Software Engineering - Design

Summer Term 2024
03 Designing classes

Prof. Dr. Robert Lokaiczyk
Frankfurt University of Applied Sciences
Faculty of Computer Science and Engineering
robert.lokaiczyk@fb2.fra-uas.de

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Träumst du von einem eigenen Startup?

Das kann Wirklichkeit werden!

Entwickle deine Business-Idee, suche dir ein passendes Team und startet gemeinsam durch.

Wir unterstützen dich dabei.

Unser Projekt

Wirtschaft und Gesellschaft sind im Wandel und brauchen Innovationen. Aus Hochschulen heraus gegründete Startups spielen dabei eine zunehmend wichtige Rolle. Dazu braucht es gute Ideen, Expertise und eine solide Gründungsvorbereitung.

Unser Angebot

Wir unterstützen Gründungen im technischen, wirtschaftlichen, ökologischen oder sozialen Bereich und beraten Gründungsinteressierte der Hochschule.

Ideen-Workshops und Veranstaltungsreihen

AppliedIdea Ideenwettbewerb

Individuelle Gründungsberatung

Zugang zu Finanzierung und Vernetzung

Jetzt mehr erfahren
www.frankfurt-university.de

Kontakt
Abteilung HoST -
House of Science and Transfer
Mail: host@host.fra-uas.de
Telefon: +49 69 1533-2169

HoST
House of Science
and Transfer

Announcement

One-time changes to exercise schedule for Group 3/4 (Gian Luca Buono)

May 07 8:15am cancelled, new date: May 08 14:15pm

May 07 10:00am cancelled, new date: May 08 16:00pm

Room 1-236

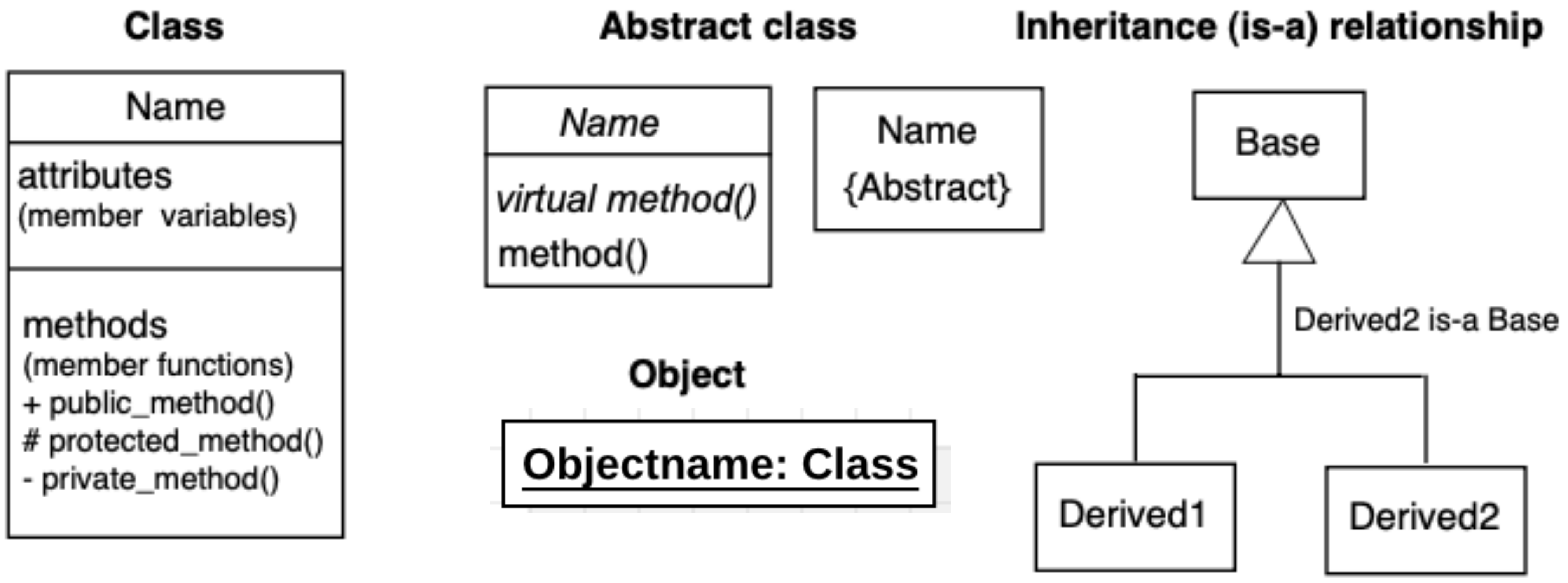
Agenda for today

- UML notation recap
- How to design classes?
    - Class identification
    - Find attributes
    - Find methods
    - Find associations between classes
- SOLID Principles of OOD
- CRC cards

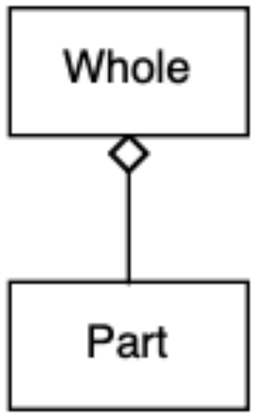
Unified Modeling language (UML)

- is a standardized general-purpose modeling language in the field of software engineering
- includes a set of graphic notation techniques to create visual models of software-intensive systems like use case diagrams, activity diagrams, class diagrams and many more
- Latest specification version is 2.5.1. by Object Modelling Group (OMG)
    - https://www.omg.org/spec/UML/2.5.1/About-UML (751 pages)

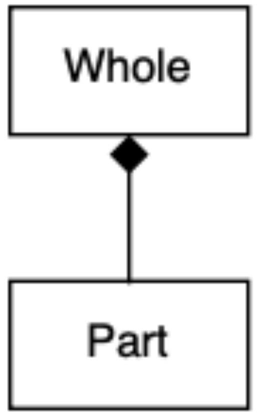
UML Notation revisited



UML Notation revisited

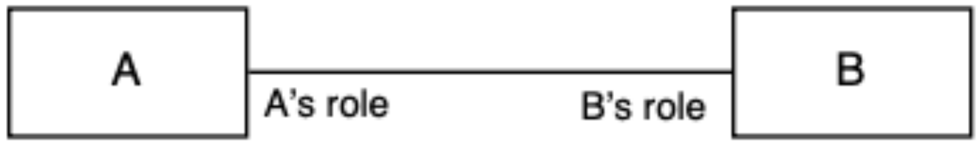


Whole has Part as a part;
lifetimes might be different;
Part might be shared with other Wholes.
(aggregation)

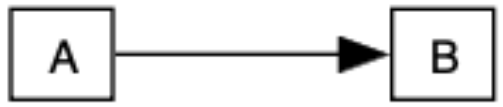


Whole has Part as a part;
lifetime of Part controlled by Whole,
Part objects are contained in one Whole object.
(composition)

Association (uses, interacts-with) relationship



Navigability - can reach
B starting from A

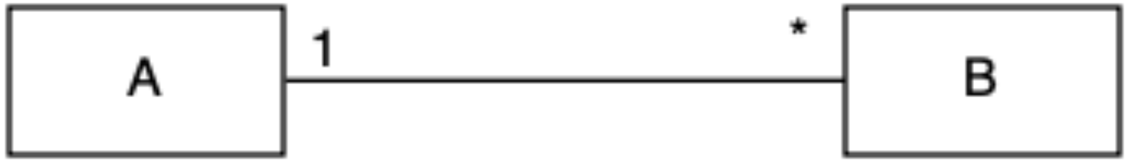


UML Notation revisited

Multiplicity in Aggregation, Composition, or Association

* - any number	0..1 - zero or one
1 - exactly 1	1..* - 1 or more
n - exactly *n*	*n* .. *m* - *n* through *m*

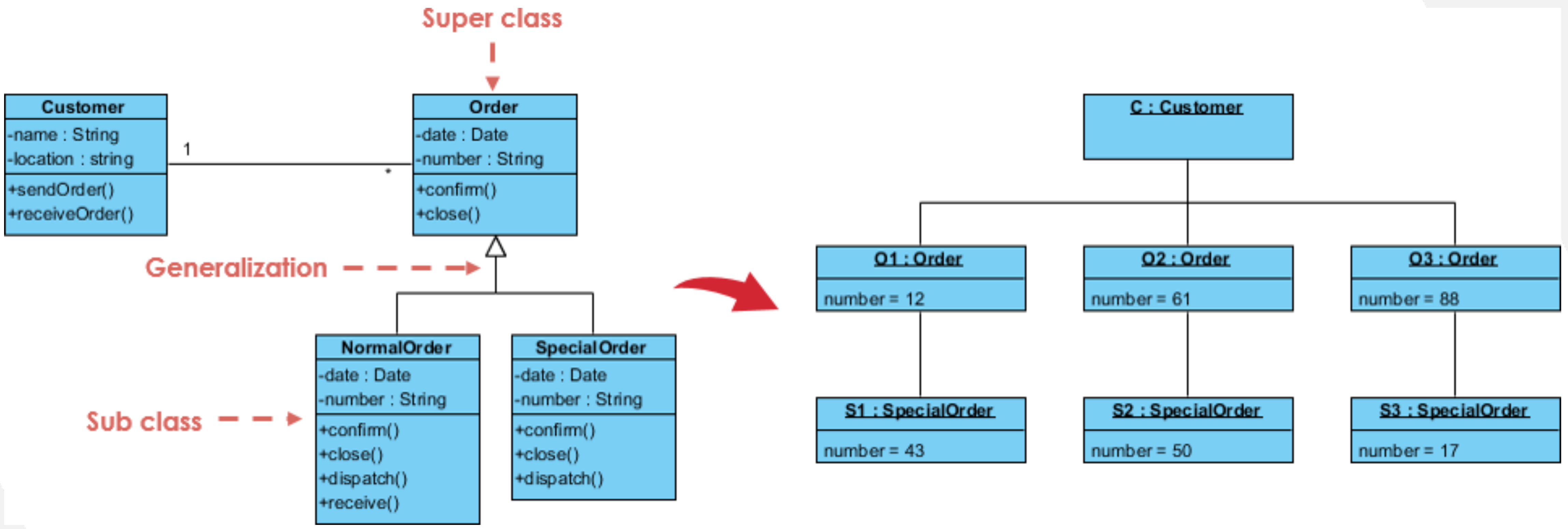
Follow line from start class to end class,
note the multiplicity at the end.
Say “Each <start> is associated with <multiplicity> <ends>”



Each A is associated with any number of B’s.
Each B is associated with exactly one A.

UML Notation revisited

Example: Class and object diagram



Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-object-diagram/

Steps to build an object model

1. Class identification
2. Find attributes
3. Find methods
4. Find associations between classes
5. Add multiplicity

Deriving classes

From the textual description of a software system

A library management software is a comprehensive system designed to streamline the operations of a **library**, facilitating efficient management of **books**, patrons, and library resources. It allows **librarians** to catalog, organize, and track library materials, including books, journals, multimedia items, and more. The software typically features functionalities such as inventory management, circulation control, patron registration, fine management, and reporting capabilities. **Users** can search for and borrow books, renew items, place holds, and manage their accounts online. Additionally, the software may include features for librarians to generate **reports** on library usage, analyze collection data, and optimize resource allocation.

Deriving classes

Linguistic heuristics

Grammatical construct	Model component	Example
Name	Instance / Object	Mike, Mazda, FRA UAS
Noun	**Class**	Customer, Account
Transitive verb 1)	**Operation**	help so., buy sth,
Intransitive verb	Event	appear
Classifying verb, „to be"	Generalization	is a type of, is one of, is a kind of
Possessive verb 2), „Have"	Aggregation	has, consists of
Modal verb	Constraint	must be
Adjective	**Attribute**	four-sided, heavy, green

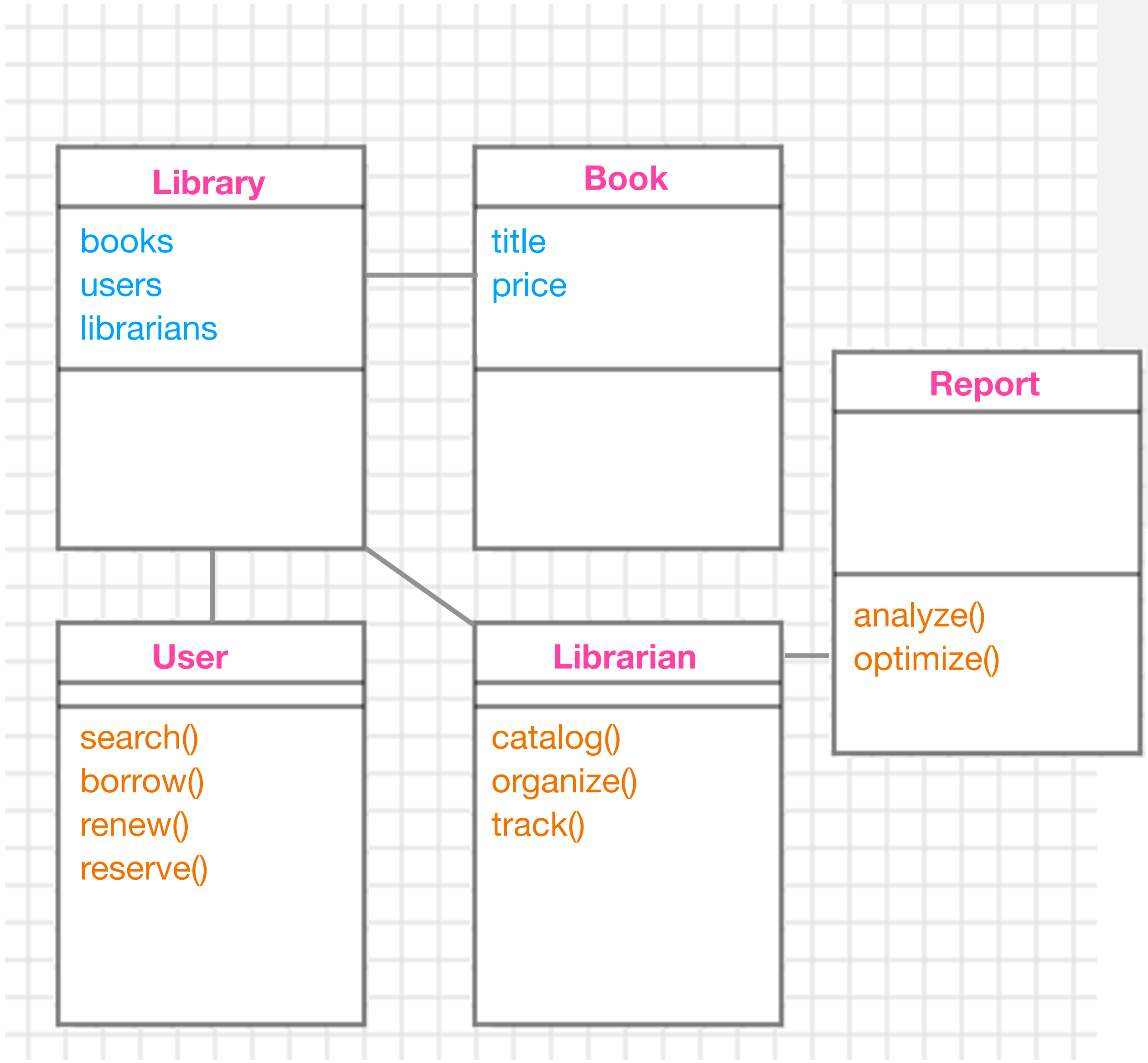
1) Does have an object in accusative

2) Signalizes ownership

Source: Abbott 1983

Result

A library management software is a comprehensive system designed to streamline the operations of a library, facilitating efficient management of books, patrons, and library resources. It allows librarians to catalog, organize, and track library materials, including titled books, journals, multimedia items, and more. The software typically features functionalities such as inventory management, circulation control, patron registration, fine management, and reporting capabilities. Users can search for and borrow books, renew items, place holds, and manage their accounts online. Additionally, the software may include features for librarians to generate reports on library usage, analyze collection data, and optimize resource allocation.



Deriving classes

Abbott provides rules but does not replace your own thinking

- Nouns are good candidates for classes
- Verbs are good candidates for operations
- Adjectives are often candidates for attributes

Manual control and check for correctness needed!

Table has legs.

Child is a person.

Customer has account.

Mike is a person.

Questions

- Which class **holds** information?
- Which class is **responsible for creating** objects?
- Which class is **responsible for calling** operations?
- How are the classes *linked* together?
- How are we **grouping** classes together into **components**?

Naming classes

Names:

- Should be a noun in singular
- Should be unique in package
- Be descriptive, not generic: ~~*Manager, Controller, DataHandler, Helper, Misc, Thing*~~
- Be consistent, use same terms or naming conventions in other classes
- In Java classes typically start Uppercase and use CamelCase: e.g. MyClassName
- Avoid abbreviations, they make code less readable and understandable
- Using domain terminology makes alignment with domain experts easier

Deriving attributes

- Choose attributes by the responsibility of the class to manage the data (CRC cards)
- Should be nouns or adjective nouns, not verbs
- Should not repeat class name
- No abbreviations again
- Attributes are a class attribute if the have the same value for all objects
- Use access modifiers such as public, private, protected
- Avoid redundancy, do not store the same value in multiple classes (inconsistency)
- Outsource sets of attributes that have their own object identity to other classes (e.g. Person, Street, Postcode, City) or exist independently

Deriving methods

- Defined by the behaviour the class is expected to exhibit externally
- Model the real world interaction with other classes
- Follow the single responsibility principle, do atomic things or split. ~~orderAndPay()~~
- Prioritize by importance to class responsibility
- Be descriptive: ~~processData(), calculateResults(), helperMethod(), handleRequest(~~)

General Responsibility Assignment Software Patterns

GRASP

- **Information expert**:
  - Where to assign responsibilities to classes?
  - Assign responsibility to the class that has the information needed to fulfill it
- **Creator**
  - Who creates instances of class A ?
  - Class B create objects of A if:
    - B contains or compositely aggregate A
    - B record A
    - B closely uses A
    - B has initializing information for A
- **Controller**: See MVC

Source: Craig Larman - Applying UML and Patterns, 1997

Deriving associations between classes

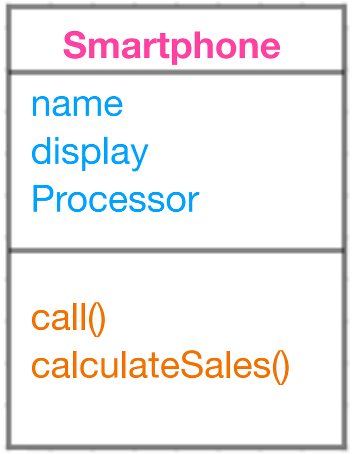
- Start with simple association (single line) between class A and B (no cardinality, no aggregations)
- Check for specific type of association:
  - A is a physical component of B?
  - A is a logical component of B?
  - A is an element / member of B?
  - A uses B?
  - A owns B?
- Name associations
- Add roles
- Add multiplicity

SOLID: 5 object-oriented design principles

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single responsibility principle (SRP)

- Introduced by Robert C. Martin (2003)
- „A class should have only one reason to change“
- meaning:
  - A class should only do one thing / have one responsibility



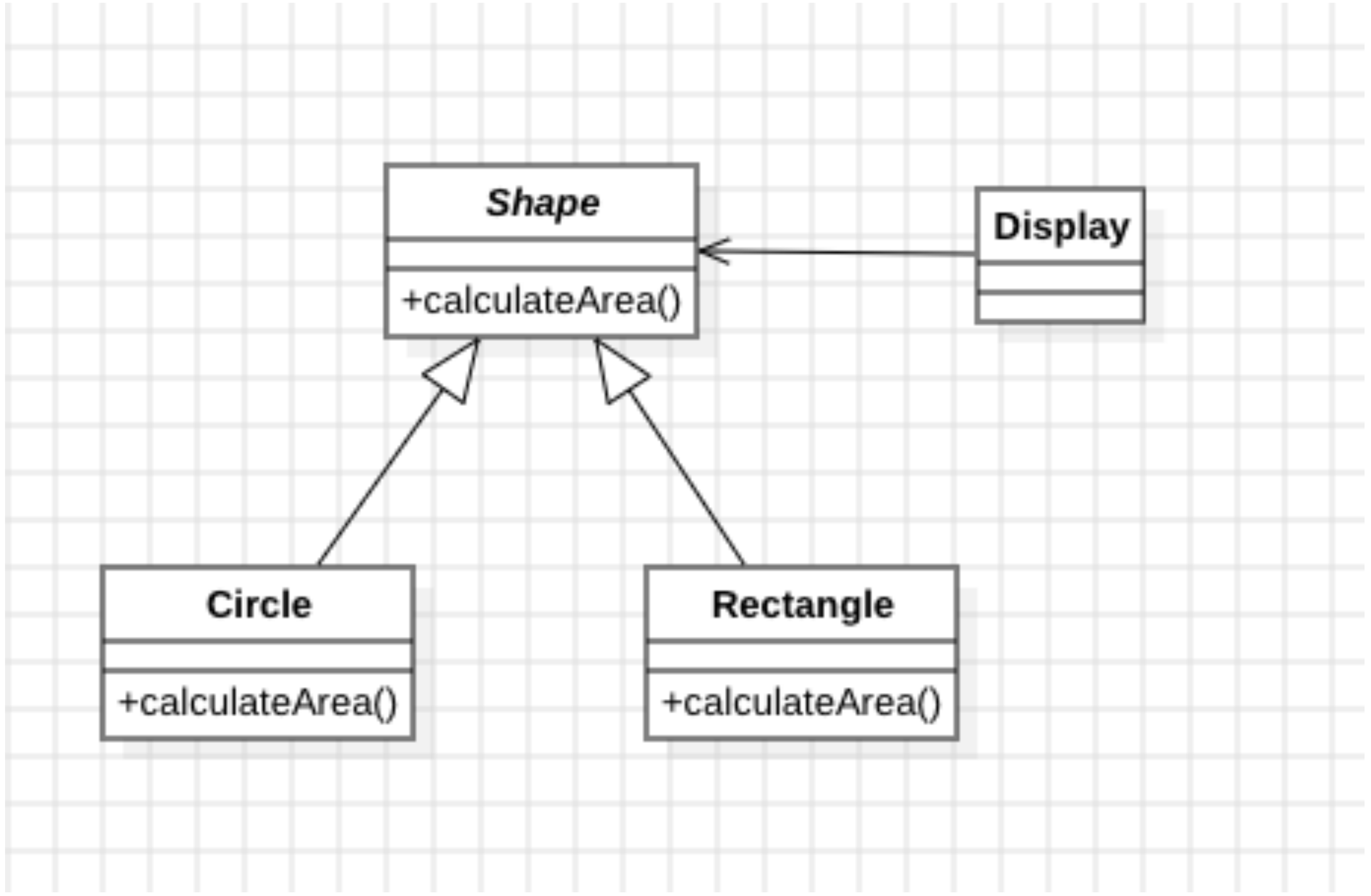
Source: Angelacleancoder - CC BY-SA 4.0

Open/Close principle

- Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- This can be achieved via inheritance and dynamic binding. According to Robert C. Martin [Mar96], modules that conform to the open-closed principle have two primary attributes.
- 1. They are “Open For Extension”. This means that the **behavior** of the module **can be extended**. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
- 2. They are “Closed for Modification”. The source **code** of such a module is **inviolate**. No one is allowed to make source code changes to it.

Source: Mayer 1988

Open/Close principle example



Liskov substitution principle

- The Liskov Substitution Principle (**LSP**) is one of the principles of object-oriented programming, named after **Barbara Liskov**, who formulated it in 1987. The principle states that objects of **a superclass should be replaceable with objects of its subclass without affecting the correctness** of the program.
- In practice, this means that any subclass should **adhere to the contracts and behaviors** defined by its superclass. It should not introduce new behaviors that are **not** part of the superclass interface, nor should it **violate** any **invariants** established by the superclass. Violating the Liskov Substitution Principle can lead to unexpected behavior and logical errors in the program.

Source: Kenneth C. Zirkel - CC BY-SA 3.0

Definition

- The Liskov Substitution Principle (LSP) states the following:
- Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T .

$$S \leq T \to \forall x{:}T.\, \phi(x) \to \forall y{:}S.\, \phi(y)$$

S=Subtype

T=Type

- Objects of S can be replaced with objects of T:
- Same or weaker preconditions
- Same or stronger postconditions
- Derived methods should not assume more or deliver less

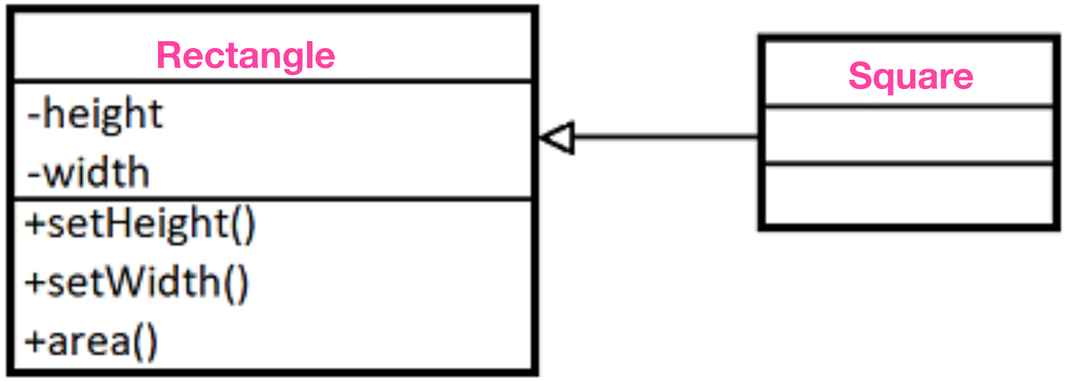
Example

- LSP violation when a square class is derived from a rectangle class, assuming setter methods exist for both width and height:

  A square is a rectangle but

  `setWith(20)`

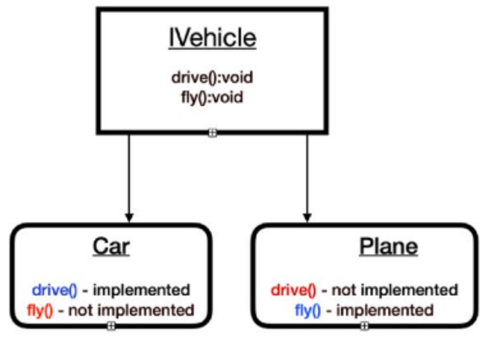
  to a square makes it no longer a square

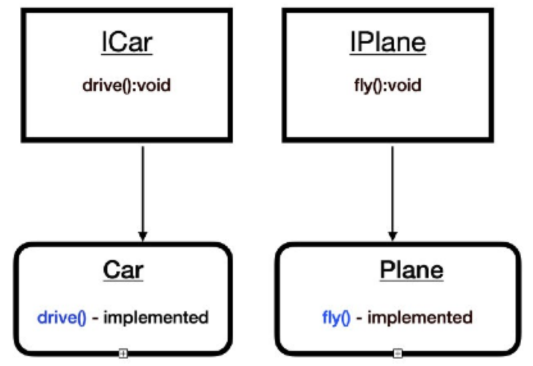


- Solution: Abstract class Shape that must override the methods

Interface segregation

- No code should depend on methods it does not use. **No class should be compelled to implement interfaces it does not need**.
- Instead, **shrink interfaces** to smaller versions that are specific to the requirements of the class
- This **reduces** the **coupling** between classes and interfaces
- This brings advantages in code flexibility and maintainability

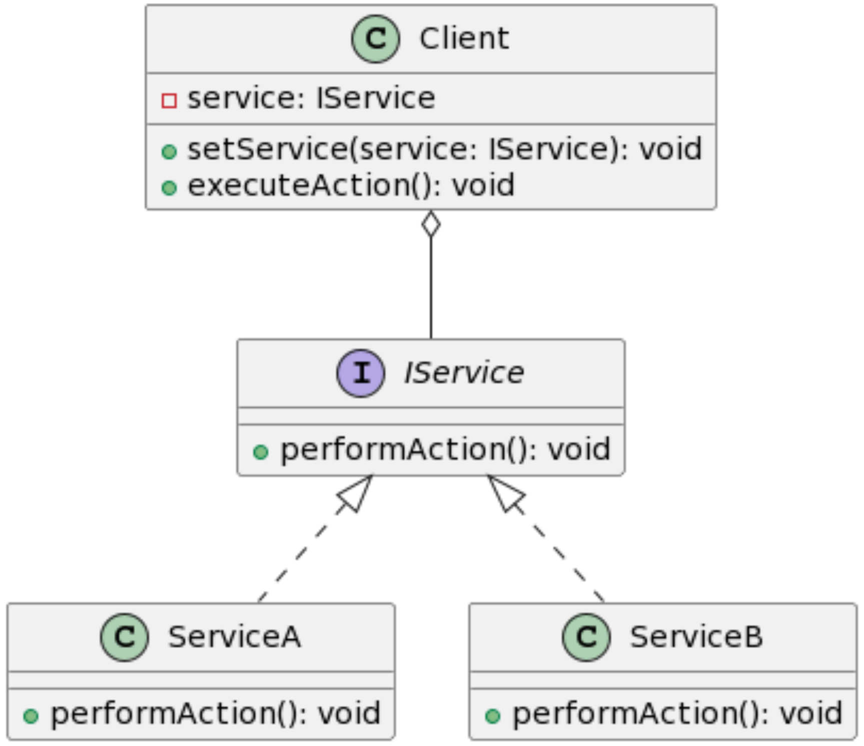




Source: Vagdevi Kommineni

Dependency inversion

- Inverses the conventional high-level, policy-setting modules dependency to low-level modules:
- 1. The Dependency Inversion Principle (DIP) asserts that high-level modules should not depend on low-level modules; both should depend on abstractions.
- 2. Abstractions should not depend on concrete implementations details. Details should depend an abstractions.



Created with PlantUML https://www.planttext.com/

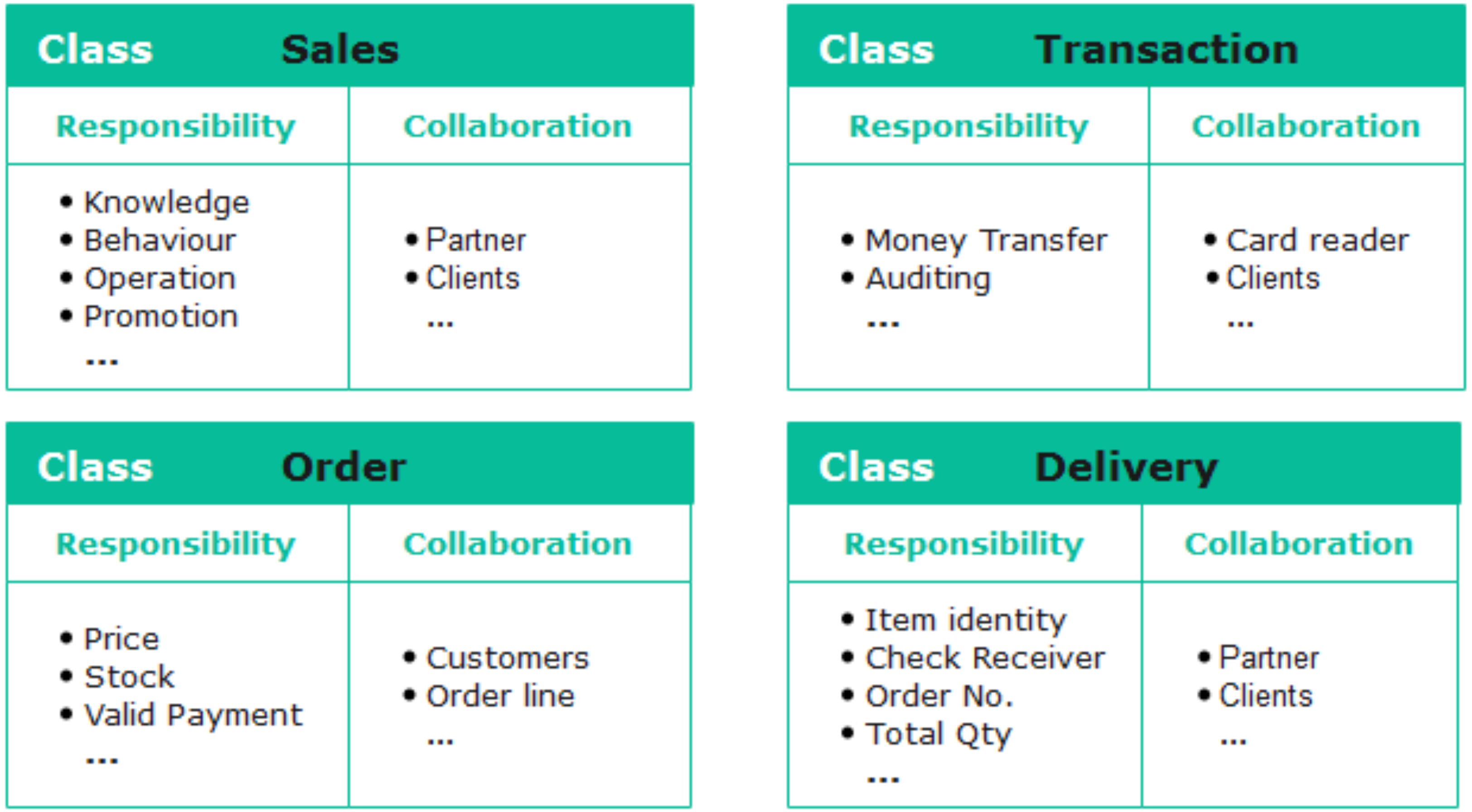
Class-Responsibility-Collaboration (CRC cards)

- CRC (Class-Responsibility-Collaboration) cards are a **brainstorming and design tool** used in **object-oriented software development**.
- popularized by Ward Cunningham and Kent Beck in the early 1990s
- facilitate collaborative design discussions among team members, typically in the early stages of a project, **to define classes, their responsibilities, and their interactions**.
- Each CRC card represents a **single class** in the system
- During a CRC card session, team members collaboratively write CRC cards for various classes in the system on physical index cards or via digital tools. They discuss the responsibilities and collaborations of each class, **identifying potential design flaws**, refining class definitions, and ensuring a **clear understanding of the system's structure and behavior**.

CRC cards structure

Class name of the class	
Responsibilities *What is the class supposed to do?* *Responsibilities are the tasks or behaviors that the class should perform to fulfill its role in the system. Consider what actions the class needs to take and what data it needs to manage.*	**Collaborators** *Which other classes is the class collaborating with?* *Identify which other classes each class needs to interact with to fulfill its responsibilities. This helps in understanding the relationships and dependencies between classes in the system.*

CRC cards example



Class	Sales
Responsibility	Collaboration
• Knowledge • Behaviour • Operation • Promotion ...	• Partner • Clients ...

Class	Transaction
Responsibility	Collaboration
• Money Transfer • Auditing ...	• Card reader • Clients ...

Class	Order
Responsibility	Collaboration
• Price • Stock • Valid Payment ...	• Customers • Order line ...

Class	Delivery
Responsibility	Collaboration
• Item identity • Check Receiver • Order No. • Total Qty ...	• Partner • Clients ...

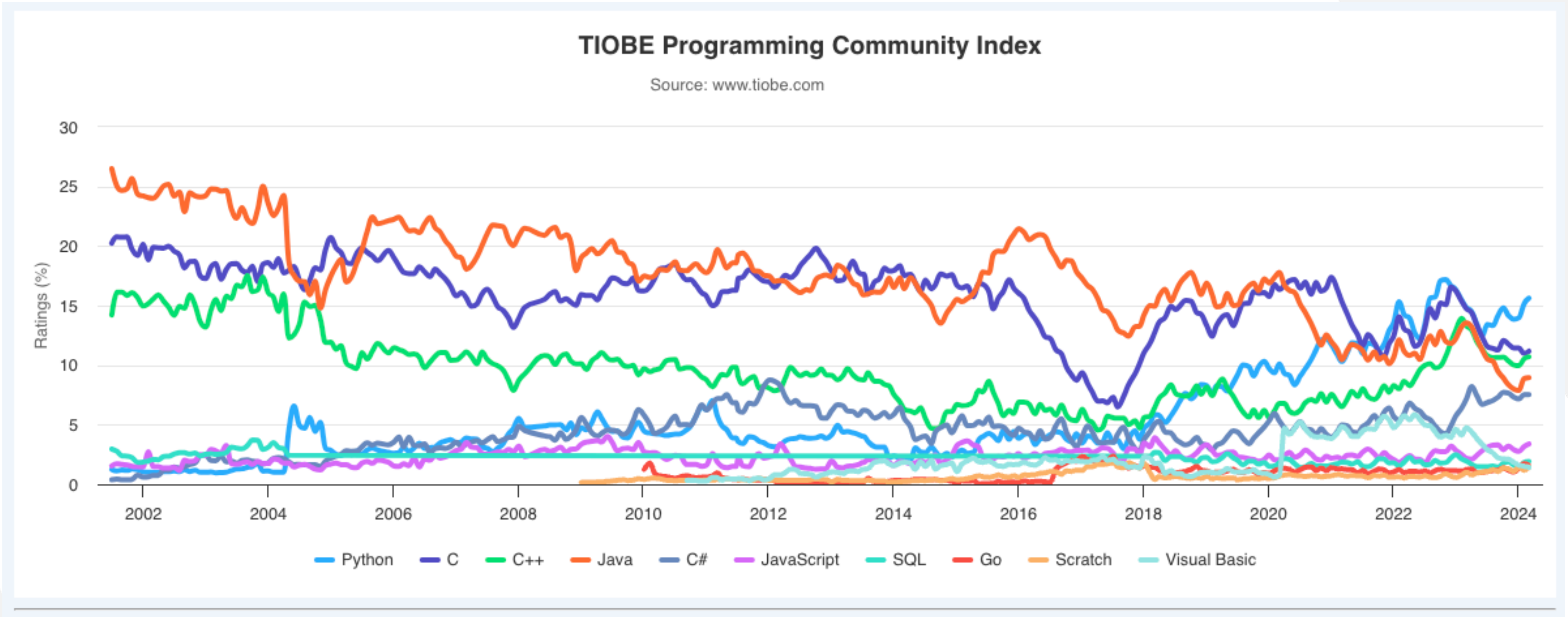
Source: © 2024 Edrawsoft

Purpose

„One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. **It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code**."

—Roger Pressman

OOP is not everything



- OO is a popular programming paradigm. But there are others, such as Functional Programming, Aspect Oriented Programming, Procedural Programming, Logic Programming

Final remarks

- **Summary**:
  - You understand how to build a design model in UML
  - You understand and are able to apply the SOLID principles
  - You can create CRC cards.
- **Next week**: Modular design
- **Please read**: Aldrich - GRASP as recap