

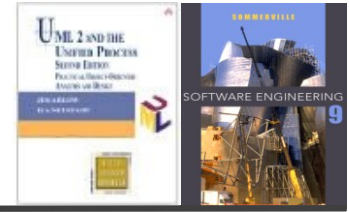
---

# Low-level Design and Implementation

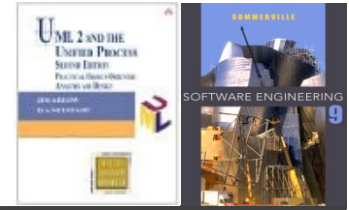
## Lecture 8

# Outline

---



- ✧ Low-level design
- ✧ Implementation issues
- ✧ UML Interaction diagrams



---

# Low-level Design

## Lecture 8/Part 1

# Low-level design



## Purpose:

- ✧ Include all **code-level details** into the model
- ✧ Decide how exactly the system shall be **implemented**
- ✧ Typically an **implicit part of implementation**

## ✧ Techniques

- Design patterns
- SOLID principles
- Guidelines for dependable/testable/.. programming

# Design patterns



- ✧ A design pattern is a way of reusing abstract knowledge about a problem and its solution in object-oriented world.
  - Pattern descriptions make use of object-oriented characteristics such as inheritance, polymorphism and interface realization.
- ✧ A pattern is a description of the problem and the essence of its solution.
  - Not a concrete design but a **template** for a design solution that can be **instantiated in different ways**.
- ✧ It should be sufficiently abstract to be reusable in different settings.

# The “Gang of Four” design patterns



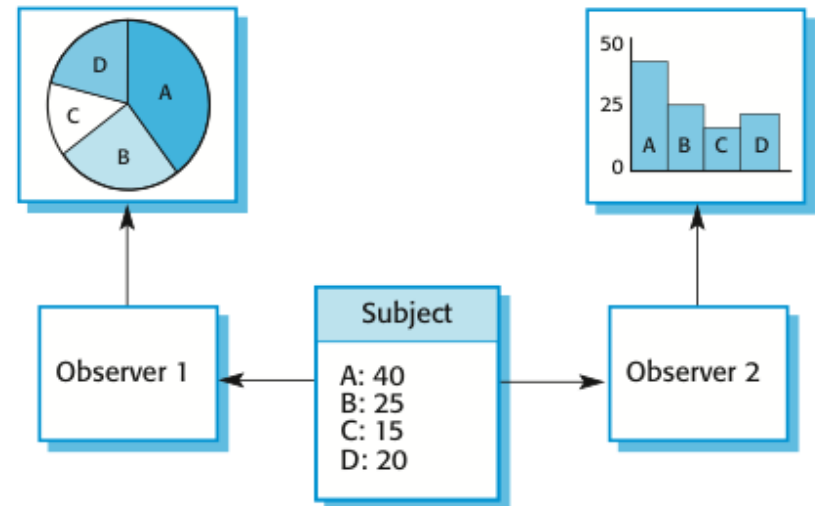
- ✧ Introduced in a book by GoF in 1995
- ✧ Collection of 23 classic software design patterns divided into three groups:

- Creational
- Structural
- Behavioral

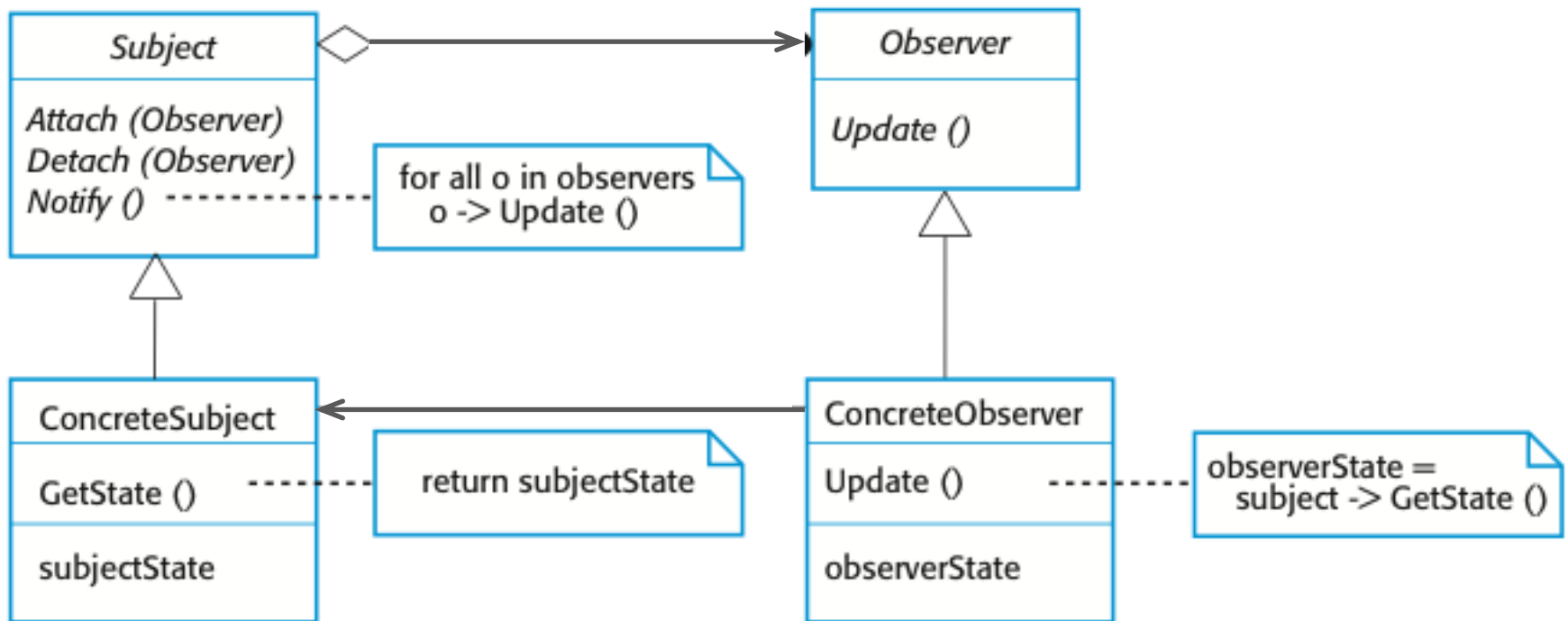


## ✧ Observer pattern

- Behavioral pattern
- Separates the display of object state from the object itself when multiple displays of state are needed.



# A UML model of the Observer pattern



# Design problems



- ✧ Be aware that any design problem you are facing **may have an associated pattern** that can be applied.
- Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Allow classes with incompatible interfaces to work together by wrapping a new interface around that of an already existing class (Adapter pattern).
  - Reduce the cost of creating and manipulating a large number of similar objects (Flyweight pattern).
  - Restrict object creation for a class to only one instance (Singleton pattern).



# SOLID principles



- ✧ The “first five principles” identified by Robert C. Martin in the early 2000s that stand for five basic principles of object-oriented programming and design.
- ✧ **S**ingle responsibility
- ✧ **O**pen/closed
- ✧ **L**iskov substitution
- ✧ **I**nterface segregation
- ✧ **D**ependency inversion

# Single responsibility principle

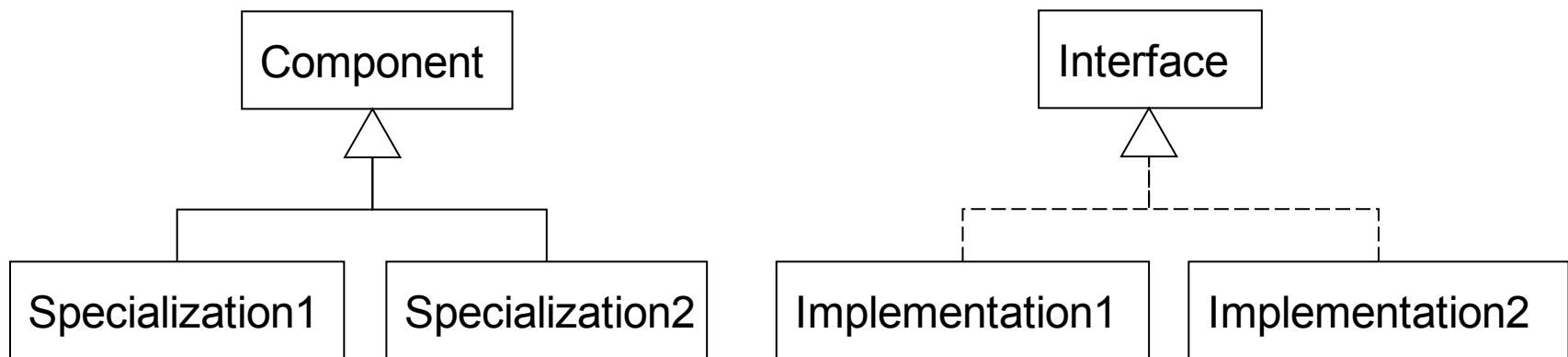


- ✧ The principle states that every class should have a **single responsibility**, and that responsibility should be **entirely encapsulated** by the class.
- ✧ A **responsibility** can be understood as a reason to change, so a class or module should have one, and only one, reason to change.
- ✧ As an example, consider a module that compiles and prints a report. Such a module can be changed for two reasons – because the **content** or the **format** changes.
  - If there is a change to the report compilation process, there is greater danger that the printing code breaks.

# Open/closed principle



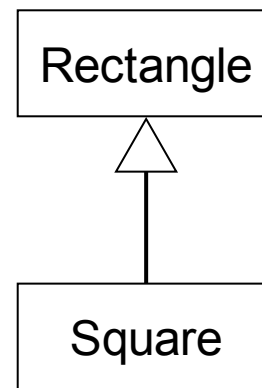
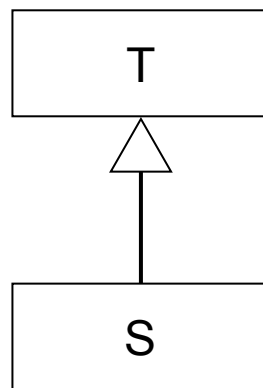
- ✧ The principle states that software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- ✧ Use **inheritance** and **interfaces** to avoid code changes when extending system functionality.



# Liskov substitution principle



- ✧ The principle states that, in a computer program, if S is a subtype of T, then **objects of type T may be replaced with objects of type S** without altering any of the desirable properties of that program (correctness, task performed, etc.).



width and height can be changed independently

width and height must not be changed independently

# Interface segregation principle



- ✧ The principle states that no client should be forced to **depend on methods it does not use**.
- ✧ ISP splits large interfaces into smaller and more specific “role” interfaces so that clients will only have to know about the methods that are of interest to them.
- ✧ ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

iATM

iCheckBalance

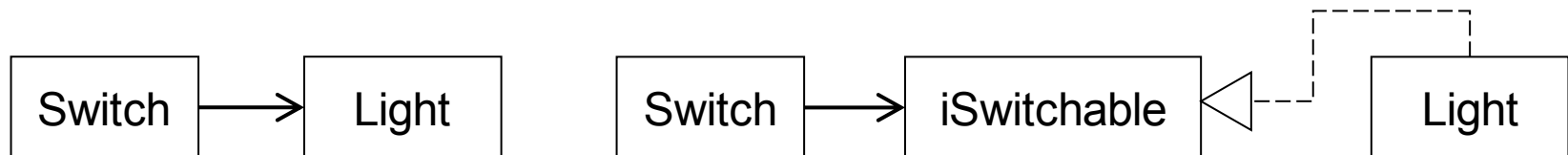
iWithdraw

iChangePIN

# Dependency inversion principle



- ✧ The principle refers to a specific form of **decoupling** where conventional dependency relationships **established from high-level modules to low-level modules are inverted**. The principle states:
  - ✧ **A.** High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - ✧ **B.** Abstractions should not depend upon details. Details should depend upon abstractions.



# Clean code by Robert C. Martin

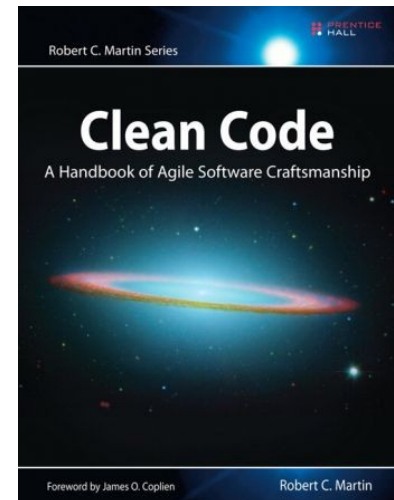


✧ A handbook of agile software craftsmanship

✧ Guidelines for:

- Meaningful names
- Functions
- Comments
- Formatting
- Objects and data structures
- Error handling
- Concurrency
- ... and others

✧ Smells and heuristics



# Design for non-functional qualities



- ✧ Design patterns and programming principles help us to implement specific functionality while maintaining high code quality
  - Respect of design patterns and principles improves system maintainability
- ✧ What if also other non-functional qualities are of high importance?
- ✧ Are there any “patterns” for dependability, performance, testability, etc.?



# Programming guidelines for Dependability



- 1. Limit the visibility of information in a program**
- 2. Check all inputs for validity**
- 3. Provide a handler for all exceptions**
- 4. Minimize the use of error-prone constructs**
- 5. Provide restart capabilities**
- 6. Check array bounds**
- 7. Include timeouts when calling external components**
- 8. Name all constants that represent real-world values**

# Limit the visibility of information in a program

---



- ✧ Program components should only be **allowed access** to data that they need for their implementation.
- ✧ This means that accidental corruption of parts of the program state by these components is impossible.
- ✧ You can control visibility by making data representation **private** and only allowing access to the data through predefined operations such as **get()** and **set()**.

# Check all inputs for validity



## ✧ Range checks

- Check that the input falls within a known range.

## ✧ Size checks

- Check that the input does not exceed some maximum size e.g. 40 characters for a name.

## ✧ Representation checks

- Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.

## ✧ Reasonableness checks

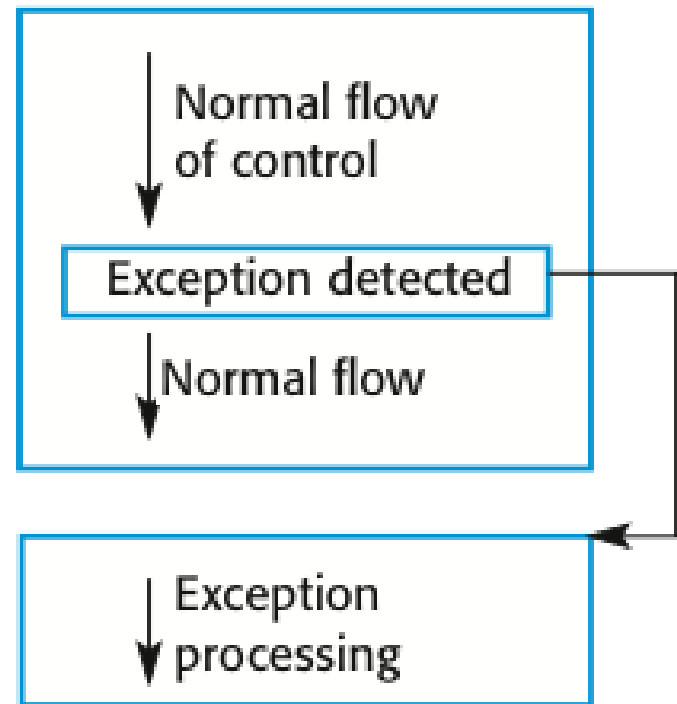
- Use information about the input to check if it is reasonable rather than an extreme value.

# Provide a handler for all exceptions



- ✧ A program exception is an error or unexpected event.
- ✧ Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.

Code section



Exception handling code

# Exception handling



- ✧ Exception handling is a mechanism that implements some level of **fault tolerance**.
- ✧ Exception handling strategies delegate responsibility to:
  - **Caller.** Signal to a calling component that an exception has occurred and provide information about the type of exception.
  - **Callee.** Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
  - **Controller.** Pass control to a run-time support system to handle the exception.

# Minimize the use of error-prone constructs



- ✧ Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
- ✧ This is made worse by error-prone constructs in that they are inherently complex or do not check for mistakes.
  
- ✧ Unconditional branch (goto) statements
- ✧ Pointers
  - When referring to the wrong memory areas can corrupt data.
- ✧ Dynamic memory allocation
  - Run-time allocation can cause memory overflow.

# Error-prone constructs



## ✧ Parallelism

- Can result in unforeseen interaction between processes.

## ✧ Recursion

- Errors in recursion can cause memory overflow.

## ✧ Aliasing

- Using more than 1 name to refer to the same state variable.

## ✧ Floating-point numbers

- Inherently imprecise, leading to invalid comparisons.

## ✧ Interrupts

- Interrupts can cause a critical operation to be terminated.

# Provide restart capabilities



- ✧ For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
  
- ✧ Restart depends on the type of system
  - Keep copies of forms so that users don't have to fill them in again if there is a problem
  - Save state periodically and restart from the saved state



# Check array bounds



- ✧ In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- ✧ This leads to the well-known ‘bounded buffer’ vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- ✧ If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

# Include timeouts when calling external components



- ✧ In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- ✧ To avoid this, you should always include timeouts on all calls to external components.
- ✧ After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

# Name all constants that represent real-world values



- ✧ Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- ✧ You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- ✧ This means that when these ‘constants’ change (for sure, they are not really constant), then you only have to make the change in one place in your program.

# Programming guidelines for Performance

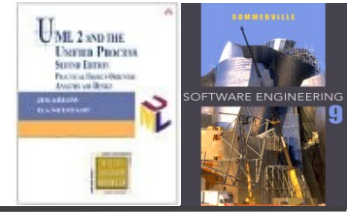


- ✧ Reduce the resources required for processing an event stream.
  - Increase computational efficiency.
  - Reduce computational overhead.
- ✧ Reduce the number of events processed.
  - Manage event rate.
  - Control frequency of sampling.
- ✧ Control the use of resources.
  - Bound execution times.
  - Bound queue sizes.

# Programming guidelines for Testability



- ✧ **Record/playback.** The information crossing an interface during normal operation can be saved and accessed.
- ✧ **Separate interface from implementation.** This allows substitution of implementations for testing purposes.
- ✧ **Specialize access routes/interfaces.** Having specialized testing interfaces allows the capturing or specification of variable values for a component through a test harness, independently from its normal execution.
- ✧ **Built-in monitors.** The component can maintain state, performance load, capacity, security, or other information accessible through an interface.



---

# Implementation Issues

## Lecture 8/Part 2

# Implementation issues



- ✧ Some implementation issues that are often not covered in programming texts:
  - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

# Reuse



- ✧ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse of software was the reuse of functions and objects in programming language libraries.
- ✧ Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- ✧ An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.



# Reuse levels



## ✧ The object level

- At this level, you directly reuse objects from a library rather than writing the code yourself.

## ✧ The component level

- Components are collections of objects and object classes that you reuse in application systems.

## ✧ The system level

- At this level, you reuse entire application systems.

## ✧ The abstraction level

- At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.

# Reuse costs



- ✧ The costs of the time spent in **looking for software to reuse** and assessing whether it meets your needs.
- ✧ Where applicable, the costs of **buying the reusable software**. For large off-the-shelf systems, these costs can be very high.
- ✧ The costs of **adapting and configuring** the reusable software components or systems to reflect the requirements of the system that you are developing.
- ✧ The costs of **integrating reusable software elements** with each other (if you are using software from different sources) and with the new code that you developed.

# Configuration management



- ✧ Configuration management is the name given to the general **process of managing a changing** software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

# Configuration management activities



- ✧ **Version management**, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ **System integration**, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ✧ **Problem tracking**, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

# Host-target development

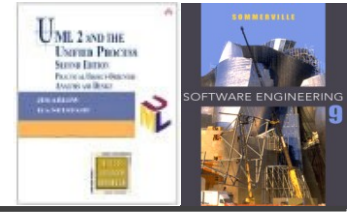


- ✧ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ✧ More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- ✧ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

# Key points



- ✧ When developing software, you should always consider the possibility of **reusing existing software**, either as components, services or complete systems.
- ✧ **Configuration management** is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is **host-target development**. You use a development environment on a host machine to develop the software, which is transferred to a target machine for execution.



---

# Interaction Diagrams

## Lecture 5/Part 3

# Interaction diagrams



## ✧ Sequence diagrams

- Emphasize time-ordered sequence of message sends
- Show interactions arranged in a time sequence
- Are the richest and most expressive interaction diagram
- Do not show object relationships explicitly - these can be inferred from message sends

## ✧ Communication diagrams

- Emphasize the structural relationships between objects
- Use communication diagrams to make object relationships explicit

## ✧ Timing diagrams

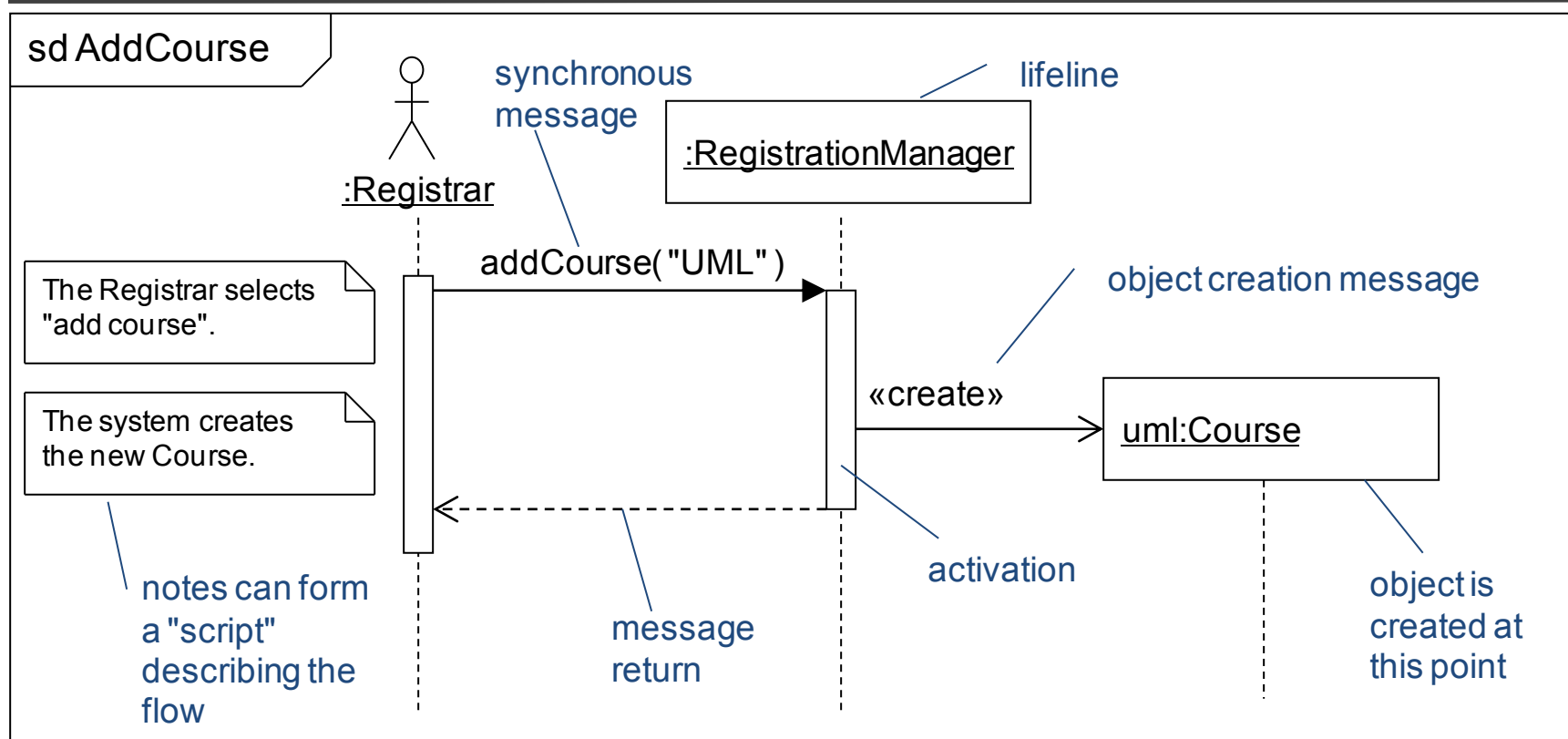
- Emphasize the real-time aspects of an interaction

## ✧ Interaction overview diagrams

- Show how complex behavior is realized by a set of simpler interactions (discussed earlier together with Activity diagrams)



# Sequence diagram syntax



✧ Interactions are captured via **lifelines** (participants in the interaction) and **messages** (communications between lifelines)



✧ Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

# Lifelines



`jimsAccount [ id = "1234" ] : Account`

name

selector




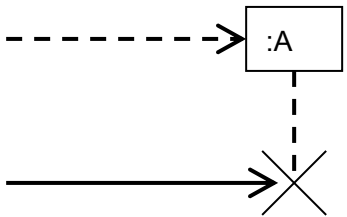
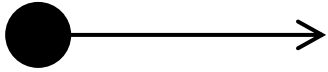
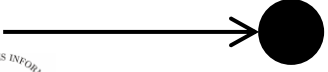
type

- ✧ A lifeline represents a single participant in an interaction
  - Shows how a classifier instance may participate in the interaction
- ✧ Lifelines have:
  - name - the name used to refer to the lifeline in the interaction
  - selector - a boolean condition that selects a specific instance
  - type - the classifier that the lifeline represents an instance of
- ✧ They must be uniquely identifiable within an interaction by name, type or both
- ✧ The lifeline has the same icon as the classifier that it represents

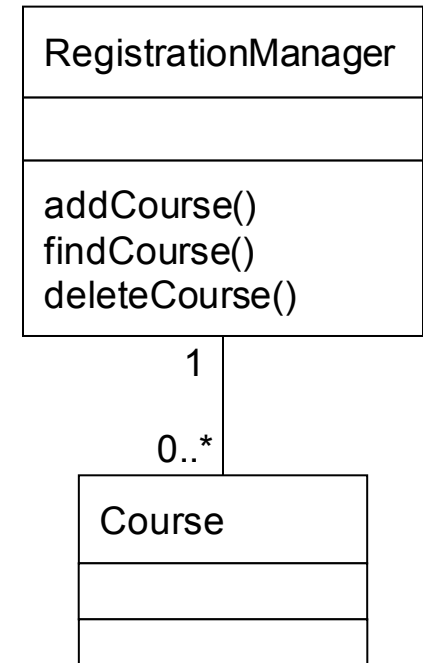
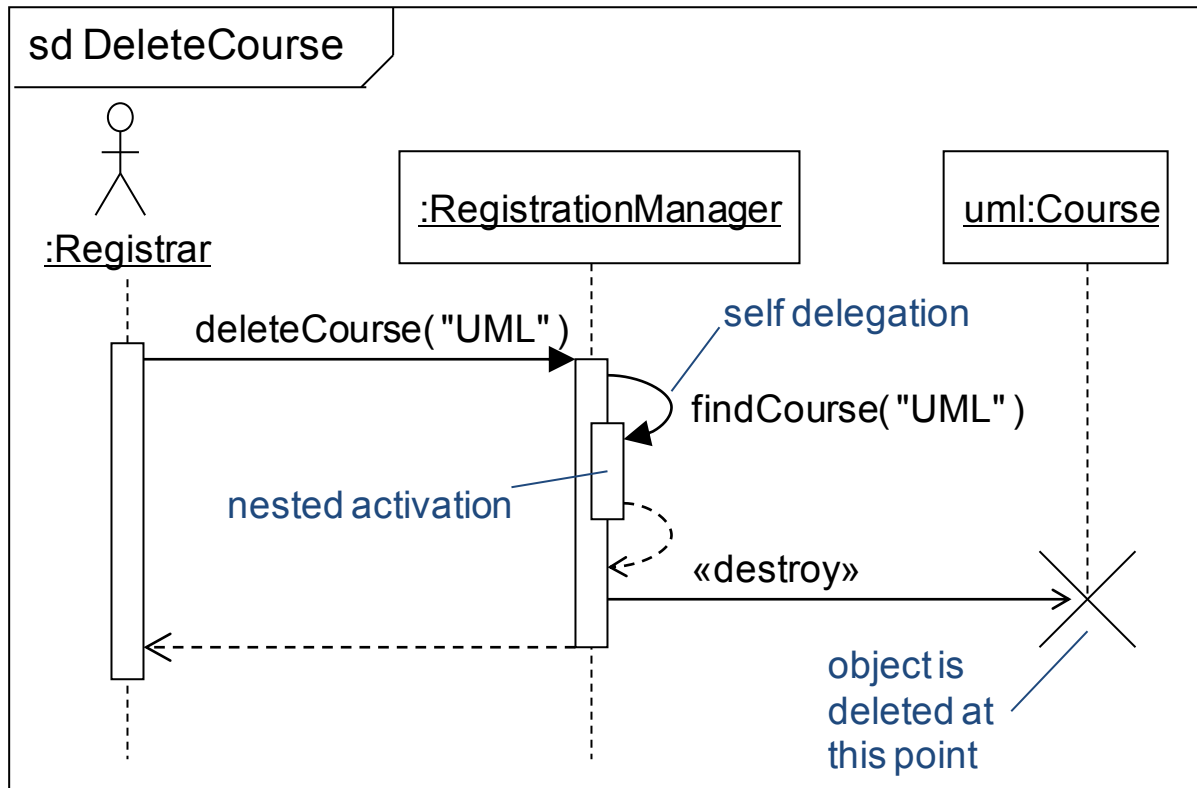
# Messages



✧ A message represents a communication between two lifelines

arrow type	type of message	semantics
	synchronous message	calling an operation synchronously the <b>sender waits</b> for the receiver to complete
	asynchronous send	calling an operation asynchronously, sending a signal the <b>sender does not wait</b> for the receiver to complete
	message return	returning from a synchronous operation call the receiver returns focus of control to the sender
	creation  destruction	the sender creates the target  the sender destroys the receiver
	found message	the message is sent from outside the scope of the interaction
	lost message	the message fails to reach its destination

# Deletion and self-delegation



- ✧ Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation

# Combined fragments – opt and alt

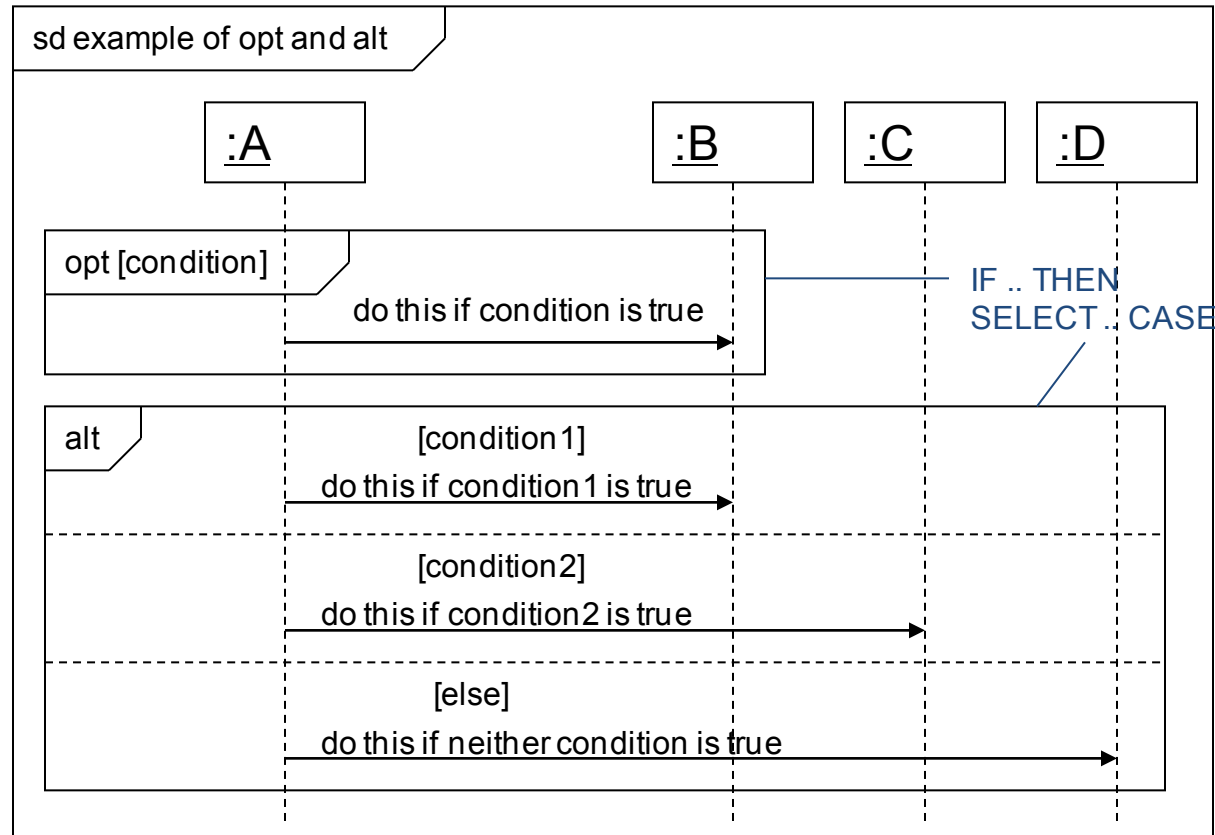


## ✧ OPT semantics:

- single operand that executes if the condition is true

## ✧ ALT semantics:

- two or more operands each protected by its own condition
- an operand executes if its condition is true
- use **else** to indicate the operand that executes if none of the conditions are true



# Combined fragments – loop and break



## LOOP semantics:

- Loop min times, then loop (max – min) times while condition is true

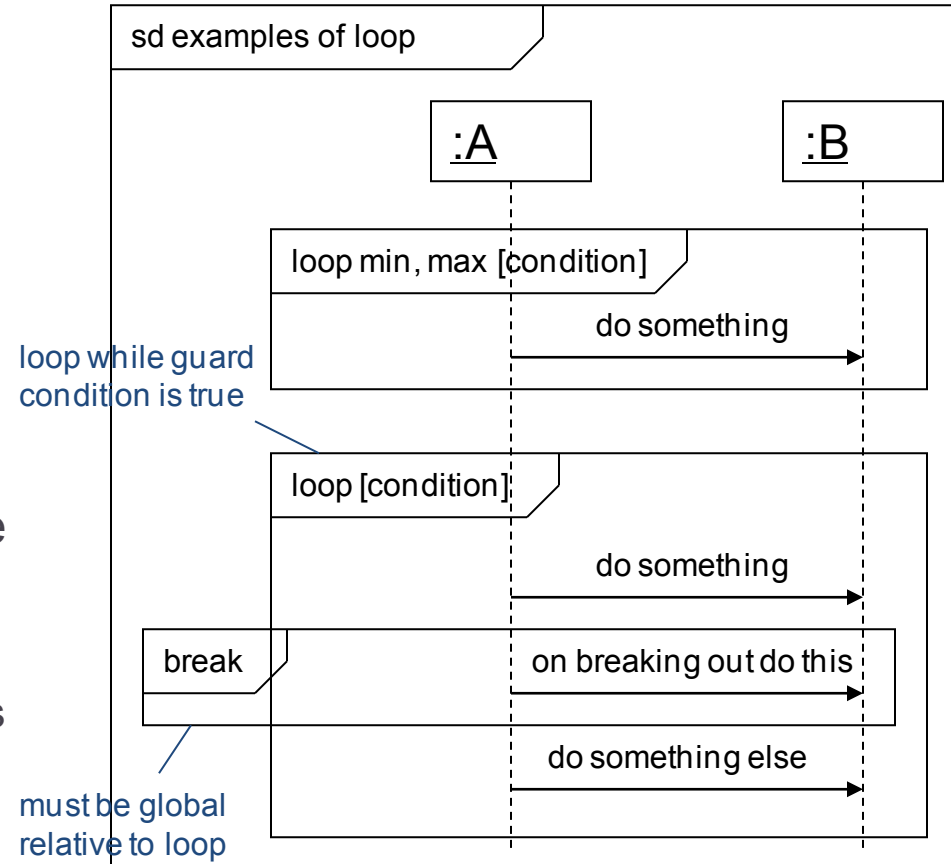
## LOOP syntax:

- A loop without min, max or condition is an infinite loop
- condition can be
  - Boolean expression
  - Plain text expression provided it is clear!

Break specifies what happens when the loop is broken out of:

- The break fragment executes
- The rest of the loop after the break does not execute

The break fragment is outside the loop and so should overlap it as shown



# Loop idioms



type of loop	semantics	loop expression
infinite loop	keep looping forever	loop *
for i = 1 to n {body}	repeat ( n ) times	loop n
while( booleanExpression ) {body}	repeat while booleanExpression is true	loop [ booleanExpression ]
repeat {body} while( booleanExpression )	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]
forEach object in collection {body}	Execute the loop once for each object in a collection	loop [for each object in collection]
forEach object in ObjectType {body}	Execute the loop once for each object of a particular type	loop [for each object in :ObjectType]

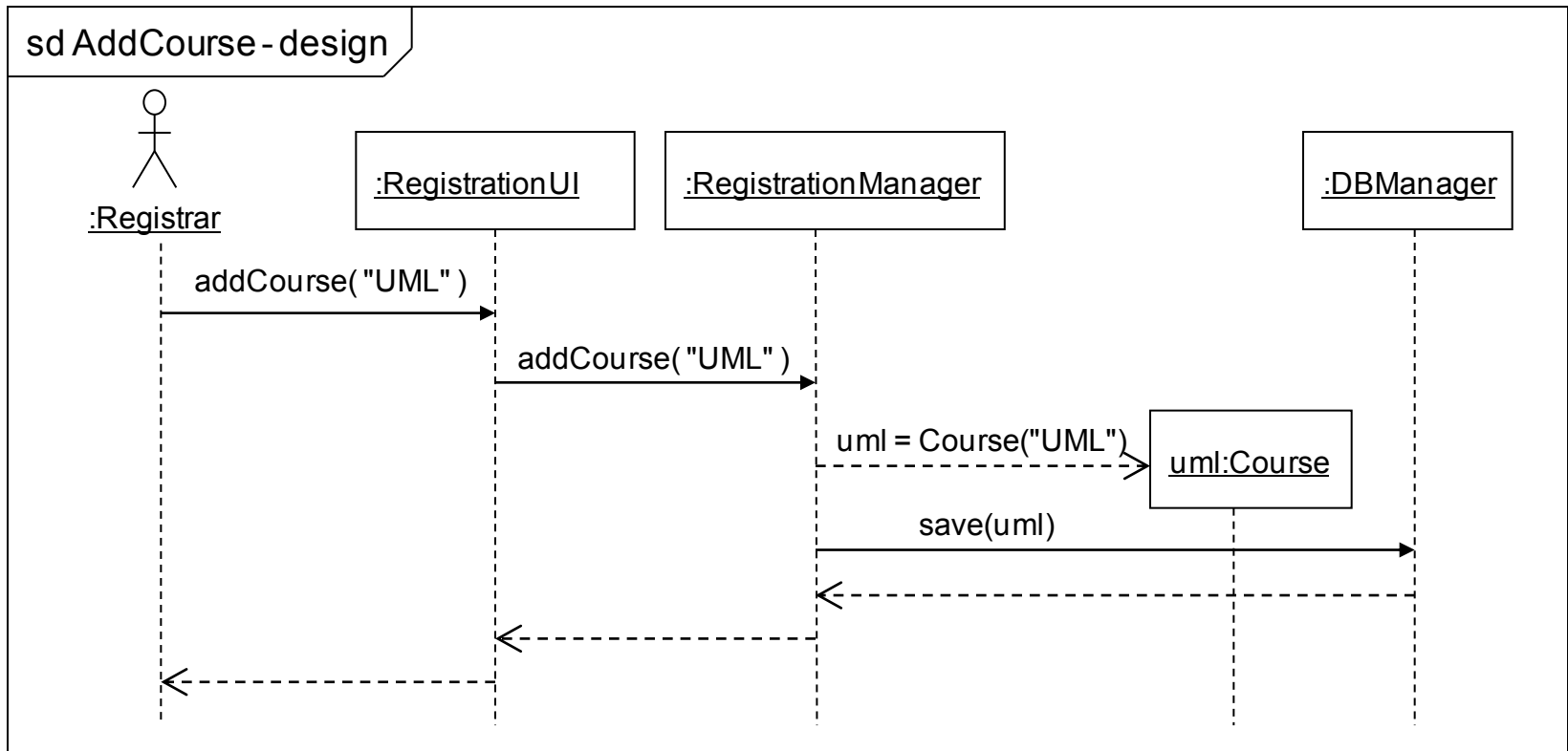
# The rest of the operators



operator	long name	semantics
<b>par</b>	parallel	Both operands execute in <b>parallel</b>
<b>seq</b>	weak sequencing	The operands execute in <b>parallel</b> subject to the constraint that event occurrences on the same lifeline from different operands must happen in the same sequence as the operands
<b>ref</b>	reference	The combined fragment refers to another interaction
<b>strict</b>	strict sequencing	The operands execute in strict <b>sequence</b>
<b>neg</b>	negative	The combined fragment represents interactions that are <b>invalid</b>
<b>critical</b>	critical region	The interaction must execute <b>atomically</b> without interruption
<b>ignore</b>	ignore	Specifies that some messages are intentionally <b>ignored</b> in the interaction
<b>consider</b>	consider	Lists the messages that are <b>considered</b> in the interaction (all others are ignored)
<b>assert</b>	assertion	The operands of the combined fragments are the only <b>valid</b> continuations of the interaction



# Sequence diagrams in design

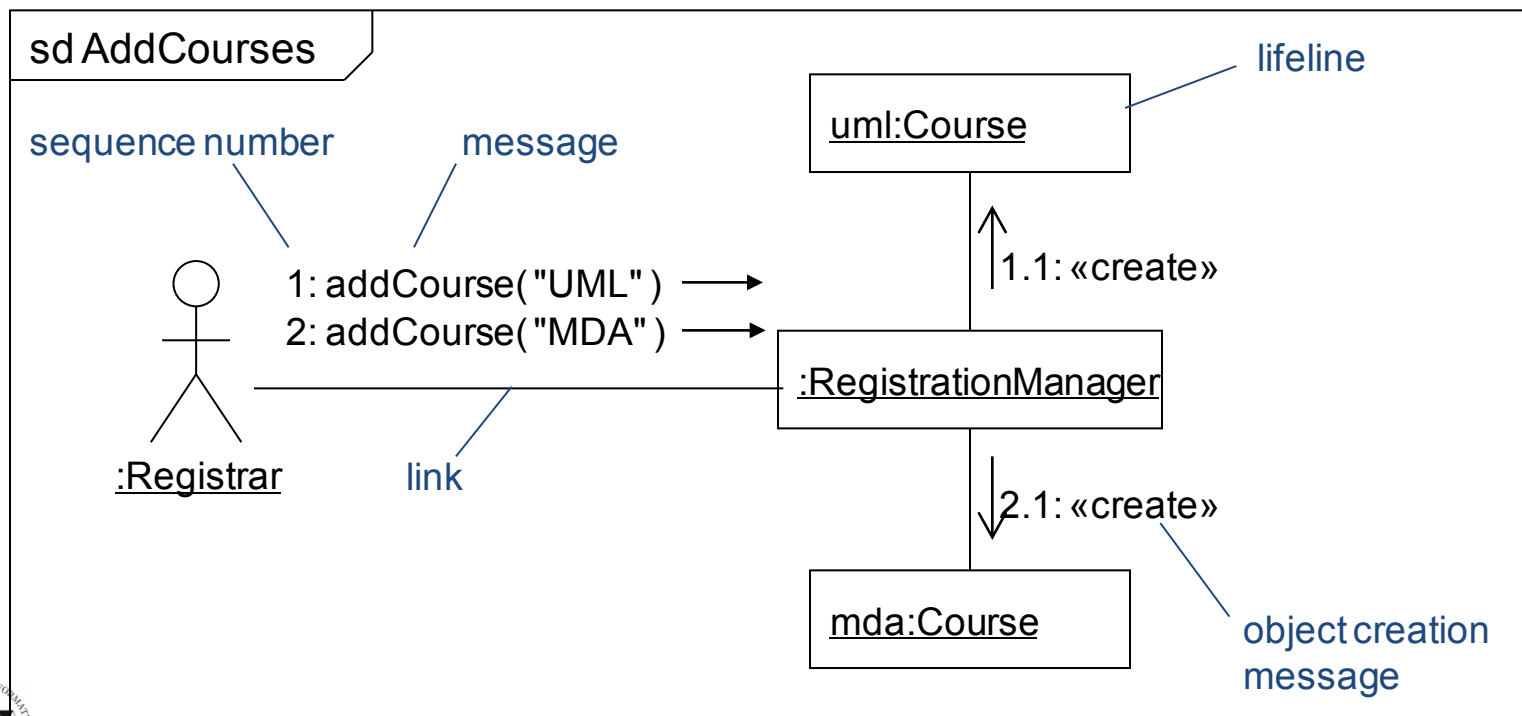


❖ Could you draw a UML Class diagram corresponding to the sequence diagram above?

# Communication diagram syntax



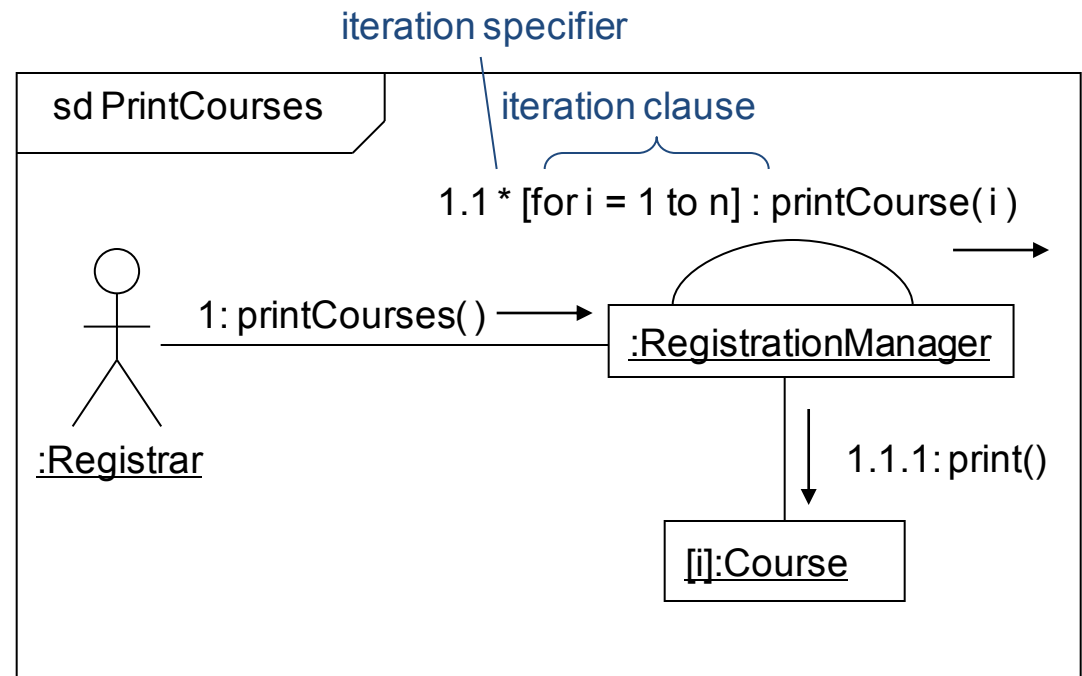
- ✧ Communication diagrams emphasize the structural aspects of an interaction - how lifelines connect together
  - Compared to sequence diagrams they are semantically weaker
  - Object diagrams are a special case of communication diagrams



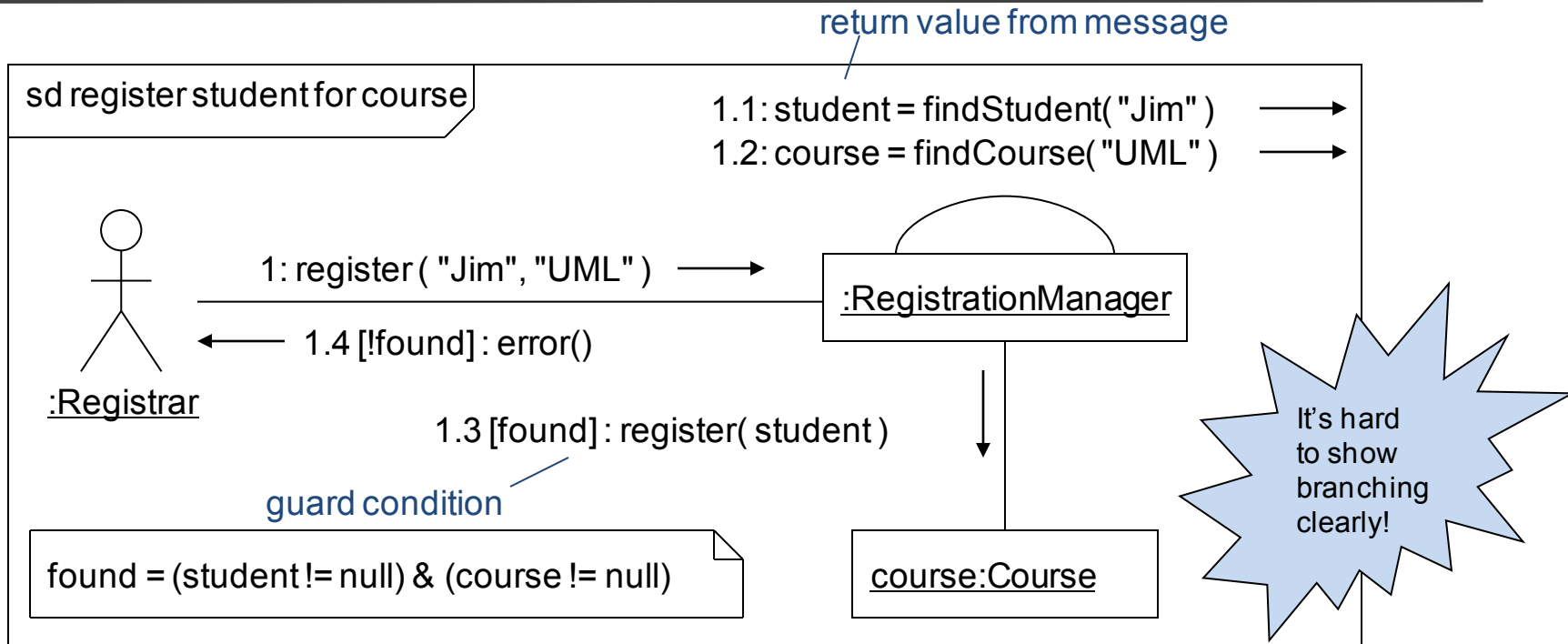
# Iteration



- ✧ Iteration is shown by using the iteration specifier (\*), and an optional iteration clause
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- ✧ To show that messages are sent in parallel use the parallel iteration specifier, \*//



# Branching



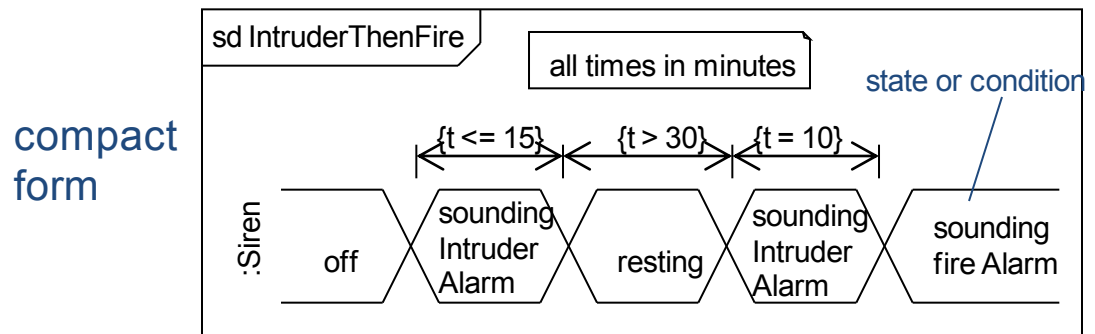
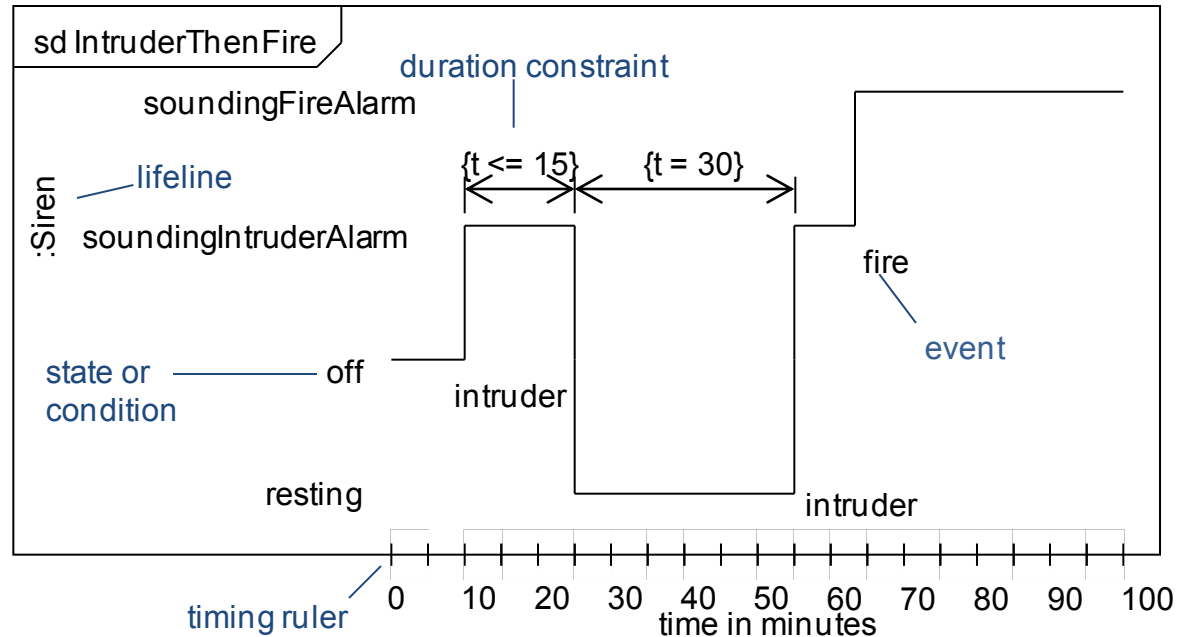
✧ Branching is modelled by prefixing the sequence number with a **guard condition**

- There is no prescribed UML syntax for guard conditions
- In the example above, we use the variable **found**. This is true if both the student and the course are found, otherwise it is false

# Timing diagrams



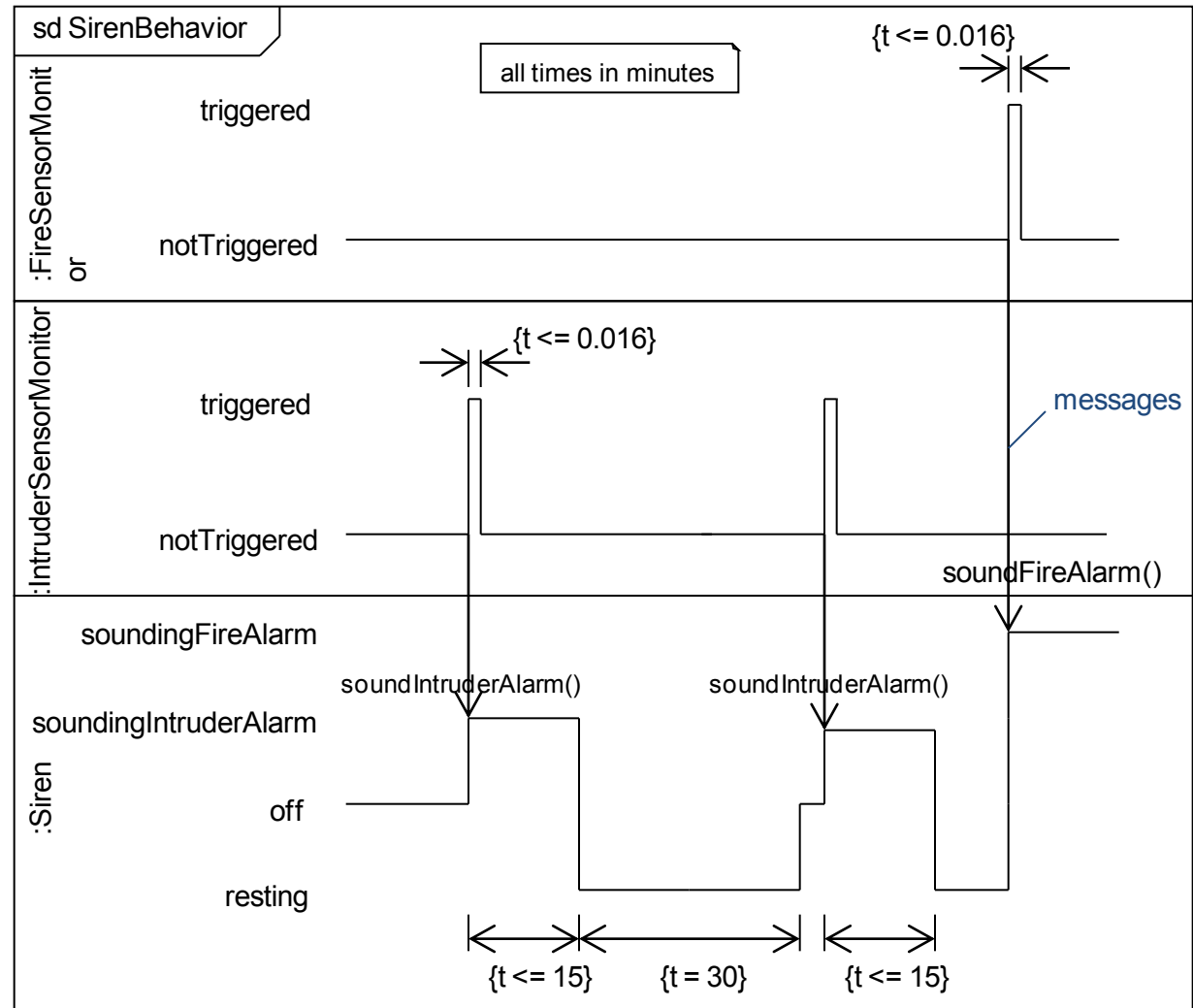
- ✧ Emphasize the real-time aspects of an interaction
- ✧ Used to model timing constraints
- ✧ Lifelines, their states or conditions are drawn vertically, time horizontally
- ✧ It's important to state the time units you use in the timing diagram



# Messages on timing diagrams



- ✧ You can show messages between lifelines on timing diagrams
- ✧ Each lifeline has its own partition



# Key points

---



- ✧ There are four types of interaction diagrams:
  - Sequence diagrams – emphasize time-ordered sequence of message sends
  - Communication diagrams – emphasize the structural relationships between lifelines
  - Timing diagrams – emphasize the real-time aspects of an interaction
  - Interaction overview diagrams – show how complex behavior is realized by a set of simpler interactions; presented together with Activity diagrams