# AMSI-TUTORIALS

Mikel Vandeloise

2025-08-13

# Table of contents

# Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

# 1 Introduction to Modelling

This chapter introduces the principles of modelling in the context of information systems engineering. By the end of this section, you will be able to define what a model is, identify the qualities of a good model, and understand why modelling is an indispensable activity in modern software engineering.

### 1.0.1 The Central Challenge: Managing Complexity

As software systems become increasingly complex, the primary challenge for engineers is not just writing code, but managing the immense complexity of the systems they build. How do we ensure that a large-scale system is correct, robust, and aligned with user needs before investing thousands of hours in implementation? This is the core problem that modelling addresses.

The following quotes highlight the human-centric and real-world implications of this challenge:

> "Concern for man himself and his fate must always form the chief interest of all technical endeavors. […] Never forget this in the midst of your diagrams and equations."
>
> — **Albert Einstein**

> "In Software Engineering people often believe that a state is a node in a graph and do not even care about what a state means in reality."
>
> — **David L. Parnas**

These reflections guide our approach: modelling is not an abstract exercise but a critical tool for building systems that are both technically sound and meaningful in the real world.

## 1.1 What is Modelling?

In software engineering, "modelling" is often misunderstood and reduced to clichés: drawing diagrams because a process demands it, or simply creating documentation.

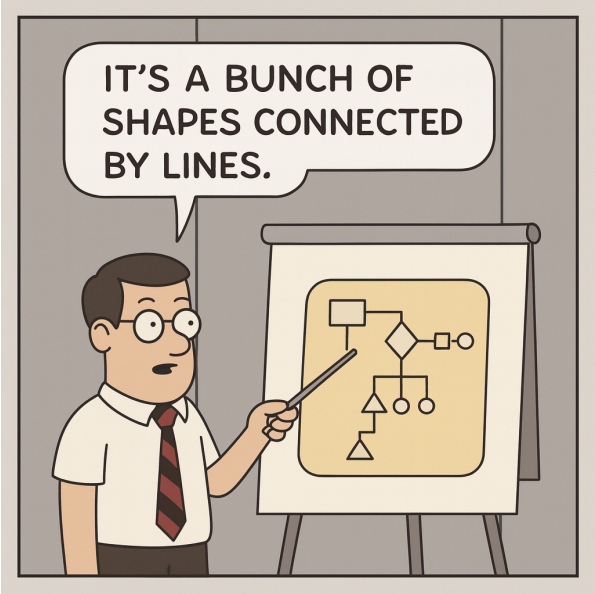



Figure 1.1: A cliché representation of modelling, with complex and confusing shapes connected by arrows.

In reality, **modelling is a rigorous intellectual activity focused on understanding and communicating complex systems**.

### 1.1.1 The Problem Space vs. The Solution Space

A critical distinction in advanced modelling is the separation between two domains:

- **The Problem Space:** This refers to the real-world environment, with all its complexities, rules, and stakeholders. A model in this space aims to understand and formalize the "what" — what is the problem to be solved?
- **The Solution Space:** This refers to the computational system we intend to build. A model in this space specifies the "how" — how will our software be structured and behave to solve the problem?

Effective modelling involves creating a clear and verifiable bridge between these two spaces.

### 1.1.2 A More Formal Definition

According to Jeff Rothenberg, a model is a simplification of reality that allows us to reason about the world in a more manageable way. Building on this, we can define a model as:

- A **simplification** of a real or imagined reality…
- …that captures its **essential properties**…
- …to help us **understand and reason** about it…
- …for a specific **cognitive purpose** (e.g., analysis, simulation, code generation).

A good model must not only be an abstraction but also possess qualities of the original system, allowing us to perform meaningful analysis and make predictions about the system's behaviour.

## 1.2 The Role of Models in Engineering

In any engineering discipline, building a complex system without prior modelling is unthinkable. Before investing significant resources, engineers create and learn from models to manage complexity and ensure the final product meets its requirements. In software engineering, models serve three critical roles throughout the development lifecycle:

### 1.2.1 Models for Analysis (Understanding the Problem)

Models are first and foremost tools for thought and communication. They help us analyze the problem space by:

- **Clarifying Requirements:** Translating ambiguous client needs into a precise, structured representation that can be discussed and validated.
- **Exploring the Domain:** Capturing the essential concepts, rules, and relationships of the business domain, ensuring the development team shares a common understanding.

### 1.2.2 Models for Synthesis (Designing the Solution)

Once the problem is understood, models guide the design of the solution space. They serve as blueprints for construction, helping to:

- **Architect the System:** Defining the high-level structure, components, and interactions of the software.
- **Plan Development:** Providing a clear plan for implementation, allowing for better task allocation and project management.

### 1.2.3 Models for Validation (Mitigating Risks)

Finally, models allow us to validate our design choices before writing a single line of production code, thereby mitigating risks. This can be done through:

- **Simulations and Walkthroughs:** "Executing" the model mentally or with tools to identify logical flaws or unintended consequences.
- **Formal Verification:** Applying mathematical techniques to prove that a model satisfies certain critical properties (e.g., security, safety).

## 1.3 The Spectrum of Modelling Languages

To describe an information system, engineers use various languages that fall along a spectrum, balancing the trade-off between intuitive expressiveness and formal precision. Choosing the right language depends on the context, the audience, and the goals of the model.

### 1.3.1 Informal Languages

These languages prioritize ease of communication and are accessible to all stakeholders, including non-technical clients.

#### 1.3.1.1 Natural Language (e.g., English, French)

Natural language is the default for initial requirements gathering.

- **Strengths:** Universally understood, requires no special training.
- **Weaknesses:** Highly prone to what are known as the "7 deadly sins" of specification: ambiguity, contradiction, vagueness, noise (irrelevant information), silence (missing information), over-specification, and wishful thinking. These issues make it unsuitable for detailed and rigorous system design.

#### 1.3.1.2 Ad-hoc Notations (e.g., whiteboard sketches)

These are informal diagrams drawn without strict rules.

- **Strengths:** Excellent for brainstorming and collaborative sessions; highly flexible and fast to create.
- **Weaknesses:** Lacks a well-defined syntax and semantics. The meaning of a symbol can be interpreted differently by each person, leading to significant ambiguity and making it impossible to automate any analysis.

### 1.3.2 Formal and Semi-Formal Languages

As we move toward implementation, the need for precision increases, requiring languages with well-defined rules.

#### 1.3.2.1 Semi-Formal Notations (e.g., UML)

This is the dominant category in modern software engineering. UML (Unified Modelling Language) is the industry standard.

- **Strengths:** Provides a well-defined visual syntax that is more intuitive than purely formal languages. It offers a partial common interpretation (semantics) and supports partial automation, such as code generation and model checking. The learning curve is relatively fast.
- **Weaknesses:** While the syntax is standardized, UML's semantics can sometimes remain ambiguous, leaving room for interpretation in complex scenarios.

#### 1.3.2.2 Formal Notations (e.g., Z, VDM, Alloy)

These languages are based on mathematical principles, providing the highest level of precision.

- **Strengths:** Possess a well-defined syntax and an unambiguous mathematical semantics. This rigor eliminates ambiguity and allows for extensive automation, including formal verification and proof of correctness.
- **Weaknesses:** Their mathematical nature makes them difficult for non-experts to read and requires a significant learning investment. They are typically used for safety-critical or mission-critical systems where absolute correctness is paramount.

### 1.3.3 Summary Comparison

| Language Type | Syntax | Semantics | Ambiguity | Tool Support |
|---|---|---|---|---|
| **Natural Language** | Undefined | Informal | Very High | Low |
| **Ad-hoc** | Undefined | Informal | High | Very Low |
| **Semi-Formal (UML)** | Defined | Partially Formal | Low | High |
| **Formal (Z, VDM)** | Defined | Formal | Very Low | Very High |

## 1.4 Qualities of a Good Model

To be effective, a model must be more than just a picture; it must possess specific qualities that make it useful for its intended purpose. The famous statistician George E.P. Box aptly noted:

> "Essentially, all models are wrong, but some are useful."
>
> — **George E.P. Box**

The usefulness of a model can be evaluated across three distinct dimensions: its syntax, its semantics, and its pragmatics.

### 1.4.1 Syntactic Quality: Is the Model Well-Formed?

This dimension concerns the structure and comprehensibility of the model itself, independent of what it represents.

- **Understandable:** The model must use a notation that is clear and intuitive for its intended audience, whether they are domain experts, developers, or clients. A model that cannot be easily understood fails its primary purpose of communication.

### 1.4.2 Semantic Quality: Does the Model Represent Reality Faithfully?

This is about the relationship between the model and the system it represents. The goal is to ensure the model is a truthful representation.

- **Abstract:** A model must be an effective abstraction, focusing on the essential aspects of the system while intentionally omitting irrelevant details. This is the primary mechanism for managing complexity.
- **Precise:** Precision is a measure of how faithfully the model reflects reality. It encompasses several sub-qualities:
  - **Correctness:** The model does not contain any information that is false with respect to the system.
  - **Completeness:** The model includes all relevant information for its purpose.
  - **Lack of Ambiguity:** The statements made by the model have one and only one interpretation.

### 1.4.3 Pragmatic Quality: Is the Model Useful for its Purpose?

This dimension evaluates the model's utility in practice. A model can be well-formed and accurate but still be useless if it doesn't serve its purpose.

- **Predictive:** The model must allow us to deduce non-trivial properties about the system it represents. It should be a tool for analysis and reasoning, helping us answer "what if" questions.
- **Inexpensive:** The cost of creating, analyzing, and maintaining the model must be significantly lower than the cost of experimenting with the actual system. If the model is as complex or expensive as the system itself, it loses its pragmatic value.

# 2 Introduction to UML

This chapter provides an overview of the Unified Modelling Language (UML), the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

## 2.1 UML in a Nutshell

UML is a **general-purpose visual Modelling language** designed to provide a standard way to visualize the design of a system. It is crucial to understand what UML is and what it is not:

- **Unified:** It is an industry standard managed by the Object Management Group (OMG), an international consortium of major software companies like IBM and Microsoft. Its goal is to be the *lingua franca* for business analysts, software architects, and developers.
- **Modelling:** It is primarily a graphical language used to create models of both application domains (the problem) and software systems (the solution).
- **Language:** It provides a set of notations (diagrams) but is **not a method or a process**. A process provides guidance on the order of activities and the artifacts to be developed, whereas UML is intentionally process-independent.

Despite some criticisms, UML remains the undisputed leader in software Modelling, supported by a vast number of tools and methods.

## 2.2 History and Philosophy

UML was created in the mid-1990s from the unification of several popular object-oriented modelling methods, most notably those of Booch, Rumbaugh (OMT), and Jacobson (OOSE), often called the "three amigos". It was standardized by the OMG in November 1997 (UML 1.1) and has evolved significantly since then, with the current version being UML 2.5.

The philosophy behind UML is one of **unification by union, rather than intersection**. This means it aims to be highly versatile and capable of representing entire systems across various domains like real-time systems, banking, and scientific applications.

This versatility, however, comes with a trade-off:

- **Pros:** UML is a well-accepted standard, versatile, and supported by many tools.
- **Cons:** It can be overly complex, with "too many constructs for some concerns and missing ones for others". It also has a relative lack of formality and coordination between its various notations, which can lead to ambiguity.

## 2.3 UML Diagrams: An Overview

UML provides a wide range of diagrams to model different aspects of a system. These are broadly categorized into two main types: **Structure Diagrams** and **Behaviour Diagrams**.

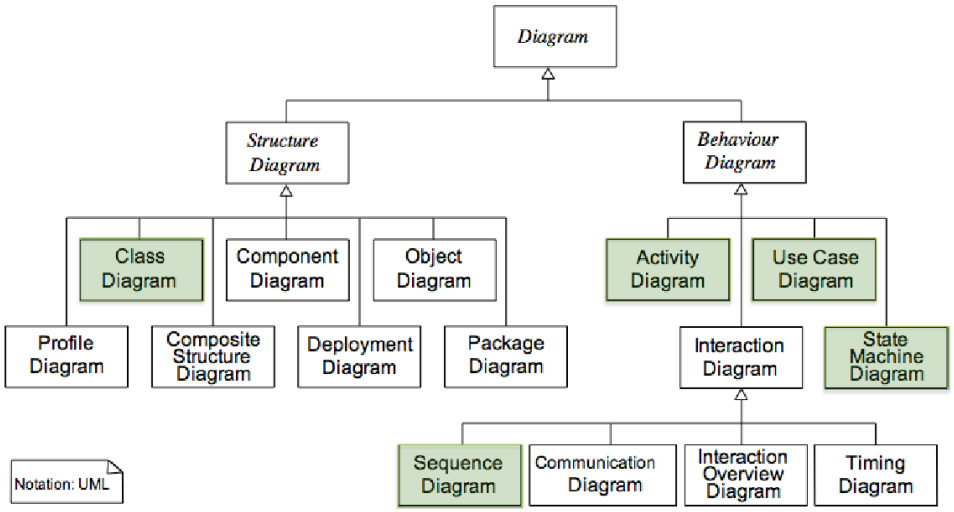



Figure 2.1: A diagram showing the hierarchy of UML diagrams, separating Structure and Behaviour diagrams.

### 2.3.1 Structure Diagrams (Static View)

These diagrams depict the static structure of the system, its components, and the relationships between them at different levels of abstraction. They represent the elements that must be present in the system being modelled.

- **Class Diagram:** The cornerstone of object-oriented modelling. It describes the structure of a system by showing its classes, attributes, operations, and the relationships among them.
- **Component Diagram:** Depicts how a software system is split into modular components and shows the dependencies and interfaces among these components.

- **Deployment Diagram:** Models the physical deployment of artifacts on nodes. It shows the hardware of your system and the software deployed on that hardware.
- **Object Diagram:** Shows a snapshot of the instances in a system, the objects and their relationships, at a particular point in time. It is often considered a special case of a class diagram.
- **Package Diagram:** Organizes elements of a system into groups (packages) to manage complexity. It shows the dependencies between these packages.
- **Composite Structure Diagram:** Shows the internal structure of a class and the collaborations that this structure makes possible.
- **Profile Diagram:** A specialized diagram used for extending the UML metamodel for specific platforms or domains.

### 2.3.2 Behaviour Diagrams (Dynamic View)

These diagrams illustrate the dynamic behaviour of the system, focusing on how its components interact and change over time. They represent what happens in the system being modelled.

- **Use Case Diagram:** Describes the high-level functions of a system from an external point of view, showing how users (actors) interact with it to achieve specific goals.
- **Activity Diagram:** Shows the flow of control or data through a system, useful for Modelling business processes, complex operational workflows, or algorithms.
- **State Machine Diagram:** Models the behaviour of a single object, specifying the sequence of states it goes through during its lifetime in response to events.
- **Sequence Diagram:** An interaction diagram that emphasizes the time-ordering of messages between objects or components. It is particularly useful for understanding the chronology of method calls.
- **Communication Diagram:** (Formerly Collaboration Diagram) Another type of interaction diagram that emphasizes the structural organization of the objects that send and receive messages, rather than the time-ordering.
- **Interaction Overview Diagram:** A hybrid of an Activity Diagram and Sequence Diagrams. It provides a high-level view of the control flow of an interaction, where individual steps can be represented as more detailed sequence diagrams.
- **Timing Diagram:** An interaction diagram that focuses on the timing constraints of messages, showing how the state of an object changes over a specific timeline.

While this course will not cover every diagram in detail, this complete overview provides the necessary context for understanding the full scope and power of UML. Our focus will remain on a core subset most critical for the analysis and design phases.

# 3 Class and Object Diagrams

This chapter focuses on the cornerstone of UML's structural modelling: the **Class Diagram**. We will explore its fundamental components, the different types of relationships between classes, and its practical applications in both the analysis and design phases of software development. We will also briefly introduce the **Object Diagram**, which provides a snapshot of a class diagram's instances at a specific moment in time.

## 3.1 Overview of Class Diagrams

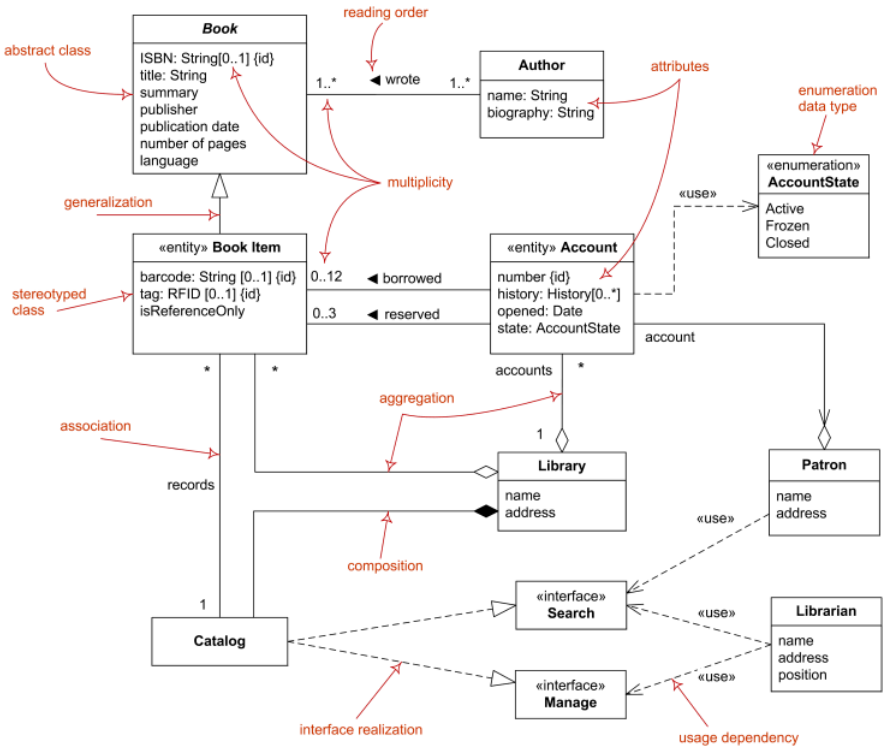
A UML Class Diagram specifies the **static structure** of a system. It is the most common and essential diagram in object-oriented modelling, with its origins in earlier formalisms like Entity-Relationship-Attribute (ERA) models and the Object-modelling Technique (OMT).

Its primary purpose is to show the system's classes, interfaces, their attributes and operations, and the relationships between them.

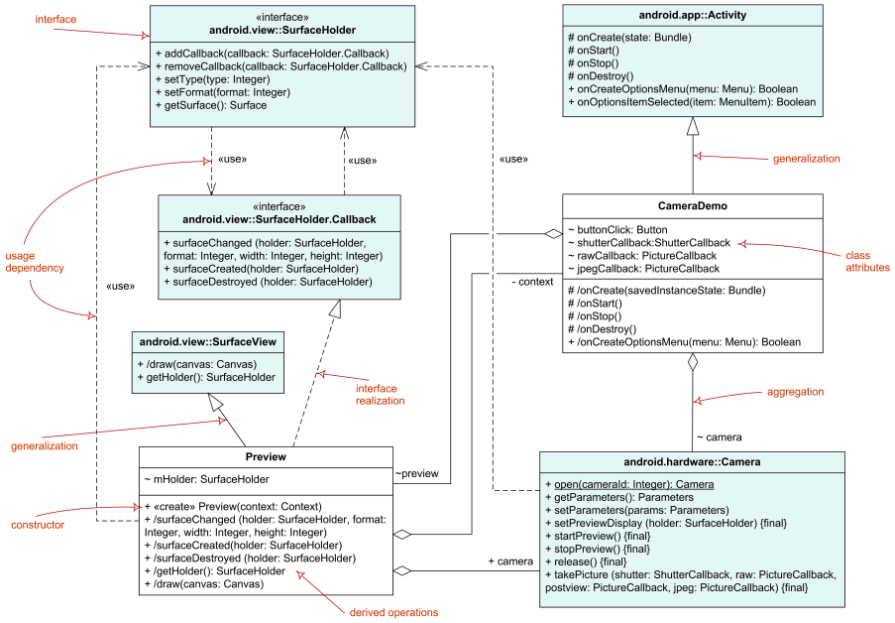
### 3.1.1 Multi-Purpose Usage

Class diagrams are versatile and used throughout the software lifecycle. Their purpose often determines which constructs are used and how they are interpreted:

- **Analysis (Conceptual Modelling):** In this phase, class diagrams are used to model the concepts of the application domain. These models, often called **Domain Models**, focus on real-world entities and their relationships, deliberately omitting software-specific details like operations or visibility.




- **Design (Software Specification):** During design, the focus shifts to the solution space. Class diagrams, now called **Diagrams of Implementation Classes**, specify the software classes, including their attributes, operations (methods), and visibility. They serve as a blueprint for implementation, and class skeletons can often be generated directly from them.




## 3.2 Core Concepts: The Blueprint and the Product

The object-oriented approach is built upon two fundamental concepts: the **class** and the **object**. The easiest way to understand their relationship is through an analogy: a class is like a **cookie cutter**, and an object is the **cookie** it creates.

### 3.2.1 The Class: A Blueprint or Template

A **class** is an abstraction that serves as a blueprint for creating objects. It defines a common structure and behaviour that all objects of that type will share. Just as a cookie cutter defines the shape and pattern for all cookies made from it, a class defines:

- **Properties (Attributes):** The data or characteristics that each object will have (e.g., a `User` class defines that every user will have a `name` and an `email`).
- **behaviour (Operations):** The actions or functions that each object can perform (e.g., a `User` class defines that every user can `login()` or `logout()`).

> **i** Formal Definition
>
> A **class** is an abstraction that describes a group of objects with common properties (attributes), behaviour (operations), and relationships.

### 3.2.2 The Object: A Concrete Instance

An **object** is a concrete instance of a class. It's a specific "thing" that exists in the system, created from the blueprint defined by its class. Just as you can use one cookie cutter to make many individual cookies, you can use one class to create many objects.

Every object has:

- **Identity:** It is a unique entity, distinct from all other objects, even those of the same class (e.g., two different `User` objects, `user1` and `user2`).
- **State:** It holds specific values for the attributes defined in its class (e.g., `user1`'s name is "Alice", while `user2`'s name is "Bob").
- **Lifespan:** It exists in the system for a certain period, from its creation to its destruction.

> **i** Formal Definition
>
> An **object** represents a specific "thing" in the modelled world that has a unique identity, a state, and a lifespan. Every object is an instance of a class.

## 3.3 Anatomy of a Class: A Comparative Example

A class in UML is rendered as a rectangle with three compartments: Name, Attributes, and Operations. To understand how these components are used in practice, we will compare two common but fundamentally different design patterns: a **stateless utility class** (`Math`) and a **stateful service class** (`SearchService`).

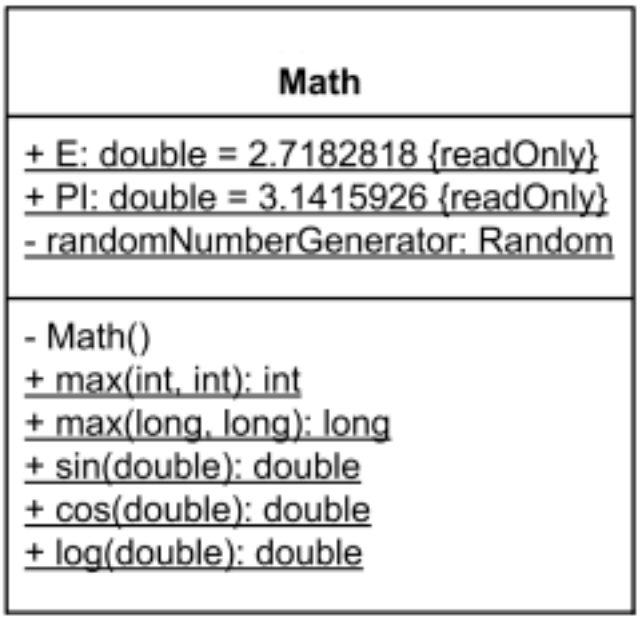



Figure 3.1: A UML diagram of the Math utility class.

A utility class groups related functions but has no state. It is not meant to be instantiated.

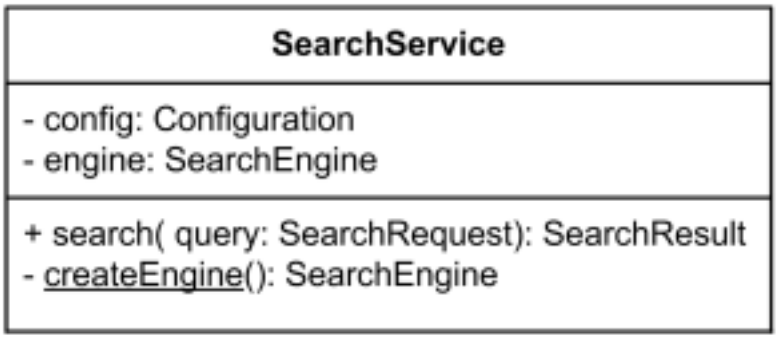



Figure 3.2: A UML diagram of the SearchService class.

A service class encapsulates data (state) and provides operations that act on that data. It is designed to be instantiated.

### 3.3.1 Attributes: State vs. Constants

An attribute represents a property of a class. Our two examples illustrate the critical difference between class-level constants and instance-level state.

- In the `Math` **class**, attributes like `+ PI: double = 3.14` are **constants**. They are public, read-only, and their values are shared across the entire system. In a strict UML diagram, they would be underlined to indicate they are **static**.
- In the `SearchService` **class**, attributes like `- config: Configuration` and `- engine: SearchEngine` are **instance attributes**. This means every `SearchService` object has its *own* configuration and search engine. This is the internal **state** of the object, which is kept `private` (`-`) to ensure **encapsulation**.

### 3.3.2 Operations: Static Functions vs. Instance Methods

An operation is a service that can be requested. This is where the difference between our two classes is most apparent.

- In the `Math` **class**, all operations like `+ sin(double): double` are **static** (underlined). They are self-contained functions that don't depend on any object's state. They are called on the class itself (e.g., `Math.sin(x)`).
- In the `SearchService` **class**, the main operation `+ search(...)` is an **instance method**. It relies on the internal state of the object (e.g., the `engine` attribute) to perform its function. You must create an instance of the class to call it (e.g., `mySearchService.search(...)`).

### 3.3.3 Constructors and Object Creation

The constructor is a special operation that creates an object. Its visibility reveals the intended use of the class.

- The `Math` **class** has a **private constructor** (`- Math()`). This is a deliberate design choice to **prevent instantiation**. You cannot create a `Math` object, reinforcing its role as a purely static utility class.
- The `SearchService` **class** would have a **public constructor** (often implicit if not drawn). This allows users to create multiple instances of the service, each with its own state.

By comparing these two examples, we can see how the same UML building blocks, classes, attributes, and operations, can be used to model fundamentally different design patterns, each suited for a different purpose.

## 3.4 Relationships Between Classes

Relationships describe the connections between classes.

### 3.4.1 Association

An association is a **structural relationship** that specifies that objects (instances) of one class are connected to objects of another. It is the most fundamental way to represent connections in a class diagram.

An association is typically drawn as a **solid line** connecting two classes. Several adornments can be added to this line to specify the relationship with greater precision.

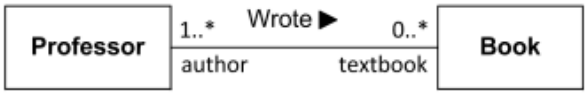



Figure 3.3: A UML diagram showing a detailed association between Professor and Book.

Let's break down the components of an association using the example above:

- **Association Name:** The name of the association is typically a verb that describes the relationship. It is placed near the middle of the association line. In our example, the name is `Wrote`. A small triangle can be added to indicate the reading direction (e.g., "Professor Wrote Book").
- **Association Ends (Roles):** Each end of the association connects to a class and represents a "role" that the class plays in the relationship. These ends can have several properties:
  - **Role Name:** An optional name describing the role played by the class at that end. For example, the `Professor` class plays the role of `author`, and the `Book` class plays the role of `textbook`.
  - **Multiplicity:** This is a mandatory constraint that specifies how many instances of the class can participate in one instance of the association. In the example, `1..*` on the `author` side means a `Book` must be written by at least one `Professor`. The `0..*` on the `textbook` side means a `Professor` can have written zero or many `Book`s.
  - **Navigability:** This specifies whether instances of one class can be efficiently accessed from an instance of the class at the other end of the association. It is a critical concept for bridging the gap between design models and implementation. The notation is as follows:
    * An **open arrowhead** on an association end indicates that it is **navigable**.
    * A small **'x'** on an end indicates it is explicitly **not navigable**.
    * **No adornment** on an end means that navigability is **unspecified**.

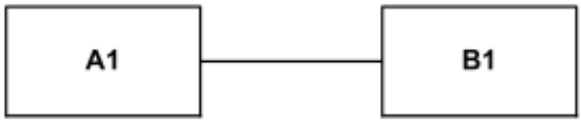



Figure 3.4: Both ends of the association have unspecified navigability.

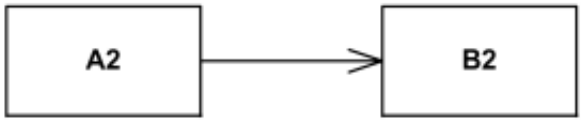



Figure 3.5: A2 has unspecified navigability while B2 is navigable from A2.

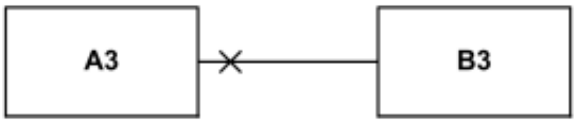



Figure 3.6: A3 is not navigable from B3 while B3 has unspecified navigability.

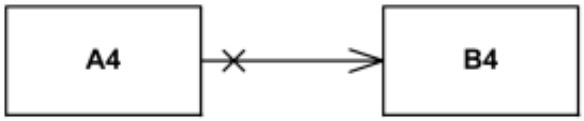



Figure 3.7: A4 is not navigable from B4 while B4 is navigable from A4.

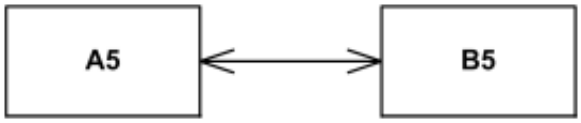



Figure 3.8: A5 is navigable from B5 and B5 is navigable from A5.

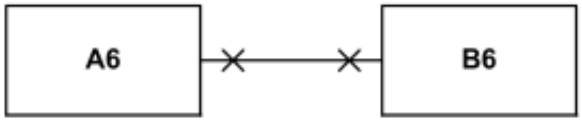



Figure 3.9: A6 is not navigable from B6 and B6 is not navigable from A6.

It is important to understand the sometimes confusing semantics behind non-navigability in the UML specification. While one definition states that a

non-navigable end means "access from the other ends may or may not be possible, and if it is, it might not be efficient," this can be functionally ambiguous.

A more formal, though sometimes contradictory, definition in UML links navigability to ownership: an end is considered navigable if it is owned by the opposite class. This highlights an advanced concept where a navigable role is often implemented as an attribute in the opposite class.

- **Ownership:** An association end can be formally "owned" by either the class at the opposite end or by the association itself. This is an advanced concept that hints at the implementation. Ownership by the opposite class is indicated by a small **filled circle (dot)** at the end of the line. This notation implies that the role will be implemented as an attribute in the owning class.

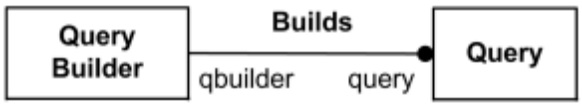



Figure 3.10: An example of ownership in a UML association.

In the example above, the dot on the `Query` end signifies that the `query` role is owned by the `Query Builder` class. This suggests that the `QueryBuilder` class will contain an attribute named `query` of type `Query`.

### 3.4.2 Aggregation: The "Has-A" Relationship

Shared aggregation, commonly known as **Aggregation**, represents a "weak" whole-part relationship. It signifies that a composite object groups together a set of part instances, but these parts can exist independently of the whole.

Aggregation has the following key characteristics:

- It is a **binary** and **asymmetric** relationship; only one end of the association (the "whole") can be marked as an aggregate.
- The relationship is **transitive**, meaning aggregation links must form a directed, acyclic graph. An object cannot be a direct or indirect part of itself.
- The "part" can be included in **several** composites simultaneously, and if the composite "whole" is deleted, the part may still exist.

**Notation:** Aggregation is depicted as an association with a **hollow diamond** at the "whole" or aggregate end of the line.

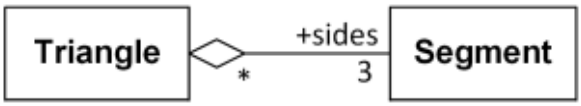



Figure 3.11: A UML diagram showing that a Triangle is an aggregation of three Line Segments.

In the example above, a `Triangle` is an aggregate of exactly three `Segment`. The `*` multiplicity at the `Triangle` end signifies that a `Segment` can be a part of multiple triangles or none at all. Deleting a `Triangle` object does not delete the `Segment` objects.

#### 3.4.2.1 Common Mistakes with Aggregation

{#sec-common-mistakes}

It is crucial to use aggregation correctly to avoid creating logically inconsistent models. Here are two common mistakes to avoid:

**Mistake 1: Marking both ends as an aggregate.** Aggregation is an asymmetric relationship. The UML specification does not allow a diamond on both ends of an association line. This is an incorrect attempt to model a many-to-many relationship where, for example, a `Student` has a list of `Course` and a `Course` has a list of `Student`. The correct way to model this is with a simple association or an association class.


Figure 3.12: An incorrect UML diagram with aggregation diamonds on both ends of an association.

**Mistake 2: Creating cyclic relationships.** Aggregation links must form a directed, acyclic graph. This means a composite object cannot be a direct or indirect part of itself. Creating a cycle where `Student` is part of `Course` and `Course` is part of `Student` is a logical error.


Figure 3.13: An incorrect UML diagram showing a cyclic aggregation.

### 3.4.3 Composition: The "Owns-A" Relationship

**Composition** represents a "strong" form of aggregation with co-incident lifetime of the parts with the whole. It's a whole/part relationship where a part can belong to **at most one** composite (whole) at a time.

**Notation:** Composition is depicted as an association with a **filled black diamond** at the aggregate (whole) end.


Figure 3.14: A UML diagram showing that a Folder is a composition of multiple Files.

Composition has the following key characteristics:

- **Existential Dependency:** If a composite (whole) is deleted, all of its composite parts are normally deleted with it. In the example above, if a `Folder` is deleted, all `File`s it contains are also deleted.
- **Exclusive Ownership:** A part can only belong to one whole at a time. A `File` cannot be in two different `Folder` simultaneously.
- **Figurative Interpretation:** In domain models, the concept of "deletion" should be interpreted figuratively. For example, if a `Hospital` is composed of `Department`, closing the hospital also implies closing all its departments, even if the objects are not physically destroy.

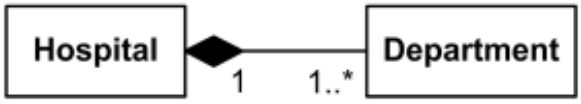



Figure 3.15: A UML diagram showing that a Hospital is a composition of multiple Departments.

It's also important to note that the UML specification does not dictate *how* or *when* the parts of a composite are created. Furthermore, a part can sometimes be removed from a composite before the whole is deleted, allowing it to survive independently in specific cases.

Interestingly, the multiplicity of the composite (whole) end can be `0..1`, which means that a part is allowed to exist as a "stand-alone" instance, not owned by any specific composite at a particular time. Because Composition is a strong form of aggregation, it is subject to the same structural rules, and the common mistakes, previously discussed, apply to it as well.

### 3.4.4 Generalization (Inheritance)

Generalization is a directed, taxonomic relationship between a more general classifier (the **superclass**) and a more specific classifier (the **subclass**). Each instance of the subclass is also considered an instance of the superclass, which is why this is informally known as an **"Is A"** relationship (e.g., a `Patient` is a `Person`).

**Notation:** Generalization is shown as a solid line with a **hollow triangle arrowhead** pointing from the subclass to the superclass.

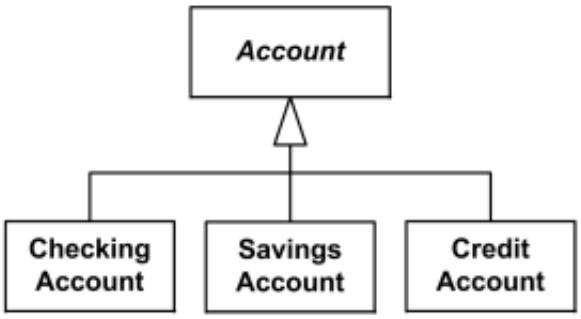



Figure 3.16: A UML diagram showing Checking, Savings, and Credit Account classes inheriting from a general Account class.

#### 3.4.4.1 Inheritance Mechanism

Generalization is the conceptual relationship, while **inheritance** is the mechanism that implements it. Through inheritance, a subclass incorporates the structure (attributes) and behaviour (operations) of its superclass. In UML, this means the subclass inherits the features of the more general classifier, and any constraints applying to the superclass also apply to the subclass.

##### 3.4.4.1.1 Multiple Inheritance

UML implicitly allows a class to have more than one superclass, a concept known as **multiple inheritance**. This is often used when combining orthogonal classification schemes (e.g., classifying an `Employee` by contract type and by role).

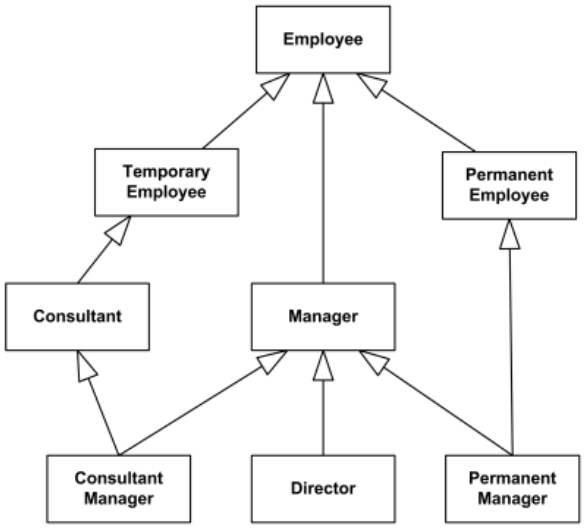



Figure 3.17: A diagram illustrating the multiple inheritance.

However, multiple inheritance introduces complexities, most famously the **"diamond problem."** If a class inherits from two superclasses that both provide an implementation for the same operation, it becomes ambiguous which implementation the subclass should use. UML does not define a standard resolution for this issue, leaving it to the implementation language (e.g., Java prohibits multiple inheritance of classes to avoid this).

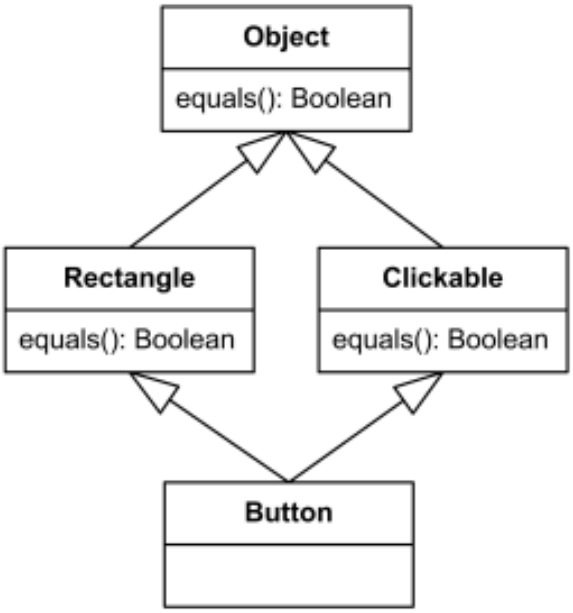



Figure 3.18: A diagram illustrating the diamond problem in multiple inheritance.

#### 3.4.4.2 Generalization Sets

To add more precision to classification hierarchies, UML provides the **Generalization Set**. This is a way to group related generalization relationships and apply constraints to them. The two main constraints are:

- **{complete} vs. {incomplete}:**
  - **`{complete}`:** Specifies that every instance of the superclass **must** also be an instance of at least one of the subclasses in the set. There are no "standalone" instances of the superclass.
  - **`{incomplete}`:** Allows for instances of the superclass that do not belong to any of the specified subclasses. This is the default.
- **{disjoint} vs. {overlapping}:**
  - **`{disjoint}`:** Specifies that an instance of the superclass can be an instance of **at most one** of the subclasses in the set.
  - **`{overlapping}`:** Allows an instance of the superclass to be an instance of **multiple** subclasses in the set simultaneously.

A generalization set that is `{disjoint, complete}` is known as a **partition**. This means every instance of the superclass is an instance of exactly one of the subclasses.

These constraints are written near the generalization arrowhead, often next to a dashed line connecting the generalization arrows.

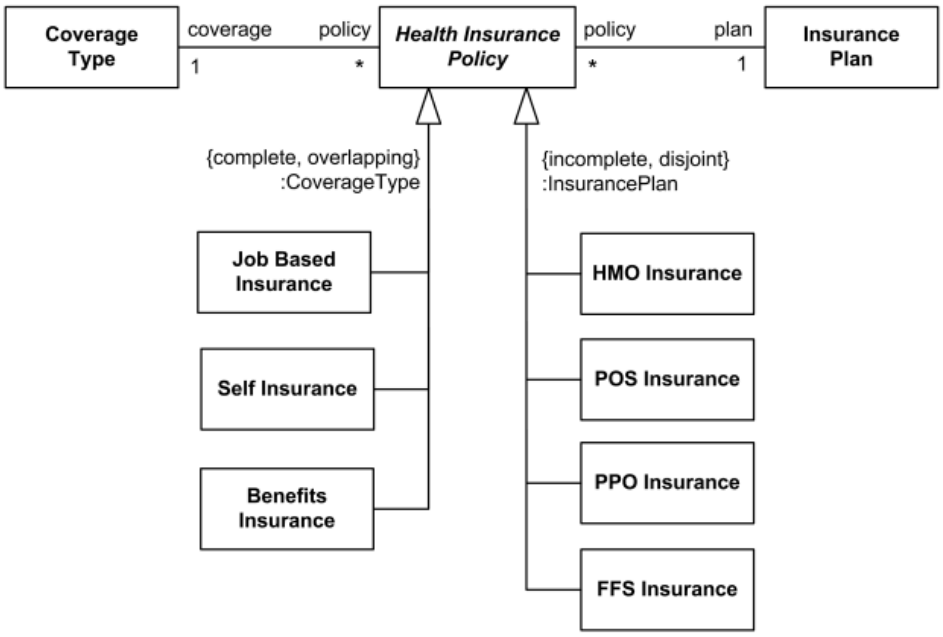



Figure 3.19: An example of generalization sets for a Health Insurance Policy.

In the example above, the `Health Insurance Policy` has two different generalization sets:

1. The set based on `CoverageType` is **`{complete, overlapping}`**. This means every policy must have at least one coverage type, and a single policy could potentially be both a `Job Based Insurance` and a `Self Insurance`.
2. The set based on `InsurancePlan` is **`{incomplete, disjoint}`**. This means a policy can be at most one type of plan (e.g., either `HMO` or `PPO`, but not both), and there might be other types of plans not listed in the diagram.

## 3.5 Object Diagrams: A Snapshot of Reality

While a class diagram models the abstract *types*, an **Object Diagram** shows a snapshot of the concrete *instances* in a system at a particular point in time. It provides a real-world example of the structures defined in a class diagram.

An object diagram shows:

- **Objects:** Instances of classes.

- **Slots:** The current values for the attributes of each object.
- **Links:** Instances of the associations between classes.

Object diagrams are invaluable for illustrating complex class diagrams and validating that your abstract model can represent real-world scenarios correctly.

### 3.5.1 Representing Instances

In UML, objects are rendered as **instance specifications**. The notation is flexible, allowing you to show as much or as little detail as necessary.

The most complete notation is `InstanceName : ClassName :: Namespace`, with all the name underlined.

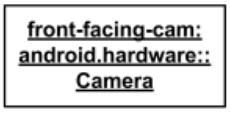



Figure 3.20: A UML diagram showing an object.

Here are the common variations:

- **Named Instance:** `order12 : Order` - An object named `order12` of the `Order` class.
- **Anonymous Instance:** `:Order` - An anonymous (unnamed) object of the `Order` class.
- **Instance of Unknown Class:** `newPatient :` - An object named `newPatient` whose class is either unknown or not specified.
- **Fully Specified Instance:** `front-facing-cam : android.hardware :: Camera` - Shows the instance name, class, and the package it belongs to.

### 3.5.2 Showing an Object's State (Slots)

The primary purpose of an object diagram is to show the state of objects at a specific moment. This is done by listing **slots**, which are the attribute names followed by an `=` and their current value.

The type of the attribute can also be specified.

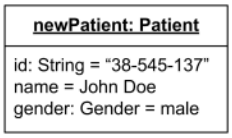



Figure 3.21: An object named ‘newPatient’ of class ‘Patient’, showing the values for its id, name, and gender attributes.

In the example above, the `newPatient` object shows the following slots:

- The attribute `id`, of type `String`, has the value `"38-545-137"`.
- The attribute `name` has the value `"John Doe"`.
- The attribute `gender`, of type `Gender`, has the value `"male"`.

This notation provides a clear and concise snapshot of an object's state at a point in time.

### 3.5.3 Putting It All Together: A Complex Example

Object diagrams truly shine when they are used to visualize a snapshot of a complex, real-world system. They serve as a powerful tool to validate that the static model (the class diagram) can accurately represent the dynamic state of the application at runtime.

The diagram below shows a snapshot of a user authentication module at a specific moment:

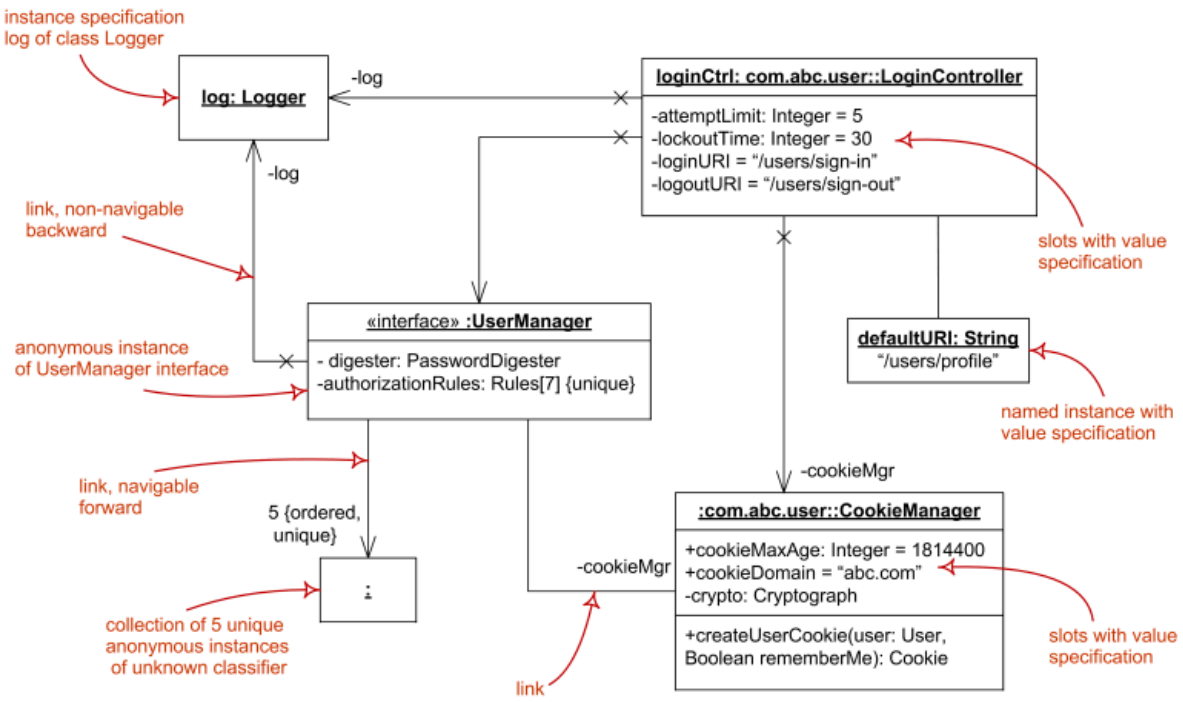



Figure 3.22: A complex object diagram showing various instances like LoginController, UserManager, and CookieManager, and the links between them.

This single diagram illustrates several key concepts simultaneously:

- **Named and Anonymous Instances:** We can see named instances like `loginCtrl` and anonymous instances like the `:UserManager` interface.
- **Slots with Values:** Objects like `loginCtrl` and `:CookieManager` show their internal state with specific values assigned to their attributes (e.g., `attemptLimit = 5`).
- **Links:** The lines connecting the objects are **links**, representing runtime instances of associations. They can show roles (`-cookieMgr`) and navigability.
- **Collections:** The diagram even shows a collection of five anonymous instances, demonstrating how multiplicity is represented at the object level.

By creating such diagrams, developers can reason about specific scenarios, debug potential issues, and communicate the runtime structure of the system to other team members with a high degree of precision.

# 4 Practical Exercises: Class Diagrams

This chapter provides a series of practical exercises to apply the concepts of UML Class Diagrams discussed previously. Each exercise presents a problem description followed by a detailed solution and an explanation of the key concepts illustrated.

---

### 4.0.1 Exercise 1: People and Cars

**Problem:** Model a system of people and cars. A person is identified by a unique number and is the sole owner of the cars they possess. A car is characterized by a license plate number, a brand, and a date of first registration.

Click to see the solution

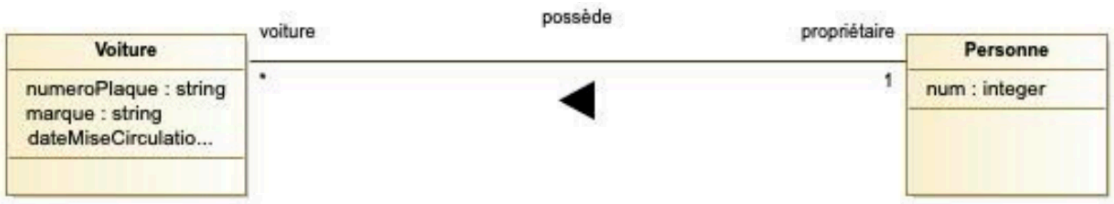



Figure 4.1: UML diagram for the People and Cars exercise.

**Correction Details:**

- **Personne (Person) Class:**
  - **Attributes:** Contains `num: integer` as a unique identifier.
  - **Multiplicity:** The `*` on the `voiture` (car) role indicates that an instance of `Personne` can be linked to zero, one, or many instances of `Voiture`.
- **Voiture (Car) Class:**
  - **Attributes:** Contains `numeroPlaque: string`, `marque: string`, and `dateMiseCirculation`.

- **Multiplicity:** The `1` on the `propriétaire` (owner) role is a crucial constraint. It means an instance of `Voiture` **must** be linked to exactly one instance of `Personne`. This directly models the "sole owner" requirement from the problem description.

- **possède (owns) Association:**
  - This association connects the two classes. The name `possède` clarifies the nature of the relationship.
  - The role names `propriétaire` and `voiture` specify the function each class plays in the relationship, making the diagram easier to read and understand.

**Key Concepts Illustrated:** This introductory exercise is designed to solidify several fundamental concepts:

- **Class:** A class is represented as a rectangle containing its name and attributes. It acts as a blueprint for objects.
- **Association:** An association models a relationship between two classes. Naming the association and its roles is a critical best practice for clarity.
- **Multiplicities (Cardinalities):** This is the most important concept in this exercise. Multiplicities are constraints that define the exact number of instances that can participate in a relationship, allowing the model to precisely capture business rules like "a car has a single owner".

---

### 4.0.2 Exercise 2: Company Organization

**Problem:** Model a company's organization. A department is identified by a number and has a location. An employee has a unique number within their department (but not necessarily within the company), a name, a role, and the department they work in. An enterprise is composed of departments.

Click to see the solution

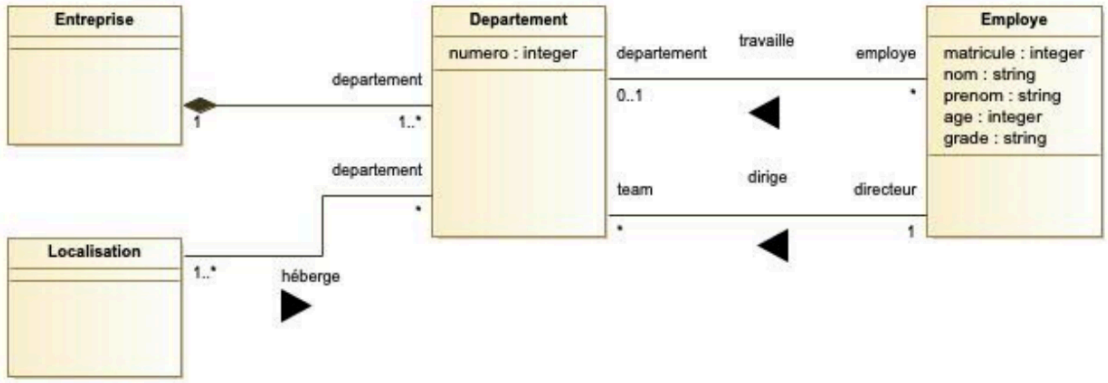



Figure 4.2: UML diagram for the Company Organization exercise.

**Correction Details:**

- **Entreprise (Company) Class:** The central element of the model. The problem states an enterprise is *composed of* departments, which strongly implies a whole-part relationship where the parts (departments) cannot exist without the whole. This is modelled with a **Composition** (filled diamond) relationship to `Departement`. The multiplicity `1..*` signifies that a company must have at least one department.
- **Departement Class:**
  - It has a `numero: integer` as an attribute. The problem states an employee's number is unique *within their department*, which means the department number itself doesn't have to be unique across the entire company (though it often is in reality).
  - It is linked to `Localisation` with a `1..*` multiplicity, meaning a department must have at least one location.
  - It has two distinct associations with the `Employe` class:
    1. The `travail` (working in) association has a `*` multiplicity, meaning a department has zero or more employees.
    2. The `dirige` (manage) association links a department to exactly one `directeur` (manager), who is an `Employe`.
- **Employe Class:**
  - The diagram shows the attributes `matricule: integer`, `nom: string`, `prenom: string`, `age: integer`, and `grade: string` for this class.
  - **Important Note:** The problem statement specifies that the employee's number (`matricule`) is unique *within* a department, not across the entire company.

modelling it as a direct attribute is a valid choice, and this uniqueness constraint would typically be managed at a higher level or with a formal constraint language like OCL.
  - The association to `Departement` has a multiplicity of `0..1`, meaning an employee works for at most one department.
  - An employee (multiplicity of `1`) can manage multiple departments, as shown by the `directeur` role on the `dirige` (manage) association, which has a multiplicity of `*` on the `Departement` side.

**Key Concepts Illustrated:**

- **Composition:** This exercise provides a classic example of Composition. The existential dependency is key: a `Departement` is fundamentally part of an `Entreprise`. If the company ceases to exist, its departments do as well. This is a much stronger and more precise relationship than a simple association or aggregation.
- **Multiple Associations between Classes:** It is common for two classes to be related in more than one way. Here, an `Employe` has a relationship of "working in" a `Departement` and a separate relationship of "managing" a `Departement`. Using distinct, named associations is crucial to model these different roles accurately.

---

### 4.0.3 Exercise 3: Geometric Figures

**Problem:** A geometric figure can be either simple or composite. A composite figure is made up of several other figures. A simple figure can be a point, a line, a circle, or a square. Any figure can be drawn or translated.

Click to see the solution

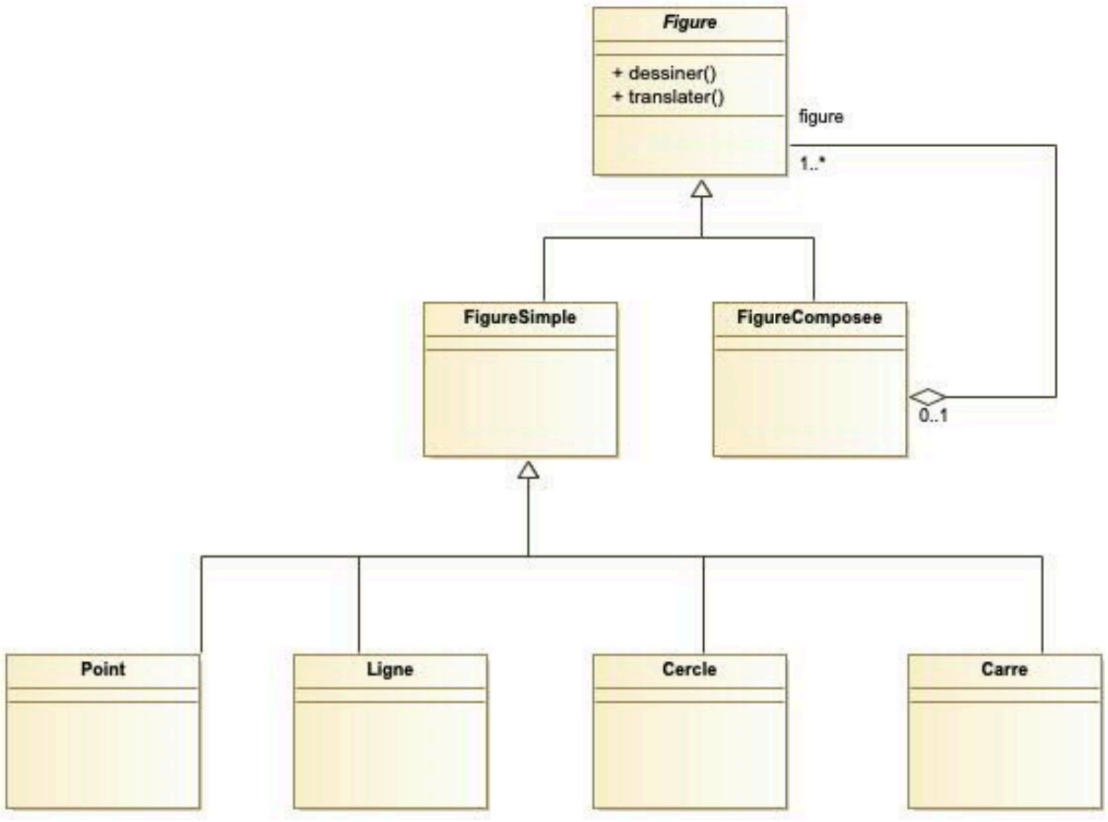



Figure 4.3: UML diagram for the Geometric Figures exercise.

**Correction Details:**

- `Figure` **Class:** This class is the core of the hierarchy. It is **abstract** (indicated by the italicized name) because there is no such thing as a generic "Figure" in this system; a figure must be either simple or composite. It defines a common interface for all figures by declaring two abstract operations: `dessiner()` and `translater()`. Any class that inherits from `Figure` will be required to provide its own implementation for these methods.
- **Generalization Hierarchy:** The model uses **Generalization** (inheritance) to create a clear taxonomy.
  - `FigureSimple` and `FigureComposee` both inherit from `Figure`. This establishes the primary classification.
  - `FigureSimple` acts as another superclass for the figure types: `Point`, `Ligne`, `Cercle`, and `Carre`. This creates a neat, two-level hierarchy.

- `FigureComposee` **Class:** This class models a figure that is made up of other figures.
  - The relationship between `FigureComposee` and `Figure` is modelled using **Aggregation** (the hollow diamond). This is a crucial design choice. It's a "whole-part" relationship, but it's "weak." If a `FigureComposee` is deleted, the `Figure` it was composed of are not necessarily deleted, as they could be part of another composite figure or exist independently. The `1..*` multiplicity indicates that a composite figure must contain at least one other figure.

**Key Concepts Illustrated:**

- **Abstraction & Inheritance:** This is the central theme of the exercise. A common interface (`dessiner()`, `translater()`) is defined in an abstract superclass (`Figure`) and inherited by all subclasses. This allows us to treat all figures uniformly, regardless of their specific type.
- **Aggregation vs. Composition:** This exercise provides a classic example of when to use Aggregation. Since a `Figure` can exist on its own or be part of multiple composite figures, its lifetime is not tied to a single "whole." This makes the weaker "has-a" relationship of Aggregation the correct choice over the stronger "owns-a" relationship of Composition.
- **Polymorphism:** This is the powerful result of the abstraction and inheritance structure. Thanks to polymorphism, we can have a collection of `Figure` objects and call the `dessiner()` method on each one. The system will automatically execute the correct implementation of `dessiner()` based on the object's actual type (`Point`, `Cercle`, `FigureComposee`, etc.). This allows for highly flexible and extensible code.

---

### 4.0.4 Exercise 4: Family Links

**Problem:** Define a schema describing the family links of a population of people, identifiable by their national register number.

Click to see the solution

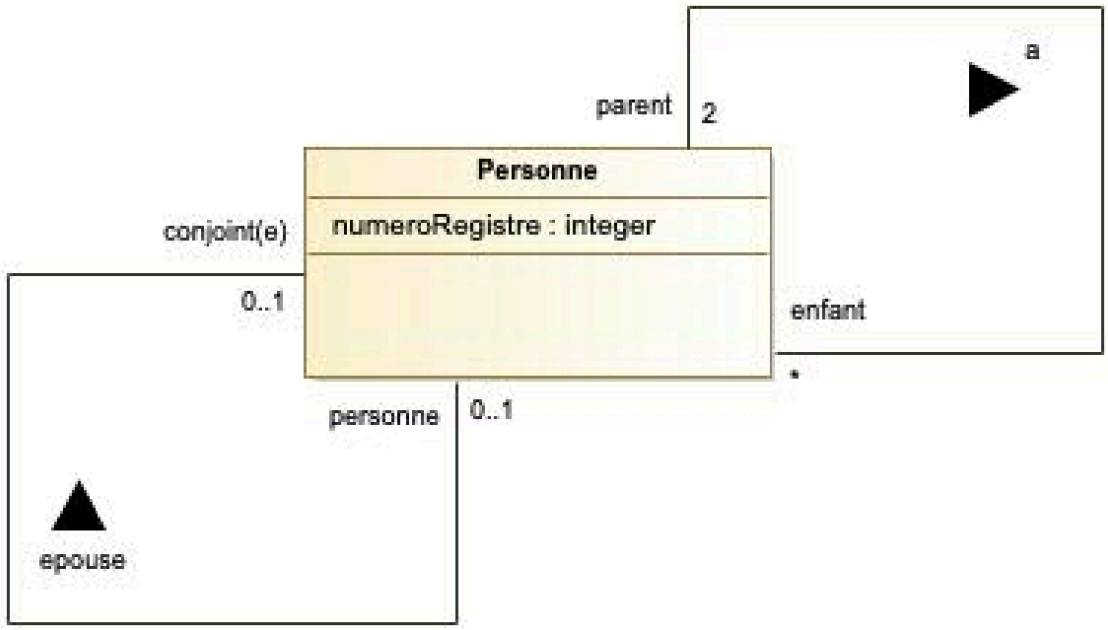



Figure 4.4: UML diagram for the Family Links exercise.

**Correction Details:**

- **`Personne` Class:** This single class is used to model all individuals in the system. It contains a `numeroRegistre: integer` attribute, which serves as a unique identifier for each person.
- **"épouse" (spouse) Association:** This is a **reflexive association**, meaning it connects the `Personne` class to itself. It models the marriage relationship.
  - **Roles:** The roles `conjoint(e)` and `personne` are used to clarify the nature of the link.
  - **Multiplicities:** The `0..1` multiplicity on both ends is a critical constraint. It precisely models a monogamous relationship, where a person can have either zero or one spouse at a time.
- **"a" (parent/child) Association:** This is a second reflexive association on the `Personne` class, modelling parent-child relationships.
  - **Roles:** The roles `parent` and `enfant` are absolutely essential here to make the diagram unambiguous. They allow us to read the relationship in two directions: a `parent` "has" an `enfant`, and an `enfant` "has" a `parent`.

– **Multiplicities:** The cardinalities are key to capturing the rules of the domain: a person must have exactly `2` parents, while a person acting as a `parent` can have zero, one, or many (`*`) children.

**Key Concepts Illustrated:**

- **Reflexive Association:** This exercise is a classic example of how a class can be related to itself. This pattern is fundamental for modelling any kind of network or hierarchical structure, such as organizational charts, social networks, or, as seen here, family trees.
- **The Importance of Roles:** In reflexive associations, role names are not just helpful; they are indispensable. Without them, it would be impossible to distinguish a parent from a child or a spouse from themselves. They are the primary tool for removing ambiguity in these situations.
- **modelling Domain Constraints:** The power of a class diagram comes from its ability to enforce real-world rules through multiplicities. The constraints `0..1` for a spouse and `2` for parents are not arbitrary; they are a direct translation of the rules of our specific domain into a formal model.

---

### 4.0.5 Exercise 5: Hotel

**Problem:** A hotel is composed of at least two "rooms". Each room consists of several parts such as bedrooms, washrooms, living rooms, and meeting rooms. Each room has a minimum of one bedroom and one washroom. The washrooms can be bathrooms (with a tub) or shower rooms. A room is also characterized by a price and a number. The hotel itself has a category, and an address. Furthermore, the hotel can host clients, employ staff, and is managed by a staff member.

Click to see the solution

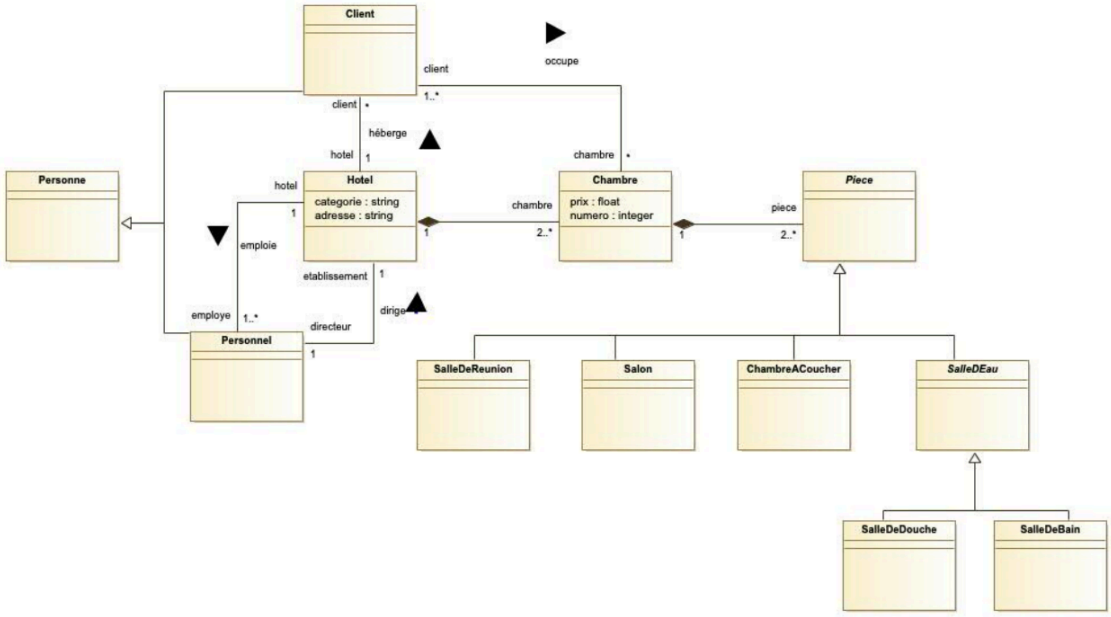



Figure 4.5: UML diagram for the Hotel exercise.

**Correction Details:**

- `Personne` **Generalization:** The diagram correctly uses **Inheritance** to model that both `Client` and `Personnel` are types of `Personne`. This is a classic "is-a" relationship that factors out common attributes (like `nom`, `age`, `adresse`) into a general superclass, avoiding redundancy.
- **Nested Compositions (Whole-Part Relationships):** The model features two levels of **Composition**, indicated by the filled diamonds. This choice is critical and deliberate:
  1. **Hôtel-Chambre:** An `Hôtel` is composed of `2..*` (at least two) `Chambre`. The filled diamond signifies that a `Chambre`'s existence is dependent on the `Hôtel`. If the hotel is demolished, its rooms cease to exist.
  2. **Chambre-Pièce:** Similarly, a `Chambre` is composed of `2..* Pièce` (parts). A `Pièce` (like a specific bedroom or bathroom) cannot exist independently of the room it belongs to. This creates a strong structural hierarchy.
- `Pièce` **Hierarchy:** The `Pièce` class itself is the root of another inheritance hierarchy. It's an abstract superclass for more specific types like `SalleDeReunion`, `Salon`, `ChambreACoucher` and `SalleDEau`. `SalleDEau` is further specialized into

SalleDeBain and SalleDeDouche. This demonstrates how inheritance can be used to classify and organize related concepts.

- **Client Associations:** The Client class is central to two key relationships:
  1. The héberge (hosts) association links Hôtel to Client. An hotel can host zero or more (*) clients, and a client is hosted by exactly one (1) hotel.
  2. The occupe (occupies) association links Client to Chambre. A client can occupy one or more (1..*) rooms, and a room can be occupied by zero or more (*) clients (e.g., a family in one room).
- **Associations with Personnel:** The Hôtel class has multiple, distinct associations with the Personnel class, highlighting how different relationships can coexist:
  - **emploie (employs):** A general association indicating that a hotel employs one or more (1..*) staff members.
  - **dirige (manages):** A more specific association with a directeur role, indicating that the hotel is managed by exactly one (1) member of staff. This shows how roles can be used to add precision to a relationship.

**Key Concepts Illustrated:**

- **Combining Relationship Types:** This is the most important takeaway. A real-world model rarely uses just one type of relationship. This exercise skillfully combines **Generalization**, **Composition**, and **Association** to create a rich and accurate representation of the domain.
- **Multiple, Overlapping Hierarchies:** This exercise perfectly illustrates how a system can feature different kinds of hierarchies simultaneously.
  - There is a clear **"is-part-of" hierarchy** established through **Composition**: An Hôtel is composed of Chambres, which are in turn composed of Pièces.
  - Running in parallel, there are two distinct **"is-a" hierarchies** established through **Inheritance**:
    1. The first classifies people: Client and Personnel are specific types of Personne.
    2. The second classifies the parts of a room: ChambreACoucher, SalleDEau, Salon, and SalleDeReunion are all types of Pièce. This hierarchy goes even deeper, as SalleDeBain and SalleDeDouche are specific types of SalleDEau. Understanding how to model these different, coexisting structures is a key skill in object-oriented analysis.
- **The Power of Constraints:** The specific multiplicities (2..*, 1, etc.) are not arbitrary. They are a direct translation of the problem description's rules ("at least two rooms", "managed by a staff member") into the formal language of UML, ensuring the model is precise.

### 4.0.6 Exercise 6: Petri Net

**Problem:** A Petri net is a directed graph composed of places, transitions, and arcs. An arc always connects two nodes of different types: either a place and a transition, or the reverse.

Model a class diagram to represent Petri nets, both with and without tokens. What is the difference?

Click to see the solution

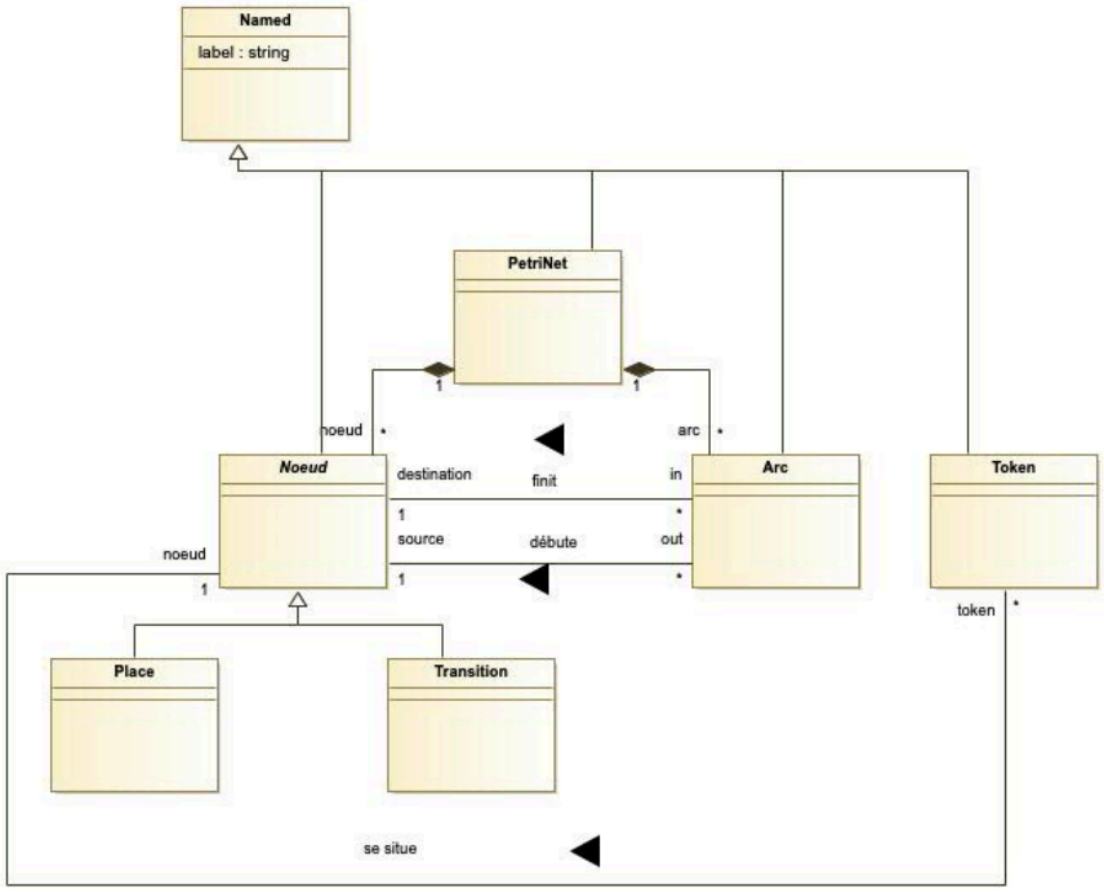



Figure 4.6: UML diagram for the Petri Net exercise.

**Correction Details:**

- **`PetriNet` Class:** This class acts as the root of the model. It has a **Composition** relationship (filled diamond) with `Nœud` (Node) and `Arc`. This is a critical choice: it signifies that the nodes and arcs are integral parts of a specific Petri Net and cannot exist independently. If a `PetriNet` is deleted, all its constituent elements are deleted as well.
- **`Nœud` (Node) Hierarchy and its Associations:**
  - `Nœud` is an **abstract class**, indicated by its italicized name. This is because a node in a Petri Net must be either a `Place` or a `Transition`; it cannot be just a generic "Node".
  - `Place` and `Transition` are concrete classes that **inherit** from `Nœud`, correctly modelling the "is-a" relationship.
  - The `Nœud` class is connected to the `Arc` class via two associations: `débute` (starts) and `finit` (ends). The multiplicities must be read in both directions:
    1. From the `Arc`'s perspective: An `Arc` must have **exactly one** source `Nœud` and **exactly one** destination `Nœud`. This is shown by the multiplicity of `1` at the `Nœud` end of both associations.
    2. From the `Nœud`' perspective: A `Nœud` can be the source of **zero or more** (`*`) `Arc` and the destination of **zero or more** (`*`) `Arc`. This is shown by the `*` multiplicity at the `Arc` end of the associations.
- **`Arc` Class and its Associations:**
  - An `Arc` has two mandatory (`1`) associations with the `Nœud` class: one for its source (`débute`) and one for its destination (`finit`). This accurately models that every arc connects exactly two nodes.
  - The problem states that an arc must connect a `Place` to a `Transition` or vice-versa. This is a complex constraint that is difficult to represent visually in a class diagram alone. It would typically be specified using a formal constraint language like **OCL (Object Constraint Language)**.
- **`Token` Class:**
  - The `Token` class represents the dynamic part of the model. Its association `se situe` (is located) connects it to the `Nœud` class.
  - **Multiplicities:** The cardinalities are critical. The `*` (many) on the `Token` side and `1` (exactly one) on the `Nœud` side mean that a `Nœud` can contain zero or more `Token`, but a `Token` must be located in **exactly one** `Nœud` at any given time.

**Key Concepts Illustrated:**

- **modelling a Meta-Structure (Metamodelling):** This is the most advanced concept in these exercises. The class diagram is not modelling a typical business

domain (like a hotel or a company). Instead, it is modelling the **structure of another model**, the Petri Net. This is a powerful technique used to define the "rules" of a language or a diagramming technique.

- **Static vs. Dynamic State:** This exercise perfectly illustrates the difference between modelling structure and state.
  - **Without the `Token` class:** The diagram would only describe the **static structure** of the Petri Net graph (which places and transitions exist and how they are connected by arcs).
  - **With the `Token` class:** The model can now represent the **dynamic state** of the network at any given moment (i.e., the "marking" of the net). The number and location of tokens are what determine which transitions can be fired, allowing the model to represent the system's behaviour over time.

# 5 Practical Exercises: Object Diagrams

This chapter provides practical exercises to apply the concepts of UML Object Diagrams. The goal is to learn how to represent a specific, concrete situation (a "snapshot" of a system) using instances of classes from a given class diagram.

---

### 5.0.1 Exercise 1: Petri Net

**Problem:** Given the following class diagram for a Petri Net, model the specific state of the Petri Net shown in the image below as a UML Object Diagram.

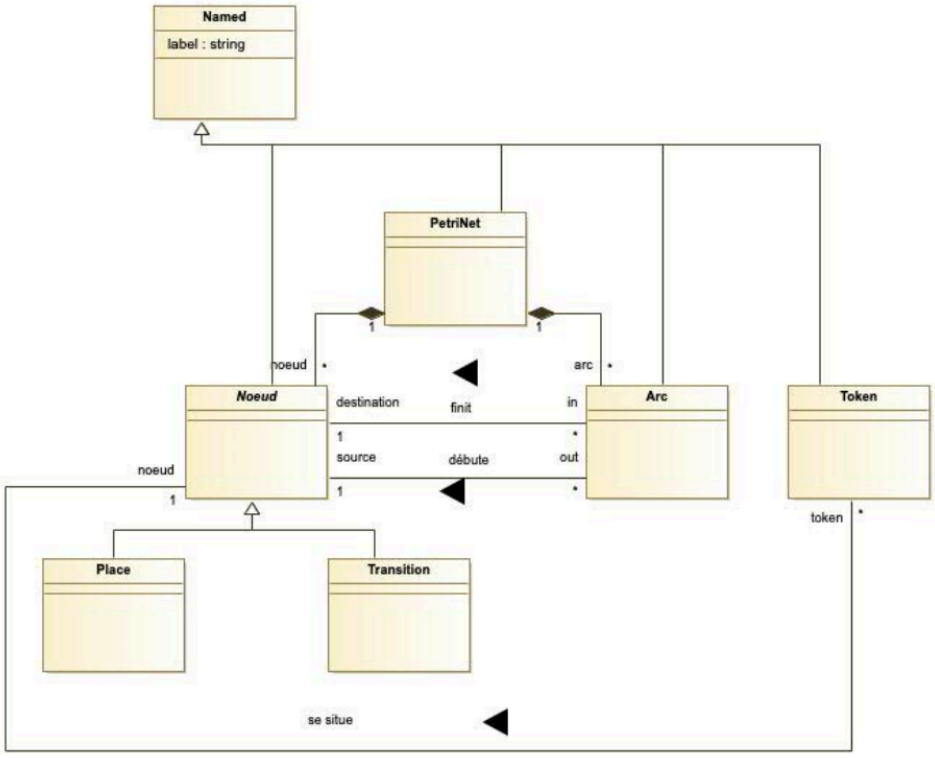



Figure 5.1: Class diagram for a Petri Net.

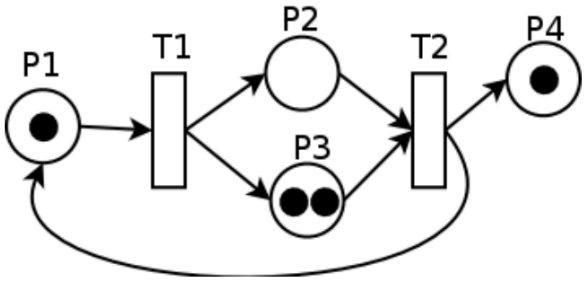



Figure 5.2: The specific state of a Petri Net to be modelled.

Click to see the solution

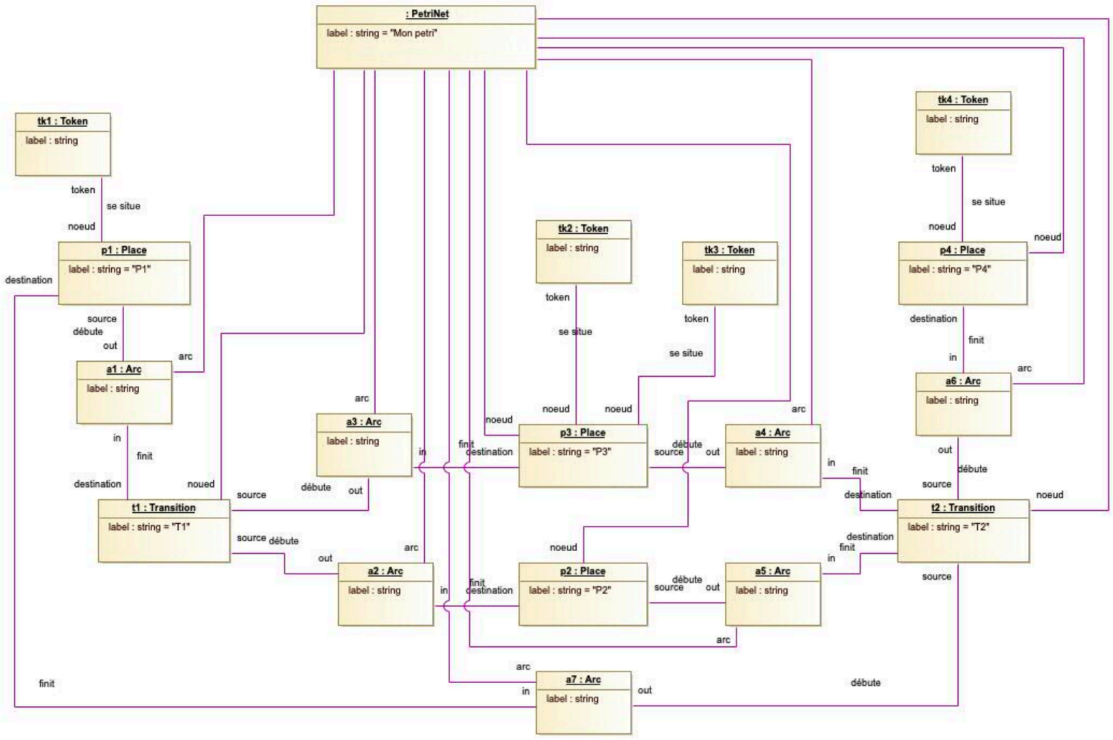



Figure 5.3: Object diagram solution for the Petri Net exercise.

**Correction Details:**

- **Instances of `Place` and `Transition`:** The solution correctly creates four named instances of the `Place` class (`p1`, `p2`, `p3`, `p4`) and two named instances of the

`Transition` class (`t1`, `t2`). Each instance is given a `label` corresponding to its name in the diagram (e.g., `"P1"`, `"T1"`).

- **Instances of `Arc` (Links):** The seven arcs from the problem are modelled as seven distinct, named instances of the `Arc` class (`a1` to `a7`). The `source` and `destination` links for each arc instance are correctly connected to the appropriate `Place` and `Transition` objects, modelling the directed graph structure.
- **Instances of `Token` (State):** The state of the Petri Net is represented by four instances of the `Token` class (`tk1` to `tk4`).
  - `p1` and `p4` each contain one token.
  - `p3` contains two tokens (`tk2` and `tk3`).
  - Each `Token` object is linked to the `Place` object that contains it via an instance of the `se situe` association.
- **Instance of `PetriNet`:** The entire system is contained within a single instance of `PetriNet`, named `mon_petri`. All `Place`, `Transition`, and `Arc` objects are linked to this main object, respecting the composition relationships defined in the class diagram.

---

### 5.0.2 Exercise 2: Email Service

**Problem:** Based on the class diagram modelling a university's email service, create the object diagram that represents the specific email situation shown in the image below.

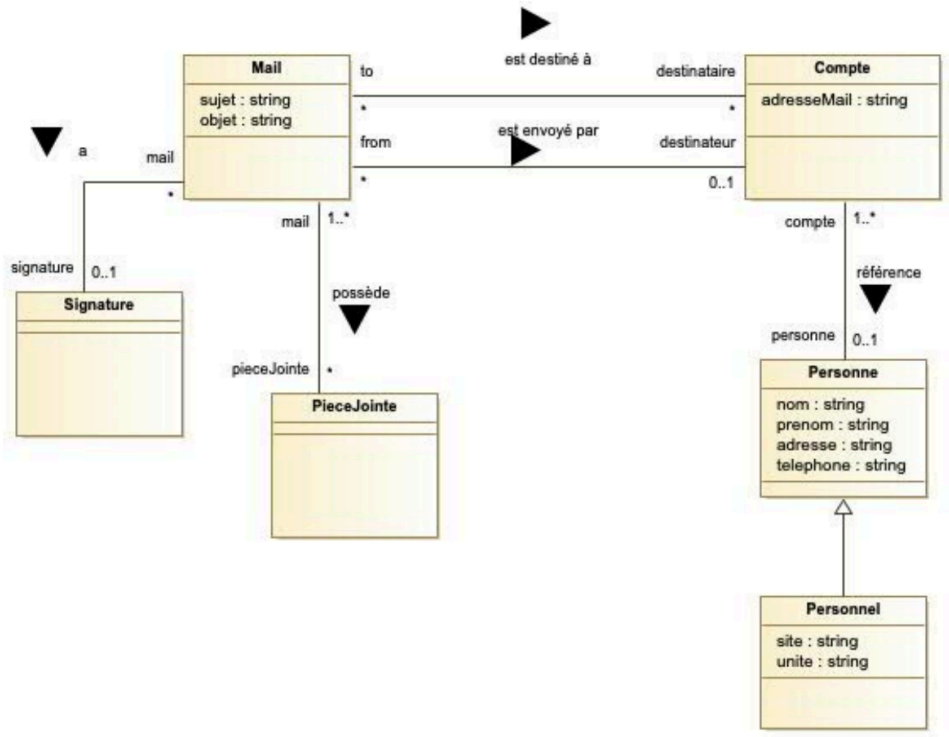



Figure 5.4: Class diagram for a university email service.

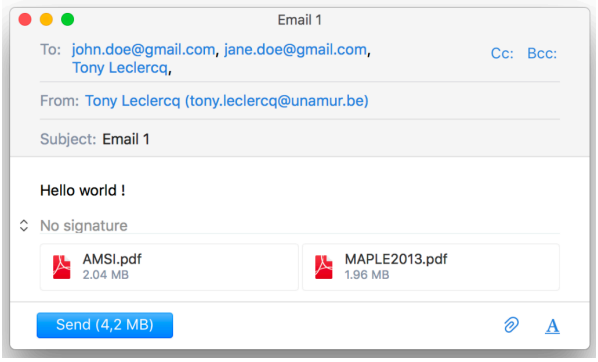



Figure 5.5: A screenshot of an email sent by Tony Leclercq.

Click to see the solution

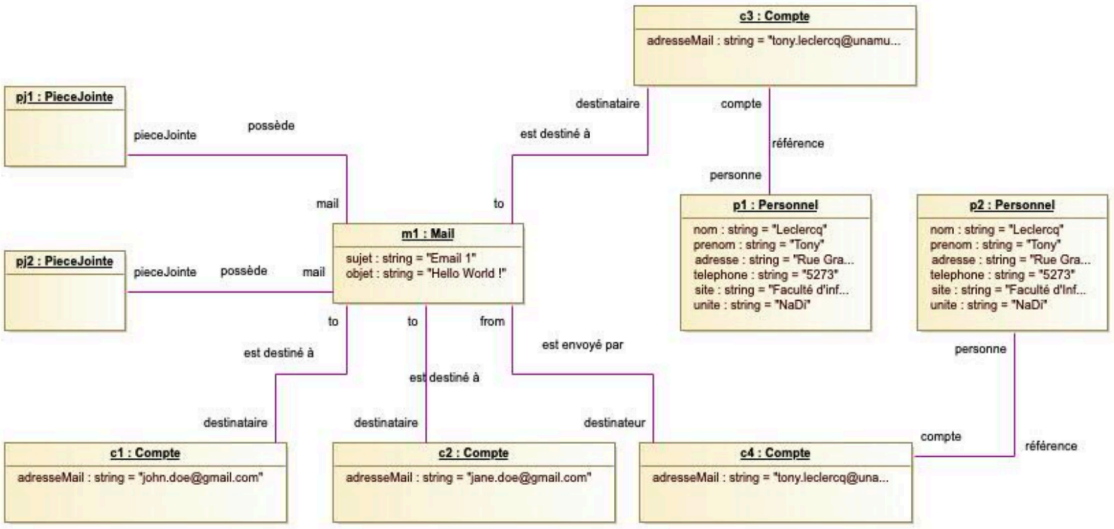



Figure 5.6: Object diagram solution for the Email exercise.

**Correction Details:**

- **Instances of `m1: Mail`:** A single instance of the `Mail` class represents the email. Its slots are filled with the corresponding data: `sujet = "Email 1"` and `objet = "Hello World !"`.
- **Instances of `pj1, pj2: PieceJointe`:** The two attachments, "AMSI.pdf" and "MAPLE2013.pdf", are modelled as two distinct instances of the `PieceJointe` class. Both are linked to the `m1` mail object via the `possède` (owns) composition link.
- **Instances of `c1, c2, c3, c4: Compte`:** The four email addresses involved are represented by four instances of the `Compte` (Account) class, with their `adresseMail` slots correctly filled.
  - `c3` and `c4` represent the accounts of Tony Leclercq.
  - `c1` and `c2` represent the external recipients.
- **Links for Senders/Recipients:** The links correctly model the email's flow:
  - The `est envoyé par` (is sent by) link connects `m1` to `c4` (Tony's sending account).
  - The `est destiné à` (is sent to) links connect `m1` to `c1`, `c2`, and `c3` (the three recipient accounts).

- **Instances of `p1, p2: Personnel`:** Two instances of `Personnel` are used to model Tony Leclercq, who is both the sender and a recipient. This might seem redundant, but it correctly represents his dual role in this specific transaction. Both instances are linked to their respective `Compte` objects (`c3` and `c4`).

# 6 The Object Constraint Language (OCL): An Overview

This chapter introduces the **Object Constraint Language (OCL)**, a formal language used to add precise and unambiguous rules to UML models. We will explore why UML diagrams alone are often insufficient and how OCL helps to create more rigorous and verifiable software specifications.

## 6.1 The Need for Precision: What UML Diagrams Don't Say

As we have seen, UML diagrams, especially Class Diagrams, are powerful tools for modelling the static structure of a system. However, they cannot capture all the business rules and constraints that govern a system's behaviour. A diagram can be syntactically correct but still allow for situations that are logically incorrect or violate business rules.

Consider the following class diagram for a car rental system:

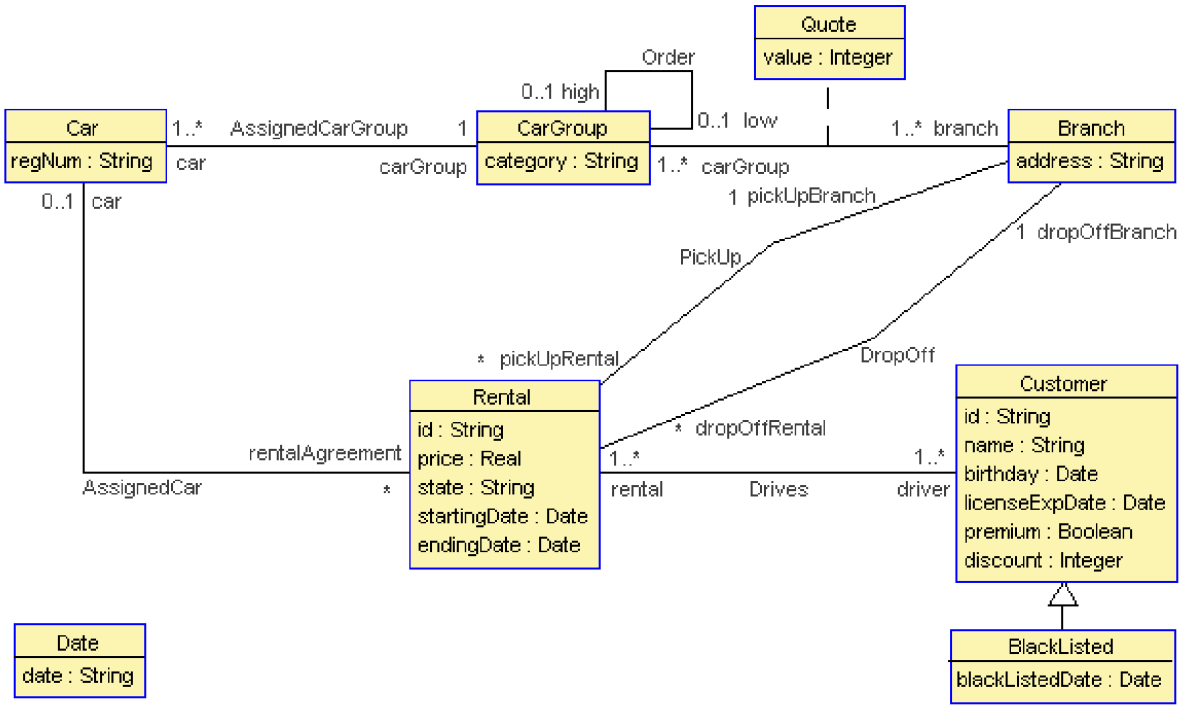



Figure 6.1: A class diagram for a car rental system.

This diagram leaves many important questions unanswered:

- Can a customer who is on the blacklist rent a new car?
- How is the price of a rental calculated?
- Can the pickup and drop-off branches be different?
- Should a driver's license be valid for the entire rental period?

While we could add this information in natural language, this approach is notoriously unreliable. Natural language specifications are prone to what are often called the **"7 Deadly Sins"**:

The 7 Deadly Sins of Natural Language Specification

- **Ambiguity:** A statement can be interpreted in multiple ways.
- **Contradiction:** Two or more rules in the specification conflict with each other.
- **Silence (Incompleteness):** A required rule or feature is not mentioned at all.
- **Noise:** The text contains irrelevant information that obscures the actual requirements.
- **Over-specification:** The text unnecessarily describes implementation details ("how") instead of requirements ("what").

- **Vagueness:** The requirements are not stated precisely enough to be verified.
- **Wishful Thinking:** A requirement is described that is impossible to implement.

These issues make automated verification impossible and lead to costly errors in implementation. To solve this, we need a formal, unambiguous way to express these rules, which is precisely the role of OCL.

## 6.2 Introduction to OCL

To solve the problems of ambiguity and imprecision, the **Object Constraint Language (OCL)** was created. It is the official standard from the OMG (Object Management Group) for adding formal constraints to UML models.

OCL is a **declarative, textual, and formal language**. This means:

- **Declarative:** You specify *what* should be true, not *how* to check it.
- **Textual:** It integrates seamlessly with the graphical nature of UML diagrams.
- **Formal:** It has a well-defined mathematical foundation, which removes ambiguity.

It is designed to be a middle ground: more precise than natural language, but less complex and easier to learn than purely mathematical formalisms.

### 6.2.1 Language Philosophy

To use OCL correctly, it is crucial to understand its core philosophy: OCL is a **pure specification language**, not a programming language. This has two major consequences:

**No Side Effects**

An OCL expression is a query that can only **read** the state of the model, it can **never change it**. You cannot use OCL to assign a value to an attribute, create an object, or call an operation that modifies the system's state. This guarantees that checking a constraint will never alter the system you are verifying.

**Instantaneous Evaluation**

The evaluation of an OCL expression is considered to be **instantaneous**. This is a powerful abstraction that means the state of the system is "frozen" during the evaluation of a constraint. You don't have to worry about other parts of the system changing while your expression is being checked, which greatly simplifies writing complex rules.

## 6.3 Main Applications of OCL

OCL is a versatile language used to add precision to a model in several key ways. Let's explore its main applications with concrete examples based on our car rental system.

1. **Constraining Models (Invariants):** The primary use of OCL is to write **invariants**. An invariant is a rule that must be true for all instances of a class at all times.

   *Example: A customer's discount must always be a positive value.*

   ```
   context Customer inv:
     self.discount >= 0
   ```

   **Explanation:** This simple invariant ensures that for any `Customer` object, the `discount` attribute (which is an `Integer` in the diagram) can never be negative.

2. **Specifying Contracts (Preconditions & Postconditions):** OCL is used to define formal contracts for operations. A **precondition** must be true *before* an operation is executed, and a **postcondition** must be true *after*.

   *Example: To extend a rental, its state must be 'active'.*

   ```
   context Rental::extendRental(newDate: Date)
     pre must_be_active:
       self.state = 'active'
   ```

   **Explanation:** This precondition checks that the `state` attribute of the `Rental` instance is equal to the string `'active'`. The operation cannot be called otherwise.

3. **Specifying Derived Values:** OCL can define how the value of an attribute should be calculated from other information. This is for values that are not stored directly.

   *Example: Let's define a derived boolean attribute `isPrivileged` for a `Customer`.*

   ```
   context Customer derive: isPrivileged: Boolean =
     self.premium = true and self.discount > 10
   ```

   **Explanation:** This rule states that a customer is considered "privileged" if and only if their `premium` status is true and their `discount` is greater than 10.

4. **Defining Queries:** You can use OCL to navigate through the model to retrieve information, much like a query.

   *Example: A query to get the set of all active rentals for a specific customer.*

   ```
   context Customer
     def: getActiveRentals(): Set(Rental) =
       self.rental->select(r | r.state = 'active')
   ```

**Explanation:** This defines a new operation `getActiveRentals` on the `Customer` class. It navigates to the customer's collection of `rental` and uses the `select` iterator to return a new set containing only those rentals whose `state` is 'active'.

## 6.4 Fundamental OCL Concepts

To write meaningful constraints, we first need to understand the two pillars of the OCL language: its **robust typing system**, and its **precise syntax and the formal semantics** that underpin it.

### 6.4.1 OCL is a Typed Language

Every element in OCL, from an attribute to the result of a complex navigation, has a **type**. This strong typing system ensures that expressions are well-formed and prevents errors, much like in languages like Java. OCL's type system is built from three sources:

#### 6.4.1.1 Primitive Types

OCL includes four fundamental, built-in types that are the basis for most expressions:

- `Integer` (e.g., `5`, `-10`)
- `Real` (e.g., `3.14`, `-0.5`)
- `String` (e.g., `'active'`, `'John Doe'`)
- `Boolean` (`true`, `false`)

#### 6.4.1.2 UML Model Types

This is a powerful feature: **every classifier you define in your UML model automatically becomes a new type in OCL**. If you have a class named `Customer` in your diagram, `Customer` becomes a valid type in your OCL expressions. This allows you to write constraints that are perfectly tailored to your specific domain.

#### 6.4.1.3 Collection Types

Because navigating associations often results in multiple objects, OCL has a rich set of built-in collection types to handle them. The collection is an abstract type with four concrete subtypes:

| Collection Type | Order Matters? | Duplicates Allowed? | Example Literal |
|---|---|---|---|
| `Set(T)` | No | No | `Set{1, 2, 3}` |
| `OrderedSet(T)` | Yes | No | `OrderedSet{1, 2, 3}` |
| `Bag(T)` | No | Yes | `Bag{1, 2, 2, 3}` |
| `Sequence(T)` | Yes | Yes | `Sequence{1, 3, 2, 3}` |

By default, navigating a `*` multiplicity association returns a `Set`. Understanding these collection types is essential, as most non-trivial OCL expressions involve manipulating collections.

### 6.4.2 Special OCL Types

Beyond the primitive and model-based types, OCL provides a set of special, built-in classifiers that are essential for writing advanced constraints. The hierarchy of these types is shown in the diagram below.

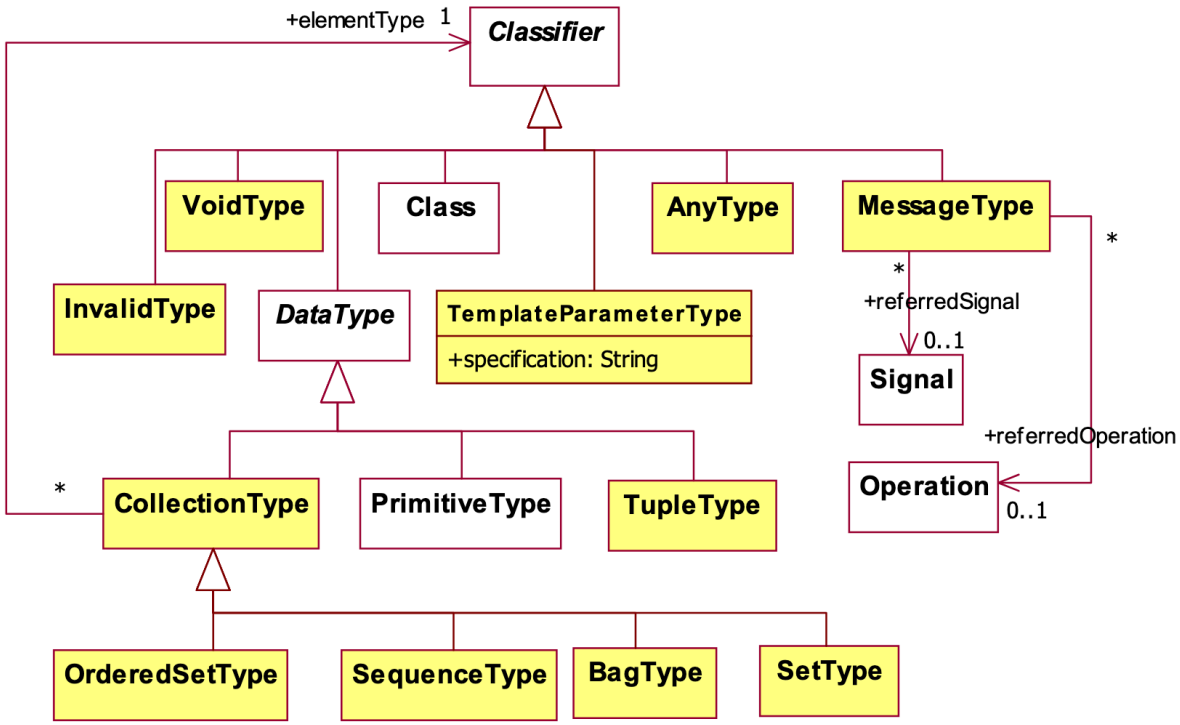



Figure 6.2: A class diagram showing the OCL Type Hierarchy, with Classifier at the top.

Let's explore the most important ones:

- `OclAny`: This is the supertype of all other types in OCL (except for collection and tuple types). Any operation defined on `OclAny` (like `oclIsTypeOf()`) is available on all objects in your model.
- `OclVoid`: This type has only one possible value: `null`. It is used to check if a property has been set or not. Any expression that results in a division by zero, for example, will evaluate to `null`. You can check for this value using the `oclIsUndefined()` operation.
- `OclInvalid`: This type has only one value: `invalid`. It is used to represent the result of an ill-formed expression, for instance, when trying to navigate from `null`. It is a subtype of all other types, meaning an `invalid` value can "poison" an entire expression. You can check for it with `oclIsInvalid()`.
- `TupleType`: A tuple is a structured type that groups together several named parts, each with its own type. It's similar to a struct or a record and is very useful for returning multiple values from a query operation.

  ```
  Tuple{name: String = 'Mikel', id: Integer = 123}
  ```

- `OclState`: This special type is used exclusively within state machine diagrams. The operation `oclIsInState(stateName: OclState)` allows you to write a constraint that checks if an object is currently in a specific state, which is very powerful for defining state-dependent invariants.
- `OclMessage`: Used in interaction diagrams (like sequence diagrams), this type allows you to write constraints on the messages exchanged between objects. For example, you can check if a specific message has been sent or received.

### 6.4.3 Type Conformance and Subtyping

The OCL type system is not flat, it has a rich hierarchy that defines which types are considered subtypes of others. This is known as **type conformance**. Understanding these rules is crucial, as it determines when a value of one type can be used where a value of another type is expected.

The main rules of type conformance in OCL are as follows:

1. For primitive numbers, the hierarchy is logical: `UnlimitedNatural` is a subtype of `Integer`, which is a subtype of `Real`.
2. `OclAny` is the ultimate supertype for all non-collection types. This means any object from your UML model is conformant to `OclAny`.
3. The hierarchy of your **UML model types** mirrors the generalization (inheritance) relationships in your class diagram. If you have a `Student` class that inherits from `Person`, then `Student` is a subtype of `Person` in OCL.

4. For collections, `Set(T)`, `Bag(T)`, `Sequence(T)`, and `OrderedSet(T)` are all subtypes of the general `Collection(T)` type.
5. The special types have their own rules:
    - `OclVoid` (representing `null`) is a subtype of all other types (except `OclInvalid`).
    - `OclInvalid` is the ultimate subtype, it is a subtype of *every other type*. This explains why an `invalid` value can propagate through and "poison" any expression.

This hierarchy is visualized in the diagram below.

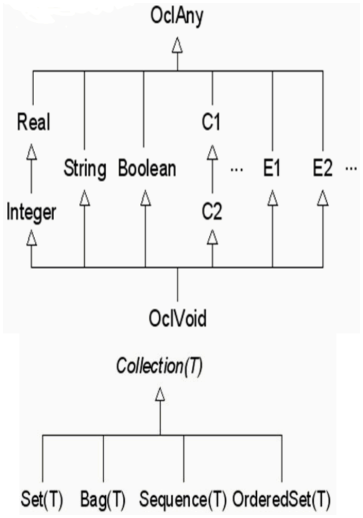



Figure 6.3: A class diagram showing the OCL Type Hierarchy and a list of the 7 conformance rules.

### 6.4.4 Standard OCL Operators

Like any language, OCL comes with a standard set of operators to perform logical, relational, and arithmetic operations. These are the building blocks for creating complex boolean expressions within your constraints.

#### 6.4.4.1 Logical Operators

OCL supports the standard boolean logic operators. Note that they are written in lowercase.

| Operator | Syntax | Description |
|---|---|---|
| **and** | `a and b` | True if both `a` and `b` are true. |
| **or** | `a or b` | True if either `a` or `b` is true. |

| Operator | Syntax | Description |
|---|---|---|
| **xor** | `a xor b` | True if exactly one of `a` or `b` is true. |
| **not** | `not a` | True if `a` is false. |
| **implies** | `a implies b` | False only if `a` is true and `b` is false. |

*Example: A premium customer must have a discount.*

```
context Customer inv:
  self.premium implies (self.discount > 0)
```

#### 6.4.4.1.1 The `if-then-else` Expression

Unlike in many programming languages, `if` in OCL is not a statement but an **expression** that always returns a value.

```
context Customer
  def: customerCategory: String =
    if self.premium then
      'High Value'
    else
      'Standard'
    endif
```

> ⚠ Important Rules for `if` expressions
>
> - **The `else` clause is mandatory.** Since `if` is an expression, it must always result in a value, so the `else` part can never be omitted.
> - **Type Conformance.** The type of the `if` expression is the most specific common supertype of the `then` expression and the `else` expression.

#### 6.4.4.1.2 A Note on Evaluation

OCL has a very specific rule for evaluating boolean expressions that differs from many common programming languages.

> ⚠ No Short-Circuit Evaluation
>
> The logical operators `and`, `or`, and `xor` in OCL are **strict**. This means that **both operands are always evaluated**, even if the result of the first operand is enough to determine the outcome. There is no "short-circuit" or "lazy" evaluation. This is a common source of errors, especially when dealing with potentially null values.

#### 6.4.4.2 Relational, Arithmetic, and String Operators

OCL provides a rich set of standard operators for the primitive types, allowing for the construction of detailed and precise expressions.

##### 6.4.4.2.1 Relational Operators

These operators are used for comparison and work across most primitive types to produce a `Boolean` result.

| Operator | Description |
|---|---|
| `=` | Equality |
| `<>` | Inequality (not equal) |
| `<` | Less than |
| `>` | Greater than |
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |

##### 6.4.4.2.2 Arithmetic Operators for (`Integer` and `Real`)

OCL provides a standard library of arithmetic functions for numerical types.

| Operator | Description |
|---|---|
| `+`, `-`, `*`, `/` | Addition, Subtraction, Multiplication, Division |
| `abs()` | Absolute value |
| `floor()`, `round()` | Floor and Rounding functions |
| `max(n)`, `min(n)` | The maximum or minimum of two numbers |
| `mod(n)` | Modulo operation |

#### 6.4.4.2.3 Operators for Strings (`String`)

OCL includes a powerful set of operations for manipulating strings.

| Operation | Description | Example |
|---|---|---|
| `size()` | Returns the number of characters in the string. | `'hello'.size() = 5` |
| `concat(s2)` | Concatenates another string `s2`. | `'hello'.concat(' world')` results in `'hello world'` |
| `toUpper()` | Converts the string to uppercase. | `'hello'.toUpper()` results in `'HELLO'` |
| `toLower()` | Converts the string to lowercase. | `'HELLO'.toLower()` results in `'hello'` |
| `substring(i1, i2)` | Returns the substring from index `i1` to `i2`. | `'hello'.substring(2, 4)` results in `'ell'` |

*Example: An invariant stating that a car group's category must be a single uppercase letter.*

```
context CarGroup inv:
  self.category.size() = 1 and self.category = self.category.toUpper()
```

### 6.4.4.3 Operator Precedence

OCL defines a strict order of precedence to determine how complex expressions are evaluated. The following table lists the operators in decreasing order of priority, from highest (evaluated first) to lowest (evaluated last).

| Priority | Operator(s) | Description |
|---|---|---|
| **1** (Highest) | `@pre` | Time marker (used in post-conditions) |
| **2** | `.` , `->` | Navigation (dot and arrow) |
| **3** | `not` , `-` | Unary operators (negation) |
| **4** | `*` , `/` | Multiplication, Division |
| **5** | `+` , `-` | Addition, Subtraction (binary) |
| **6** | `if-then-else-endif` | Conditional expression |
| **7** | `=` , `<>` , `>` , `<` , `>=` , `<=` | Relational operators |
| **8** | `and` , `or` , `xor` | Logical operators |
| **9** | `implies` | Logical implication |
| **10** (Lowest) | `in` | (Not covered yet) |

Best Practice: Use Parentheses

Even if you know the precedence rules, it is always a good practice to use parentheses `()` to make your expressions explicit and easier to read. This avoids any ambiguity for you and for others who will read your model.

### 6.4.5 The `context` and `self` Keywords

Every OCL constraint is anchored to a specific element in the UML model. The **`context`** keyword declares this anchor, defining the perspective from which the OCL expression will be evaluated. Within this context, the keyword **`self`** always refers to the specific instance of the element being constrained.

OCL defines three primary types of context:

#### 6.4.5.1 Classifier Context

This is the most common context, used for defining **invariants**. The context is a classifier, typically a class, and `self` refers to an instance of that class.

**Syntax:** `context <ClassName>`

```
context Customer inv:
  self.discount >= 0
```

#### 6.4.5.2 Operation Context

This context is used to define **preconditions** and **postconditions** for an operation. The context is a specific operation within a class, and `self` refers to the instance of the class on which the operation is being called.

**Syntax:** `context <ClassName>::<operationName>(...)`

```
context Rental::extendRental(newDate: Date)
  pre: self.state = 'active'
```

#### 6.4.5.3 3. Attribute Context

This context is used to define constraints on an attribute, typically for **derived values** or **initial values**. The context is a specific attribute within a class, and `self` refers to the instance of the class that owns the attribute.

**Syntax:** `context <ClassName>::<attributeName>: <Type>`

```
context Customer::isPrivileged: Boolean
  derive: self.premium = true and self.discount > 10
```

### 6.4.6 Writing Constraints: Invariants

The most common type of constraint you will write in OCL is the **invariant**.

> **What is an Invariant?**
>
> An invariant is a constraint that must be **true** for all instances of a class at **all times** during the system's execution. It defines a condition of integrity that can never be violated. If an operation causes an invariant to become false, the system is considered to be in an invalid state.

#### 6.4.6.1 Syntax

The general syntax for an invariant is straightforward:

`context <Classifier> inv []:  <boolean_expression>`

- **`context <Classifier>`**: Specifies the class to which the invariant applies.
- **`inv`**: The stereotype indicating this is an invariant.
- **`[<constraint_name>]`**: An optional but highly recommended name for the constraint.
- **`<boolean_expression>`**: The OCL expression that must always evaluate to `true`.

#### 6.4.6.2 Invariant Examples

Let's illustrate with a few examples based on our car rental system.

#### 6.4.6.2.1 Example 1: Simple Attribute Constraint

A very common use for invariants is to restrict the possible values of an attribute.

- **Rule:** A customer's discount must be a positive value.

```
context Customer inv:
  self.discount >= 0
```

#### 6.4.6.2.2 Example 2: Constraint Across an Association

Invariants are powerful for defining rules that involve multiple, associated classes.

- **Rule:** A car that is currently assigned to an active rental cannot be part of a `CarGroup` marked as 'in_maintenance'.

```
context Car inv:
  self.rental->exists(r | r.state = 'active') implies
    self.carGroup.category <> 'in_maintenance'
```

#### 6.4.6.2.3 Example 3: Constraint with Inheritance

Invariants can also enforce rules related to a class hierarchy.

- **Rule:** A customer who is on the blacklist (i.e., is of type `BlackListed`) cannot have any active rentals.

```
context Customer inv:
  self.oclIsTypeOf(BlackListed) implies
    self.rental->select(r | r.state = 'active')->isEmpty()
```

### 6.4.7 Writing Contracts: Preconditions and Postconditions

While invariants define the rules for a class's state, **preconditions** and **postconditions** define a **contract** for its operations. They specify what an operation requires to run and what it guarantees to accomplish.

> **i** What are Preconditions and Postconditions?
>
> - A **precondition** is a constraint that must be `true` **before** an operation is executed. It is the responsibility of the *caller* to satisfy the precondition. If it's false, the operation should not be called.
> - A **postcondition** is a constraint that must be `true` **after** an operation has suc-

> cessfully completed. It is the responsibility of the *operation itself* to establish the postcondition.

#### 6.4.7.1 Syntax

The syntax is similar to invariants, but the context is an operation, and the stereotypes are `pre` and `post`.

`context <Classifier>::<operationName>(...) pre []: <boolean_expression> post []: <boolean_expression>`

##### 6.4.7.1.1 Precondition Example

A precondition specifies the conditions under which an operation can be legally called.

- **Rule:** A customer can only be added to the blacklist if they are not already on it.

```
context Customer::addToBlackList(d: Date)
  pre is_not_already_blacklisted:
    self.oclIsTypeOf(BlackListed) = false
```

**Explanation:** This contract states that before calling `addToBlackList` on a `Customer` object, that object must not already be of the type `BlackListed`.

##### 6.4.7.1.2 Postcondition Example and the `@pre` Keyword

A postcondition specifies the state of the system after the operation has finished. A crucial tool for this is the `@pre` keyword, which allows you to refer to a value as it was *before* the operation started.

- **Rule:** After a customer is successfully added to the blacklist, they must be of the type `BlackListed`.

```
context Customer::addToBlackList(d: Date)
  post is_now_blacklisted:
    self.oclIsTypeOf(BlackListed) = true
```

Here's a more complex example using `@pre`:

- **Rule:** After a rental is extended, the new end date must be the one provided, and the number of associated cars must not have changed.

```
context Rental::extendRental(newDate: Date)
  post end_date_is_updated:
    self.endingDate = newDate
  post number_of_cars_is_unchanged:
    self.assignedCar->size() = self.assignedCar@pre->size()
```

**Explanation:**

- The first postcondition checks the new value of `endingDate`.
- The second postcondition is critical: it uses `self.assignedCar@pre` to refer to the collection of cars *before* the operation ran and ensures its size has not changed. This prevents unintended side effects.

### 6.4.8 Initial and Derived Values, Query and Body Definitions

Beyond invariants and contracts, OCL can be used to precisely define how attributes and operations get their values.

#### 6.4.8.1 `init`: Specifying Initial Values

The **`init`** stereotype allows you to define a constraint for the initial value of an attribute when an object is created. This is more powerful than a simple default value because you can use a complex OCL expression.

**Syntax:** `context <ClassName>::<attributeName>: <Type> init: <expression>`

*Example: When a `Rental` object is created, its `state` is initialized to 'planned'.*

```
context Rental::state: String
  init: 'planned'
```

#### 6.4.8.2 `derive`: Specifying Derived Values

The **`derive`** stereotype is used for attributes whose value is not stored but is always calculated based on other properties of the model. This is used for derived attributes, which are often denoted with a `/` in front of their name in a class diagram.

**Syntax:** `context <ClassName>::<attributeName>: <Type> derive: <expression>`

*Example: A customer is considered "privileged" if they are premium and have a discount greater than 10.*

```
context Customer::isPrivileged: Boolean
  derive: self.premium = true and self.discount > 10
```

#### 6.4.8.3 `def`: Defining Query Operations

The **`def`** stereotype allows you to define the body of a query operation directly in OCL. This is useful for specifying operations that retrieve information without changing the system's state.

**Syntax:** `context <ClassName> def: <operationName>(): <ReturnType> = <expression>`

*Example: Defining an operation that returns the set of all active rentals for a customer.*

```
context Customer
  def: getActiveRentals(): Set(Rental) =
    self.rental->select(r | r.state = 'active')
```

#### 6.4.8.4 `body`: Defining Operation Bodies

While `def` is used to define a new query operation not present in the UML model, the **`body`** stereotype is used to provide a **concrete implementation** for an operation that is already **declared** in a class diagram.

This is particularly useful when you want to formally specify the result of a query operation without leaving its logic ambiguous.

**Syntax:** `context <ClassName>::<operationName>(): <ReturnType> body: <expression>`

*Example: Defining the body for a `getDiscountedPrice()` operation on the `Rental` class.*

```
context Rental::getDiscountedPrice(): Real
  body: self.price * (1 - self.customer.discount/100.0)
```

**Explanation:** This OCL constraint provides the implementation for the `getDiscountedPrice()` operation. It specifies that the returned `Real` value is calculated by taking the rental's base `price` and applying the `discount` associated with the `customer`. Any programming language implementation of this class would now have a formal specification to follow for this method.

### 6.4.9 Navigating the Model

The primary power of OCL comes from its ability to navigate through a class diagram, starting from a context instance (`self`) to access its attributes, call its operations, and traverse its associations to reach other objects.

#### 6.4.9.1 Accessing Properties and Traversing Associations

Navigation is performed using the dot (`.`) notation.

- **Accessing Attributes:** `self.age`
- **Calling Operations:** `self.getAge()` (Note: operations must be query-only and have no side effects).
- **Traversing an Association:** To navigate, you use the **role name** at the opposite end of the association. If the role is unnamed, you use the name of the class itself, starting with a lowercase letter.

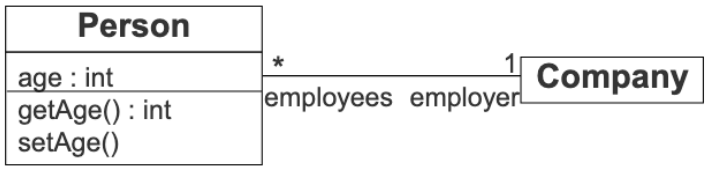



Figure 6.4: A diagram showing a Person class associated with a Company class.

```
context Person inv:
  -- Navigates from Person to Company via the 'employer' role.
  -- The result is a single object of type Company.
  self.employer
```

#### 6.4.9.2 Navigation From a Collection

This is a fundamental concept in OCL. When you start a navigation from an expression that results in a collection, the navigation is performed on **each element** of that collection. The final result is a new collection containing all the individual results.

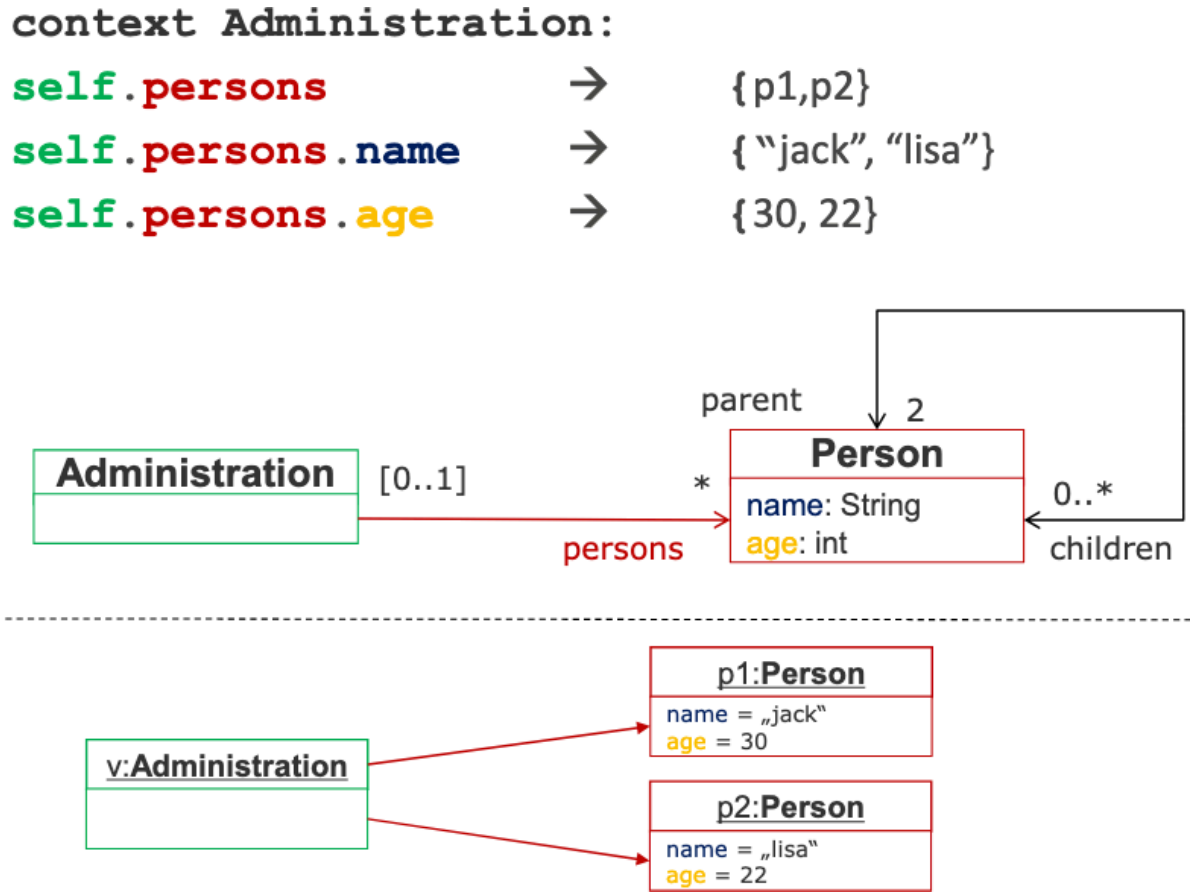



Figure 6.5: A diagram showing navigation from an Administration object to a collection of Person objects.

```
context Administration inv:
  -- 1. self.persons returns a Set of Person objects: Set{p1, p2}
  self.persons

  -- 2. Navigating to 'name' from this Set returns a Bag of all names.
  -- The result is Bag{'jack', 'lisa'}
  self.persons.name

  -- 3. Navigating to 'age' returns a Bag of all ages.
  -- The result is Bag{30, 22}
  self.persons.age
```

**Note:** When you navigate from a `Set` to an attribute, the result is a `Bag`, not a `Set`, because different objects in the original set could have the same value for that attribute.

#### 6.4.9.3 Chained Navigation

You can chain navigations together to traverse complex paths in your model. The rules of collection navigation apply at each step.

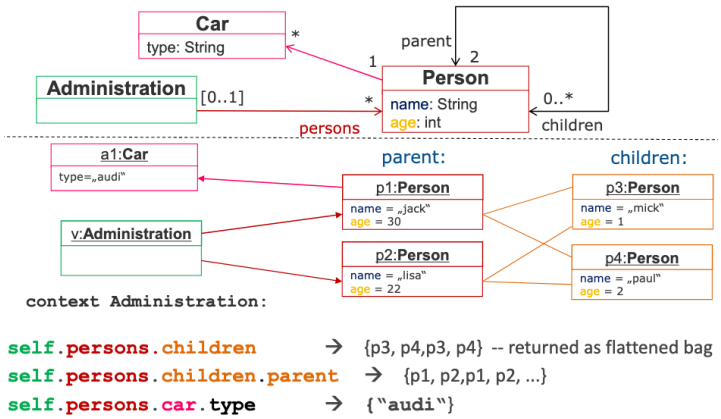



Figure 6.6: A diagram showing chained navigation from Rental to AssignedCar, then CarGroup, then category.

```
context Administration inv:
  -- self.persons -> Set{p1, p2}
  -- self.persons.children returns a Bag containing all children of p1 and p2.
  self.persons.children

  -- This expression first gets all children (a Bag),
  -- then for each child, it navigates to their parents (a Set of 2 Persons).
  -- The final result is a Bag containing all the parents of all the children.
  self.persons.children.parent
```

##### 6.4.9.3.1 A Note on Flattening Collections

When a navigation path involves multiple "to-many" associations, OCL automatically simplifies the result.

Consider the expression from the slide:

```
context Administration inv:
  self.persons.car
```

Let's break this down:

1. `self.persons` returns a `Set` of `Person` objects.
2. For each `Person` in that set, `.car` returns a `Set` of `Car` objects (because the multiplicity is `*`).
3. The result is therefore a collection of collections, for example: `Bag{ Set{car1}, Set{car2, car3} }`.

Instead of forcing you to work with this complex nested structure, OCL **automatically flattens** it into a single `Bag` containing all the elements: `Bag{car1, car2, car3}`.

This is why you can continue the navigation chain directly. The expression `self.persons.car.type` works as follows:

1. `self.persons.car` produces a flattened `Bag` of all cars owned by all persons.
2. The navigation `.type` is then applied to each `Car` in this flattened bag.
3. The final result is a `Bag` of all the `type` strings from all those cars.

### 6.4.10 Working with Collections

Because navigation often results in collections, OCL provides a rich set of predefined operations to query and manipulate them. These operations are always invoked using the arrow (`->`) notation.

#### 6.4.10.1 Basic Operations

These operations provide fundamental information about a collection or check for the presence of specific elements.

- **`size()`**: Returns the number of elements.
- **`isEmpty()` / `notEmpty()`**: Checks if the collection is empty or not.
- **`includes(obj)` / `excludes(obj)`**: Checks if a specific object is in the collection.
- **`includesAll(coll2)` / `excludesAll(coll2)`**: Checks if all (or no) elements of another collection `coll2` are present.
- **`count(obj)`**: Counts the occurrences of an object in the collection (most useful for `Bag` and `Sequence`).

*Example: An invariant stating that a Person must have exactly two parents.*

```
context Person inv:
  self.parent->size() = 2
```

#### 6.4.10.2 Iterators: The Power of Collections

The most powerful feature of OCL collections is the ability to use **iterators**. An iterator is an operation that evaluates an expression for each element in a collection, allowing you to filter, transform, or verify its properties.

##### 6.4.10.2.1 `select` and `reject` (Filtering)

These are the primary tools for filtering a collection. They return a new collection containing only the elements that satisfy (or don't satisfy) a condition.

- **`select(v | boolean_expression)`**: Returns a sub-collection with elements for which the expression is `true`.
- **`reject(v | boolean_expression)`**: Returns a sub-collection with elements for which the expression is `false`.

*Example: Select all rentals for a customer that are currently active.*

```
context Customer
  def: getActiveRentals(): Set(Rental) =
    self.rental->select(r | r.state = 'active')
```

##### 6.4.10.2.2 `forAll` and `exists` (Quantifiers)

These iterators evaluate a condition over a collection and return a single `Boolean` value. They are essential for writing precise invariants.

- **`forAll(v | boolean_expression)`**: Returns `true` if the expression is true for **all** elements.
- **`exists(v | boolean_expression)`**: Returns `true` if the expression is true for **at least one** element.

*Example: An invariant stating that all cars in a "Luxury" car group must be of the brand "Mercedes".*

```
context CarGroup inv:
  self.category = 'Luxury' implies
    self.car->forAll(c | c.brand = 'Mercedes')
```

#### 6.4.10.2.3 `collect` (Transformation)

The `collect` iterator transforms a collection by applying an expression to each of its elements. It returns a new `Bag` containing the results (similar to a `map` function).

*Example: Get a collection of all the registration numbers of the cars rented by a customer.*

```
context Customer
  def: allRentedCarRegNums(): Bag(String) =
    self.rental.assignedCar.car->collect(c | c.regNum)
```

### 6.4.10.3 `closure` (Transitive Closure)

The `closure` iterator is an advanced and extremely powerful operation that calculates the **transitive closure** of a relationship over a collection. In simpler terms, it allows you to repeatedly navigate through an association until no more new elements can be discovered.

This is the perfect tool for working with hierarchical or graph-like structures, such as finding all descendants of a person in a family tree or all sub-parts in an assembly.

**Syntax:** `collection->closure( v | navigation_expression )`

*Example: Find all descendants of a person (their children, their children's children, and so on).*

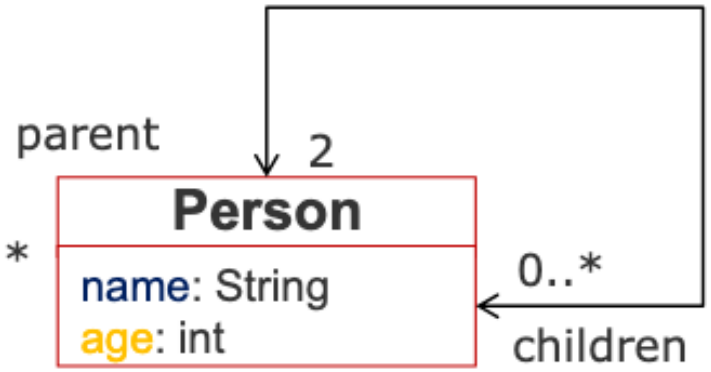



Figure 6.7: A class diagram showing a reflexive 'children' association on the Person class.

```
context Person
  def: getAllDescendants(): Set(Person) =
    self.children->closure(p | p.children)
```

**Explanation:**

1. `self.children`: The expression starts with the set of the person's direct children.
2. `->closure(p | p.children)`: The `closure` iterator then takes each person `p` from that set, finds their own `children`, adds them to the result, and repeats this process until no new descendants can be found. The final result is a flattened `Set` containing all descendants at every level.

### 6.4.11 Operations on All Objects (`OclAny`)

Because `OclAny` is the supertype of all types in your model, the operations it defines are available on **every object**. These are fundamental tools for checking equality, types, and for casting.

#### 6.4.11.1 Equality (= and <>)

These operators check if two expressions refer to the exact same object instance.

- `a = b`: Returns `true` if `a` and `b` are the same object.
- `a <> b`: Returns `true` if `a` and `b` are different objects.

#### 6.4.11.2 Type Checking: `oclIsTypeOf` vs. `oclIsKindOf`

These two operations are crucial for writing constraints in a generalization hierarchy, but they have a very important semantic difference.

- `oclIsTypeOf(Type)`: Returns `true` only if the object is an instance of that **exact** type, and not a subtype.
- `oclIsKindOf(Type)`: Returns `true` if the object is an instance of that type **or any of its subtypes**.

Let's consider the following hierarchy:

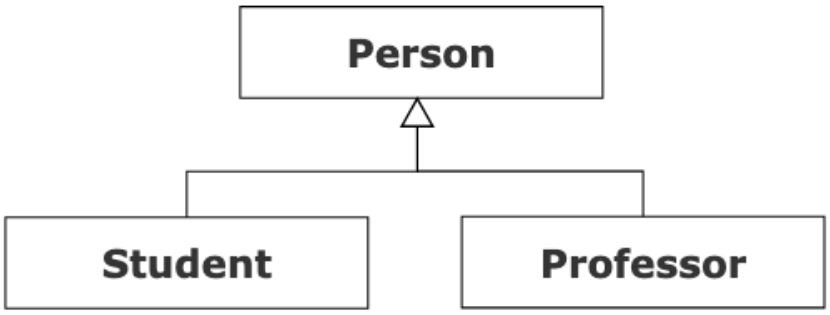



Figure 6.8: A generalization hierarchy showing Student and Professor inheriting from Person.

The following examples illustrate the difference:

```
context Person inv:
  self.oclIsKindOf(Person)   -- returns true
  self.oclIsTypeOf(Person)   -- returns true
  self.oclIsKindOf(Student)  -- returns false
  self.oclIsTypeOf(Student)  -- returns true
```

```
context Student inv:
  self.oclIsKindOf(Person)   -- returns true
  self.oclIsTypeOf(Person)   -- returns false
  self.oclIsKindOf(Student)  -- returns true
  self.oclIsTypeOf(Student)  -- returns true
  self.oclIsKindOf(Professor)-- returns false
  self.oclIsTypeOf(Professor)-- returns false
```

#### 6.4.11.3 Type Casting: `oclAsType`

This operation allows you to cast an object to a more specific subtype. This is necessary when you need to access attributes or operations that are only defined on that subtype.

- **`oclAsType(Type)`**: Casts the object to the specified `Type`. If the object is not actually of that type or one of its subtypes, the expression result is `invalid`.

*Example: An invariant on a Person object, stating that if that person is a Student, their student ID must not be empty.*

```
context Person inv:
  self.oclIsKindOf(Student) implies
    self.oclAsType(Student).studentId->notEmpty()
```

**Explanation:** We first check if the `Person` is a `Student` (or a subtype). If so, we cast `self` to `Student` to be able to safely access the `studentId` attribute.

#### 6.4.11.4 Handling Undefined Values (`OclVoid`)

In OCL, the value `null` has a specific type: **`OclVoid`**. It is used to represent an object that does not exist or an attribute that has not been set.

- **`oclIsUndefined()`**: Returns `true` if the object is `null`.

- `oclIsInvalid()`: Returns `true` if an expression is ill-formed (e.g., navigating from `null`).

*Example: An invariant stating that every* ***`Rental`*** *must be associated with a* ***`Customer`*** *(i.e., the* ***`driver`*** *role cannot be empty).*

```
context Rental inv:
  self.driver.oclIsUndefined() = false
```

This is a robust way to check that a mandatory association (1..*) is respected.

---

### 6.4.12 Special Operations: `allInstances` and `oclIsNew`

To conclude our overview of OCL, let's look at two special operations that operate at a meta-level: one that queries all objects of a certain type, and another that checks for an object's creation within a postcondition.

#### 6.4.12.1 `allInstances()`: Accessing All Objects of a Type

The `allInstances()` operation is unique because it is called directly on a **class**, not on an instance (`self`). It returns a `Set` containing all instances of that specific class that currently exist in the system at the time of evaluation.

**Syntax:** `<ClassName>.allInstances()`

This operation is extremely powerful for writing global invariants that must hold true across the entire system.

*Example: An invariant to enforce that every employee in the system has a unique employee ID.*

```
context Employee inv:
  Employee.allInstances()->forAll(e1, e2 |
    e1 <> e2 implies e1.employeeId <> e2.employeeId
  )
```

> ⚠ Use with Caution
>
> The `allInstances()` operation can be computationally expensive and conceptually dangerous. Using it on a type with a potentially huge or even infinite number of instances

(like `Integer` or `String`) can lead to problems. It is best reserved for application-specific classes where the number of instances is manageable.

#### 6.4.12.2 `oclIsNew()`: Checking Object Creation

The `oclIsNew()` operation is a special tool used **only in postconditions**. It returns `true` if the object on which it is called was created during the execution of the operation specified in the context.

*Example: A postcondition for a `createRental(...)` operation on the `Customer` class, stating that a new `Rental` object has been created and is now linked to the customer.*

```
context Customer::createRental(...)
  post: let newRental : Rental = self.rental->select(r | r.oclIsNew())->asSequence()->first()
    newRental.oclIsUndefined() = false and newRental.customer = self
```

**Explanation:** This advanced postcondition first finds the new `Rental` object created during the operation using `oclIsNew()` and then asserts that this new rental exists and is correctly associated with the current customer (`self`).

# 7 Practical Exercises: OCL Constraints

This chapter provides a series of practical exercises to apply the concepts of the Object Constraint Language (OCL) to UML Class Diagrams. Each exercise presents a problem description and a UML model, followed by a detailed solution explaining the OCL constraints used to enforce the business rules.

### 7.0.1 Exercise 1: Uniqueness of an Identifier

**Problem:** Using the diagram for people and cars, write an OCL constraint to guarantee that each person has a unique identification number (`num`) across all instances of the `Personne` class[cite: 24].

Click to see the solution

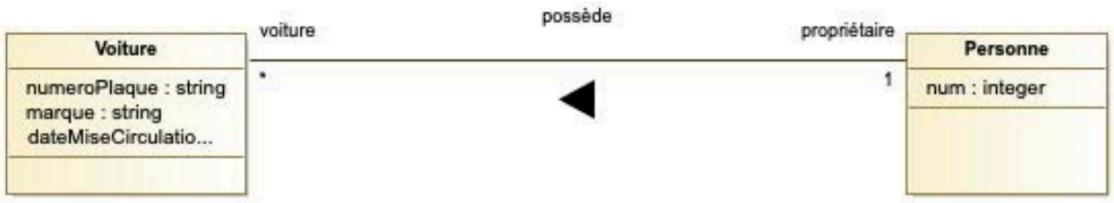



Figure 7.1: UML diagram for the People and Cars exercise.

**Correction Details:**
The goal is to ensure that the `num` attribute is unique for every person in the system[cite: 28]. In OCL, this can be expressed in two equivalent ways[cite: 29].

- **Formulation 1: Using `isUnique()`** [cite: 30] This is the most direct and readable way to express the constraint. The `isUnique()` operation checks if a given property is unique for all elements in a collection.

```
context Personne inv pKeyPersonne:
  Personne.allInstances()->isUnique(num)
```

  - **Explanation:** This invariant states that for the entire collection of

`Personne.allInstances()`, the value of the `num` attribute must be unique for each instance[cite: 32].

- **Formulation 2: Using `forAll()`** [cite: 34] This formulation uses the universal quantifier `forAll()` to express the same logic by comparing every pair of `Personne` instances.

```
context Personne inv pKeyPersonne:
  Personne.allInstances()->forAll(p1, p2 |
    p1 <> p2 implies p1.num <> p2.num
  )
```

  - **Explanation:** This invariant iterates over all possible pairs of `Personne` instances (`p1` and `p2`)[cite: 37]. It states that **for all** pairs, **if** `p1` and `p2` are different objects, **then** their `num` attributes must also be different[cite: 37].

**Key Concepts Illustrated:**

- **Context:** The `context Personne` declaration specifies that the constraint applies to the `Personne` class[cite: 31].
- **Invariant (`inv`):** An invariant is a rule that must always be true for all instances of a class throughout their lifetime[cite: 25].
- **`allInstances()`:** A class-level operation that returns a collection of all instances of that class in the system[cite: 31, 36].
- **Collection Operations:** This exercise showcases two fundamental collection operations: `isUnique()` for a direct check [cite: 31] and `forAll()` for expressing complex logical conditions on all elements[cite: 36].

### 7.0.2 Exercise 2: Uniqueness Within a Group

**Problem:** For the company organization model, an employee's ID number (`matricule`) must be unique within their department, but not necessarily across the entire company[cite: 59]. Write an OCL constraint to enforce this rule.

Click to see the solution

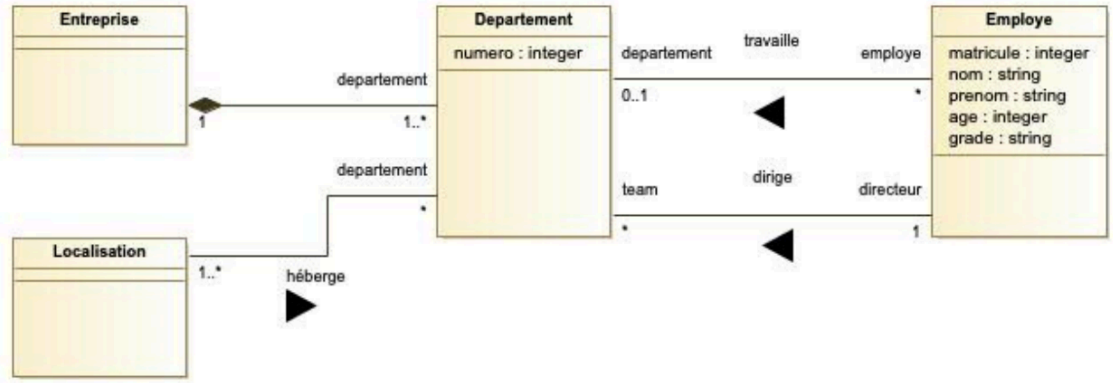



Figure 7.2: UML diagram for the Company Organization exercise.

**Correction Details:**
The key to this problem is choosing the correct context. The constraint is about the uniqueness of employees *within a department*, so the most logical context is `Departement`[cite: 72].

- **Correct Solution: Context `Departement`** [cite: 68] This approach is simple and directly models the requirement[cite: 70]. It evaluates the constraint for each department individually[cite: 72]. An idiomatic way to write this is using `isUnique()`:

  ```
  context Departement inv pKeyEmploye:
    self.employe->isUnique(matricule)
  ```

  The provided source also shows an equivalent formulation using `forAll`[cite: 70]: `ocl context Departement inv pKeyEmploye: self.employe->forAll(e1, e2 | e1 <> e2 implies e1.matricule <> e2.matricule)`

  - **Explanation:** For each instance of `Departement` (`self`), we navigate the `employe` association to get the collection of employees in that department. We then ensure that all `matricule` attributes within that collection are unique[cite: 71].

**Key Concepts Illustrated:**

- **Choosing the Right Context:** This is the most important lesson. Placing the invariant in the `Departement` context radically simplifies the expression and makes the intent clear[cite: 72].

- **Navigation:** The expression `self.employe` demonstrates navigation across an association from an instance of one class to a collection of related instances[cite: 70].
- **Scoped Uniqueness:** This exercise contrasts with the first one by showing how to enforce uniqueness within a specific scope (a department) rather than globally.

### 7.0.3 Exercise 3: Preventing Circular Composition

**Problem:** In the geometric figures model, a composite figure (`FigureComposee`) is made up of other figures. It is crucial to prevent circular compositions, where a figure could contain itself, either directly or indirectly[cite: 91]. Write an OCL constraint to forbid this.

Click to see the solution

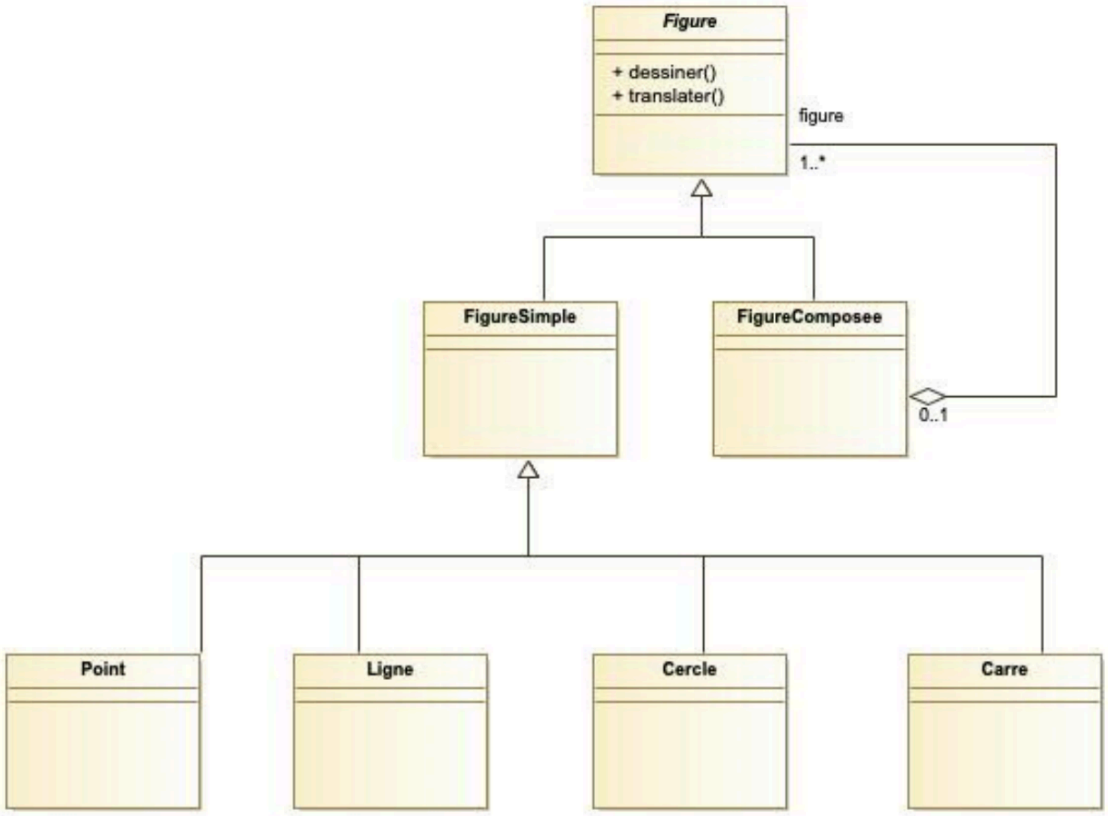



Figure 7.3: UML diagram for the Geometric Figures exercise.

**Correction Details:**
This problem requires two levels of solution: one for direct cycles and a more advanced one for indirect cycles.

- **Solution 1: Preventing Direct Cycles** This constraint prevents a `FigureComposee` from including itself in its immediate list of sub-figures[cite: 98, 101].

```
context FigureComposee inv circularComposition:
  not self.figure->includes(self)
```

  - **Explanation:** For any instance of `FigureComposee` (`self`), this invariant checks the collection of its direct sub-figures (`self.figure`) and asserts that the collection does not include the instance itself[cite: 99].

- **Solution 2: Preventing Indirect Cycles (Advanced)** The first solution is insufficient because it doesn't prevent A from containing B, which in turn contains A[cite: 102]. To solve this, we need a recursive query to find *all* sub-figures at any level of nesting (the transitive closure)[cite: 103, 114].

```
context FigureComposee
-- Define a helper function to get all nested sub-figures
def: allSubFigures(): Set(Figure) =
  self.figure->union(
    self.figure->select(f | f.oclIsTypeOf(FigureComposee))
                ->collect(f | f.allSubFigures())
                ->flatten()
  )

-- The invariant uses the helper function
inv noCircularComposition:
  not self.allSubFigures()->includes(self)
```

  - **Explanation:** We first define a helper operation `allSubFigures()` that recursively collects all direct figures and all sub-figures of any composite children[cite: 114]. The invariant `noCircularComposition` then uses this function to check that `self` is not present in the complete set of its descendants, thus preventing all direct and indirect cycles[cite: 113, 114].

**Key Concepts Illustrated:**

- **`includes()`:** A collection operation that checks for the presence of an element.

- **Helper Operations (`def`)**: OCL allows the definition of custom, reusable queries, which is essential for complex logic like recursion[cite: 103].
- **Recursion and Transitive Closure:** The `allSubFigures()` operation is a classic example of using recursion to navigate a hierarchical structure to find all connected nodes[cite: 114].

### 7.0.4 Exercise 4: Constraining Family Relationships

**Problem:** Using the family links diagram, impose a constraint to forbid a person from being both a parent and a spouse to the same other person[cite: 125].

Click to see the solution

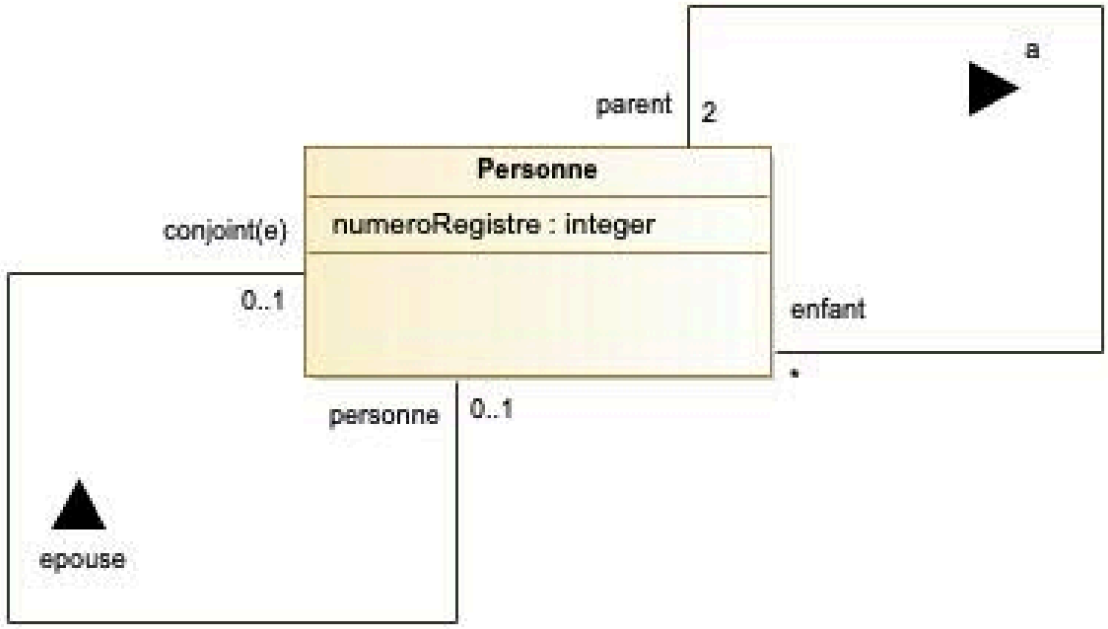



Figure 7.4: UML diagram for the Family Links exercise.

**Correction Details:**
The constraint needs to ensure that the set of a person's children and the set of a person's spouses are disjoint (have no common elements)[cite: 126].

```
context Personne inv notChildParentsMarriage:
  self.conjoint->excludesAll(self.enfant)
```

**Explanation:**

- `self`: Refers to the instance of the `Personne` class on which the constraint is being evaluated.
- `conjoint`: Refers to the association that links a person to their spouse(s). It returns a collection of `Personne` objects.
- `enfant`: Refers to the association that links a person to their children.
- `excludesAll()`: This OCL collection operation returns `true` if the first collection (the spouses) contains none of the elements from the second collection (the children).

**Key Concepts Illustrated:**

- **Reflexive Association:** The diagram shows the `Personne` class related to itself. OCL navigates these associations using the provided role names (`conjoint`, `enfant`).
- **Set Operations:** OCL provides a rich set of operations for working with collections, including `excludesAll()`, which is used here to ensure two sets are disjoint.

### 7.0.5 Exercise 5: Hotel

**Problem:** A hotel is composed of at least two "rooms". Each room consists of several parts such as bedrooms, washrooms, living rooms, and meeting rooms. Each room has a minimum of one bedroom and one washroom. The washrooms can be bathrooms (with a tub) or shower rooms. A room is also characterized by a price and a number. The hotel itself has a category, and an address. Furthermore, the hotel can host clients, employ staff, and is managed by a staff member.

Click to see the solution

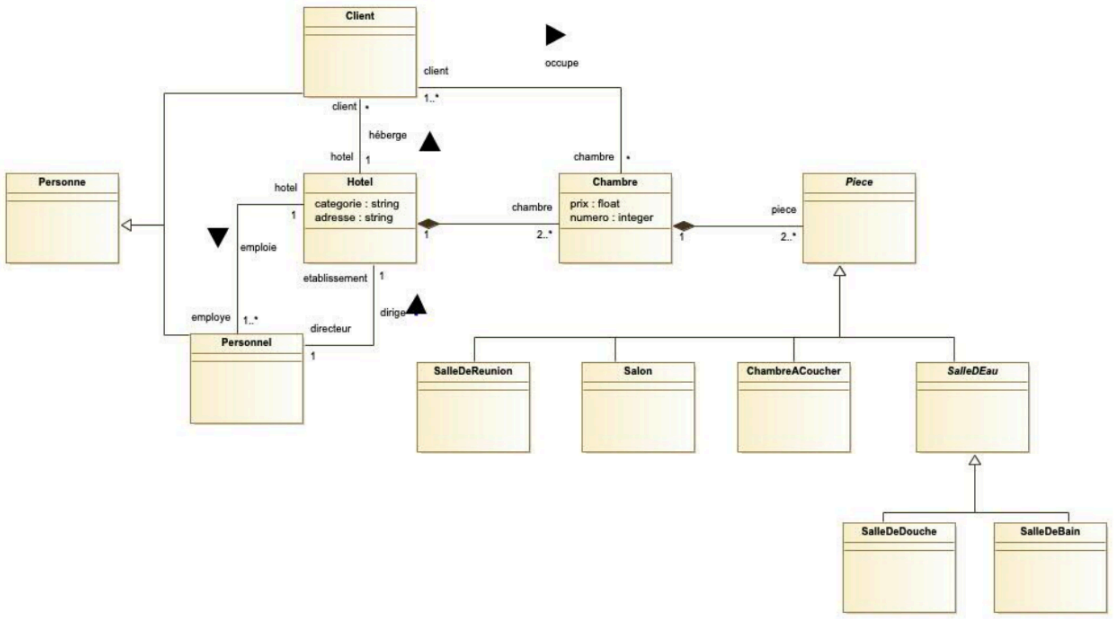



Figure 7.5: UML diagram for the Hotel exercise.

**Correction Details:**
The problem requires two OCL invariants to ensure the correct composition of a hotel room (`Chambre`).

- **Constraint 1: Each room must have at least one bedroom.** This constraint can be formulated in two ways. The first filters the collection of parts and checks its size, while the second (and more efficient) way simply checks for the existence of such a part.

  *Formulation A: Using `select` and `size`*

  ```
  context Chambre inv minChambreACoucher:
    self.piece->select(p | p.oclIsTypeOf(ChambreACoucher))->size() >= 1
  ```

  *Formulation B: Using `exists`*

  ```
  context Chambre inv minChambreACoucher:
    self.piece->exists(p | p.oclIsTypeOf(ChambreACoucher))
  ```

- **Constraint 2: Each room must have at least one washroom.** Similarly, this constraint verifies that at least one of the room's parts is a `SalleDEau` (which is a superclass for `SalleDeBain` and `SalleDeDouche`).

*Formulation A: Using `select` and `size`*

```
context Chambre inv minSalleDEau:
  self.piece->select(p | p.oclIsTypeOf(SalleDEau))->size() >= 1
```

*Formulation B: Using `exists`*

```
context Chambre inv minSalleDEau:
  self.piece->exists(p | p.oclIsTypeOf(SalleDEau))
```

**Key Concepts Illustrated:**

- **Navigation:** The constraints start from an instance of `Chambre` (`self`) and navigate the `piece` association to access the collection of parts.
- **`oclIsTypeOf()`:** This operation is used to filter a collection of objects based on their specific class.
- **`select()` vs. `exists()`:** This exercise highlights two common ways to check for the presence of an item in a collection. `select()` creates a new sub-collection, while `exists()` returns a simple boolean, which is often clearer and more performant for this type of rule.

### 7.0.6 Exercise 6: Banking System

**Problem:** Model a banking system where people own accounts in banking organizations. Enforce several business rules using OCL, including data integrity invariants (unique IDs, balance limits), initial attribute values, pre- and post-conditions for account operations (deposit, withdrawal), and queries to retrieve client information.

Click to see the solution

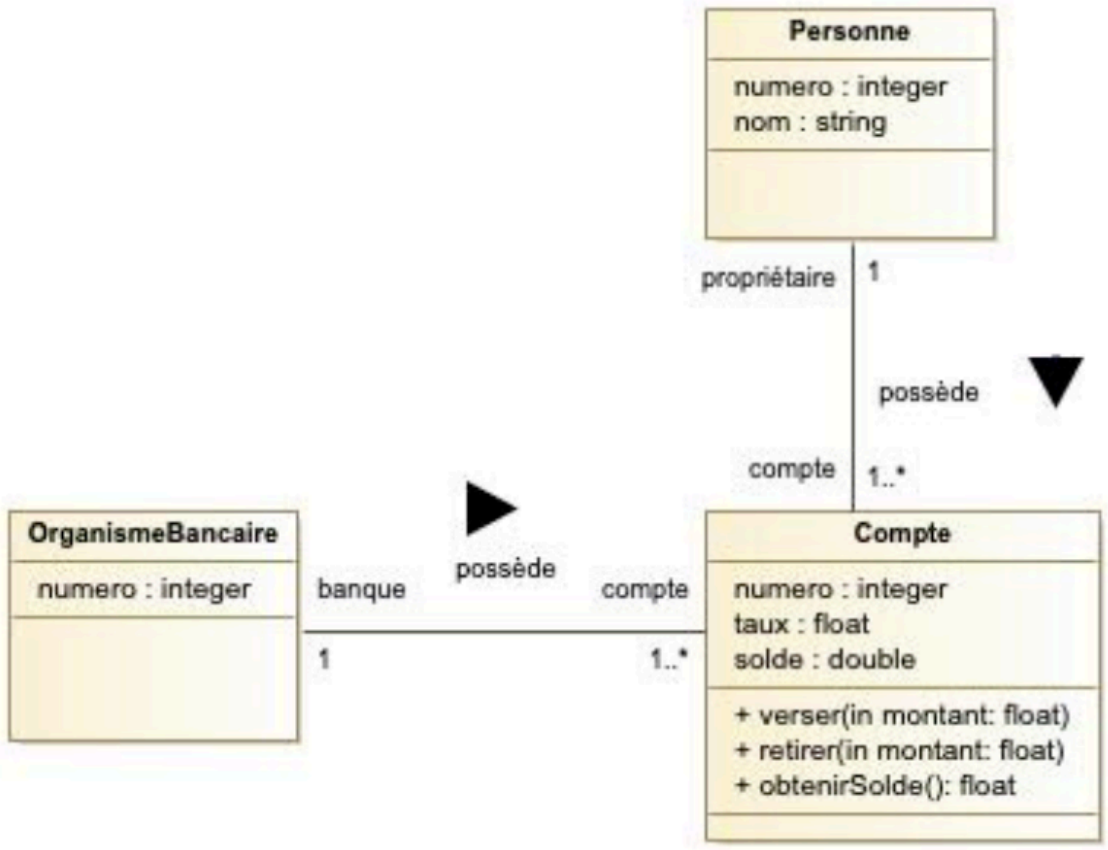



Figure 7.6: UML diagram for the Banking System exercise.

**Correction Details:**
This comprehensive exercise uses multiple OCL features to define the behaviour and constraints of a banking system.
**1. Invariants (Data Integrity Rules)**
These rules must always be true for the system to be in a valid state.

- **Unique account number per bank:**

```
context OrganismeBancaire inv pKeyCompte:
  self.compte->isUnique(numero)
```

*Explanation: For each bank, the set of its accounts must have unique numbers.*

- **The balance of an account cannot exceed 1,000,000:**

```
context Compte inv creditMax:
  self.solde <= 1000000
```

- **The interest rate must be strictly positive:**

```
context Compte inv tauxPositif:
  self.taux > 0
```

- **A person's ID must be unique:**

```
context Personne inv idUnique:
  Personne.allInstances()->isUnique(numero)
```

**2. Initial and Derived Values**
OCL can specify initial values for attributes or define attributes whose values are derived from others.

- **The balance of a new account is initialized to 0:**

```
context Compte::solde: Real
    init: 0
```

- **The interest amount is derived from the balance and rate:**

```
context Compte::interets: Real
    derive: solde * taux
```

**3. Pre- and Post-conditions (Operation Contracts)**
These define the "contract" for an operation: what must be true before it runs (`pre`) and what must be true after it finishes (`post`).

- **Deposit money (`verser`):**

```
context Compte::verser(montant: Real)
    pre montantPositif: montant > 0
    post soldeMisAJour: self.solde = self.solde@pre + montant
```

*Explanation: The precondition requires the deposit amount to be positive. The postcondition guarantees the new balance is the old balance (`@pre`) plus the amount.*

- **Withdraw money (`retirer`):**

```
context Compte::retirer(montant: Real)
    pre montantPositif: montant > 0
    pre soldeSuffisant: self.solde >= montant
    post soldeMisAJour: self.solde = self.solde@pre - montant
```

*Explanation: The preconditions require a positive amount and sufficient funds. The postcondition guarantees the balance is correctly updated.*

- **Get balance (`obtenirSolde`):**

```
context Compte::obtenirSolde(): Real
    post resultatCorrect: result = self.solde
    post soldeInchange: self.solde = self.solde@pre
```

*Explanation: The postconditions guarantee the returned value (`result`) is the current balance and that the operation did not change the state of the balance.*

**4. Queries (Helper Definitions)**
OCL can be used to define reusable queries on the model.

- **Get a list of all distinct client first names:**

```
context Personne def: listePrenomsDifferents(): Set(String) =
  Personne.allInstances()->collect(prenom)->asSet()
```

*Explanation: This defines a function that gets all `Personne` instances, collects their first names (`prenom`), and converts the resulting collection into a `Set` to ensure uniqueness.*

**Key Concepts Illustrated:**

- **Invariants (`inv`):** Rules that define a stable and consistent state for the model's objects.
- **Initializers (`init`):** Specifies the state of an attribute upon an object's creation.
- **Derived Values (`derive`):** Defines an attribute whose value is automatically calculated from other properties.
- **Design by Contract (`pre`, `post`):** A powerful method for specifying the precise behaviour of operations.
- **`@pre`:** A keyword used in post-conditions to refer to the value of a property *before* the operation was executed.

- **`result:`** A keyword used in post-conditions to refer to the value returned by the operation.
- **Querying (`def`):** Using OCL not just for constraints, but to define reusable functions that can be called to retrieve information from the model.

# 8 State Machine Diagrams

This chapter delves into one of UML's core behavioural modelling tools: the **State Machine Diagram**. While Class Diagrams describe a system's static structure, State Machine Diagrams model its dynamic behaviour. They focus on the lifecycle of a single object, detailing the sequence of states it passes through in response to events.

## 8.1 Why We Need State Machines

Class Diagrams are excellent for defining *what* an object is and *what* it can do (its attributes and operations), but they often fall short in specifying *when* those operations can be called. This temporal ordering, or **protocol**, is crucial for modelling an object's lifecycle.

Consider a simple `Door` object. Its usage is governed by rules:

- A door can only be opened if it is currently closed and unlocked.
- A door can only be closed if it is open and the doorway is clear.
- Turning the key locks an unlocked door and unlocks a locked one.

While these rules can be described using OCL preconditions, this approach provides a fragmented view and doesn't clearly show how the operations are linked together. State Machine Diagrams excel at visualizing this entire lifecycle in a single, coherent view.

## 8.2 Core Concepts of State Machines

A State Machine Diagram visualizes an object's behaviour as a journey through a finite number of **states**. It discretizes behaviour into a set of relevant, abstract states, ignoring irrelevant details. For a `Door`, the relevant states might be `Open`, `Closed`, and `Locked`; its color is an irrelevant detail.

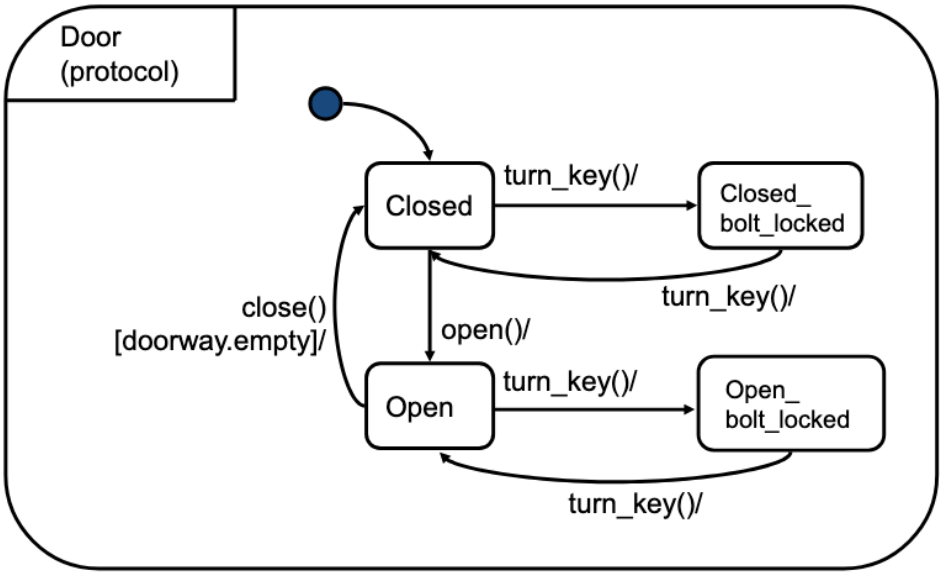



Figure 8.1: A simple protocol state machine diagram for a Door class.

The diagram is composed of a few key elements:

- **State:** A condition during the life of an object in which it satisfies some condition, performs some activity, or waits for an event. It's shown as a rectangle with rounded corners.
- **Transition:** A directed relationship between a source state and a target state. It specifies that an object in the source state will enter the target state when a specific event occurs and certain conditions are met. Transitions are considered instantaneous.
- **Initial State:** A pseudostate indicating the default starting point. It is shown as a solid black circle.
- **Final State:** A state indicating that the object has completed its lifecycle. It is shown as a circle surrounding a smaller solid circle (a bullseye).

### 8.2.1 Anatomy of a Transition

A transition moves an object from one state to another. Its full syntax is `trigger-event [guard] / effect`. Each part defines a specific aspect of the state change.

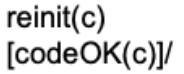



Figure 8.2: A visual representation of a UML transition, labeling the trigger, guard, and effect components.

- **Trigger (or Event):** The event that causes the transition to be considered. There are four main types of events:
  - **Call Event:** The most common type. It corresponds to the reception of a **synchronous call** to an operation of the object. The syntax is simply the operation's name, for example, `open()` or `deposit(amount)`.
  - **Signal Event:** Corresponds to the reception of an **asynchronous signal** by an object. Unlike an operation call, the sender does not wait for a reply. The syntax is the signal's name, for example, `onAlarmDetection`.
  - **Change Event:** This event occurs when a specific boolean expression becomes true. It is used to model reactions to changing conditions. The syntax uses the `when` keyword, for example, `when(sensor.isConnected)`.
  - **Time Event:** Occurs after a certain amount of time has passed or at a specific time. The syntax uses the `after` keyword (for a relative duration) or the `at` keyword (for an absolute time). For example, `after(5 seconds)` or `at(2025-12-25)`.
- **Guard:** An optional boolean condition that must be true for the transition to fire. It is written in square brackets, e.g., `[doorway.empty]`.
- **Effect (or Action):** An optional behaviour that is executed at the instant the transition fires.

## 8.3 Protocol vs. behavioural State Machines

UML distinguishes between two main types of state machines, which serve different purposes.

### 8.3.1 Protocol State Machines

A **Protocol State Machine** specifies the legal sequence of operations that can be called on an object. It acts as a "user manual" for a class, defining its lifecycle without specifying the implementation of its behaviour. Transitions in a protocol state machine have triggers and guards, but **no effects**.

> **i** Key Idea: Protocol State Machine
>
> A Protocol State Machine defines the valid order of operations. It answers the question: "What can happen next?" It is primarily used during analysis and interface specification.

A key question for protocol state machines is what happens when an unexpected event occurs (e.g., a `slam()` event arrives at a `Door`). UML leaves this undefined, meaning it's up to the modeller to specify whether such events are ignored, rejected, or cause an error.

### 8.3.2 behavioural State Machines

A **behavioural State Machine** goes further by specifying the object's reactions to events. It models the **effects** (actions or activities) that are executed when transitions occur or when the object is in a particular state.

> **Key Idea: behavioural State Machine**
>
> A behavioural State Machine defines what an object *does* in response to events. It is a specification of implementation and is used during system design and construction.

In addition to effects on transitions, states themselves can have associated behaviours:

- **Entry Action:** Executed whenever the state is entered (`entry / action`).
- **Exit Action:** Executed whenever the state is exited (`exit / action`).
- **Do Activity:** An ongoing activity performed as long as the object remains in that state (`do / activity`). It can be interrupted by an outgoing transition.

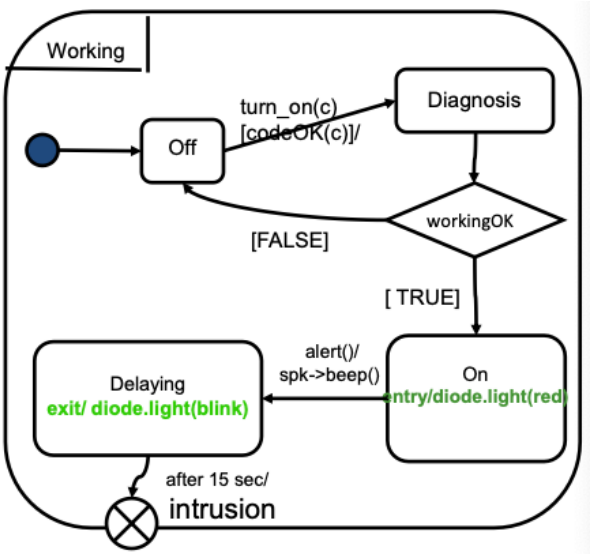



Figure 8.3: A behavioural state machine showing entry and exit actions.

In contrast to protocol state machines, behavioural state machines have a default behaviour for unexpected events: they are simply **ignored**.

## 8.4 Advanced Concepts: Structuring Complexity

Real-world objects can have complex behaviours that would lead to messy, unreadable "flat" state machines. UML provides powerful structuring mechanisms to manage this complexity.

### 8.4.1 Hierarchical States (Composite States)

A **Composite State** is a state that contains its own nested state machine. This allows for abstraction and top-down reasoning. A high-level state like `Working` can be expanded to reveal a detailed sub-machine that handles its internal logic. This is powerful because a single transition leaving the composite state (e.g., `shutdown()`) applies to all of its substates, dramatically reducing visual clutter.

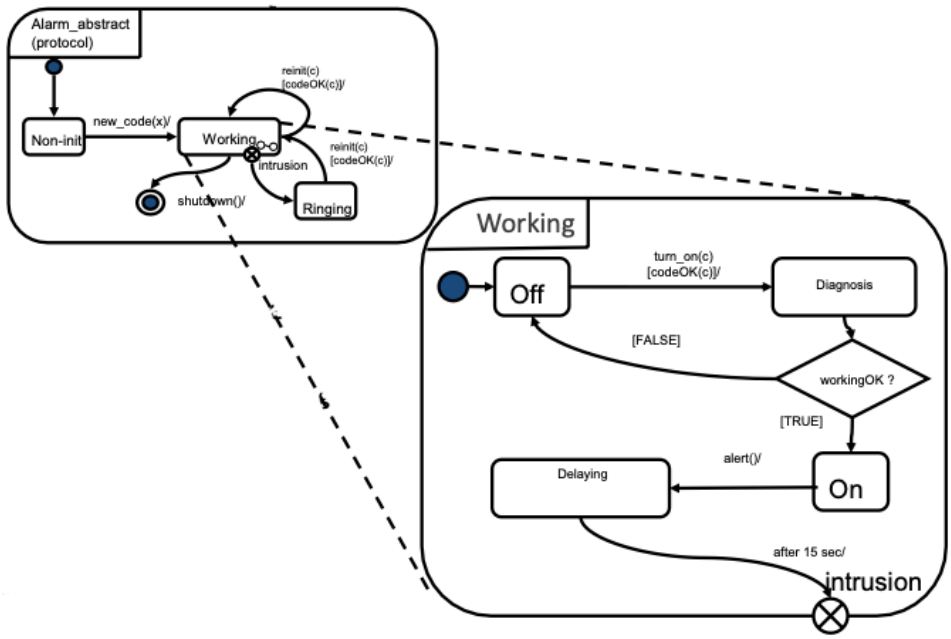



Figure 8.4: A composite state 'Working' containing a full sub-machine.

### 8.4.2 Orthogonal Regions (Concurrency)

An **Orthogonal State** is a composite state that is divided into two or more independent regions, each with its own sub-machine. When an object is in an orthogonal state, it is simultaneously in one substate from *each* region. This is UML's way of modelling independent, concurrent behaviours within a single object, which avoids a "combinatorial explosion" of states.

The primary mechanism for entering and exiting these concurrent regions is through **Fork** and **Join** pseudostates. A fork splits a single incoming transition into multiple concurrent transitions, one for each region. A join synchronizes multiple incoming transitions from different regions into a single outgoing transition.

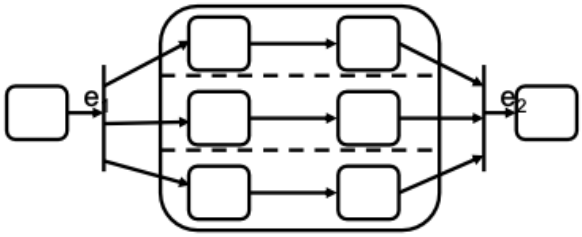



Figure 8.5: A Fork pseudostate splits one transition into three concurrent paths, which are later synchronized by a Join pseudostate.

### 8.4.3 Other Essential Pseudostates

To create sophisticated flows, UML provides several other special states (pseudostates):

- **Choice vs. Junction**: These both represent branches in a transition path, but their semantics are critically different. The key distinction is *when* the guards are evaluated relative to the transition's actions.
  - A **Choice** (diamond) models a **dynamic branch**. Its guards are evaluated *after* the actions on the incoming transition segment have been executed.
  - A **Junction** (circle) models a **static branch**. Its guards are evaluated *before* any actions are executed.


Figure 8.6: With a Junction, the path is chosen based on x's initial value. With a Choice, the path is chosen after x has been set to 0.

- **History States**: A History State (`H` or `H*`) is a pseudostate that remembers the last active substate of a composite state. This is invaluable for modelling interruptions.
  - **Shallow History (`H`)**: Remembers only the immediate substate. If `Connected` is left while in the `Running` state, a transition to `H` will resume in `Running`.
  - **Deep History (`H*`)**: Remembers the full state configuration. If the machine was in the `Fast` sub-state of `Running`, a transition to `H*` would resume in `Fast`.

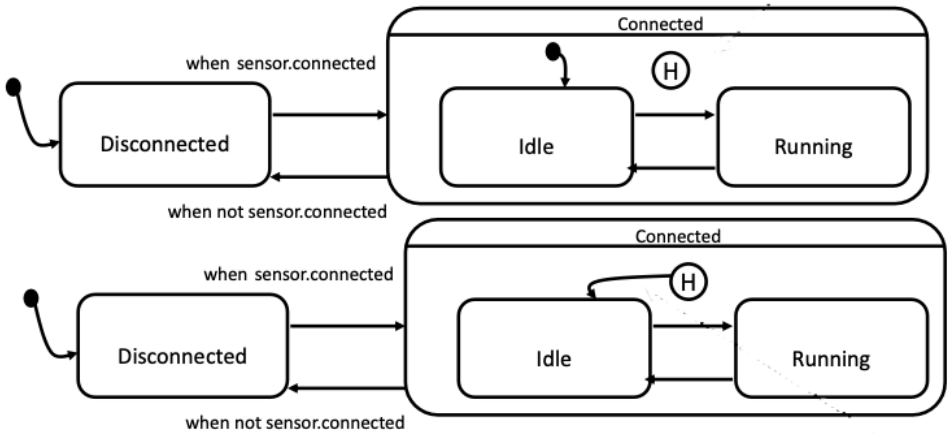



Figure 8.7: A Shallow History (H) connector inside a 'Connected' composite state.

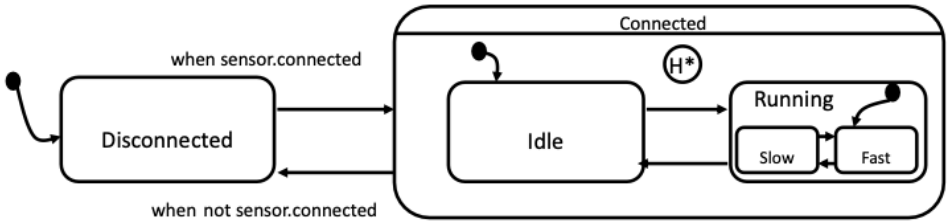



Figure 8.8: A Deep History (H*) connector inside a composite state that has its own nested states.

- **Entry and Exit Points**: These provide fine-grained control for transitions that cross the boundary of a composite state, allowing them to bypass the default initial/final states.
  - An **Entry Point** allows a transition to target a specific internal state, useful for skipping initialization steps.
  - An **Exit Point** allows an internal state to trigger a specific transition out of the composite state, useful for handling specific outcomes like errors.

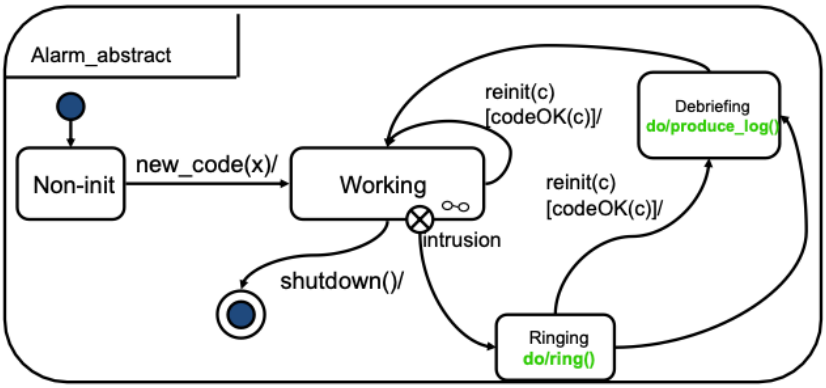



Figure 8.9: An Entry Point allows a higher-level state to transition to a specific substate, bypassing the default initial state.

By using these advanced features, State Machine Diagrams can model even the most complex object lifecycles in a structured, understandable, and precise way.

## 8.5 State Machine Semantics: The Rules of Execution

While State Machine diagrams are visually intuitive, their execution follows a precise set of rules, or semantics. Understanding these rules is essential for correctly interpreting complex diagrams and predicting system behaviour, especially when ambiguities like conflicts or concurrency arise.

### 8.5.1 Configurations and Conflicts

At any moment, a system's state is defined by its **configuration**: the set of all currently active states at every level of the hierarchy.

A **conflict** occurs when a single event could trigger multiple different transitions from the current configuration. For two transitions to be in conflict, they must:

- Be triggered by the same event.
- Have guards that are both true.
- Originate from source states that overlap (e.g., a state and its substate).

### 8.5.2 The UML Priority Rule

To resolve conflicts, UML defines a clear priority rule: **the transition originating from the most deeply nested state has priority**. This means an action defined on a substate will always be chosen over a conflicting action defined on one of its parent states. It's important to note this is a specific convention of UML, other formalisms might use different rules.

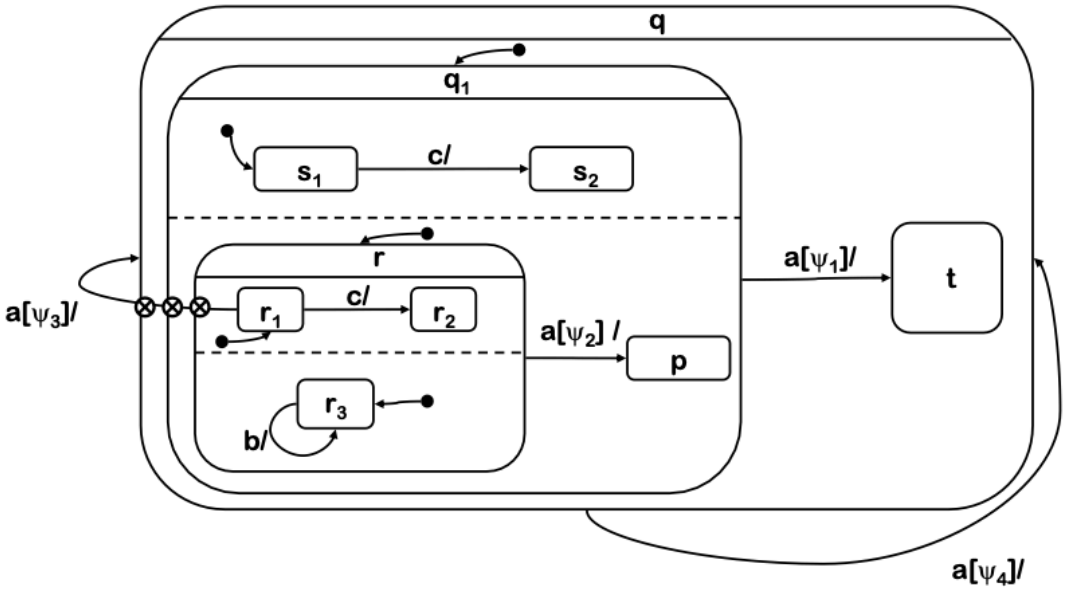



Figure 8.10: A complex hierarchical state machine where the "deepest source state" rule is needed to resolve which transition fires when event 'a' occurs.

### 8.5.3 Non-Determinism: When the Rules Aren't Enough

Sometimes, the priority rule is not sufficient to resolve all ambiguities. This leads to **non-determinism**, where the model allows for more than one possible correct behaviour.

1. **Unresolvable Conflicts**: In an orthogonal state, if an event triggers a transition in one region and a conflicting transition in another, the priority rule does not apply because neither state is "deeper" than the other. The system will make a non-deterministic choice and fire only one of the enabled transitions.

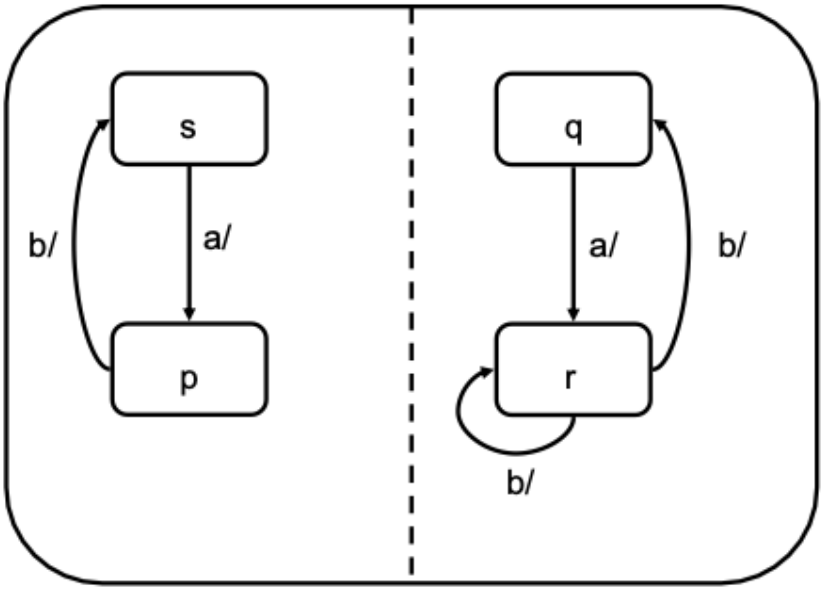



Figure 8.11: In this diagram, if event 'a' occurs, both transitions fire. The final value of 'x' is non-deterministic because the firing order is undefined.

2. **Simultaneous Transitions & Undefined Order**: If an event triggers one transition in each of several orthogonal regions, all transitions will fire. However, the **order in which their actions are executed is undefined**. If these actions modify a shared variable, the final value of that variable can be unpredictable, as it depends on which action runs last. More generally, if the actions are sequences like `a;b;c` and `e;f;g`, any interleaving of these sequences is considered a valid execution.

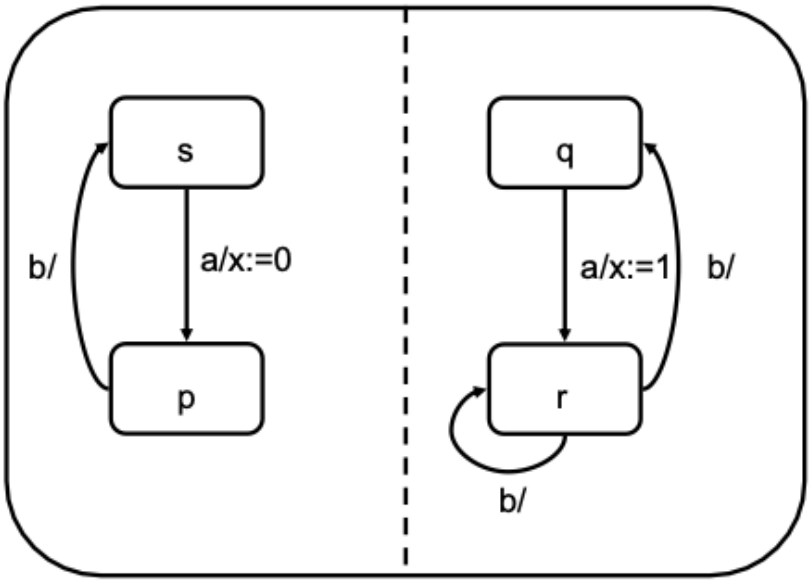



Figure 8.12: In this diagram, if event 'a' occurs, both transitions fire. The final value of 'x' is non-deterministic because the firing order is undefined.

### 8.5.4 Semantics of Junctions and Choices

The distinction between Junction and Choice pseudostates is purely semantic and relates to when guards are evaluated within a single, run-to-completion step:

- **Junction (Static)**: A junction is evaluated "beforehand." The entire path from the source state to the final target state is determined before any actions on the transitions are executed.
- **Choice (Dynamic)**: A choice is evaluated "after." The system executes the action on the incoming transition segment first, and only then evaluates the guards on the outgoing paths to determine where to go next.

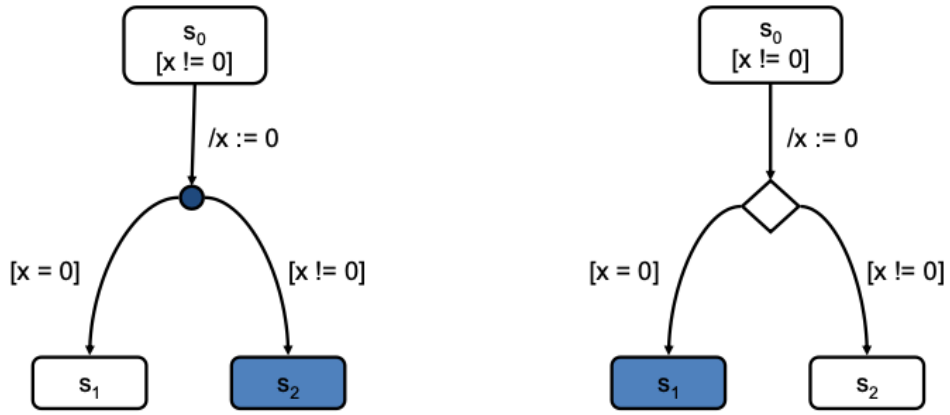



Figure 8.13: With a Junction (left), the path is chosen based on x's initial value. With a Choice (right), the path is chosen after x has been set to 0.

# 9 Practical Exercises: State Machines

This chapter provides a series of practical exercises to apply the concepts of UML State Machines discussed previously. Each exercise challenges you to trace the execution of a complex state machine by applying semantic rules to determine the stable configuration after each event.

---

### 9.0.1 Exercise 1: Orthogonal Regions and History

**Problem:** Consider the state machine diagram below. Starting from the initial configuration `{A}`, trace the system's configuration through the given sequence of events.

Click to see the solution

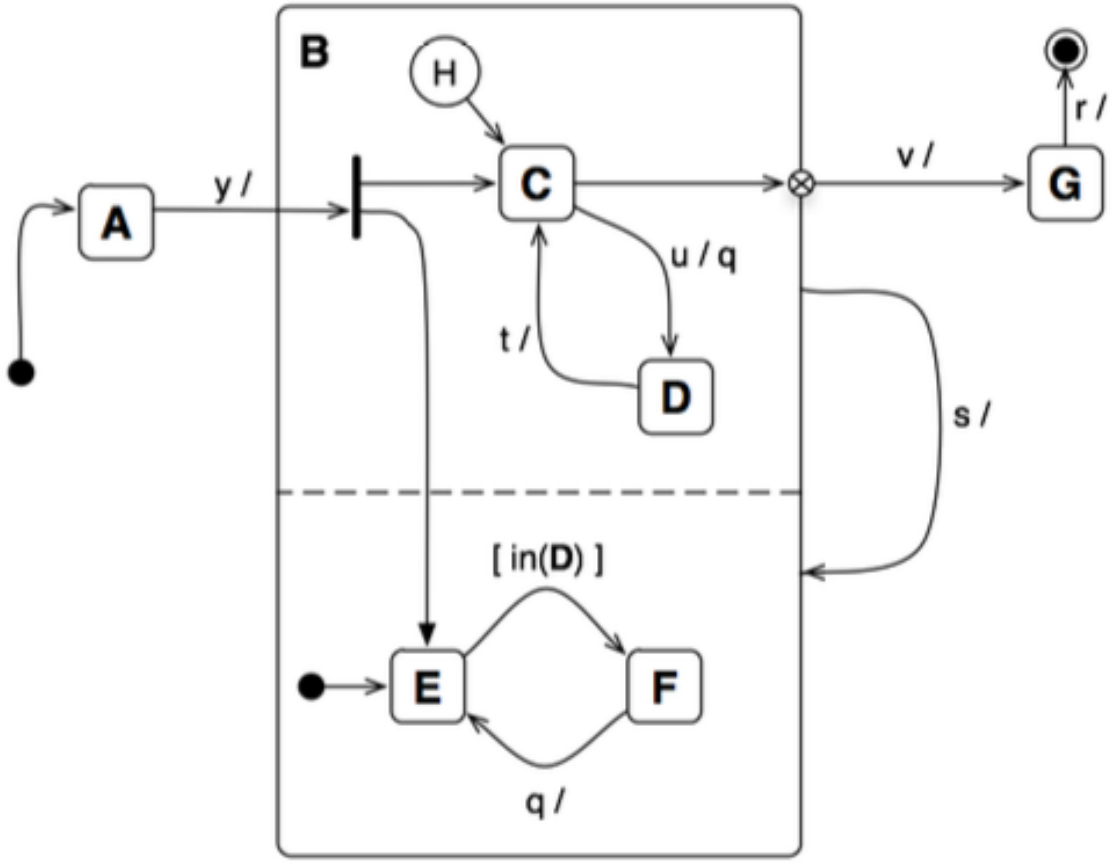



Figure 9.1: State Machine for Exercise 1.

**Correction Details:**

| Step | Initial Configuration | Event | Final Configuration | Justification |
|---|---|---|---|---|
| 1 | `{A}` | y | `{B,C,E}` | `A -> B` is the only possible transition; `C` and `E` are entered via the fork. |
| 2 | `{B,C,E}` | u | `{B,D,F}` | The `C -> D` transition fires. Its reaction sends a `q` event, which is consumed. The system enters `D`. Because the `in(D)` guard is now true, a transition to `F` is taken. The final stable state is `{B,D,F}`. |

| | | | |
|---|---|---|---|
| 3 | {B,D,F} | q | {B,D,F} The system leaves F for E. Since it is still in state D, the in(D) guard is true, causing an immediate transition back to F for a stable state of {B,D,F}. |
| 4 | {B,D,F} | t | {B,C,F} The system remains in B. The D -> C transition occurs. The system remains in F. |
| 5 | {B,C,F} | q | {B,C,E} The system remains in B. The F -> E transition occurs. The system remains in C. |
| 6 | {B,C,E} | s | {B,C,E} The system leaves B and re-enters it. C is restored because of the History state (H). E is activated because it's the initial state of its region. |
| 7 | {B,C,E} | u | {B,D,F} Same as step 2. |
| 8 | {B,D,F} | s | {B,D,F} The system leaves B and re-enters it. D is restored due to History. E is activated as the initial state, but an immediate transition to F occurs because the in(D) guard is true. |
| 9 | {B,D,F} | t | {B,C,F} The system remains in B. The D -> C transition occurs. The system remains in F. |
| 10 | {B,C,F} | v | {G} The system leaves B for G. This is enabled by the exit point, as the active state C is connected to it. |

**Key Concepts Illustrated:**

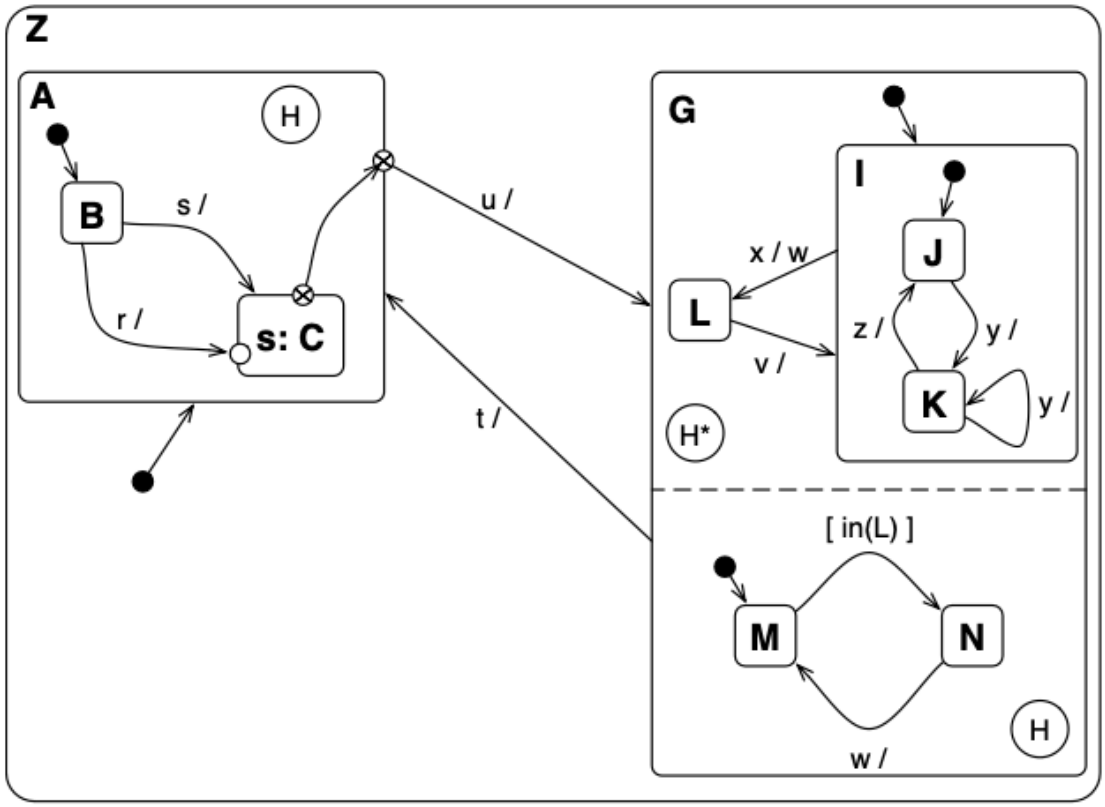
- **Orthogonal Regions:** Modeling concurrent states that are active simultaneously.
- **Run-to-Completion Semantics:** How an event and its subsequent internal reactions are processed to completion before the next external event is handled.
- **Shallow History (H):** The mechanism for remembering and restoring the last active substate within a region.
- **Forks and Exit Points:** The syntax and semantics for managing the start and end of concurrent flows.

---

### 9.0.2 Exercise 2: Submachines and Deep History

**Problem:** Consider the state machine Z, which uses the submachine C. Starting from the initial configuration {Z, A, B}, trace the system's configuration for the given event sequence.

Click to see the solution



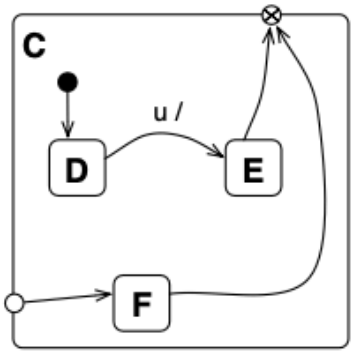




Figure 9.2: State Machine and Submachine for Exercise 2.

**Correction Details:**

| Step | Initial Config. | Event | Final Config. | Justification |
|---|---|---|---|---|
| 1 | {Z, A, B} | s | {Z, A, C, D} | A remains active; the B -> C transition occurs; D is the initial state of the submachine C. |
| 2 | {Z, A, C, D} | u | {Z, A, C, E} | The D -> E transition occurs. E is an exit state that enables the A -> G transition. |
| 3 | {Z, A, C, E} | r | {Z, A, C, E} | Nothing happens. |
| 4 | {Z, A, C, E} | u | {Z, G, I, J, M} | The A -> G transition fires via the active exit point. I and J are activated as initial states in the top region, and M is the initial state of the bottom orthogonal region. |
| 5 | {Z, G, I, J, M} | x | {Z, G, L, N} | The I -> L transition fires, launching and consuming event w. The in(L) guard is now true, so the M -> N transition also fires. |
| 6 | {Z, G, L, N} | v | {Z, G, I, J, N} | Z and G remain active. The L -> I transition fires. J is restored because of the deep history state (H*). |
| 7 | {Z, G, I, J, N} | z | {Z, G, I, J, N} | Nothing happens. |
| 8 | {Z, G, I, J, N} | y | {Z, G, I, K, N} | The J -> K transition occurs. |
| 9 | {Z, G, I, K, N} | t | {Z, A, C, D} | The system leaves G for A. C is restored due to shallow history (H). D is activated as the initial state of C because the history of A is not deep. |
| 10 | {Z, A, C, D} | u | {Z, A, C, E} | Same as step 2. |
| 11 | {Z, A, C, E} | u | {Z, G, I, K, N} | The system leaves A for G via the exit point. In the top region, {I, K} is restored due to deep history. In the bottom region, N is restored due to its shallow history. |
| 12 | {Z, G, I, K, N} | x | {Z, G, L, N} | The I -> L transition fires, launching event w. This w event causes N -> M. The system enters L, making the in(L) guard true, which causes an immediate M -> N transition. The final stable state is {Z, G, L, N}. |

**Key Concepts Illustrated:**

- **Submachine States:** How a state can be a reference to another, reusable state machine definition, promoting encapsulation.
- **Deep History (`H*`) vs. Shallow History (`H`):** This exercise provides a perfect side-by-side comparison. Shallow history restores the immediate substate, while deep history restores the entire nested configuration.
- **Interaction between Model Components:** Tracing how transitions and history mechanisms work across the boundaries of composite states and submachines.

---

### 9.0.3 Exercise 3: Advanced Semantics and Priority Rules

**Problem:** For the highly complex state machine below, trace the configuration starting from `{A, B, F, H}`. This exercise will test your understanding of conflict resolution.

Click to see the solution

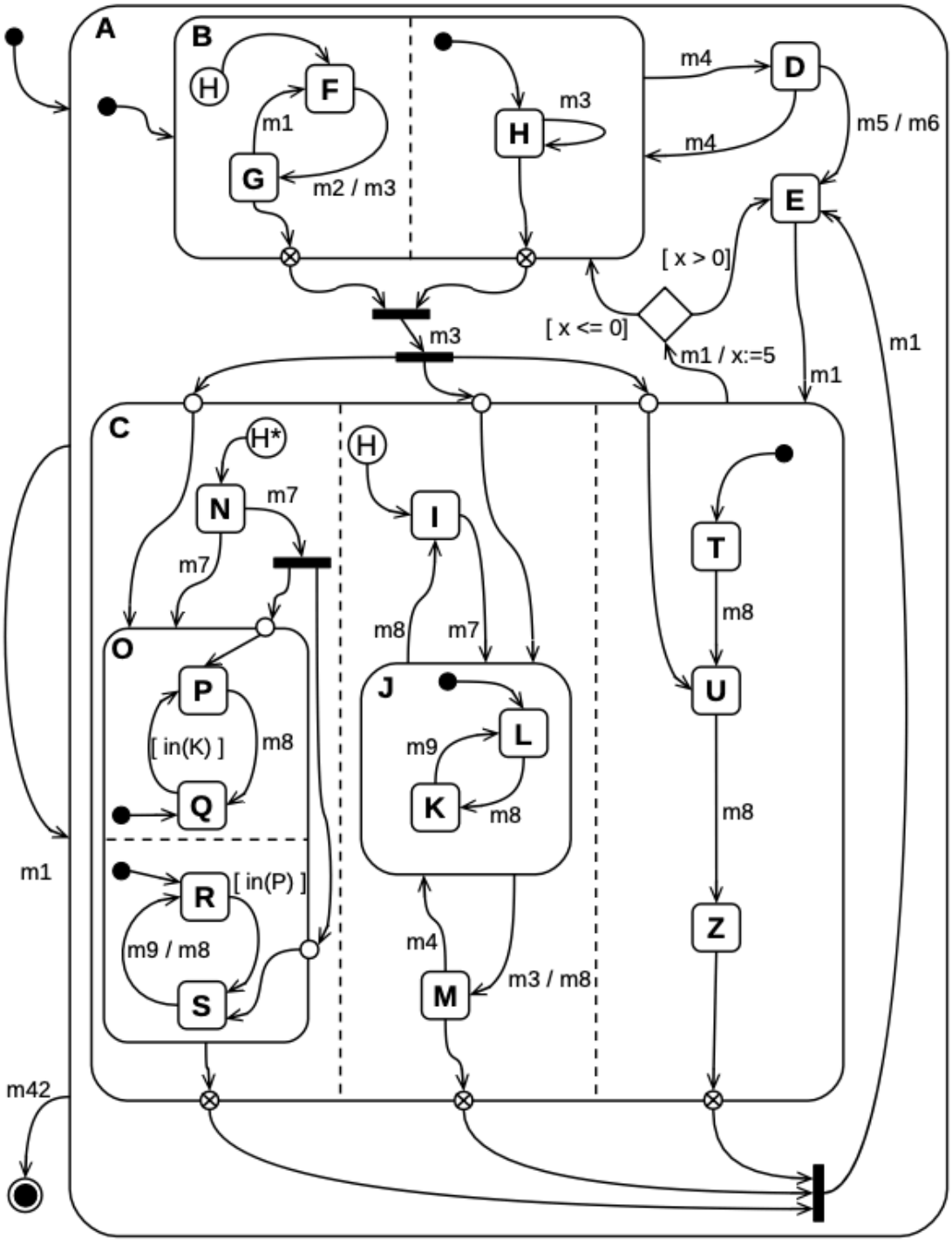



Figure 9.3: State Machine for Exercise 3.

**Correction Details:**

| No. | Initial Config. | Event | Final Config. | Justification |
|---|---|---|---|---|
| 1 | {A, B, F, H} | m1 | {A, B, F, H} | The system leaves A and re-enters it. B is the initial state, F is restored due to History, and H is the initial state. |
| 2 | {A, B, F, H} | m2/m3 | {A, B, G, H} | m2 is the event, m3 is the reaction. The F -> G transition fires, launching m3. A conflict arises between the high-level B -> C transition and the nested H -> H self-transition. The H -> H transition has priority because it is more deeply nested. m3 is consumed, and the final stable state is {A, B, G, H}. |
| 3 | {A, B, G, H} | m4 | {A, D} | The B -> D transition occurs. |
| 4 | {A, D} | m4 | {A, B, G, H} | The D -> B transition occurs. G is restored due to history, and H is activated as the initial state. |
| 5 | {A, B, G, H} | m3 | {A, B, G, H} | The system leaves H and re-enters it; the H -> H transition is the most deeply nested and thus has priority. |
| 6 | {A, B, G, H} | m1 | {A, B, F, H} | The G -> F transition is the most deeply nested. |
| 7 | {A, B, F, H} | m1 | {A, B, F, H} | The system leaves A and re-enters it. B is the initial state, F is restored due to History, and H is the initial state. |
| 8 | {A, B, F, H} | m8 | {A, B, F, H} | Nothing happens. |

| | | | | |
|---|---|---|---|---|
| 9 | {A, B, F, H} | m3/m8 | {A, B, F, H} | The system leaves `H` and re-enters it. The `H -> H` transition is the most deeply nested (the reaction `m8` is not considered here). |
| 10 | {A, B, F, H} | m1 | {A, B, F, H} | Same as step 7. |

**Key Concepts Illustrated:**

- **The UML Priority Rule:** This exercise provides a clear, practical example of conflict resolution. The rule—"deepest source state wins"—is not just theoretical; it dictates the machine's behavior.
- **Reactions and Internal Events:** Demonstrates how a transition's action can trigger a new event that is immediately processed within the same "run-to-completion" step.
- **Complex Transitions:** Shows how to trace execution across multiple levels of hierarchy and through exit points that depend on the configuration of multiple orthogonal regions.