reuse.
- For instance, if steering wheels are standardized, it doesn’t matter which one is installed.

Building in Phases

Independent Testing and Maintenance

Advantages of Composition:

- **Independent Development**: Systems and subsystems can be built separately.
- **Independent Testing**: Each part can be tested and maintained independently.
- Composition reduces complexity by breaking software into smaller, manageable parts.

Simplicity in Software:

- To ensure quality software, follow the rule of keeping things as simple as possible.
- Large software systems should be divided into manageable components for better maintainability.

Herbert Simon's Thoughts (1962):

- Nobel Prize Herbert Simon explored stable systems in, **The Architecture of Complexity**.
- He emphasized the importance of breaking down complex systems to create stable, effective architectures.

Herbert Simon’s Principles on Stable Complex Systems

Hierarchy and Functional Decomposition: *“Stable complex systems usually take the form of a hierarchy, where each system is built from simpler subsystems, and each subsystem is built from simpler subsystems still.”*

- The hierarchy concept forms the basis for functional decomposition, which underpins procedural software development.
- In object-oriented design, this same principle applies to composition, where complex objects are built from simpler pieces.

Near Decomposability: *“Stable, complex systems are nearly decomposable.”*

- Systems are identifiable by their parts, distinguishing internal interactions from those between parts.
- Stable systems have fewer links between different parts than within each part itself.
- For instance, a modular stereo system with simple links between components is inherently more stable than an integrated system that isn’t easily decomposable.

More Principles of Stable Complex Systems

Subsystem Composition: *"Stable complex systems are almost always composed of only a few different kinds of subsystems, arranged in different combinations."*

- Subsystems are generally composed of only a few types of parts.
- Different combinations of these parts provide the building blocks for complex systems.

Evolution of Stable Systems: *"Stable systems that work have almost always evolved from simple systems that worked."*

- Building on existing proven designs prevents reinventing the wheel.
- Successful systems are developed iteratively by enhancing and evolving previous designs.

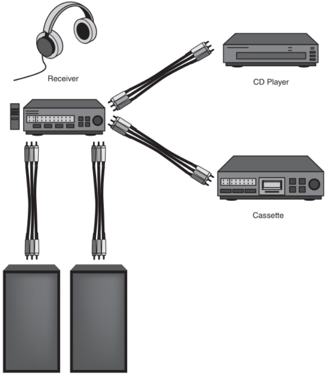
Modular vs. Integrated Systems: Stereo Example

Composition (Has-a): **Integrated System Drawbacks**:

- Imagine the stereo system as a single integrated black-box system.
- If the CD player breaks and becomes unusable, the system must be repaired.
- Repairing the entire system is more complicated and expensive.
- None of the other components can be used during the repair process.

Benefits of Modular Systems:

- Modular systems allow individual components to be repaired or replaced.
- The other stereo parts remain functional, offering more flexibility and usability.



An inheritance relationship.

Component-Based Design: Modular Stereo System

Advantages of Component-Based Systems:

- If the CD player breaks, it can be disconnected and taken for repair independently.
- Other components remain functional, allowing continued usage of the rest of the stereo system.
- Components are connected via patch cords, providing flexibility in replacement or repair.
- Replacement is easy: a new CD player can be purchased and plugged in to the system.
- Repair technicians can test and fix faulty components separately.

General Benefits of Components:

- Reduces complexity by dividing systems into smaller, manageable parts.
- Allows reuse of components developed by other teams or third-party vendors.

Challenges in Using External Components:

- Trust in third-party components requires assurance of reliability and proper testing.
- They must perform the functions advertised accurately.
- Some developers still prefer building their components over using third-party ones.

Types of Composition

Types of Composition: Association and Aggregation

Two Types of Composition:

- **Association**:
  - Represents a collaboration between objects where individual parts are visible.
  - The stereo system example used earlier is an instance of association.
- **Aggregation**:
  - Represents a relationship where the whole is more visible than its individual parts.
- Both aggregation and association are examples of **has-a** relationships.

Is Composition a Form of Association?

In object-oriented (OO) technologies, there's debate over whether composition is a form of association or vice versa. In this book, inheritance and composition are considered the two primary ways to build classes, so association is considered a form of composition.

Types of Composition - Aggregations

Aggregation as a Form of Composition

Definition of Aggregation:

- Aggregation means that a complex object is composed of other objects.
- Represents a **has-a** relationship where the whole is more prominent than the individual parts.

TV Example:

- A TV is an aggregation of transistors, a picture tube, a tuner, and other components.
- People usually perceive the TV as a single, cohesive unit rather than focusing on the individual components.
- When buying a TV, the salesperson doesn't describe it as an aggregation of parts but simply as a TV.

Intuitive Understanding:

- Aggregation is an intuitive form of composition because it's how people typically perceive complex objects in daily life.

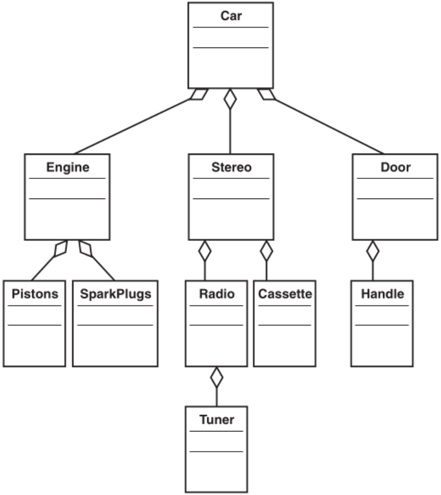
Aggregation Example: Buying a Car

Car Purchasing Analogy:

- When buying a car, you aren't focused on picking each individual component (e.g., spark plugs, door handles).
- Instead, you choose a car as a whole, which is a complex object composed of many simple and complex parts.
- While you can select some optional features, the car is still considered and purchased as a single unit (Figure).

Aggregation in Practice:

- Aggregation simplifies perception by combining multiple components into a unified concept.
- Buyers typically don't think about the individual parts but instead about the final product that fulfills their needs.



An aggregation hierarchy for a car.

Types of Composition - Associations

Aggregation vs. Association

Aggregation:

- Represents relationships where only the whole is usually visible.
- The parts are abstracted away in favor of the overall entity.

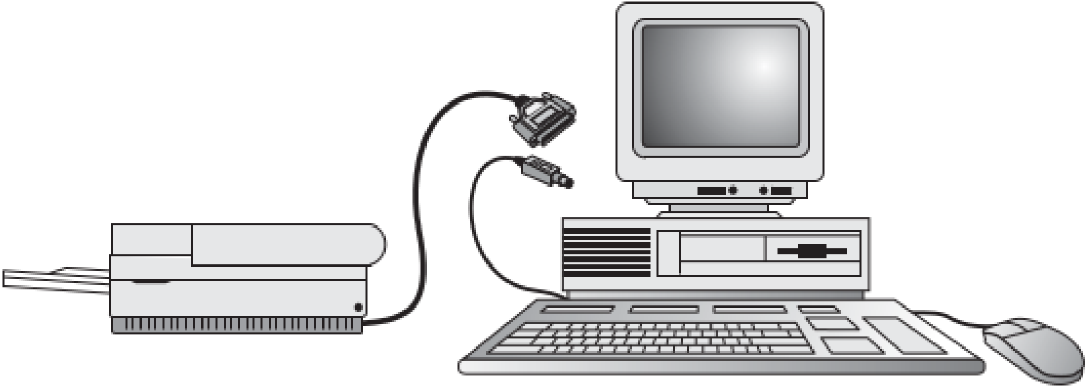
Association:

- Shows both the whole and its individual parts distinctly.
- In the stereo example, each component (like speakers or a tuner) is visible and connected to the whole via patch cords.
- Each stereo component has an interface that can be manipulated independently.

Designing Minimal Interfaces:

- Refer back to Chapter 2, "How to Think in Terms of Objects," for an example on designing for minimal interfaces.

Aggregation vs. Association



Associations.

Using Associations and Aggregations Together

Blurred Lines: Association vs. Aggregation

Distinguishing Association and Aggregation:

- In many cases, the dividing lines between association and aggregation are blurred.
- Interesting design decisions often revolve around whether to use associations or aggregations.

Computer System Example:

- The computer system example includes both association and aggregation.
- The interactions between the computer box, monitor, keyboard, and mouse represent association.
- The computer box itself is an aggregation because it contains complex internal components, such as chips, motherboards, and video cards.
- Despite this internal complexity, we only see the computer box as a whole.

Associations and Aggregations in Different Examples

Employee Example:

- An **Employee** object may have an **Address** object (aggregation) and a **Spouse** object (association).
- If the employee is fired, the spouse remains in the system, but the association is broken.

Stereo Example:

- The relationship between the receiver, speakers, and CD player is an association.
- Each of these components is a complex object made up of other objects, representing aggregation.

Car Example:

- The engine, spark plugs, and doors represent composition within the car.
- The stereo in the car is an association.

No One Right Answer

Design is not an exact science. General rules provide guidance, but there is no single correct answer for every situation.

Avoiding Dependencies

Avoid Mixing Domains in Composition

Desirability of Independence:

- Objects should not be highly dependent on one another to maintain flexibility and ease of maintenance.
- Avoid mixing domains by keeping objects within their respective areas unless necessary.

Stereo System Example:

- The receiver and CD player are maintained in separate domains, making the stereo easier to manage.
- If the CD player breaks, it can be sent for repair separately, without affecting other components.
- The CD player and MP3 player have separate domains, allowing them to be purchased from different manufacturers.
- This separation provides flexibility to swap out the CD player with a model from another brand.

Mixing Domains: TV/VCR Combination Example

Convenience vs. Stability:

- TV/VCR combinations offer the convenience of having both features in one module.
- If the TV breaks, the VCR becomes unusable as part of the integrated unit.
- In some cases, convenience might outweigh the risk of losing unit stability, depending on the application and environment.
- In the TV/VCR example (Figure), the integrated unit provides a significant convenience despite potential downtime.

Mixing Domains

Mixing domains is a design decision. If the convenience of a TV/VCR combination outweighs the risk of downtime, then mixing domains may be the preferred choice.

Cardinality

Cardinality in Object Relationships

Definition of Cardinality:

- The number of objects participating in an association and whether the participation is optional or mandatory.

Questions to Determine Cardinality:

- Which objects collaborate with other objects?
- How many objects participate in each collaboration?
- Is the collaboration optional or mandatory?

Example: Employee Class Relationships:

- **Employee Class**: Inherits from the **Person** class and has relationships with:
  - **Division**
  - **JobDescription**
  - **Spouse**
  - **Child**

Cardinality Example: Division and Job Description

Division Class:

- Contains information about the division the employee works in.
- Each employee must be associated with one division, so the relationship is mandatory.
- An employee works for one, and only one, division.

Job Description Class:

- Contains job-related information, such as salary grade and range.
- Each employee must have at least one job description, making the relationship mandatory.
- An employee can hold multiple job descriptions during their career or have multiple jobs simultaneously (e.g., a supervisor covering for a quitting employee).
- Past job descriptions can be kept as a historical record.

Cardinality Example: Spouse and Child

Spouse Class:

- Contains basic information such as the anniversary date.
- An employee can be married or not, making the spouse relationship optional.
- An employee can only have one spouse.

Child Class:

- Contains basic information like the string `FavoriteToy`.
- An employee can have children or not, making the child relationship optional.
- An employee can have no children or many children, though an upper limit could be set based on system requirements.

Summary of Cardinality: Employee Class Relationships

Table: Represents the cardinality of the associations for the previously discussed classes:

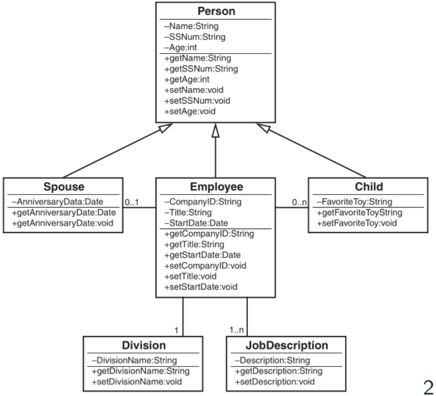
- **Division**: One-to-one, mandatory.
- **Job Description**: One-to-many, mandatory.
- **Spouse**: Zero or one, optional.
- **Child**: Zero to many, optional.

Figure: Shows the class diagram for this system, indicating cardinality along association lines.

Cardinality Notation

*The notation **0...1** means an employee can have either zero or one spouse. The notation **0...n** means an employee can have any number of children from zero to an unlimited number. Here, **n** represents infinity.*

Optional/Association	Cardinality	Mandatory
`Employee/Division`	1	Mandatory
`Employee/JobDescription`	1...n	Mandatory
`Employee/Spouse`	0...1	Optional
`Employee/Child`	0...n	Optional



Multiple Object Associations

One-to-Many Association Example in Code

Representing One-to-Many Relationships: Classes that have one-to-many relationships, such as **Child** and **JobDescription**, are represented by arrays in code.

```java
import java.util.Date;

public class Employee extends Person {
    private String CompanyID;
    private String Title;
    private Date StartDate;
    private Spouse spouse;
    private Child[] child;
    private Division division;
    private JobDescription[] jobDescriptions;

    public String getCompanyID() { return CompanyID; }
    public String getTitle() { return Title; }
    public Date getStartDate() { return StartDate; }
    public void setCompanyID(String CompanyID) {}
    public void setTitle(String Title) {}
    public void setStartDate(int StartDate) {}
}
```

Key Insights: **Child[] child**: Represents the one-to-many relationship between Employee and Child.

JobDescription[] jobDescriptions: Indicates an employee can hold multiple job descriptions.

Optional Associations

Handling Optional Associations in Code

Checking for Optional Associations:

- Your application should handle optional associations by verifying if the relationship is **null**.
- Code must check whether an associated object exists before invoking any methods on it.

Example: Employee and Spouse Association:

- The code should not assume that every employee has a spouse.
- If an employee isn't married and the code expects a spouse, the application could fail and leave the system unstable.
- Code should check if a spouse exists before calling methods on the **Spouse** object.

Proper Handling of Optional Associations:

- If no spouse exists, the code must process the **Employee** object without invoking any spouse-specific methods.
- This ensures that the application remains stable and handles the null condition properly.

Tying It All Together: An Example

System Diagram: Inheritance, Interfaces, Composition

System Overview:

- The diagram (Figure) ties together inheritance, interfaces, composition, associations, and aggregations into a single system.
- The addition of an **Owner** class allows the owner to take the dog for walks.

Inheritance Relationship:

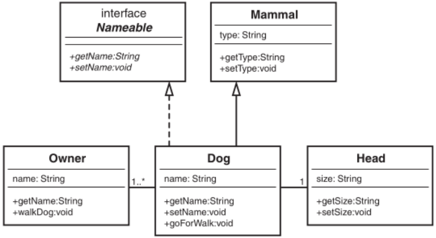
- The **Dog** class inherits directly from the **Mammal** class, indicated by the solid arrow.

Interface Implementation:

- The **Nameable** interface is implemented by the **Dog** class, shown by the dashed arrow from Dog to Nameable.

Composition and Associations:

- The **Owner** class adds an association with the **Dog** class, representing a relationship where the owner takes the dog for walks.
- Aggregation or composition could be used to represent further relationships within the system.



Cardinality in a UML diagram.

Associations and Aggregations in the Dog Class Relationships

Dog and Head (Aggregation):

- The relationship between the **Dog** class and the **Head** class is an aggregation because the head is part of the dog.
- Cardinality specifies that a dog can have only a single head.

Dog and Owner (Association):

- The relationship between the **Dog** class and the **Owner** class is an association since neither is a part of the other.
- The dog requires a service from the owner, which is taking the dog for walks.
- Cardinality specifies that a dog can have one or more owners (e.g., a husband and wife sharing responsibility).

Key OO Relationships:

- Inheritance, interfaces, composition, associations, and aggregations represent the primary design relationships in OO systems.

Conclusion

Chapter Summary and Further Exploration

Summary of Composition Concepts:

- Explored finer points of composition and its two primary types: aggregation and association.
- Inheritance represents a new kind of already-existing object, while composition represents interactions between different objects.

Inheritance and Composition Basics:

- The last three chapters covered inheritance and composition fundamentals.
- With these concepts, you're equipped to design robust classes and object models.

Further Exploration:

- This book provides a high-level overview of the object-oriented (OO) thought process.
- Seek other books for in-depth exploration of topics like UML and use cases, which have entire texts devoted to them.
- May this overview inspire you to dive deeper into OO design concepts. Good hunting!

MC322 - Object Oriented Programming

Lesson 9.1

Building Objects

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP