UNIVERSITÀ DI PISA
CDL INGEGNERIA INFORMATICA

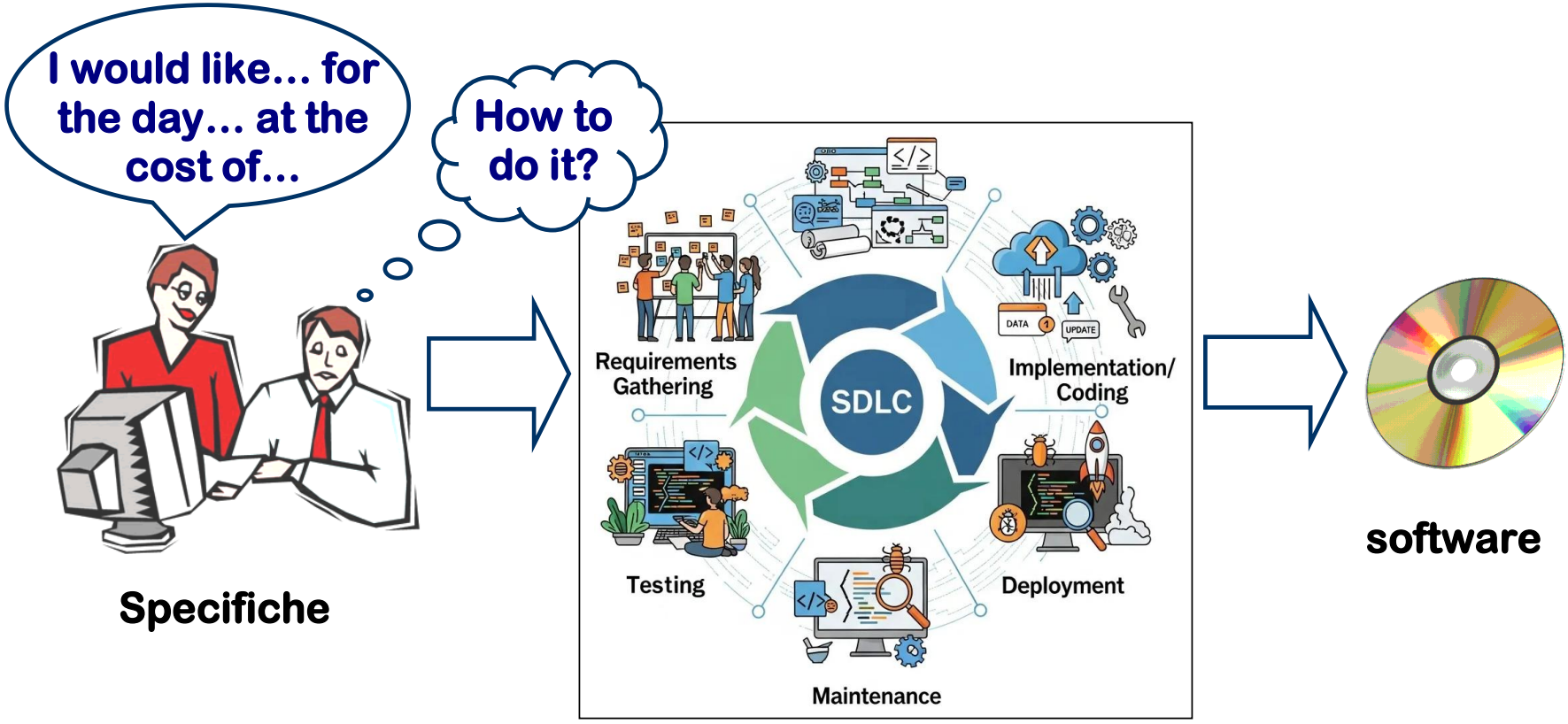
Ingegneria del Software



Mario G.C.A. Cimino

Software development process

The Software Development Process defines the *who*, *what*, *when*, and *how* of software development life cycle. Unified Process (UP) and Unified Modeling Language (UML).

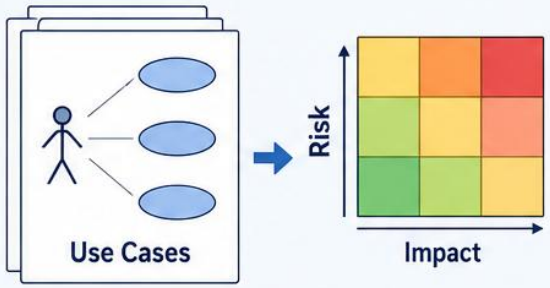


Software Development Life Cycle

UP has three basic axioms

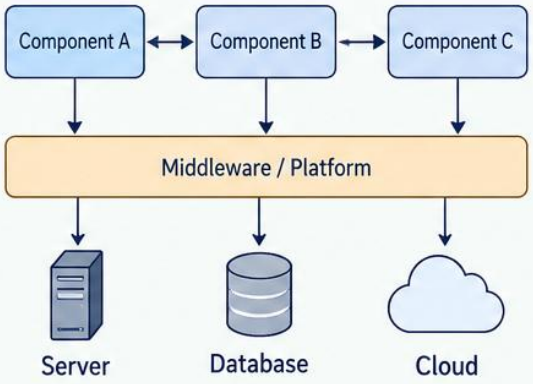
1 Use case & risk driven

Capturing requirements, risk analysis based software construction.



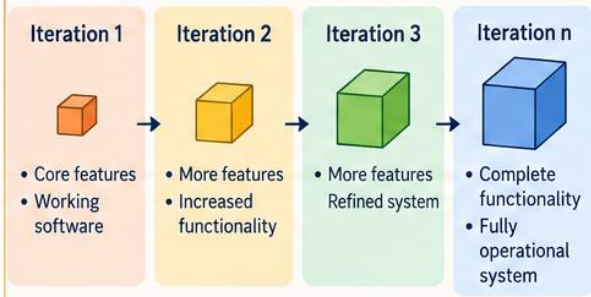
2 Architecture centric

How the system is broken down into components, how those components interact and are deployed on hardware.



3 Iterative and incremental

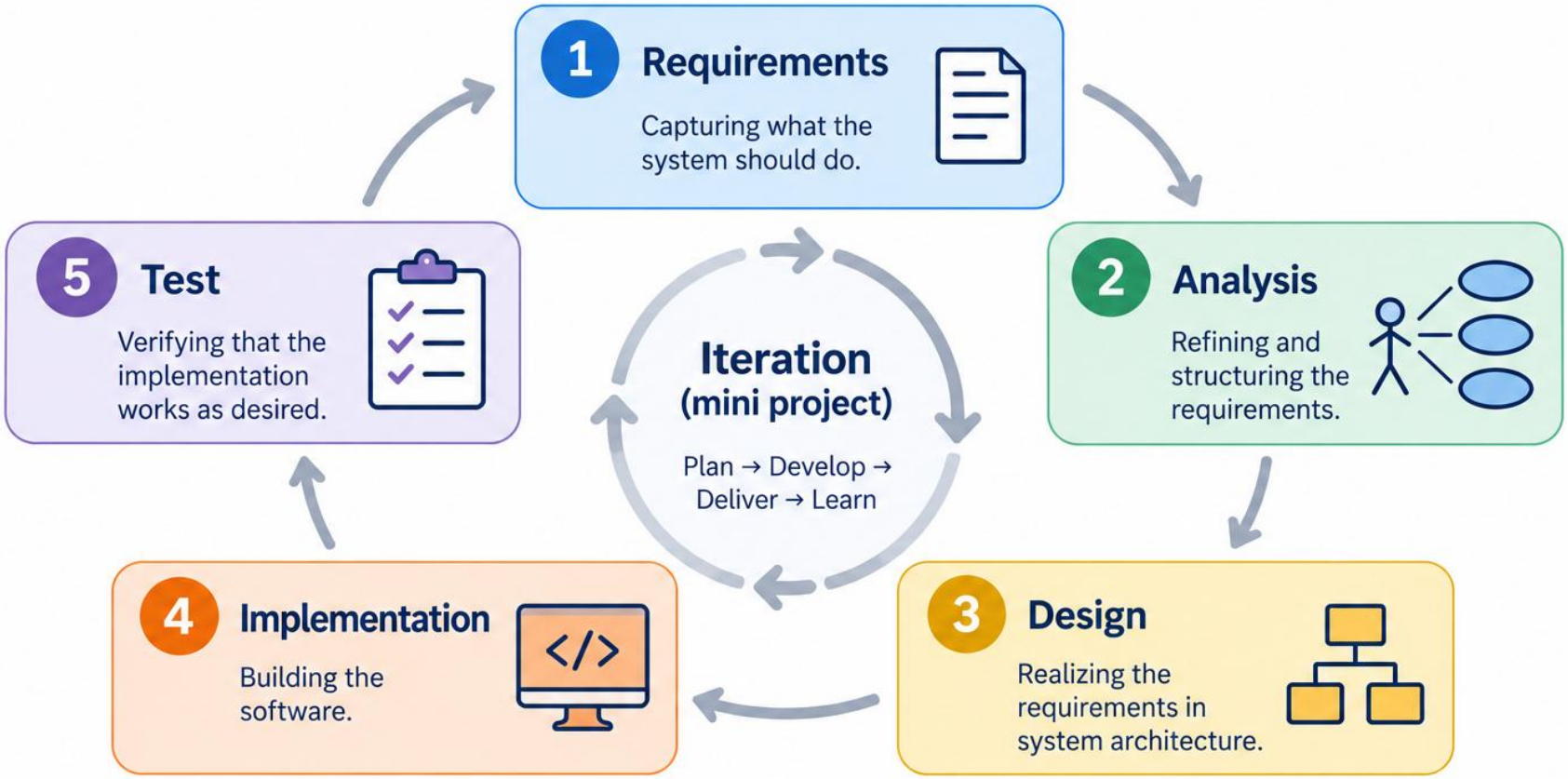
Break the project into small subprojects (the iterations) that deliver system functionality in chunks, or increments, leading to a fully functional system.



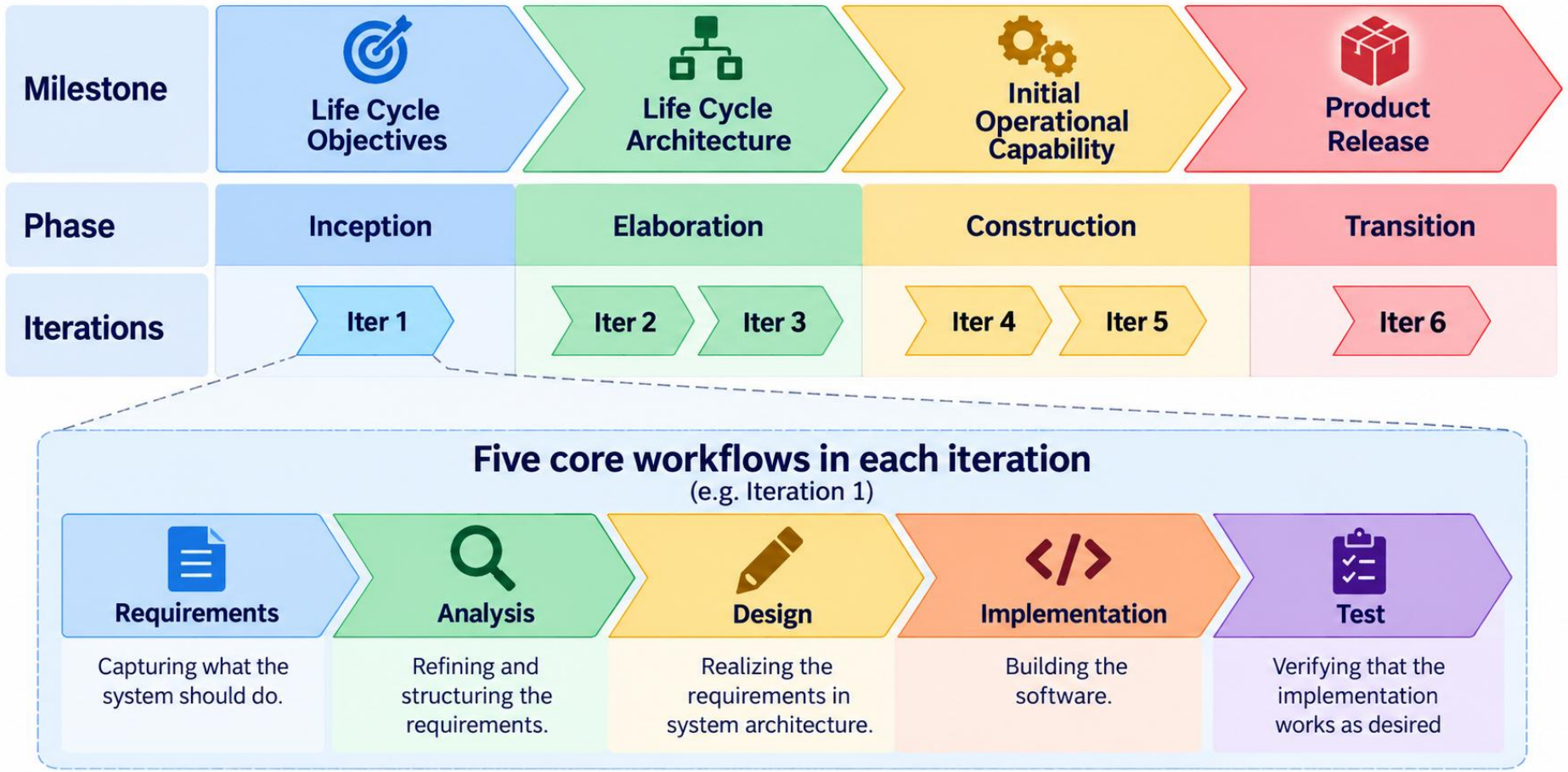
Use cases and risks guide what to build, the architecture defines how to build it, and iterative increments deliver it.

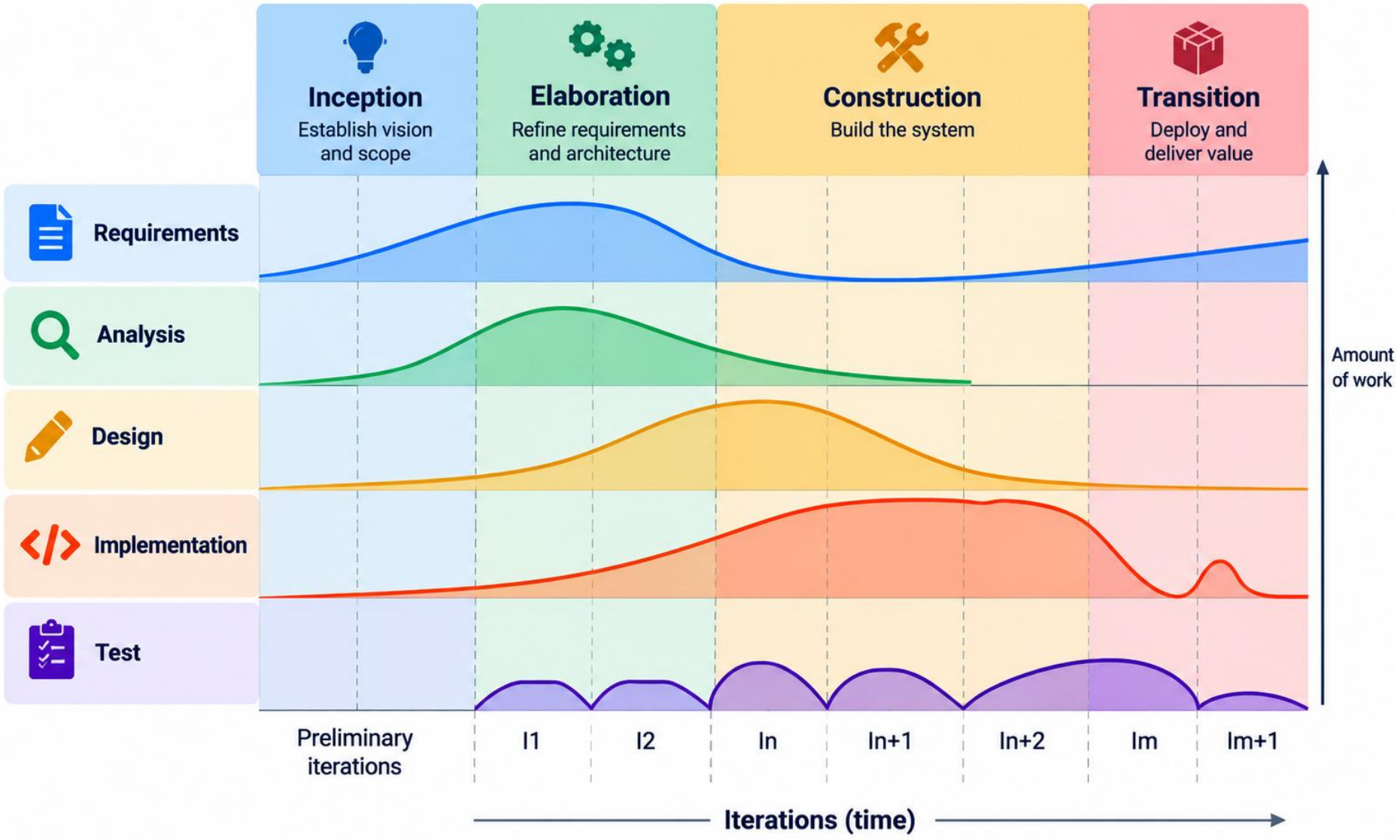
Five core workflows in each iteration

(a “mini project”)



Each iteration goes through these five workflows, delivering a working increment of the system.


Milestone
Life Cycle Objectives
Life Cycle Architecture
Initial Operational Capability
Product Release
Phase
Inception
Elaboration
Construction
Transition
Iterations
Iter 1
Iter 2
Iter 3
Iter 4
Iter 5
Iter 6
Five core workflows in each iteration
(e.g. Iteration 1)
Requirements
Capturing what the system should do.
Analysis
Refining and structuring the requirements.
Design
Realizing the requirements in system architecture.
Implementation
Building the software.
Test
Verifying that the implementation works as desired


Inception
Establish vision and scope
Elaboration
Refine requirements and architecture
Construction
Build the system
Transition
Deploy and deliver value
Requirements
Analysis
Design
Implementation
Test
Amount of work
Preliminary iterations
I1
I2
In
In+1
In+2
Im
Im+1
Iterations (time)

Phases of the Unified Process

PHASE	GOALS	FOCUS	MILESTONE
 **Inception**	• Capturing essential requirements to help scope the system • Feasibility: technical prototype and proof of concept • Business case to demonstrate value	• Requirements and analysis workflows • Some design and implementation (prototype) • No testing – throwaway prototype	**Life Cycle Objectives** (requirements / features / constraints, initial use cases)
 **Elaboration**	• Create executable architectural baseline • Capture use cases to ~80% of functional requirements • Refine risk assessment and quality attributes • Create detailed plan for construction • Formulate a bid (resources, time, cost)	• Requirements, analysis and design workflows • Implementation: build initial operational capability • Test the initial operational capability (alpha test, internal)	**Life Cycle Architecture**
 **Construction**	• Complete requirements, analysis and design • Move from architectural baseline to final system • Maintain system architecture integrity	• Implementation and testing • Build the initial operational capability • Test the initial operational capability	**Initial Operational Capability** (software system finished for beta testing)
 **Transition**	• Deploy the system after beta testing • Correct defects and prepare user site for the new software • Create user manuals and provide user consultancy • Conduct a post-project review	• No requirements or analysis • Finish implementation and complete test workflows • Modify design if problems arise in beta testing • User acceptance testing (user community)	**Product Release** (product accepted into the user community)

Inception

Conditions to attain for the Life Cycle Objectives

Conditions of satisfaction	Deliverable
The stakeholders have agreed the project objectives	Vision document that states the project's main requirements, features and constraints
System scope has been defined and agreed with the stakeholders	Initial use case model (only about 10% to 20% complete)
Key requirements have been captured and agreed with the stakeholders	Project glossary
Cost and schedule estimates have been agreed with the stakeholders	Initial project plan
A business case has been raised by the project manager	Business case
The project manager has performed a risk assessment	Risk assessment document or database
Confirmation of feasibility through technical studies and/or prototyping	One or more throwaway prototypes
An outline architecture	Initial architecture document

Elaboration

Conditions to attain for the Life Cycle Architecture

Conditions of satisfaction	Deliverable
A resilient, robust executable architectural baseline has been created	Executable architectural baseline
The executable architectural baseline demonstrates that important risks have been identified and resolved	UML Static Model UML Dynamic Model UML Use Case Model
The vision of the product has stabilized	Vision document
The risk assessment has been revised	Updated risk assessment
The business case has been revised and agreed with the stakeholders	Updated business case
A project plan has been created in sufficient detail to enable a realistic bid for time, money and resources in the next phases	Updated project plan
The stakeholders agree to the project plan	
The business case has been verified against the project plan	Business case and project plan
Agreement is reached with the stakeholders to continue the project	Sign-off document

Construction

Conditions to attain for the Initial Operational Capability

Conditions of satisfaction	Deliverable
The software product is sufficiently stable and of sufficient quality to be deployed in the user community	The software product The UML model Test suite
The stakeholders have agreed and are ready for the transition of the software to their environment	User manuals Description of this release
The actual expenditures vs. the planned expenditures are acceptable	Project plan

Transition

Conditions to attain for the Product Release

Conditions of satisfaction	Deliverable
Beta testing is completed, necessary changes have been made, and the users agree that the system has been successfully deployed	The software product
The user community is actively using the product	
Product support strategies have been agreed with the users and implemented	User support plan User manuals

What is a Risk?

Any uncertain event or condition that, if it occurs, could negatively affect the project's success: its **schedule, budget, quality, scope**, or **whether the system works at all.**

Risk-driven development in the UP

The order in which work is done is chosen largely to **confront the biggest risks as early as possible**, when they are cheapest to deal with.

Types of Risk

Technical and architectural risks

- Unproven technology
- Performance or scalability doubts
- Difficult integrations with legacy systems

Requirements risks

- Misunderstood needs
- Unstable scope
- Building the wrong product

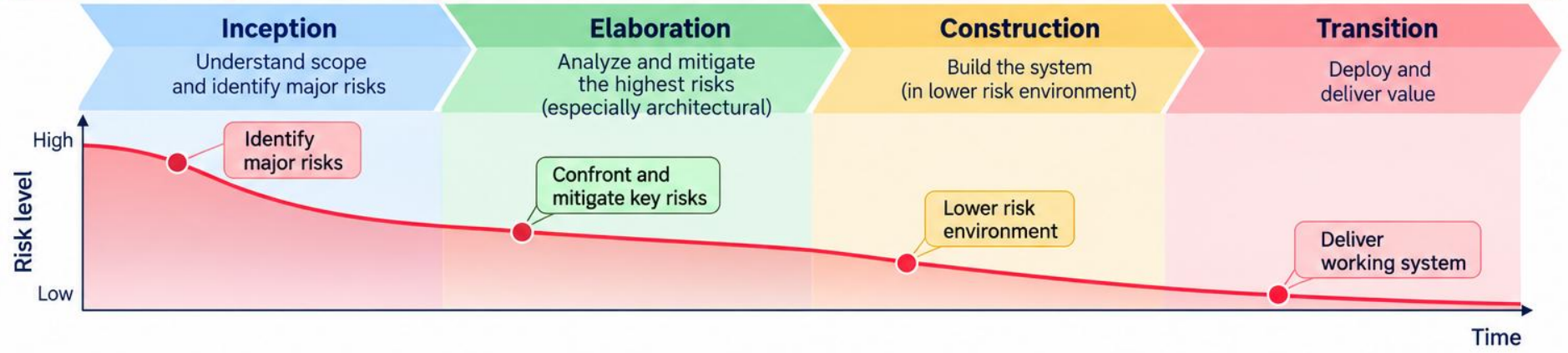
Resource and skill risks

- The team lacks experience with a tool or domain
- Key people are unavailable

Business and political risks

- Funding
- Market timing
- Stakeholder support
- Competitors

Addressing risks early in the Unified Process



Risk Management Process



Keep a risk list

Identify risks and describe them.



Rank by probability × impact

(1–5 × 1–5).



Plan a response

For each risk choose:

- Avoid
- Mitigate
- Transfer
- Accept



Revisit each iteration

Update the risk list and adjust plans.



Let top risks influence the next iteration's plan

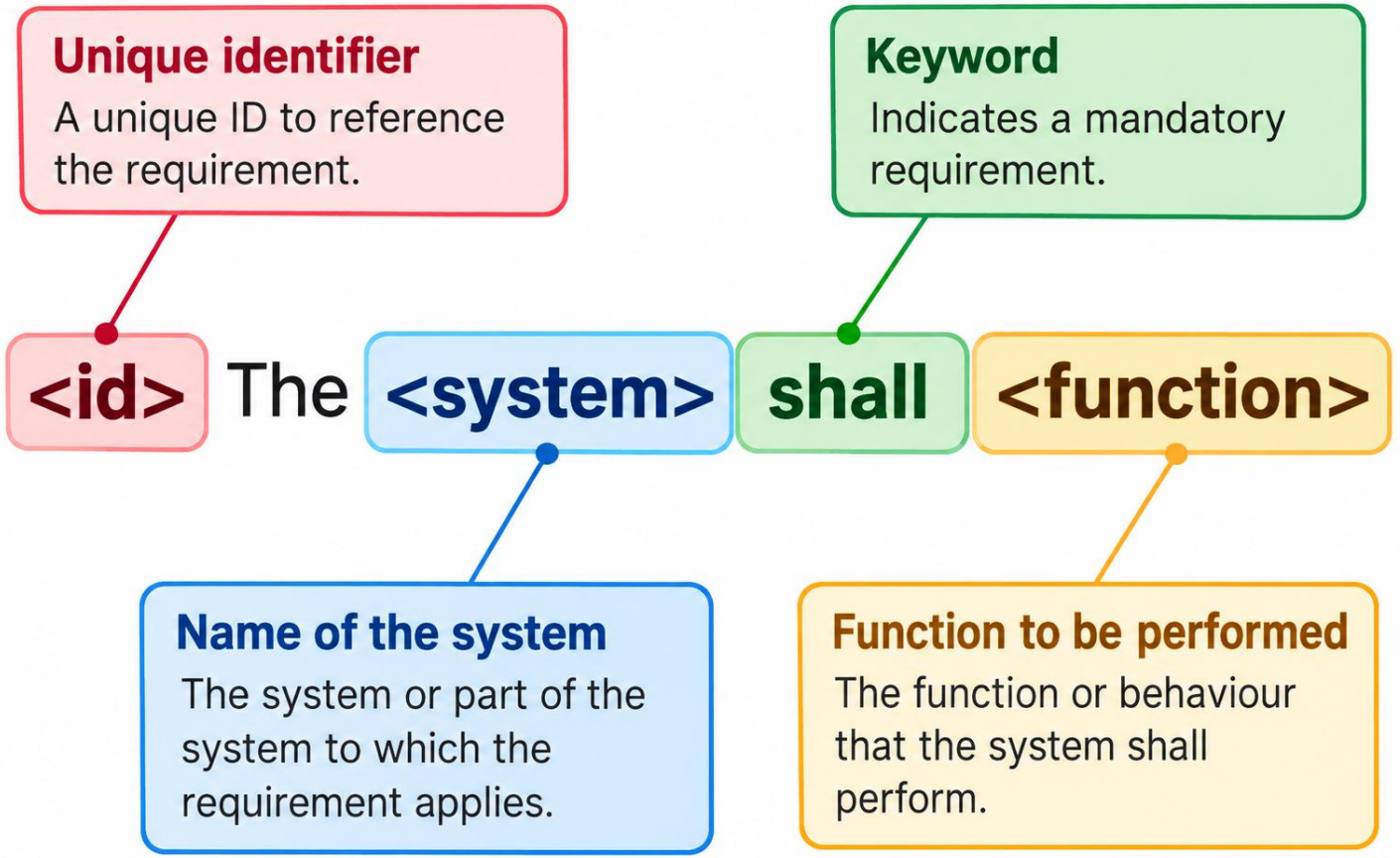
Choose work that best reduces risk.

Example Risk List (extract)

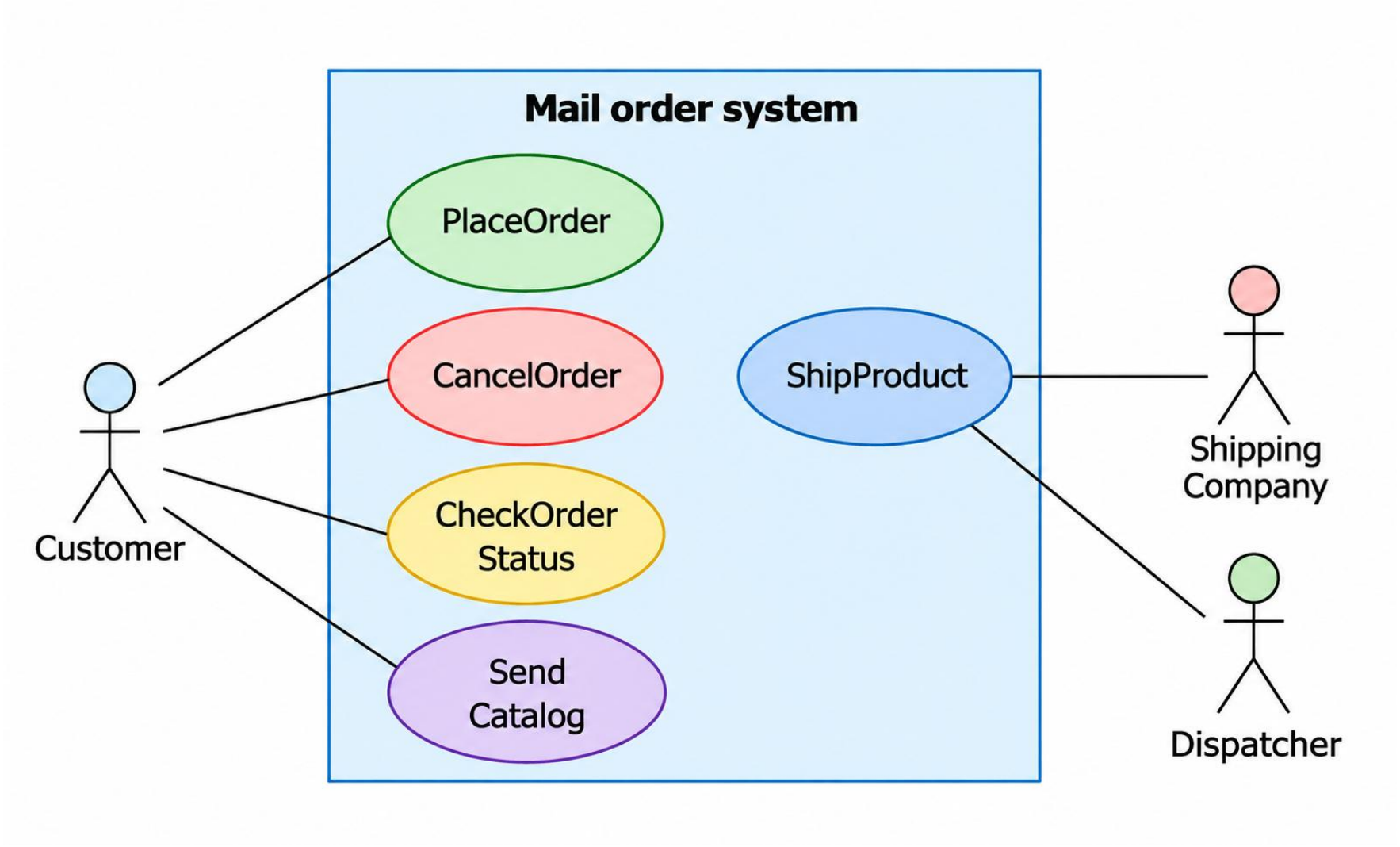
ID	Risk description	Type	Probability (1–5)	Impact (1–5)	Exposure (P × I)	Response plan
R1	New ML library may not meet performance requirements	Technical	4	5	20	Mitigate (prototype and performance tests)
R2	Requirements may change frequently	Requirements	3	4	12	Mitigate (continuous stakeholder feedback)
R3	Key developer may become unavailable	Resource	2	4	8	Transfer (cross-training)
R4	Insufficient market adoption	Business	2	5	10	Accept (monitor, prepare contingency)

Requirements workflow

Requirements: statements on what the system should do (functional) and how it should do it (constraints, properties, non-functional)



Use case: a specific goal that an actor achieves by interacting with a system. It captures what the system must do from the user's or external actor's point of view, without describing the internal implementation.



Project Glossary: a dictionary of key business terms and definitions.

Term	Definition
Customer	A person who purchases products through the Mail Order System.
Order	A request made by a customer to purchase one or more products.
Product	An item available for purchase through the system.
Catalog	The collection of products currently offered to customers.

Detailed use case: structured description of how an actor interacts with a system to achieve a specific goal. It explains the normal sequence of steps, possible alternative or exceptional flows, and the conditions that must hold before and after the interaction.
Precondition: condition that must already be true before the use case can begin.
Postcondition condition that must be true after the use case finishes successfully.

Use Case Name: Place Order
Primary Actor: Customer
Preconditions:
- The customer has selected at least one product.
Flow of Events:
1. The customer chooses to place an order.
2. The system displays the selected products and the total price.
3. The customer enters shipping information.
4. The customer provides payment information.
5. The system validates the entered data.
6. The system calculates the final cost, including shipping and taxes.
7. The customer confirms the order.
8. The system assigns a unique order number.
Postconditions:
- A valid order has been stored in the system.

Format of a step of a use case:

<number> The <something> <some action>.

Keyword IF

2. If the Customer selects “delete item”
   2.1 The system removes the item from the basket.
3. If the Customer types in a new quantity
   3.1 The system updates the quantity of the item in the basket.

Alternative Flows

Sometimes you can’t easily express branching using If. This is particularly true for things that can happen at any point in time.

3. The system displays a list of all items
Postconditions: ...
Alternative flow 1: 1. At any time the Customer may leave the shopping basket screen.
Postconditions: ...
Alternative flow 2: 1. At any time the Customer may leave the system.
Postconditions: ...

Keyword For

To repeat an action a predefined number of times within a flow of events.

5. If the system finds some matching products then
    5.1. For each product found
        5.1.1. The system displays a thumbnail sketch of the product.
        5.1.2. The system displays a summary of the product details.
        5.1.3. The system displays the product price.
6. Else
    6.1. The system tells the Customer that no matching products could be found.

n. For (iteration expression)
    n.1. Do something
    n.2. Do something else
    n.3. ...
n+1.

Keyword While

To repeat an action in a flow of events while some Boolean condition is true.

n. While (Boolean condition)

n.1. Do something

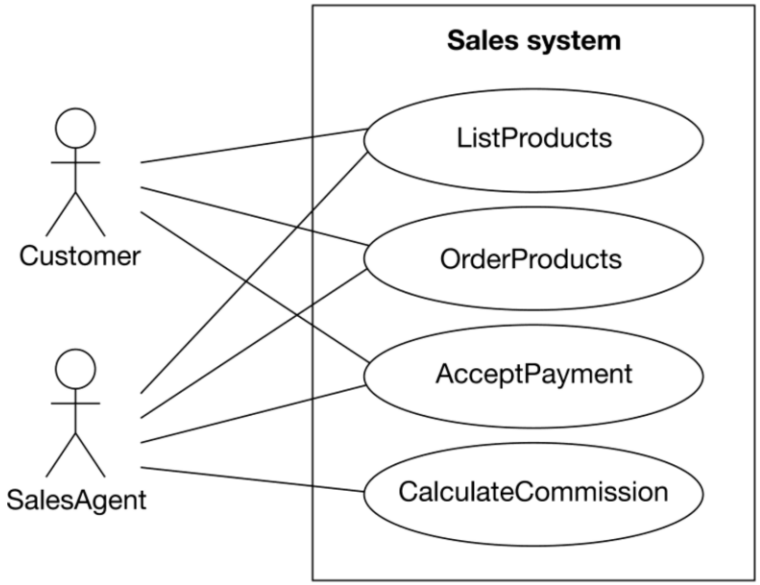
n.2. Do something else

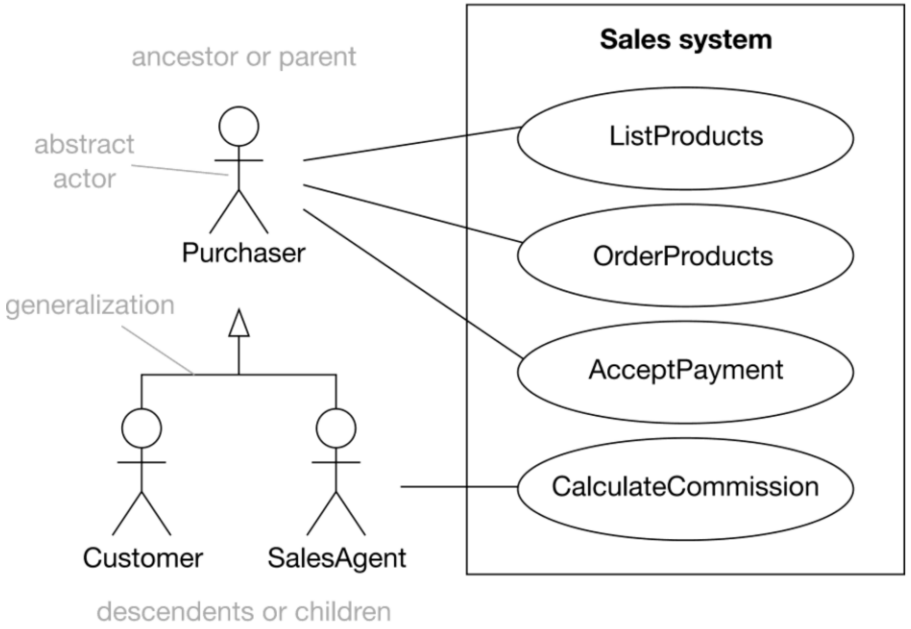
n.3. …

n+1.

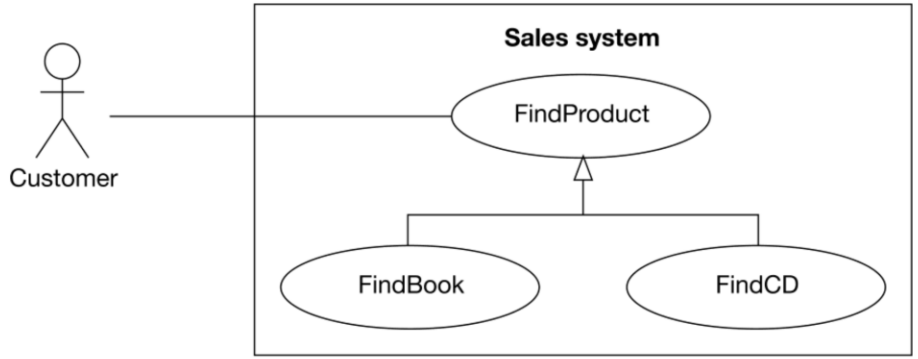
3. While the Customer is browsing the company details
   3.1. The system plays some background music.
   3.2. The system displays special offers in a banner ad.

✓ *Actor generalization*: the descendent actors inherit the roles and relationships to use cases held by the parent actor





✓ *Use case generalization*: the child use case inherits features from the parent use case, can add or change (override) inherited features (pre/post condition, steps in flow…)



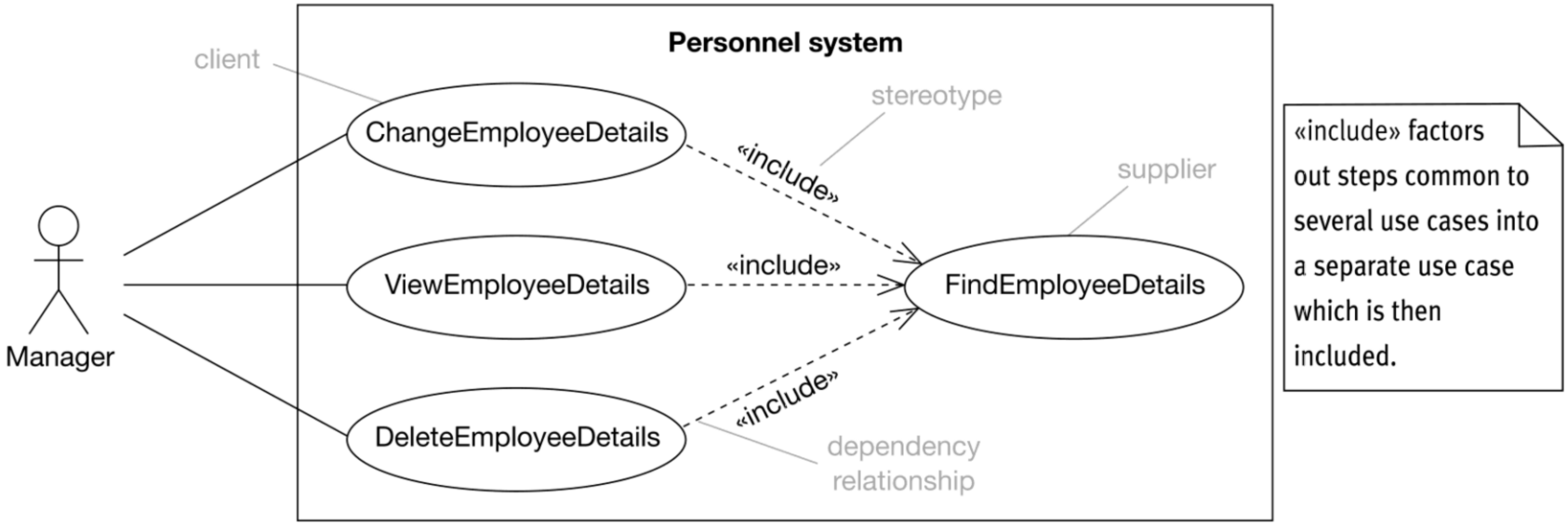
Feature is …	Typographical convention
Inherited without change from the parent	Normal text
Overridden	*Italic text*
Added	**Bold text**

Use case: FindProduct
ID: UC12
Actors: Customer
Preconditions:
Flow of events: 1. The Customer selects “find product”. 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer’s criteria. 5. If the system finds some matching products then 5.1. The system displays a list of the matching products. 6. Else 6.1. The system tells the Customer that no matching products could be found.
Postconditions:
Alternative flow: 1. At any point the Customer may move to a different page.
Postconditions:

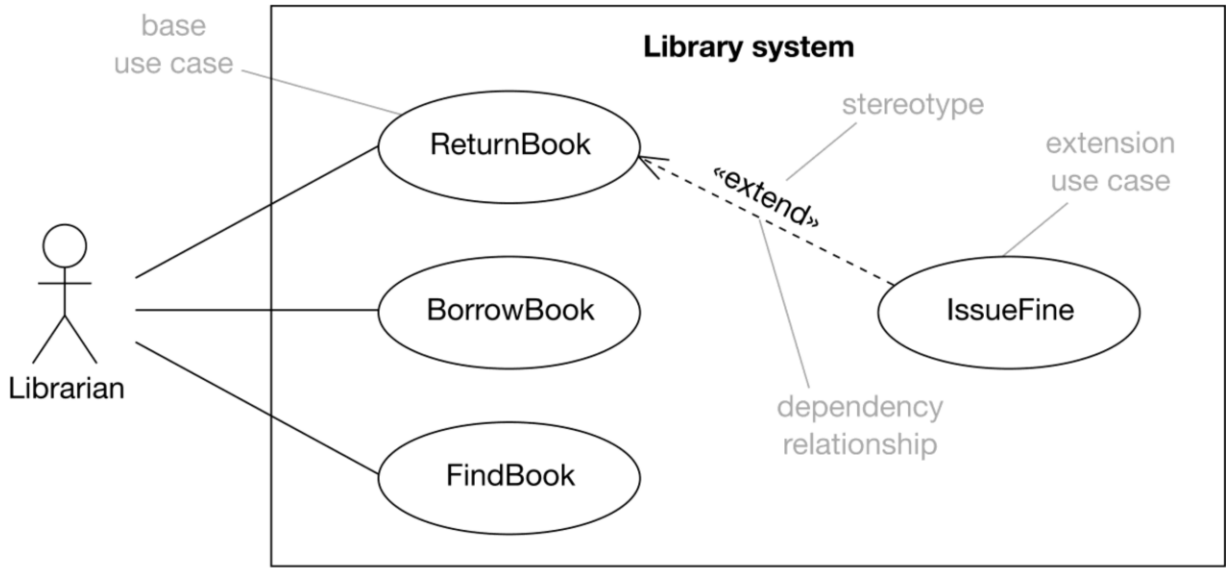
Child use case: FindBook
ID: UC16
Parent Use Case ID: UC12
Actors: Customer
Preconditions:
Flow of events: *1. The Customer selects “find book”.* *2. The system asks the Customer for book search criteria consisting of author name, title, ISBN, or topic.* 3. The Customer enters the requested criteria. *4. The system searches for books that match the Customer’s criteria.* *5. If the system finds some matching books then* *5.1. The system displays a page showing details of a maximum of five books.* **5.2. For each book on the page the system displays the title, author, price, and ISBN.** **5.3. While there are more books** **5.3.1. The system gives the Customer the option to display the next page of books.** 6. Else **6.1. The system redisplays the “find book” search page.** 6.2. The system tells the Customer that no matching products could be found.
Postconditions:
Alternative flow: 1. At any point the Customer may move to a different page.
Postconditions:

Child use case: FindCD
ID: UC17
Parent Use Case ID: UC12
Actors: Customer
Preconditions:
Flow of events: *1. The Customer selects “find CD”.* *2. The system asks the Customer for CD search criteria consisting of artist, title, or genre.* 3. The Customer enters the requested criteria. *4. The system searches for CDs that match the Customer’s criteria.* *5. If the system finds some matcing CDs then* *5.1. The system displays a page showing details of a maximum of ten CDs.* **5.2. For each CD on the page the system displays the title, artist, price, and genre.** **5.3. While there are more CDs** **5.3.1. The system gives the Customer the option to display the next page of CDs.** 6. Else **6.1. The system redisplays the “find CD” search page.** 6.2. The system tells the Customer that no matching products could be found.
Postconditions:
Alternative flow: 1. At any point the Customer may move to a different page.
Postconditions:

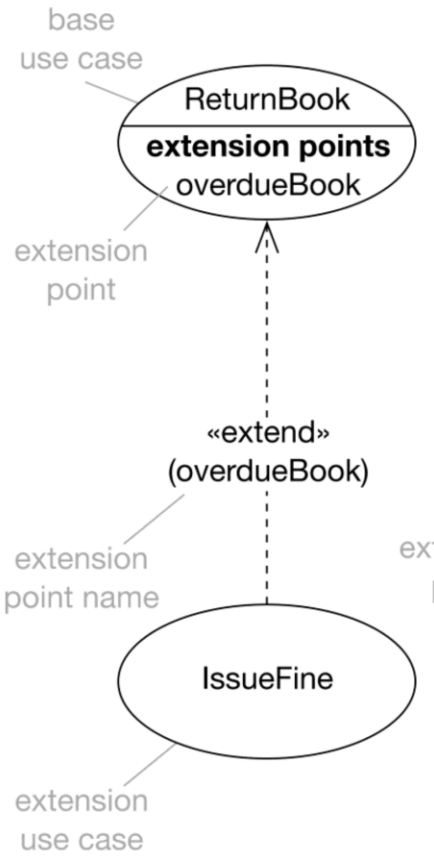
✓ *The «include» relationship* between use cases includes the behavior of a supplier use case into the flow of a client use case. The client use case is not complete without all of its supplier use cases. The supplier use cases may or may not be complete (behavior fragment, it is not instantiable, it cannot be triggered directly by actors)



✓ *The «extend» relationship* between use cases adds new behavior to a base use case. The base use case is complete without its extensions (that usually are not complete).

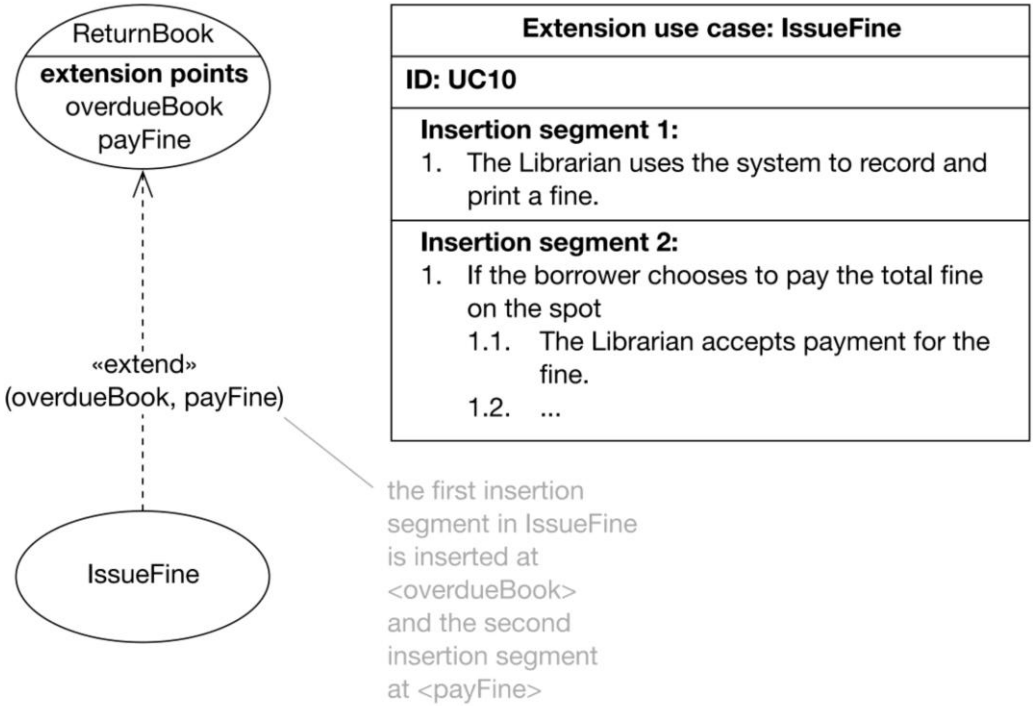


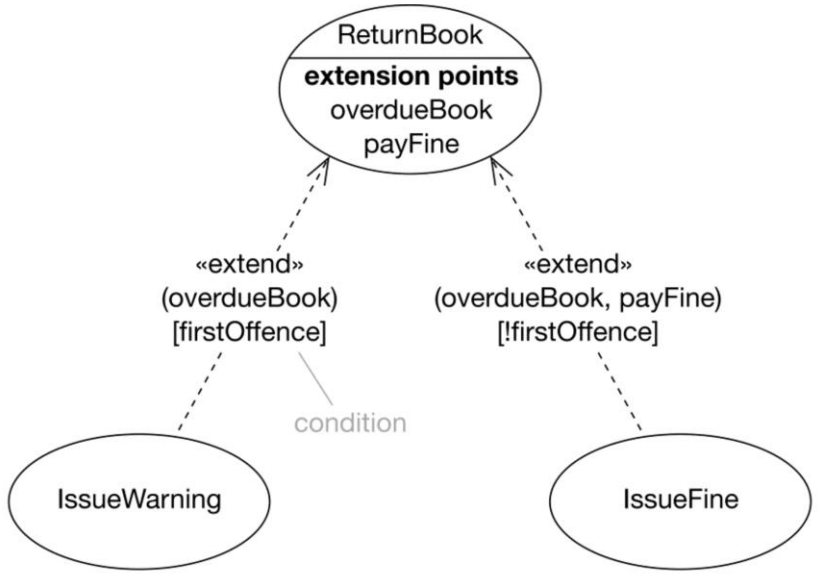
- ✓ The extension points are added to an overlay on top of the flow of events, without effects on the numbering of the flow of events of the base use case.



ReturnBook
ID: UC9
Actors: Librarian
Preconditions: 1. A valid Librarian is logged on to the system.
Flow of events: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details including the list of borrowed books. 3. The Librarian finds the book to be returned in the list of books. <overdueBook> (extension point) 4. The Librarian returns the book. 5. ...
Postconditions: The book has been returned.

- ✓ Multiple *insertion segments* can be added.
- ✓ *Conditional extensions* are also possible. A condition is a Boolean expression.


ReturnBook
extension points
overdueBook
payFine
«extend»
(overdueBook, payFine)
IssueFine
Extension use case: IssueFine
ID: UC10
Insertion segment 1:
1. The Librarian uses the system to record and print a fine.
Insertion segment 2:
1. If the borrower chooses to pay the total fine on the spot
1.1. The Librarian accepts payment for the fine.
1.2. ...
the first insertion segment in IssueFine is inserted at <overdueBook> and the second insertion segment at <payFine>


ReturnBook
extension points
overdueBook
payFine
«extend»
(overdueBook)
[firstOffence]
«extend»
(overdueBook, payFine)
[!firstOffence]
condition
IssueWarning
IssueFine

Analysis workflow

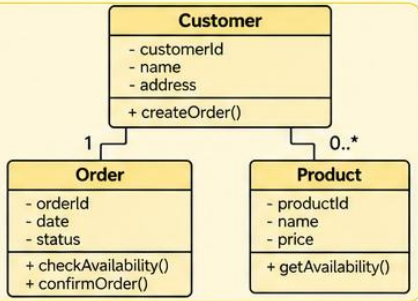
ANALYSIS WORKFLOW

- **What the system needs to do**, leaving details on how it will do it to the design workflow.
- **Key artefacts produced: analysis classes** (model key concepts in the business domain) and **use case realizations** (illustrate how instances of the analysis classes can interact).
- Only classes part of the vocabulary of the **problem domain** (no design classes such as communications or database access classes, unless the problem is about that).
- Distinguish between the **problem domain** (business requirements) and the **solution domain** (design considerations).
- Is the model useful to **all the stakeholders** (subjects with a business interest)?

Key artefacts

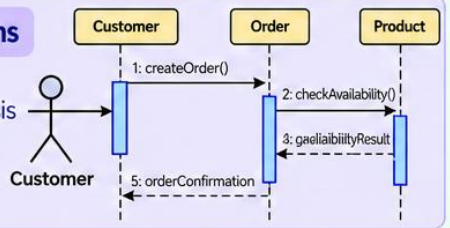
Analysis classes

Model key concepts in the business domain.



Use case realizations

Illustrate how instances of the analysis classes can interact.



Problem domain vs. Solution domain

Problem domain
(business requirements)

Concepts from the real world and the business domain (vocabulary of the problem).



Clearly distinguish between the two domains

Solution domain
(design considerations)

Technical and implementation details (not part of the problem domain, unless the problem is about that).

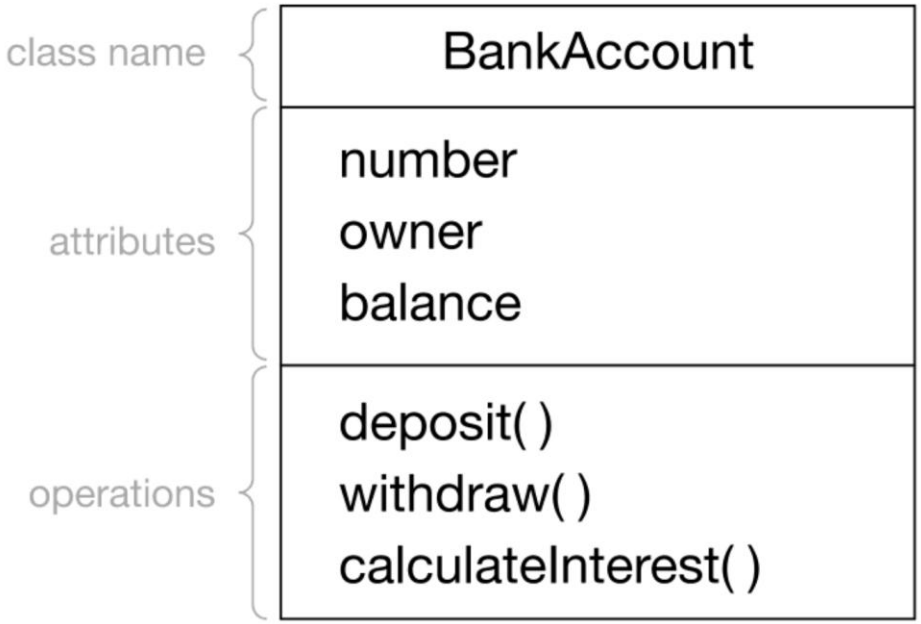


Stakeholders

The analysis model should be useful and understandable to **all the stakeholders** (subjects with a business interest), such as customers, business analysts, and domain experts.

Anatomy of an analysis class

- its name reflects its purpose;
- it is a crisp abstraction that models one specific element of the problem domain;
- it maps on to a clearly identifiable feature of the problem domain;
- it has a small, well-defined set of responsibilities;
- it has high cohesion (cohesive set of responsibilities towards the same goal);
- it has low coupling to other classes (number of relationships).



CRC: Class Responsibilities Collaboration

Class

An analysis class represents a key concept in the problem domain.

Responsibilities

What the class knows or does. They are derived from the verbs and verb phrases in the requirements.

Collaboration

Other classes that the class works with to fulfill its responsibilities.

Example CRC card

Class name: BankAccount	
Responsibilities:	**Collaborators:**
• Maintain balance • Deposit money • Withdraw money • Provide account details	• Bank • Customer • Transaction



How to find analysis classes

1. Identify nouns and verb phrases in the requirements

A customer can open a bank account. The customer deposits money into the account and withdraws money from the account. The system records each transaction and updates the account balance. The customer can view the account details.

Nouns / noun phrases (candidate classes or attributes)

Verbs / verb phrases (candidate responsibilities)

2. Derive candidate classes and responsibilities

Candidate classes (from nouns)	Candidate responsibilities (from verbs)
Customer	Open account
BankAccount (Account)	Deposit money
Money	Withdraw money
Transaction	Record transaction
Account Balance	Update account balance
Account Details	View account details

Selection of candidate classes, actors, use cases

Classes Actors Use cases

…

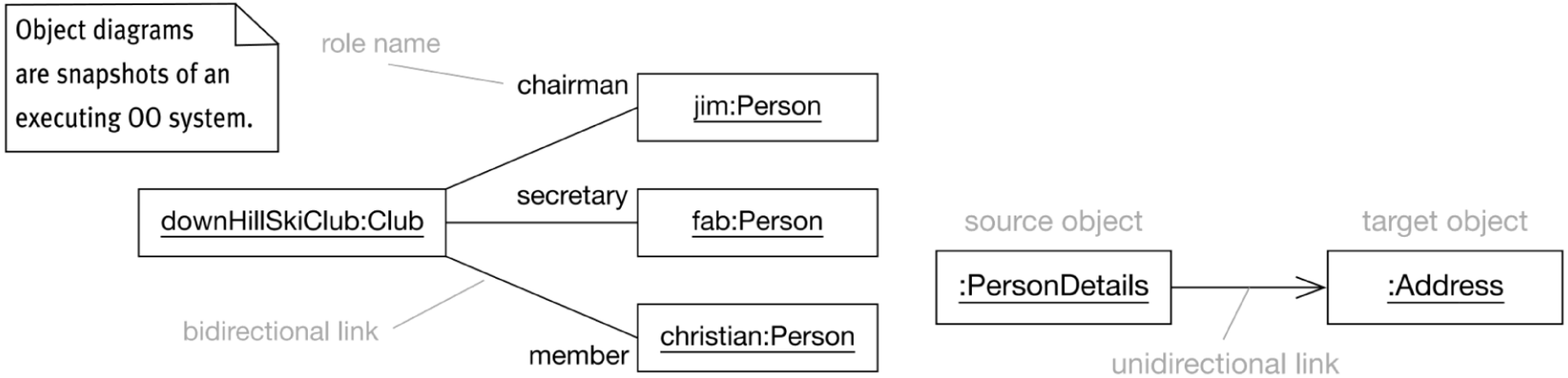
The System must retrieve the student academy record from the Student Record Management System.

…

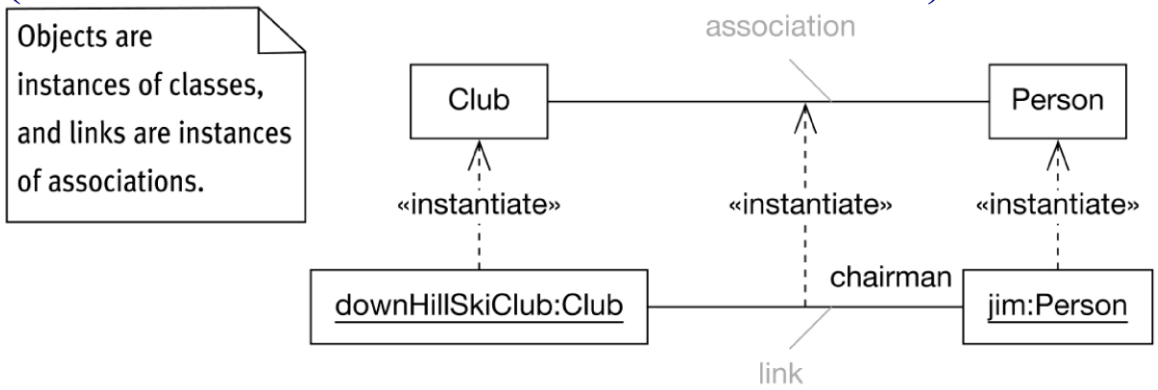
The system must enable the academic unit to generate a graduation report listing all prospective graduates for the examination board.

…

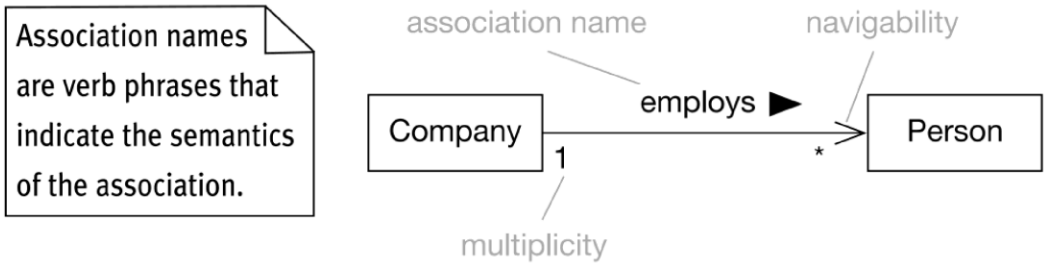
- ✓ *Link* in object diagram: it allows messages to be sent from one object to the other (pointer, references, etc.)



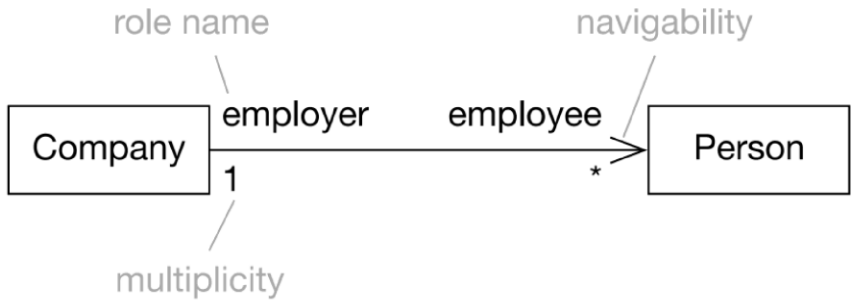
- ✓ *Association* in class diagram: relationship between classes (a link is an instantiation of an association)



- ✓ "A *Company* employs many *Persons* (a black triangle denotes the reading direction), or "Each *Person* works for one *Company*" at any point in time.
  Over time a Person object might be employed by a sequence of Company objects.

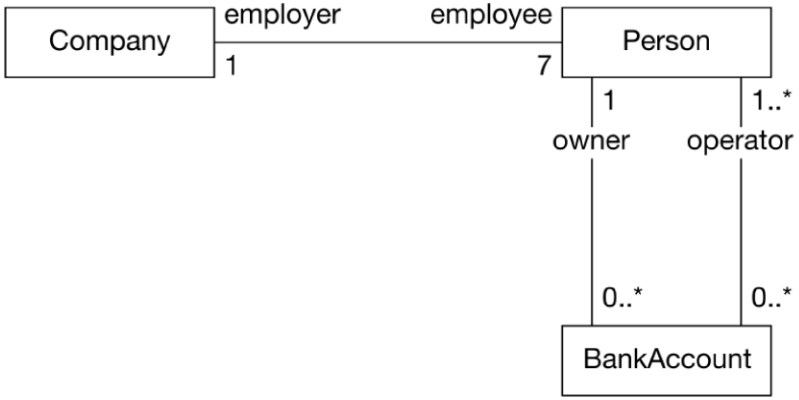


- ✓ Associations can have roles instead of association name:

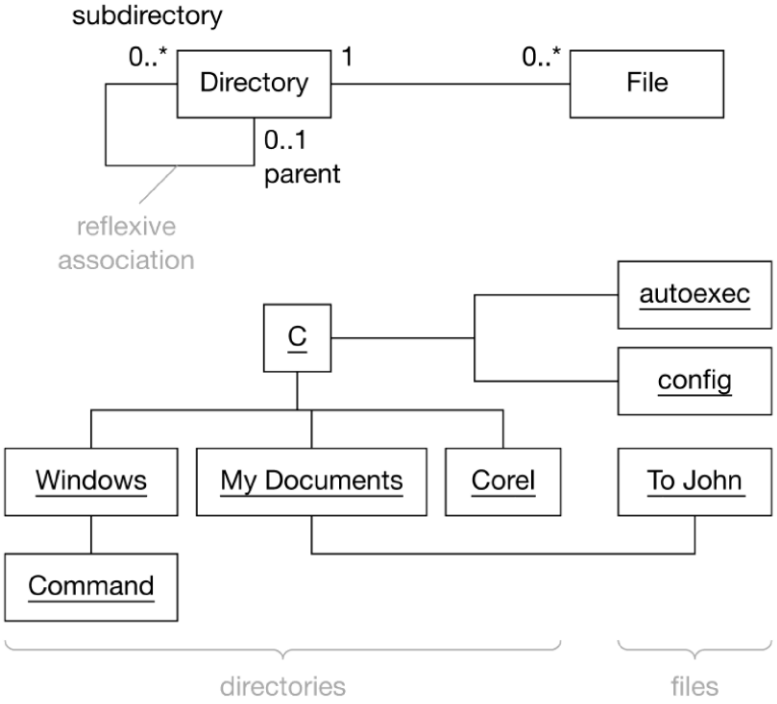


✓ Multiplicity (there is not "default" multiplicity if it is not explicitly stated):

Adornment	Semantics
0..1	Zero or 1
1	Exactly 1
0..*	Zero or more
*	Zero or more
1..*	1 or more
1..6	1 to 6
1..3,7..10,15, 19..*	1 to 3 *or* 7 to 10 *or* 15 exactly *or* 19 to many



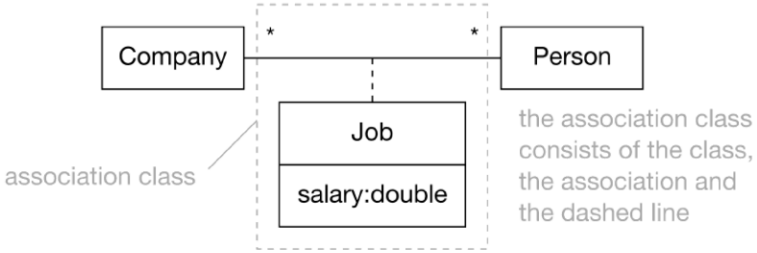
✓ Reflexive associations



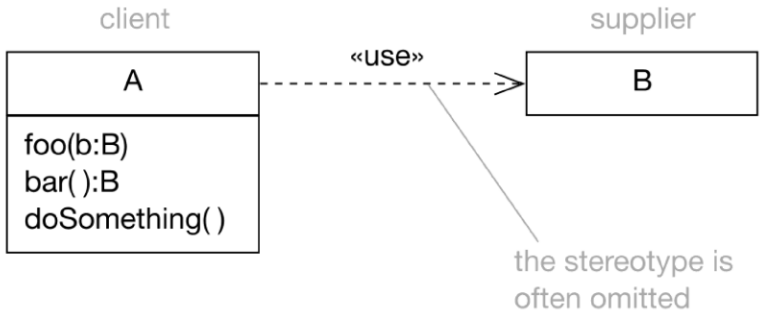



```
public class House
{   private Address address;
}
```

- ✓ Implementation of association as an attribute
- ✓ Association class (association that is also a class)

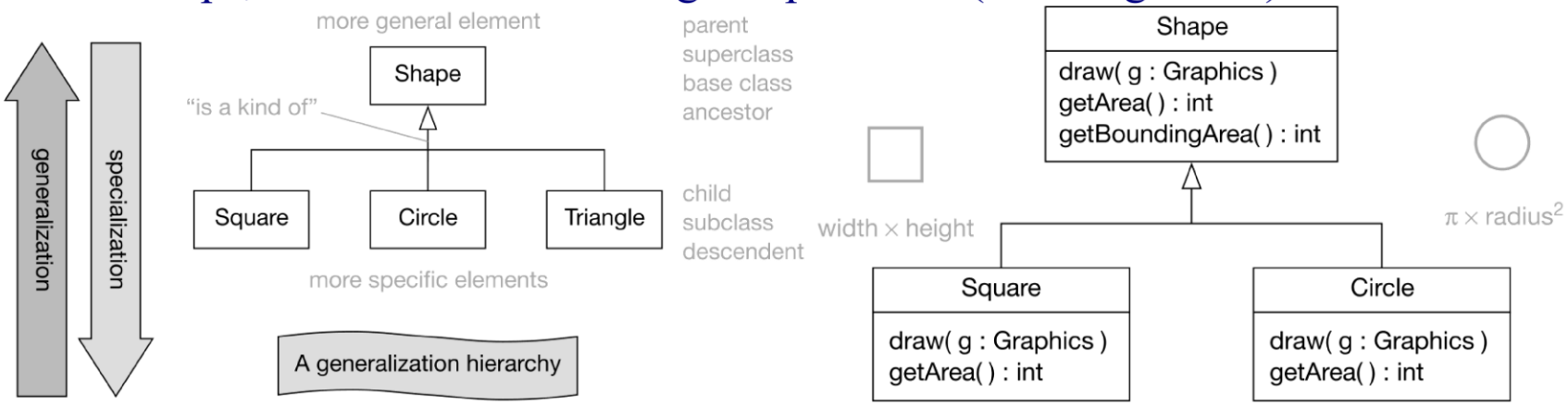


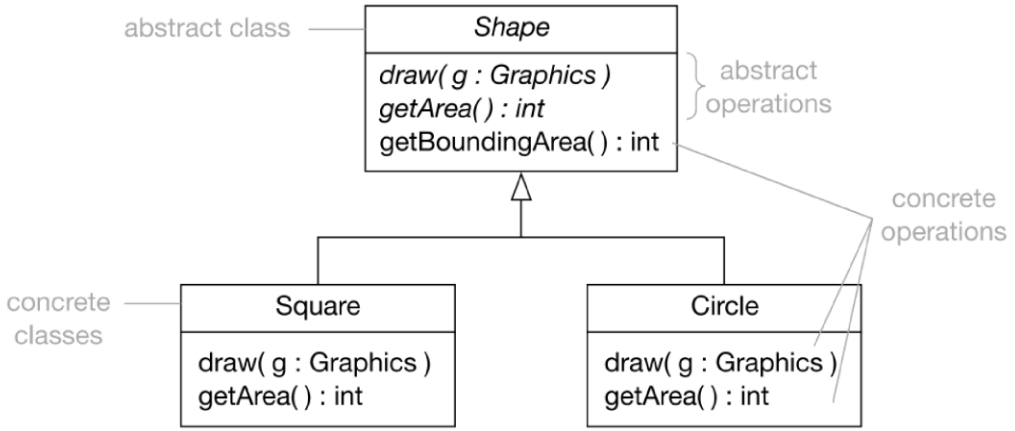
- ✓ Dependency (between classes, packages, object and classes)



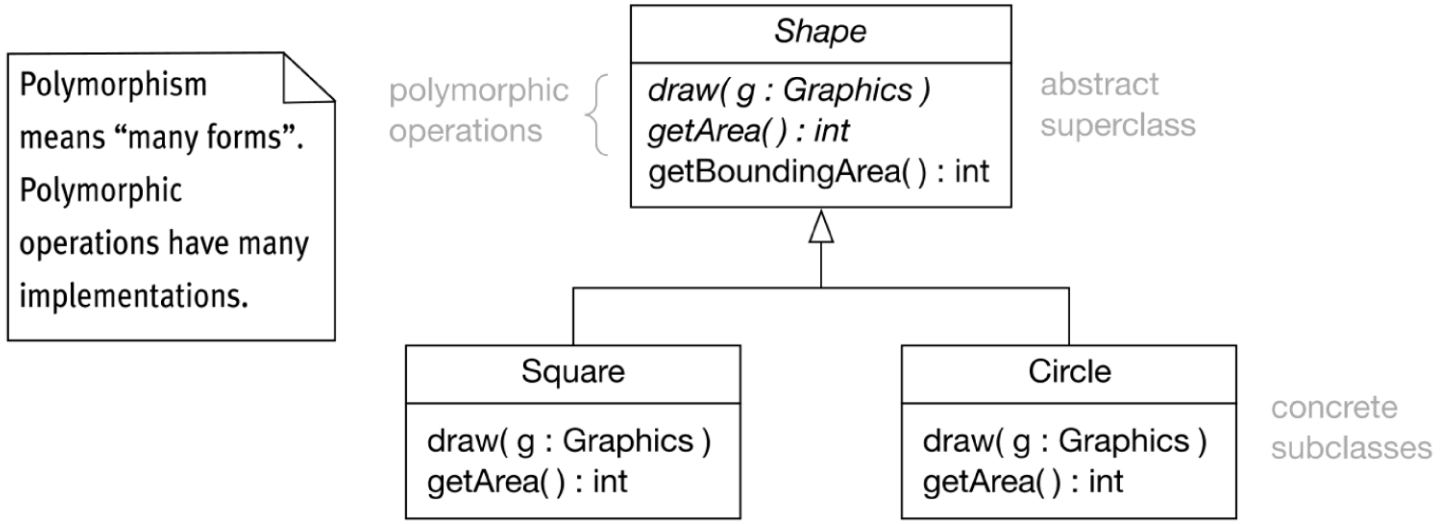
- ✓ The «use» dependency:
  An operation of class A needs a parameter, returns a value, uses an object of class B somewhere in its implementation, but not as an attribute

- ✓ The «call» dependency: an operation of class A invokes an operation of class B
- ✓ The «parameter» dependency: in class B, a parameter or returned value of class A
- ✓ The «send» dependency: a class A transfers data to a class B
- ✓ The «intantiate» dependency: an instance of class A
- ✓ The «access» dependency: a package P accesses the public content of package Q (Packages are used in UML to group things)
- ✓ The «import» dependency: the namespace of a package P is merged to the namespace of package Q (you do not need a qualified element name)
- ✓ Generalization: specialized (or extended) classes inherit attributes, operations, relationships, constraints. Overriding of operations (same signature)





✓ Abstract class cannot be instantiated.

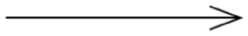
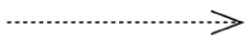


✓ Polymorphism:
there are two implementations of the Shape class, i.e., its operations have many forms (polymorphic) depending on the class of its instance (Square or Circle)

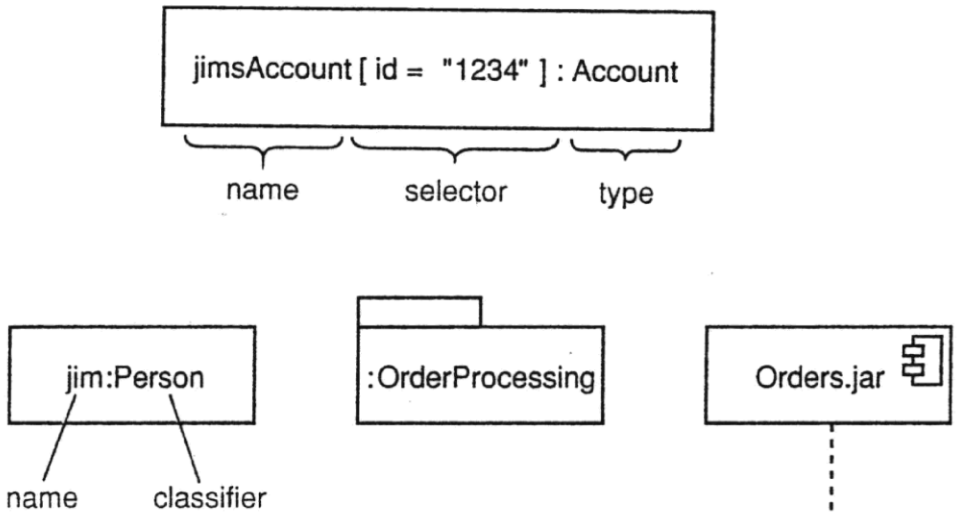
- ✓ *Dynamic view*: use case realizations show how instances of the analysis classes interact to realize the functionality of the system, via the following elements:

Element	Purpose
Analysis class diagrams	Show the analysis classes that interact to realize the use case
Interaction diagrams	Show collaborations and interactions between specific instances that realize the use case – they are "snapshots" of the running system
Special requirements	The process of use case realization may well uncover new requirements specific to the use case – these must be captured
Use case refinement	New information may be discovered during realization that means the original use case has to be updated

- ✓ Types of interaction diagrams: *communication diagram* and *sequence diagram* (dynamic interaction between instances in terms of *messages*).

Message flow	Semantics
	Procedure call – the sender waits until the receiver has finished This is the most common option
	Asynchronous communication – the sender carries on as soon as the message has been sent; it does not wait for the receiver This is often used when there is concurrency
	Return from a procedure call – the return is always implicit in a procedure call, but it may be explicitly shown using this arrow

- ✓ *Lifeline*: a participant in an interaction, an instance of a specific classifier (a classifier is a type of thing, such as actor, class, use case; an instance is a concrete example of such thing such as a specific actor, class, use case).

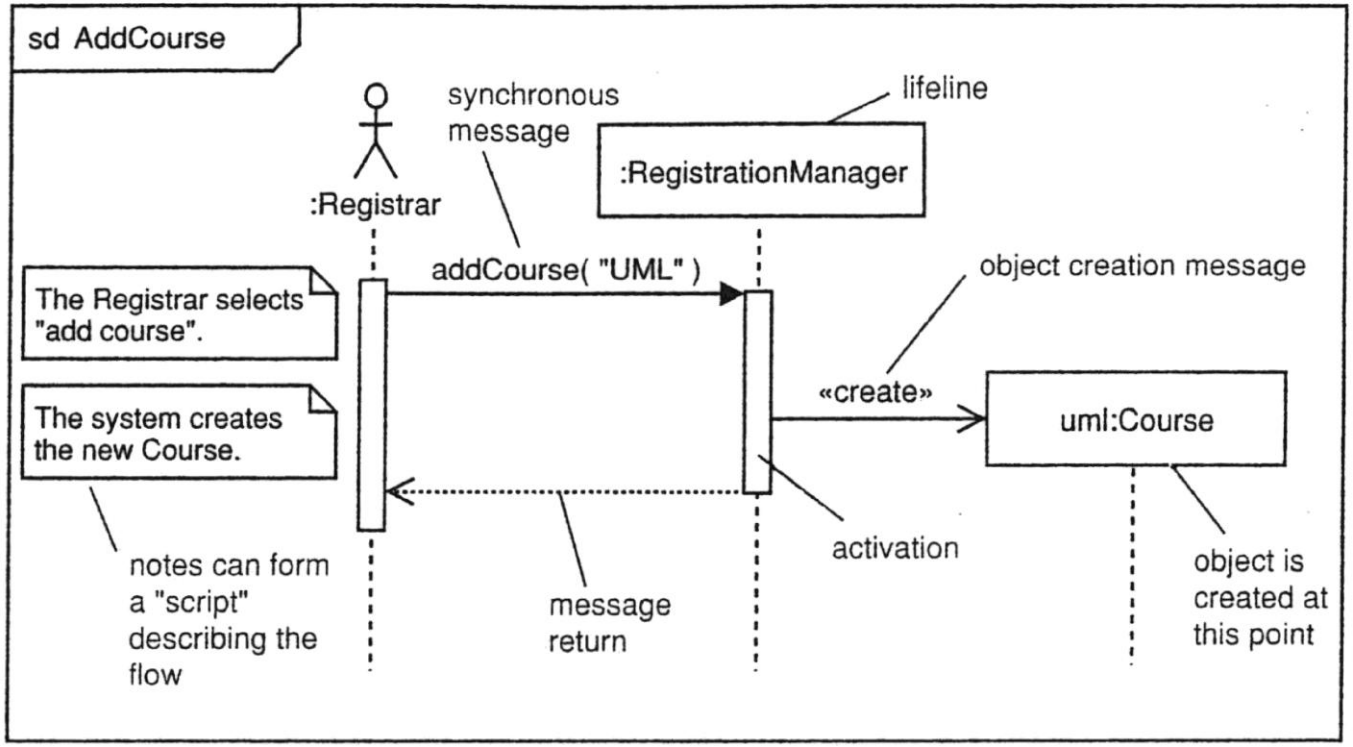


- ✓ *Selector*: a Boolean condition to select a single instance

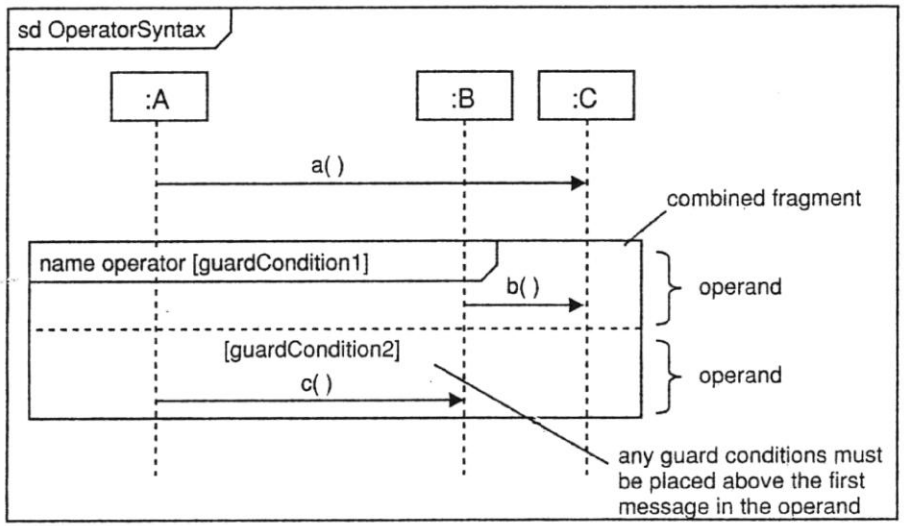
- ✓ Interaction diagrams are not verbatim transcriptions of a use case, they are illustrations of how the use case behavior is realized by analysis classes

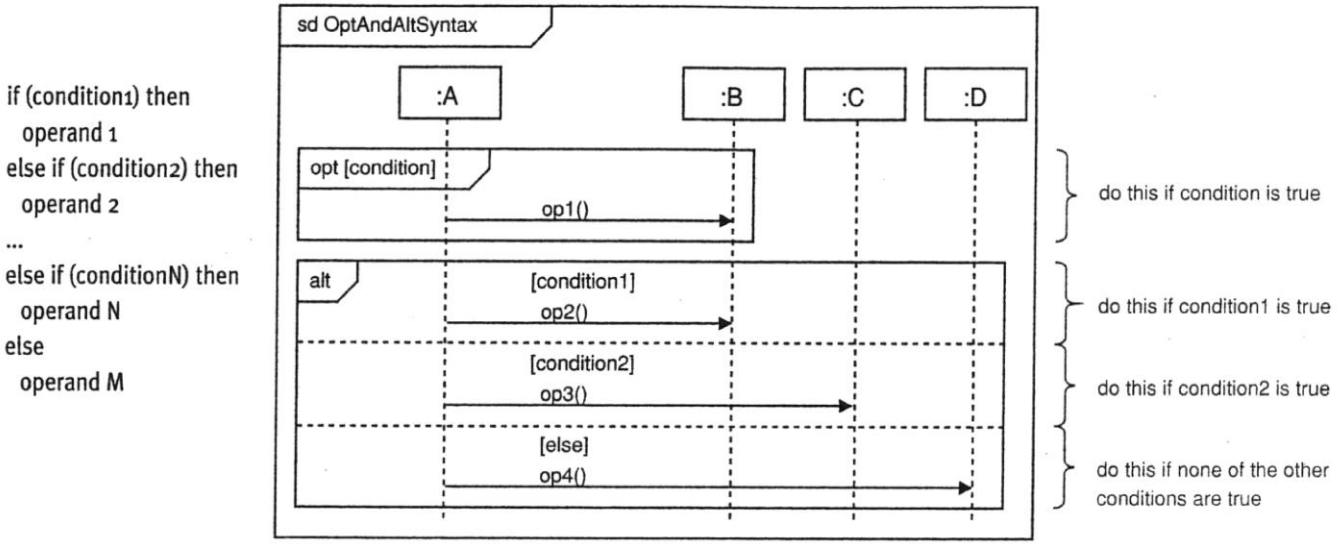
- ✓ Use case and sequence diagram

Use case: AddCourse
Actors: Registrar
Preconditions: The Registrar has logged on to the system.
Flow of events: 1. The Registrar selects “add course”. 2. The system accepts the name of the new course. 3. The system creates the new course.
Postconditions: A new course has been added to the system.

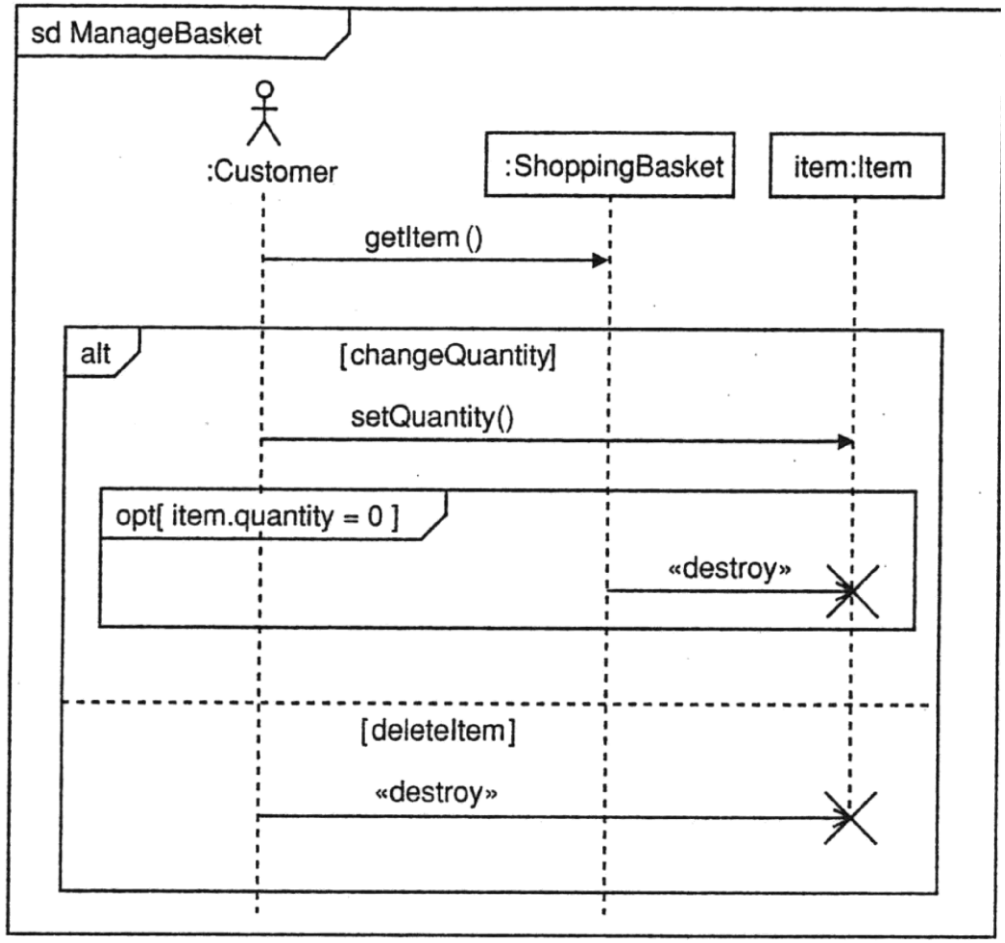


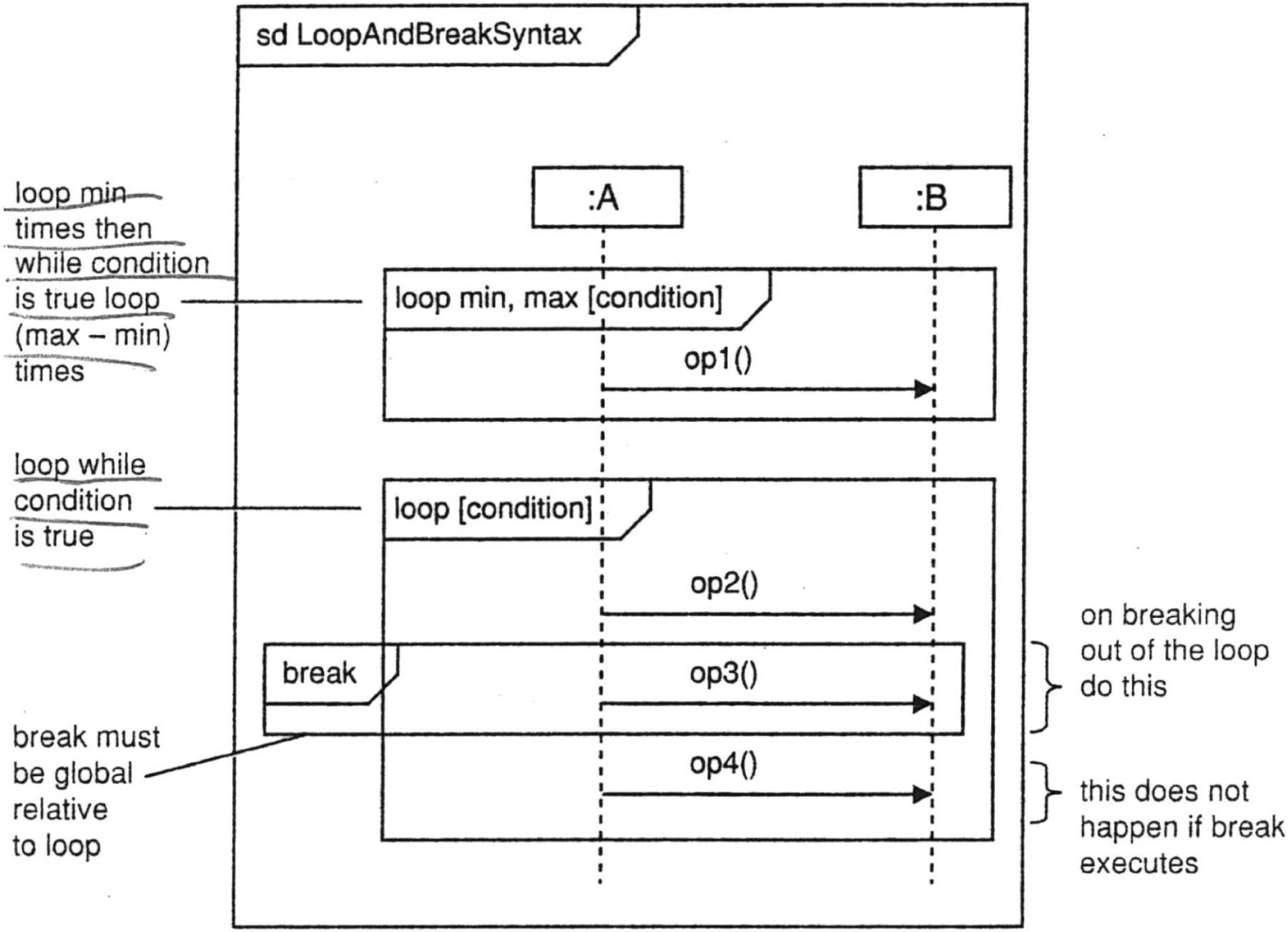
- ✓ Combined fragment and operators: combined fragments are areas of the sequence diagram; the operator determines *how* its operands are executed, whereas the guard condition determines *whether* their operand execute.

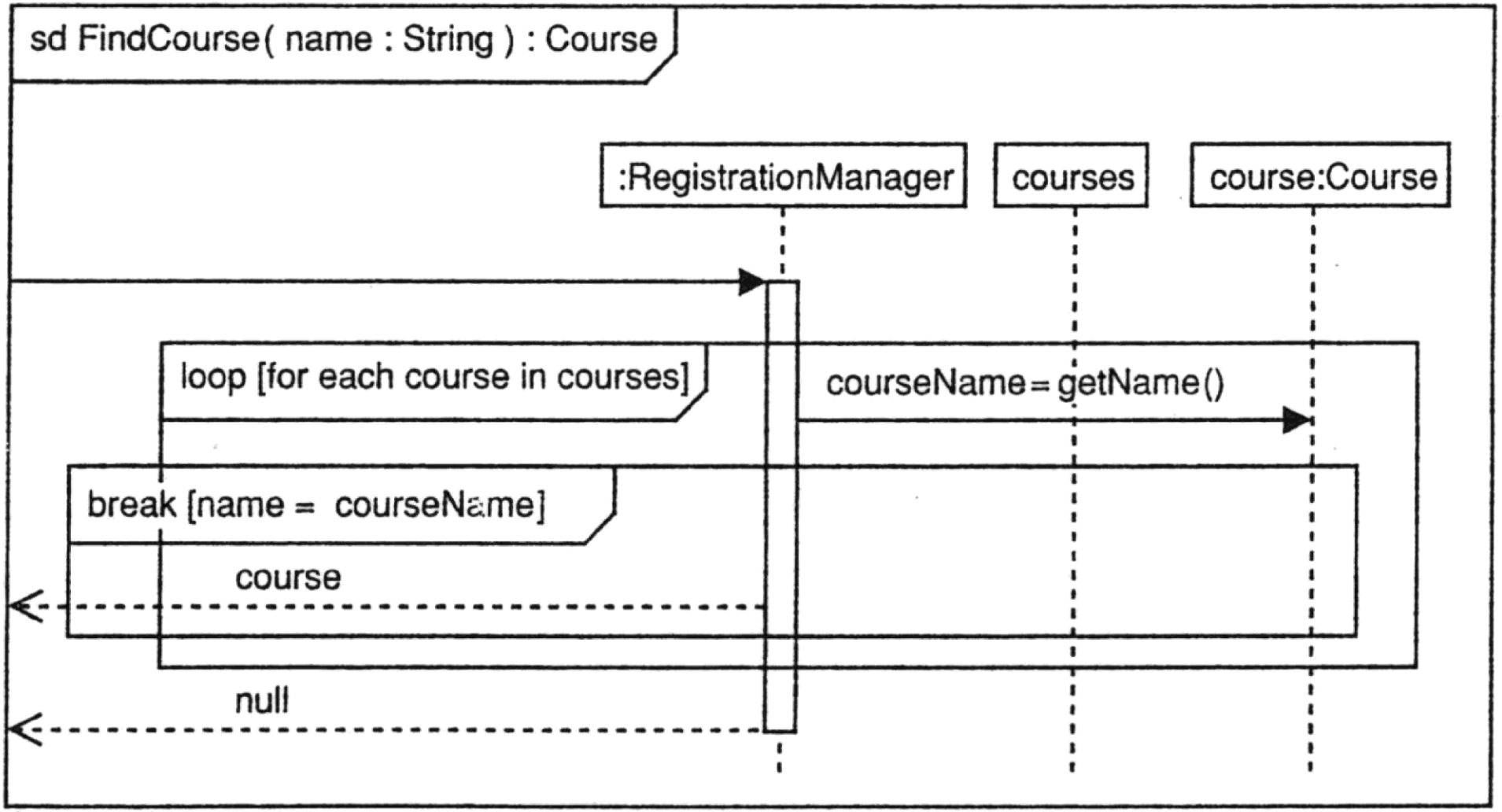




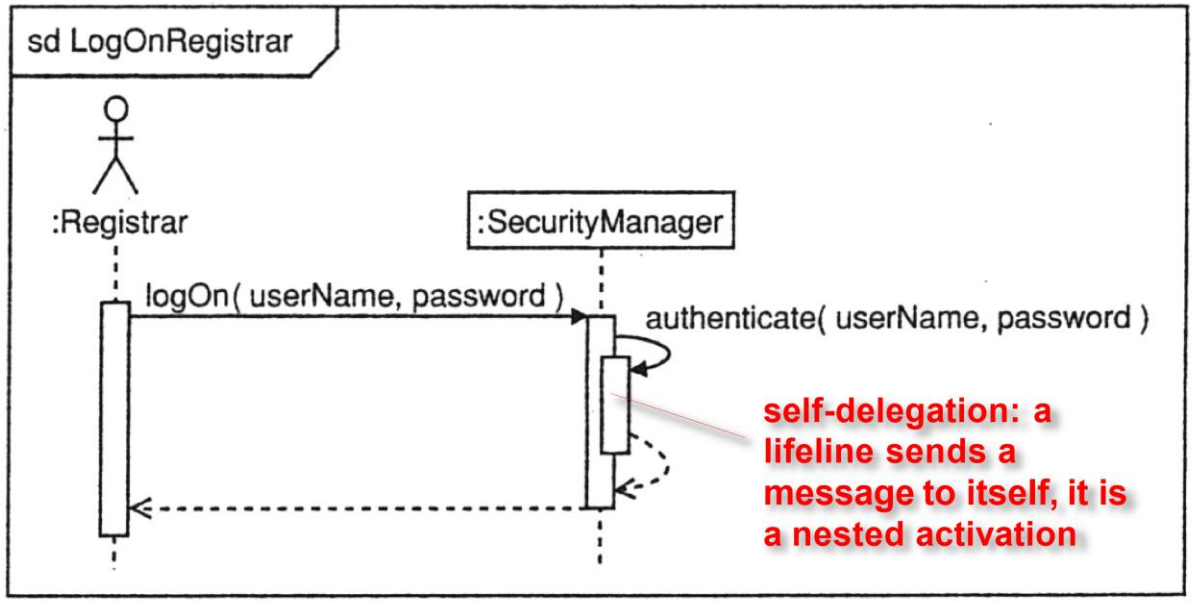
Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.

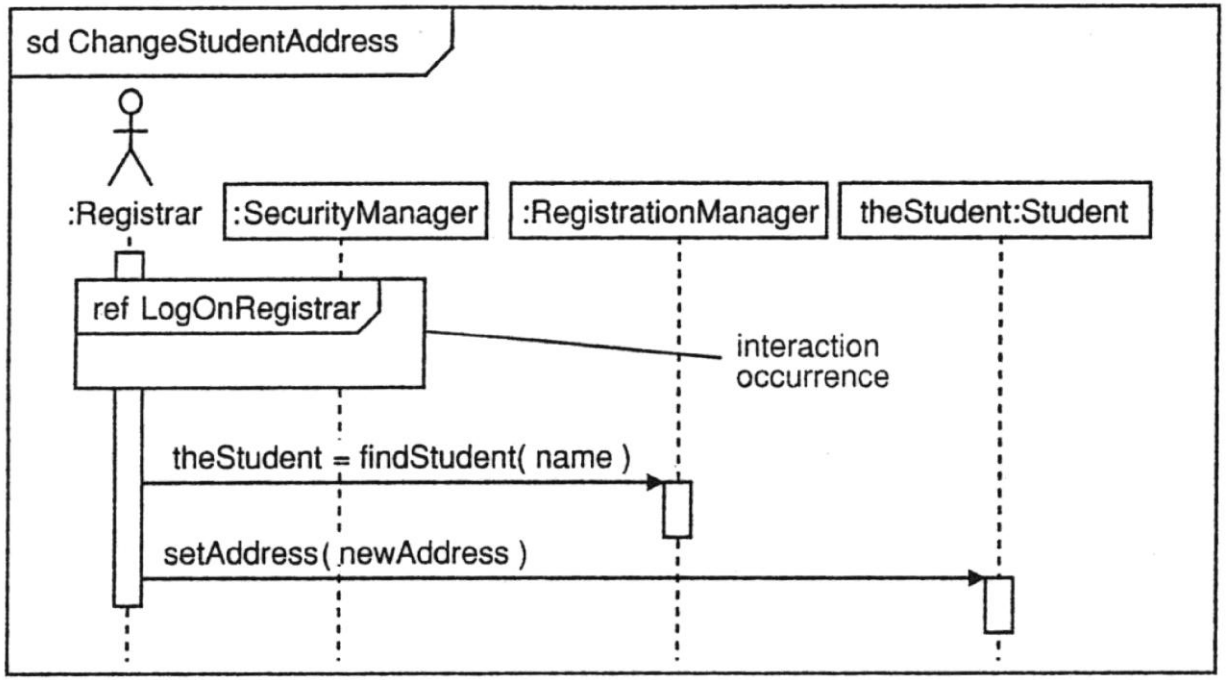



sd LoopAndBreakSyntax
:A
:B
loop min times then while condition is true loop (max – min) times
loop min, max [condition]
op1()
loop while condition is true
loop [condition]
op2()
break
op3()
break must be global relative to loop
op4()
on breaking out of the loop do this
this does not happen if break executes

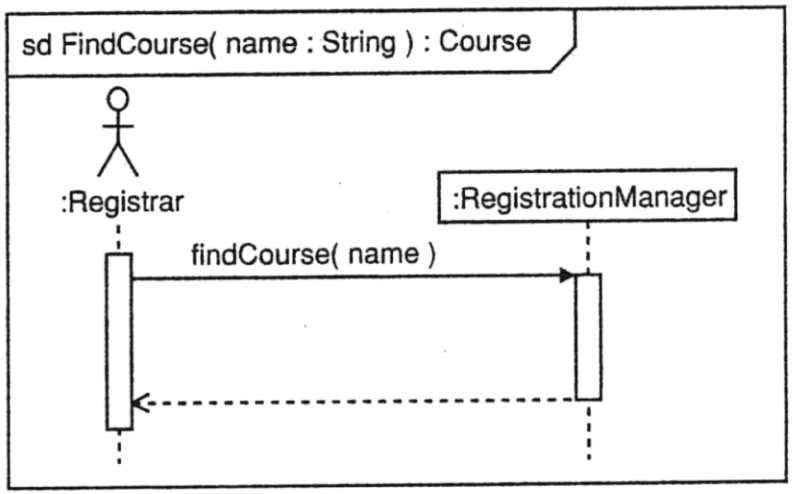

sd FindCourse(name : String) : Course
:RegistrationManager
courses
course:Course
loop [for each course in courses]
courseName = getName()
break [name = courseName]
course
null

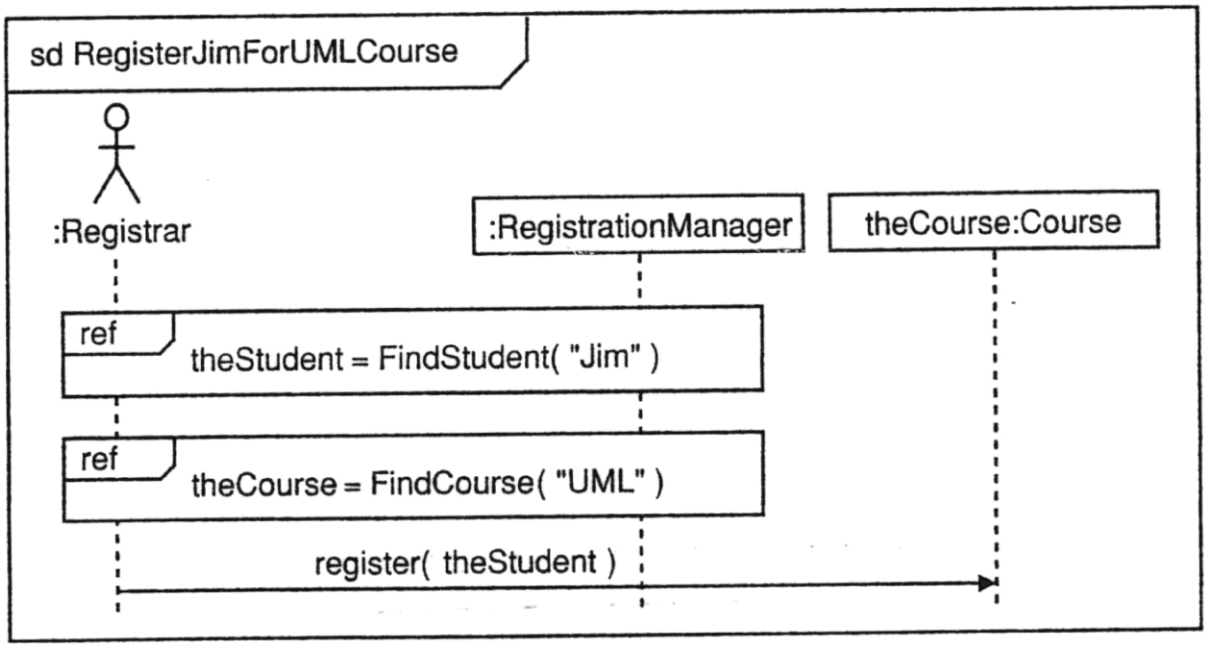
Operator	Long name	Semantics
opt	option	There is a single operand that executes if the condition is true (like if … then)
alt	alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select … case)
loop	loop	This has a special syntax: loop min, max [condition] loop min times, then while condition is true, loop (max – min) times
break	break	If the guard condition is true, the operand is executed, *not* the rest of the enclosing interaction
ref	reference	The combined fragment refers to another interaction
par	parallel	All operands execute in parallel
critical	critical	The operand executes atomically without interruption


sd LogOnRegistrar
:Registrar
:SecurityManager
logOn(userName, password)
authenticate(userName, password)
self-delegation: a lifeline sends a message to itself, it is a nested activation

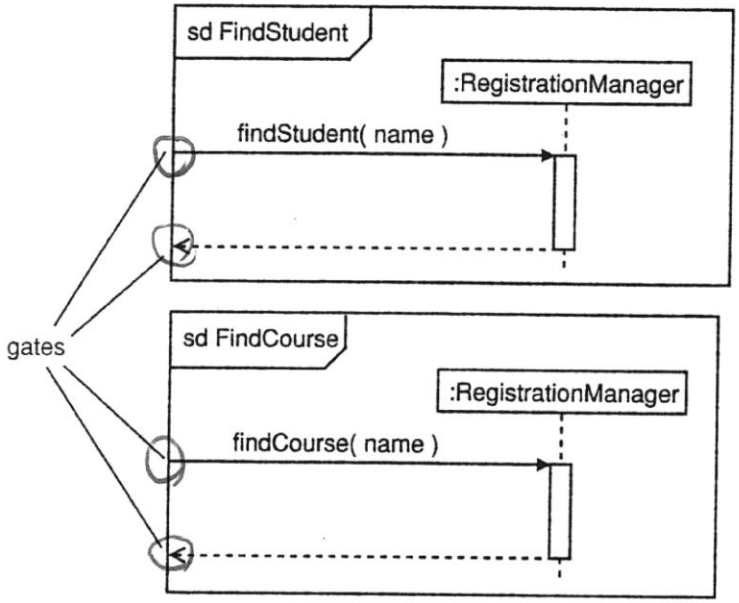

sd ChangeStudentAddress
:Registrar
:SecurityManager
:RegistrationManager
theStudent:Student
ref LogOnRegistrar
interaction occurrence
theStudent = findStudent(name)
setAddress(newAddress)

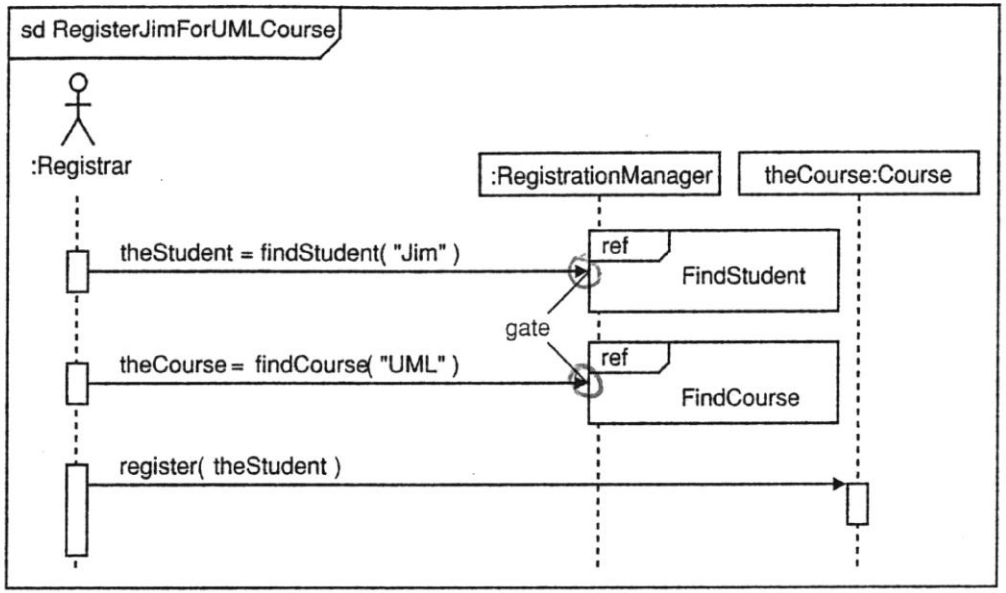
✓ Parameters in reusable interaction fragment

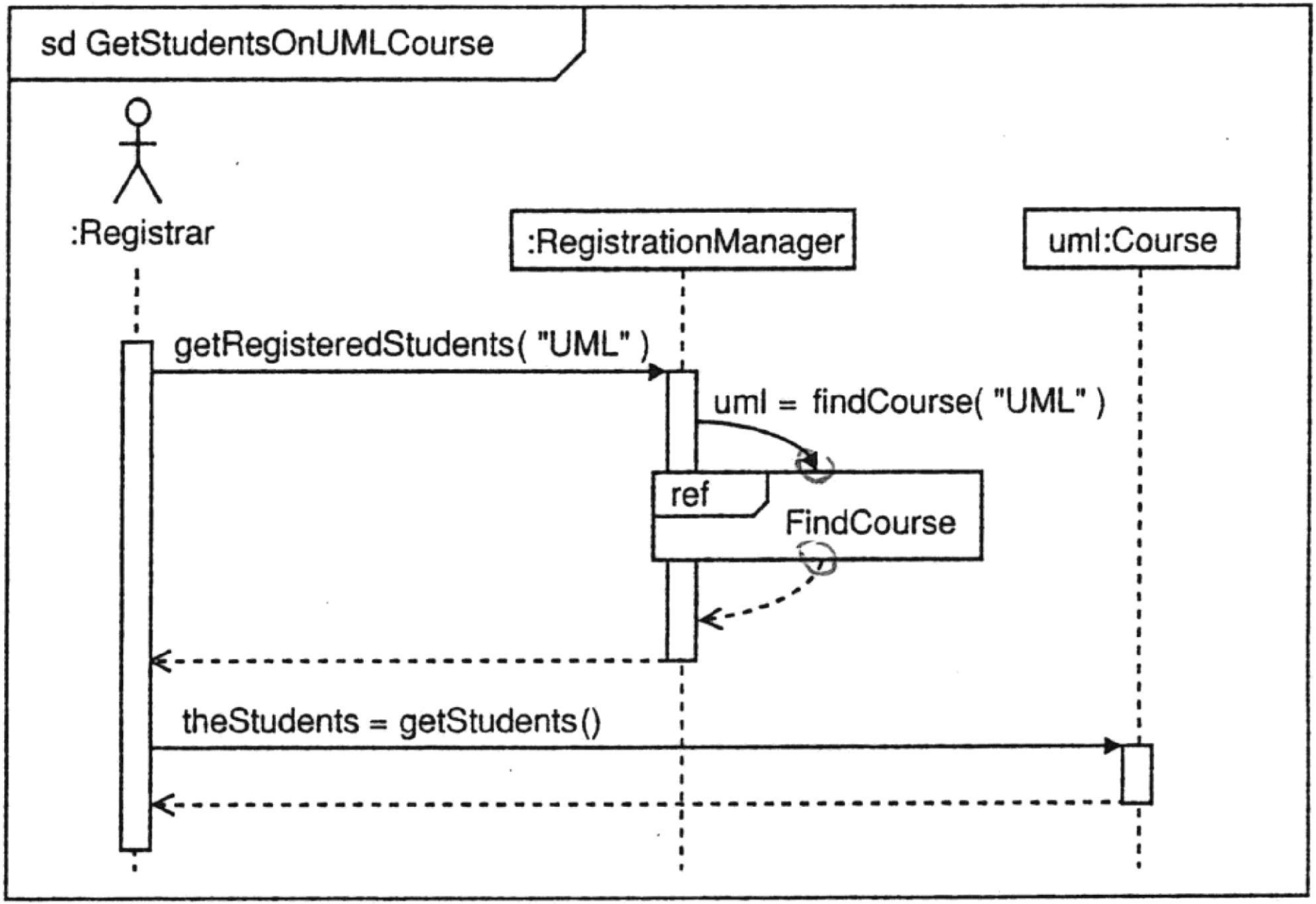




✓ Gates: inputs and outputs of interactions outside the frame



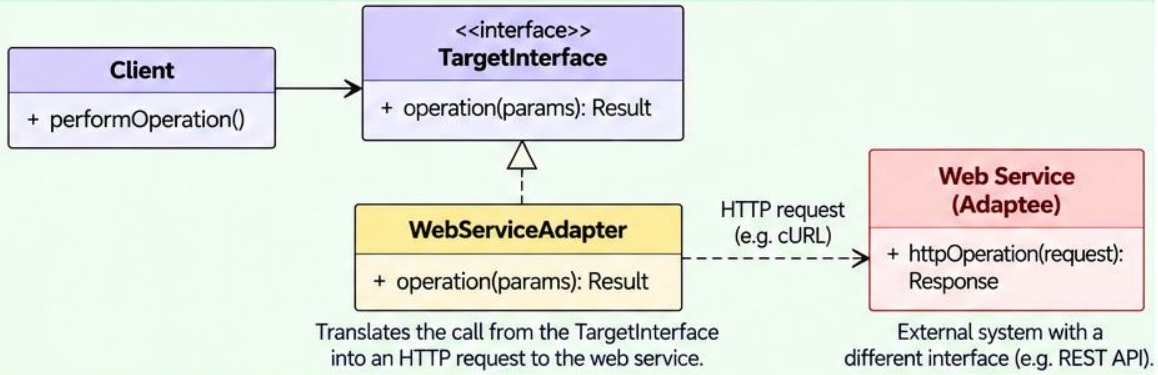



sd GetStudentsOnUMLCourse
:Registrar
:RegistrationManager
uml:Course
getRegisteredStudents("UML")
uml = findCourse("UML")
ref
FindCourse
theStudents = getStudents()

Adapter Design Pattern

The Adapter pattern allows a class to work with another class (or system) that has a **different interface**, by **translating requests** from the expected interface into the interface understood by the existing system.

Structure (UML diagram)



Real-world analogy



European plug (Client)
expects a European socket interface

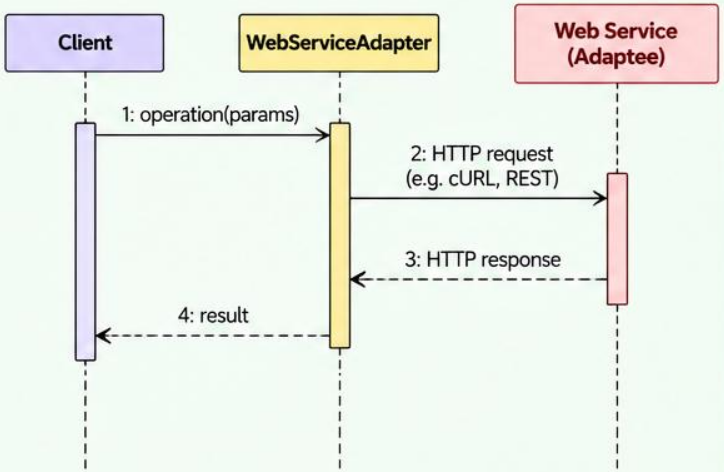
Travel adapter (Adapter)
translates the plug into a different socket interface

UK socket (Adaptee)
has a different interface

Key elements

1. **TargetInterface**
   Defines the expected interface used by the client.
2. **WebServiceAdapter**
   Implements the TargetInterface and translates calls into HTTP requests (e.g. using cURL) to the web service.
3. **Adaptee (Web Service)**
   External system with a different interface (e.g. a REST API).
4. **Client**
   Uses the TargetInterface without knowing the details of the web service.

Sequence diagram (call flow)



Benefits

- Allows incompatible interfaces to work together.
- Encapsulates the details of the external system (e.g. HTTP/REST API).
- Client code depends only on the expected interface.
- Improves maintainability and flexibility (e.g. you can switch to a different service).

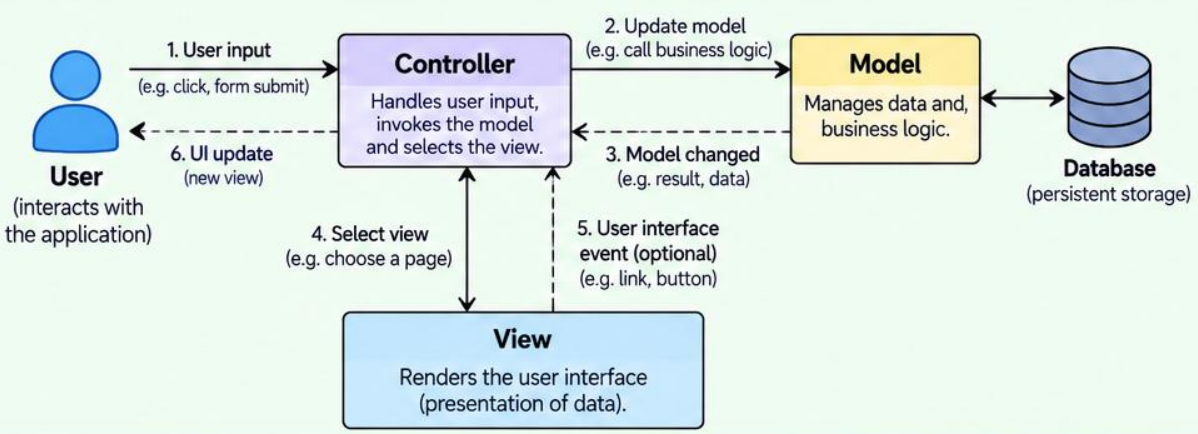
When to use

- When you need to use an existing system with a different interface (e.g. a web service, legacy system, or third-party library).
- When you want to keep your client code independent of the external system.
- When you need to translate between different communication protocols (e.g. object method calls to HTTP/REST).

Model–View–Controller (MVC) Design Pattern

The **Model–View–Controller (MVC)** pattern separates an application into three interconnected **components:** the **Model** (data and business logic), the **View** (user interface and presentation) **and the Controller** (handles user input and coordinates the flow).

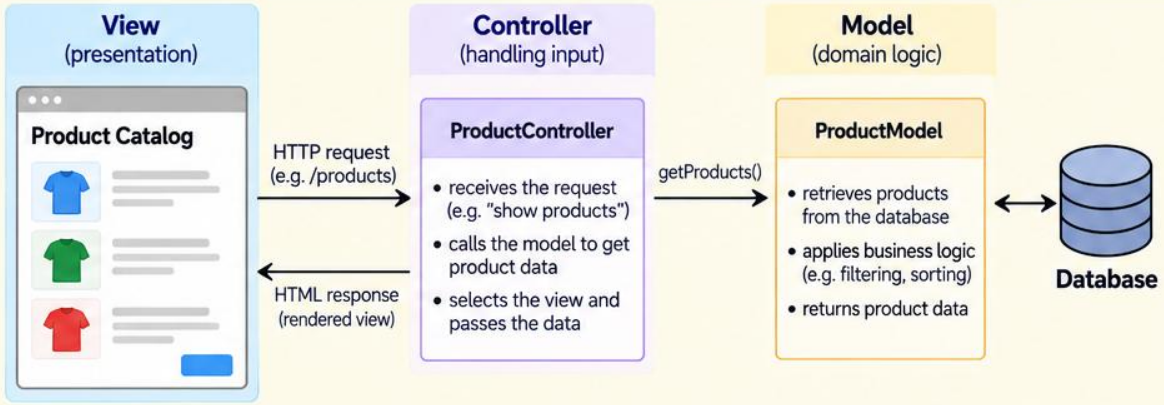
Structure



Typical flow

1. The user interacts with the application (e.g. clicks a button or submits a form).
2. The controller receives the input and calls the model to perform an operation.
3. The model processes the request (e.g. loads or updates data) and returns the result to the controller.
4. The controller selects the appropriate view (e.g. a page or template) and passes the data.
5. The view renders the user interface (e.g. HTML, GUI) and sends it back to the user.
6. The user sees the updated interface.

Example (e.g. web application: product catalog)



Benefits

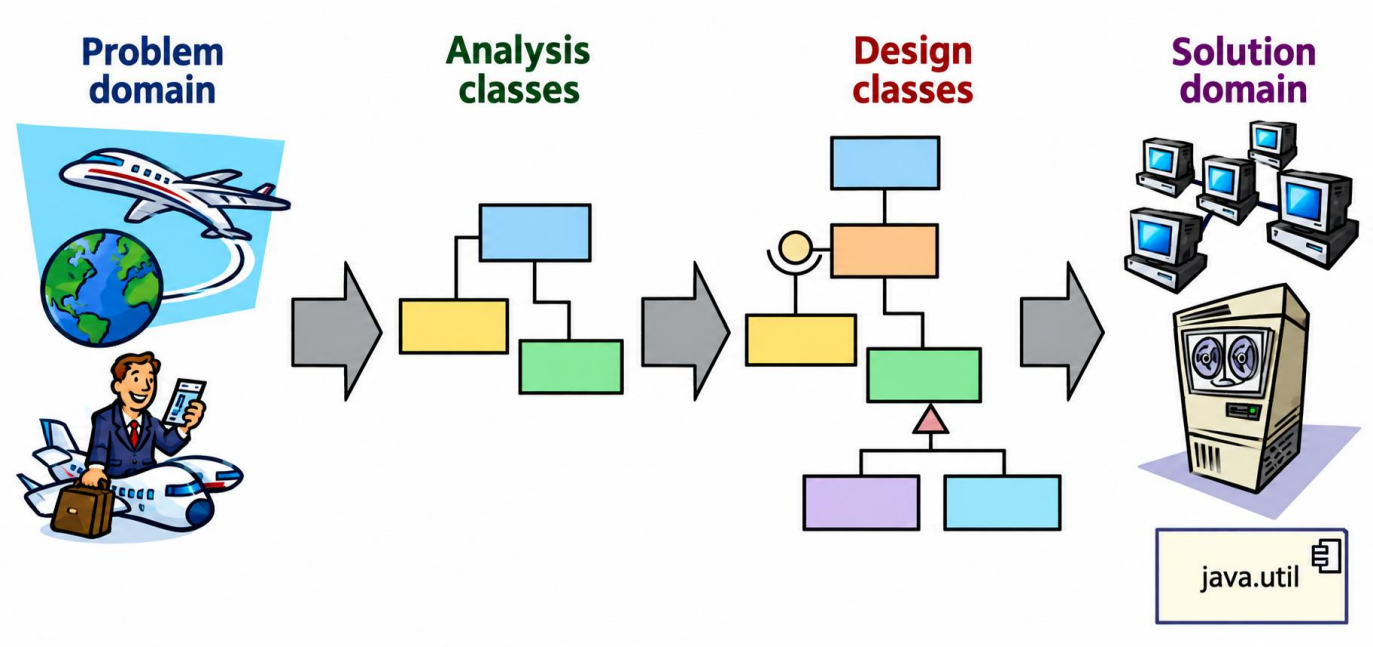
- Separates concerns (UI, logic, data).
- Improves maintainability and testability.
- Supports parallel development (UI, logic, data).
- Works well for complex applications with changing user interfaces.

When to use

- For applications with a user interface (e.g. web or desktop applications).
- When the application is moderately to highly complex and you want a clean separation between UI, business logic and data.
- When you expect the user interface to change often without modifying the business logic.

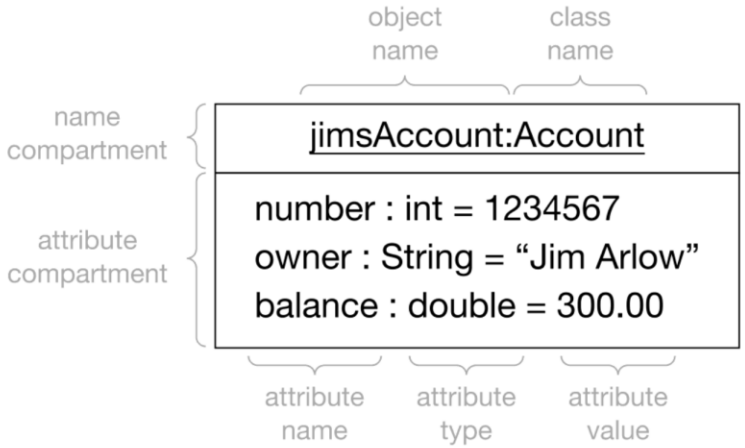
Design workflow

- ✓ While Requirements and Analysis workflows focus on the problem domain from the point of view of the system stakeholders, Design workflow focuses on the solution domain to provide: design subsystems, design classes, interfaces, use case realizations design, deployment diagrams.

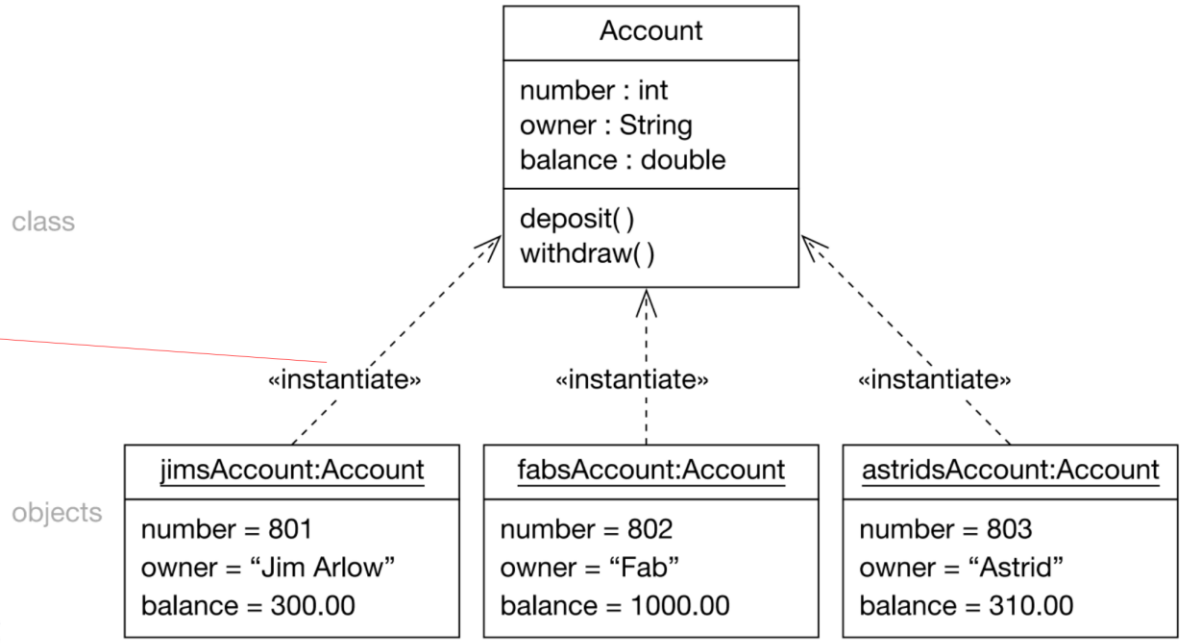


- ✓ Design classes and interfaces are first outlined and the sufficiently detailed to serve as a good basis for creating source code

- ✓ Some design classes are refinements of analysis classes. Other design classes are based on the solution domain (e.g. utility classes, communication middleware, db)

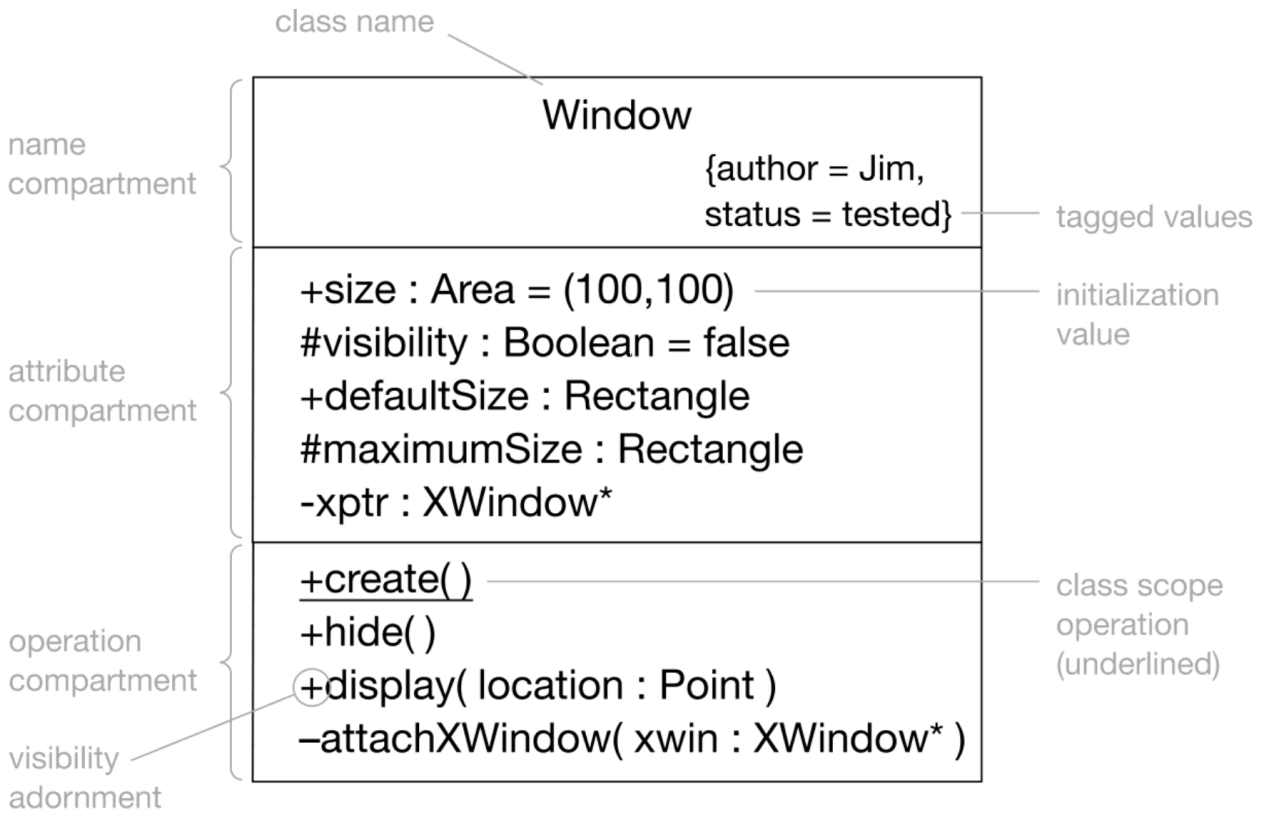


✓ UML object notation:



dependency relationship: a change to the class affects the object

✓ UML class notation:



- ✓ UML class notation:
- ✓ Class name is CamelCase (no spaces or special symbols because they are used in languages)
- ✓ Avoid abbreviations of class name
- ✓ Attribute compartment:

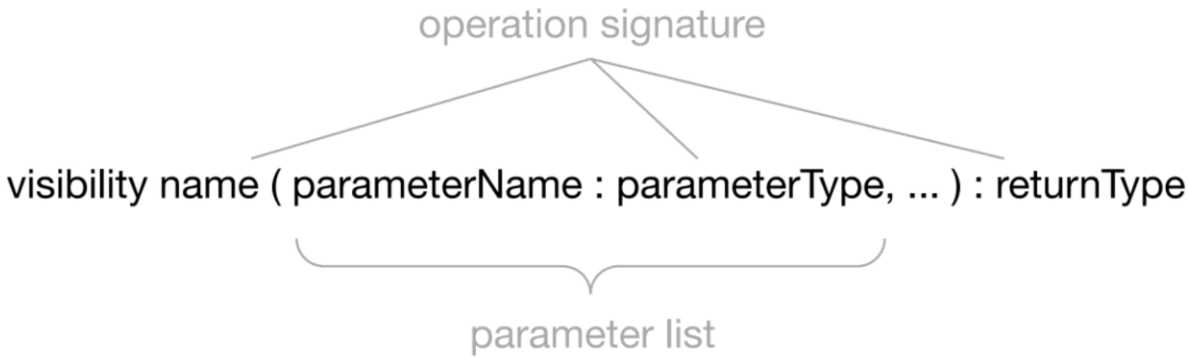


Adornment	Visibility Name	Semantics
+	Public visibility	Any element that can access the class can access any of its features with public visibility
–	Private visibility	Only operations within the class can access features with private visibility
#	Protected visibility	Only operations within the class, or within children of the class, can access features with protected visibility
~	Package visibility	Any element that is in the same package as the class, or in a nested subpackage, can access any of its features with package visibility

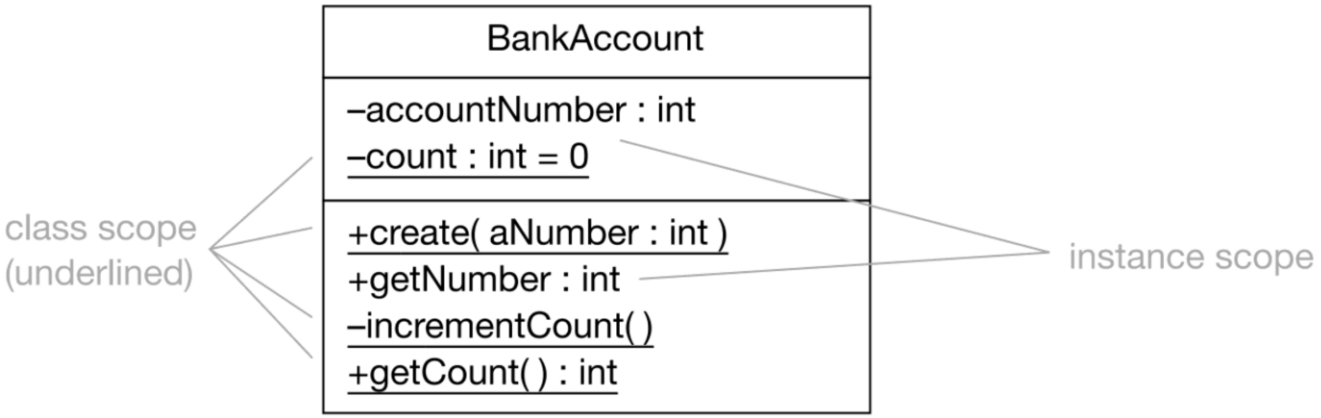
- ✓ Visibility adornment:
- ✓ Initial values and visibility are not used in the analysis model.
- ✓ Multiplicity (number of things) is more used in design, sometimes in analysis:

multiplicity expression

address [3]: String	an address is composed of an array or three Strings
name [2..*] : String	a name is composed of two or more Strings
emailAddress [0..1] : String	an emailAddress is composed of one String or null

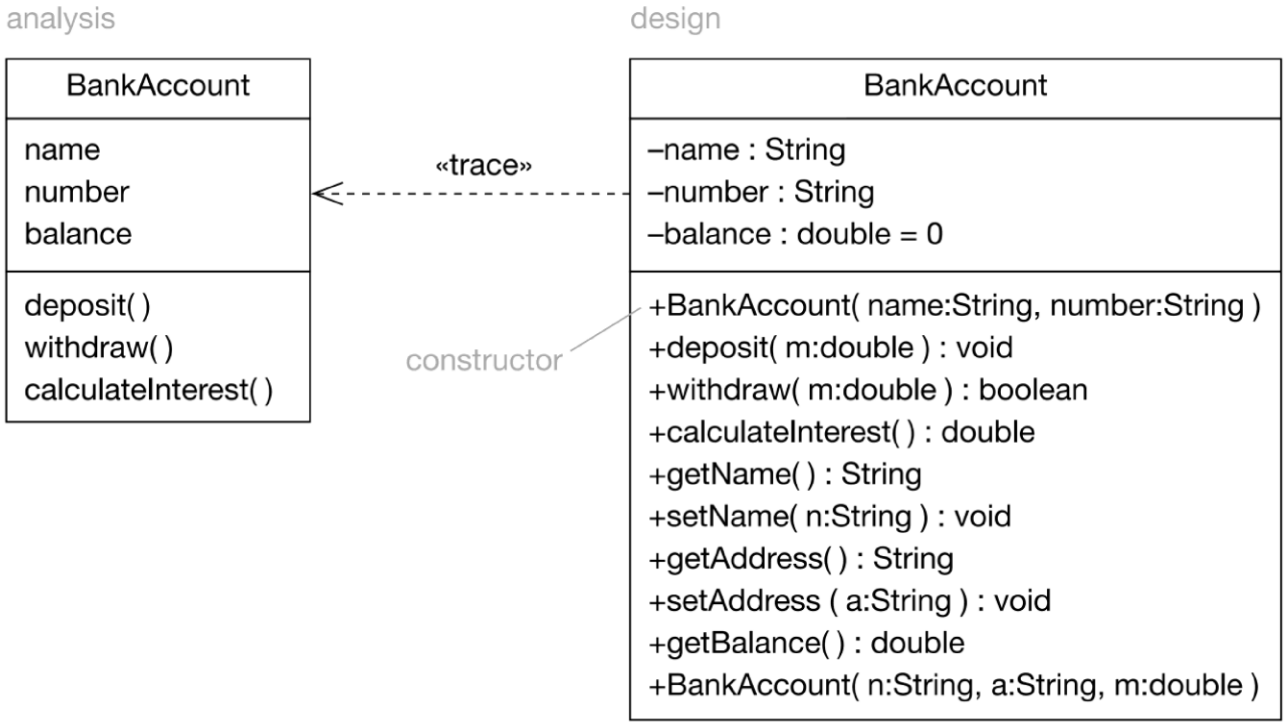


- ✓ Operation compartment:
- ✓ Instance and class scope (one version shared by all objects):



- ✓ Activity "analyze a use case": creating analysis classes and use case realizations
- ✓ *Analysis class* is in the problem domain (in which the need for the system arises)

- ✓ Complete the set of attributes and fully specify them including name, type, visibility and (optionally) a default value.
- ✓ Turn the operations specified in the analysis class into a complete set of one or more methods.



- ✓ A cohesive class has a small set of responsibilities that are closely related. Every operation, attribute, and association of the class is designed for the small, focused set of responsibilities.

- ✓ Operations offer a single primitive, atomic service. Do not offer multiple ways of doing the same thing, e.g. BankAccount class with operations for both single and multiple deposits (→maintenance and consistency problems).
- ✓ Refine analysis relationships: type, multiplicities, role names, navigability.

Aggregation

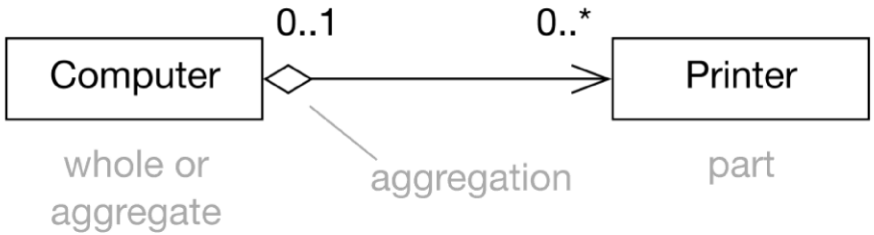


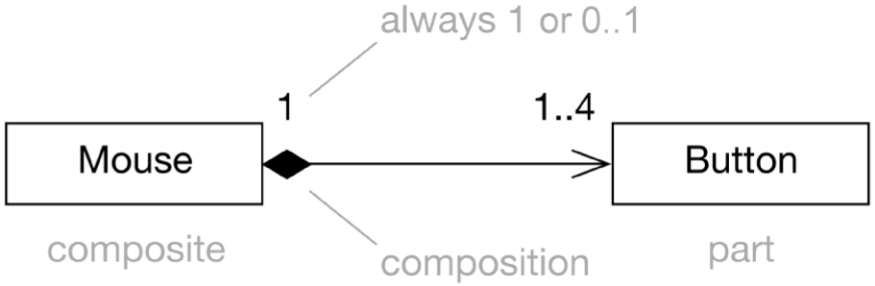
Some objects are weakly related, like a computer and its peripherals

Composition

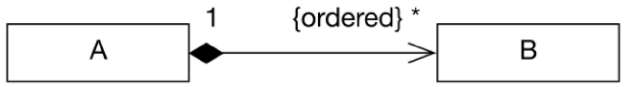


Some objects are strongly related, like a tree and its leaves





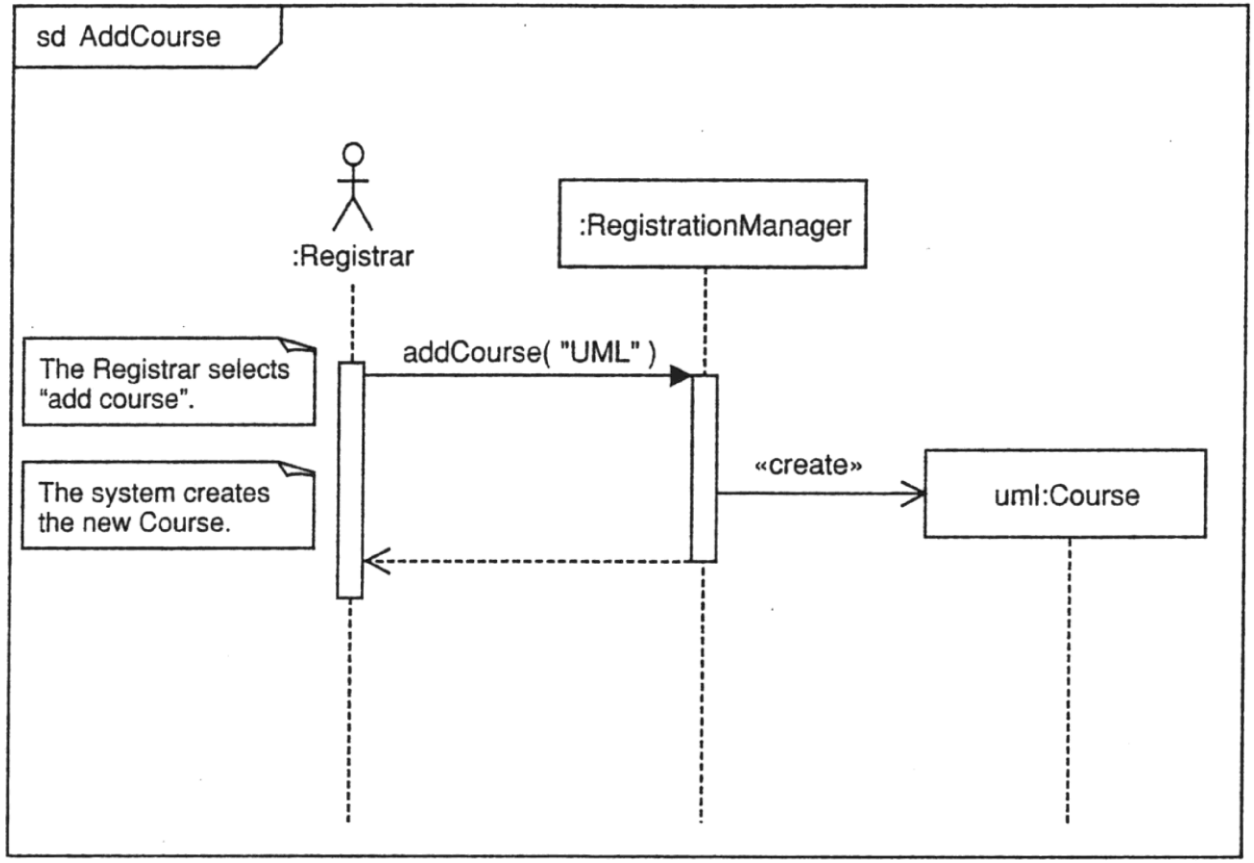
- ✓ The parts can exist (or not) independently of the aggregate, it is possible to share parts between aggregates.
- ✓ The parts can only belong to one composite at a time, no shared ownership; the composite has responsibility the creation/destruction or release of its parts.
- ✓ Multiplicity and constraints, semantics of collection (properties)



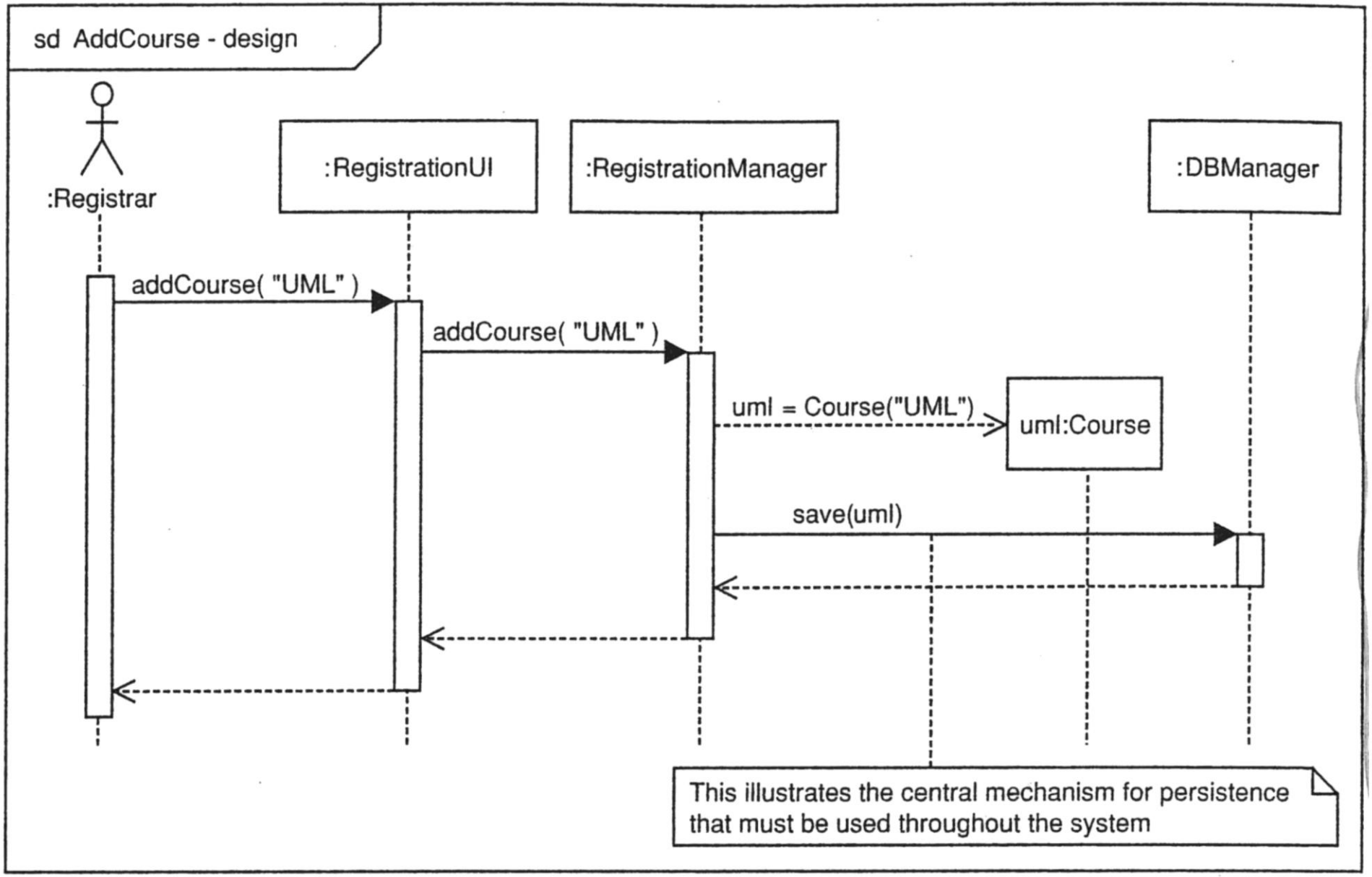
Property	Semantics
{sorted}	The collection is sorted according to some key – the key may be specified in the property, e.g. {sorted by name}
{indexed}	Each element in the set is accessible via a numeric index
{set}	Duplicates are not allowed in the collection
{lifo}	"Last in, first out" – a stack where the last element placed on the stack is the first element that can be taken off it
{queue}	A queue where the first element placed on the queue is the first element that can be taken off it

- ✓ Interfaces and components: breaking up the system into subsystems and determining their interactions via interfaces

✓ Example of an analysis sequence diagram

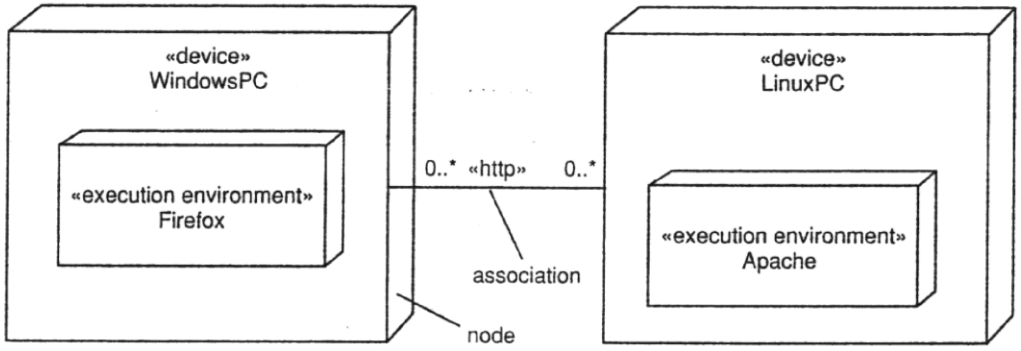


✓ In the corresponding design diagram, in the early stage of design, application layers are visible (e.g. front-end/GUI and backend/DB),


sd AddCourse - design
:Registrar
:RegistrationUI
:RegistrationManager
:DBManager
addCourse("UML")
addCourse("UML")
uml = Course("UML")
uml:Course
save(uml)
This illustrates the central mechanism for persistence
that must be used throughout the system

Implementation workflow

- ✓ Implementation: to transform a design model into executable code
- ✓ It begins in the elaboration and is the main focus of the construction phase.
- ✓ Architectural implementation: to identify architecturally significant components and to map them to hw.



- ✓ Deployment diagram

  Device: a physical type of device (PC, Server)
  Execution environment: e.g. an Apache web server

- ✓ Types of artifacts: source files, executable files, scripts, database tables, documents, outputs of previous

Testing workflow