Reliable Programming

Youssef Ghatas

Acknowledgement

- Presentation is adapted from Ian Sommerville “Engineering Software Products”

Objective

- General Brief Overview.

Software quality

- Software product quality attributes
- Programming for reliability
  - *Fault avoidance*
    - Complexities
    - Complexity reduction guidelines
    - Refactoring & Code smells
  - *Input validation*
    - *Regular Expressions*
  - *Failure Management*

Software quality

- Creating a successful software product does not simply mean providing useful features for users.
- You need to create a high-quality product that people want to use.
- Customers have to be confident that your product will not crash or lose information, and users have to be able to learn to use the software quickly and without mistakes.

Software product quality attributes

Programming for reliability

- There are three simple techniques for reliability improvement that can be applied in any software company.
  - *Fault avoidance* You should program in such a way that you avoid introducing faults into your program.
  - *Input validation* You should define the expected format for user inputs and validate that all inputs conform to that format.
  - *Failure management* You should implement your software so that program failures have minimal impact on product users.

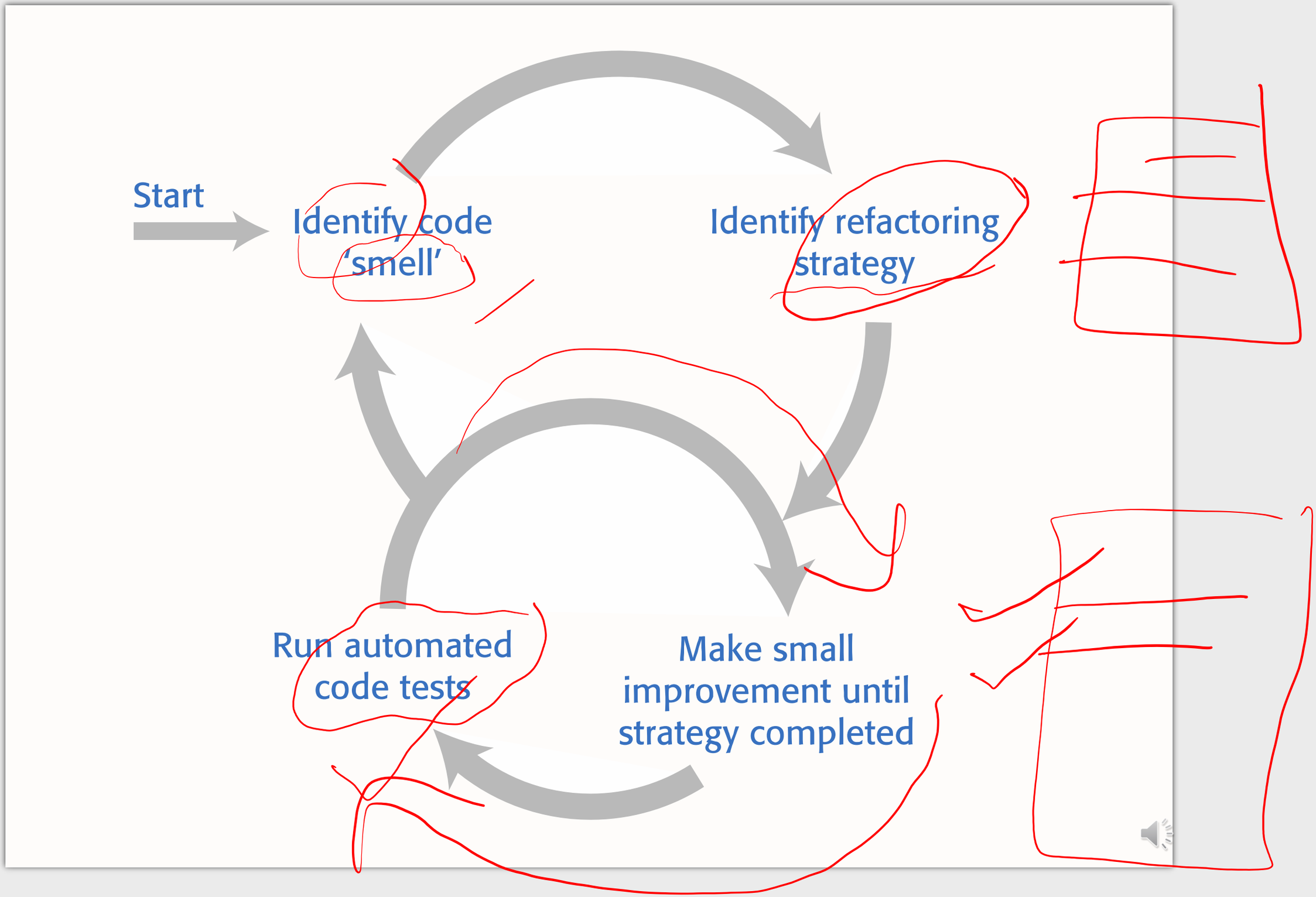
Relevant (explained in the lab)

- Clean Code
- Design Pattern

Refactoring

- Refactoring means changing a program to reduce its complexity without changing the external behaviour of that program.
  - More Readable
  - **easier to change**
- The reality of programming is that as you make changes and additions to existing code, you inevitably increase its complexity. Refactoring is inevitable.

A refactoring process



Code smells

- Martin Fowler, a refactoring pioneer, suggests that the starting point for refactoring should be to identify code ‘smells’.
- Code smells are indicators in the code that there might be a deeper problem.
  - For example, very large classes may indicate that the class is trying to do too much. This probably means that its structural complexity is high.

Code smells

Large classes
Large classes may mean that the single responsibility principle is being violated. Break down large classes into easier-to-understand, smaller classes.

Long methods/functions
Long methods or functions may indicate that the function is doing more than one thing. Split into smaller, more specific functions or methods.

Duplicated code
Duplicated code may mean that when changes are needed, these have to be made everywhere the code is duplicated. Rewrite to create a single instance of the duplicated code that is used as required

Meaningless names
Meaningless names are a sign of programmer haste. They make the code harder to understand. Replace with meaningful names and check for other shortcuts that the programmer may have taken.

Unused code
This simply increases the reading complexity of the code. Delete it even if it has been commented out. If you find you need it later, you should be able to retrieve it from the code management system.

Refactoring for complexity reduction

Reading complexity
You can **rename** variable, function and class names throughout your program to make their purpose more obvious.

Structural complexity
You can **break long classes or functions** into shorter units that are likely to be more cohesive than the original large class.

Data complexity
You can simplify data by **changing your database schema** or reducing its complexity. For example, you can merge related tables in your database to remove duplicated data held in these tables.

Input validation

- Input validation involves checking that a user's input is in the correct format and that its value is within the range defined by input rules.
- Input validation is critical for security and reliability. As well as inputs from attackers that are deliberately invalid, input validation catches accidentally invalid inputs that could crash your program or pollute your database.
- User input errors are the most common cause of database pollution.
- You should define rules for every type of input field and you should include code that applies these rules to check the field's validity.
  - If it does not conform to the rules, the input should be rejected.

Methods of implementing input validation

Built-in validation functions

You can use input validator functions provided by your web development framework. For example, most frameworks include a validator function that will check that an email address is of the correct format.

Explicit comparisons

You can define a list of allowed values and possible abbreviations and check inputs against this list. For example, if a month is expected, you can check this against a list of all months and recognised abbreviations.

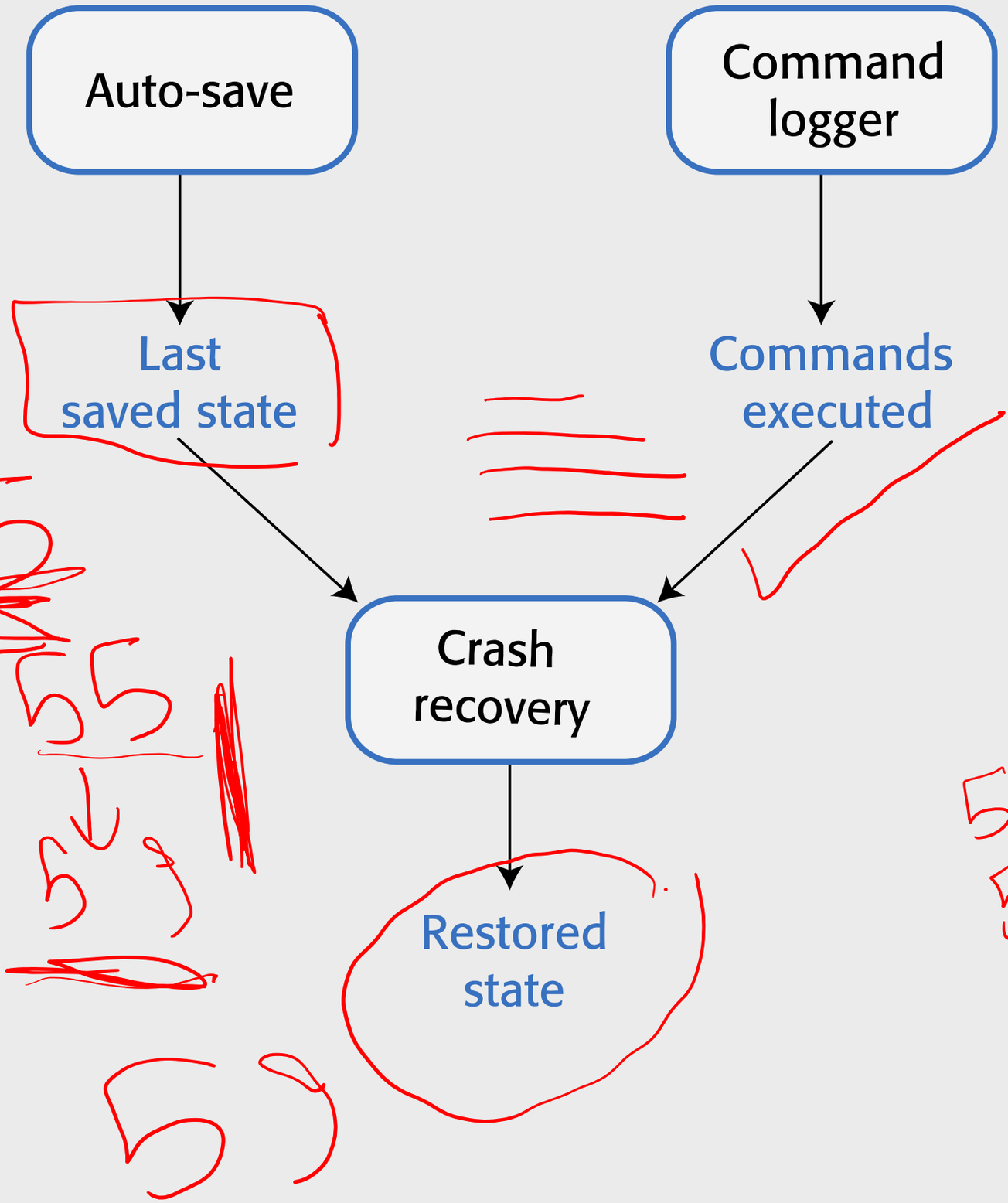
Regular expressions

You can use regular expressions to define a pattern that the input should match and reject inputs that do not match that pattern.

Auto-save and activity logging

- Activity logging
  - You keep a log of what the user has done and provide a way to replay that against their data. You don’t need to keep a complete session record, simply a list of actions since the last time the data was saved to persistent store.
- Auto-save
  - You automatically save the user’s data at set intervals - say every 5 minutes. This means that, in the event of a failure, you can restore the saved data with the loss of only a small amount of work.
  - Usually, you don’t have to save all of the data but simply save the changes that have been made since the last explicit save.

Auto-save and activity logging



External service failure

- If your software uses external services, you have no control over these services and the only information that you have on service failure is whatever is provided in the service's API.

- As services may be written in different programming languages, these errors can't be returned as exception types but are usually returned as a numeric code.

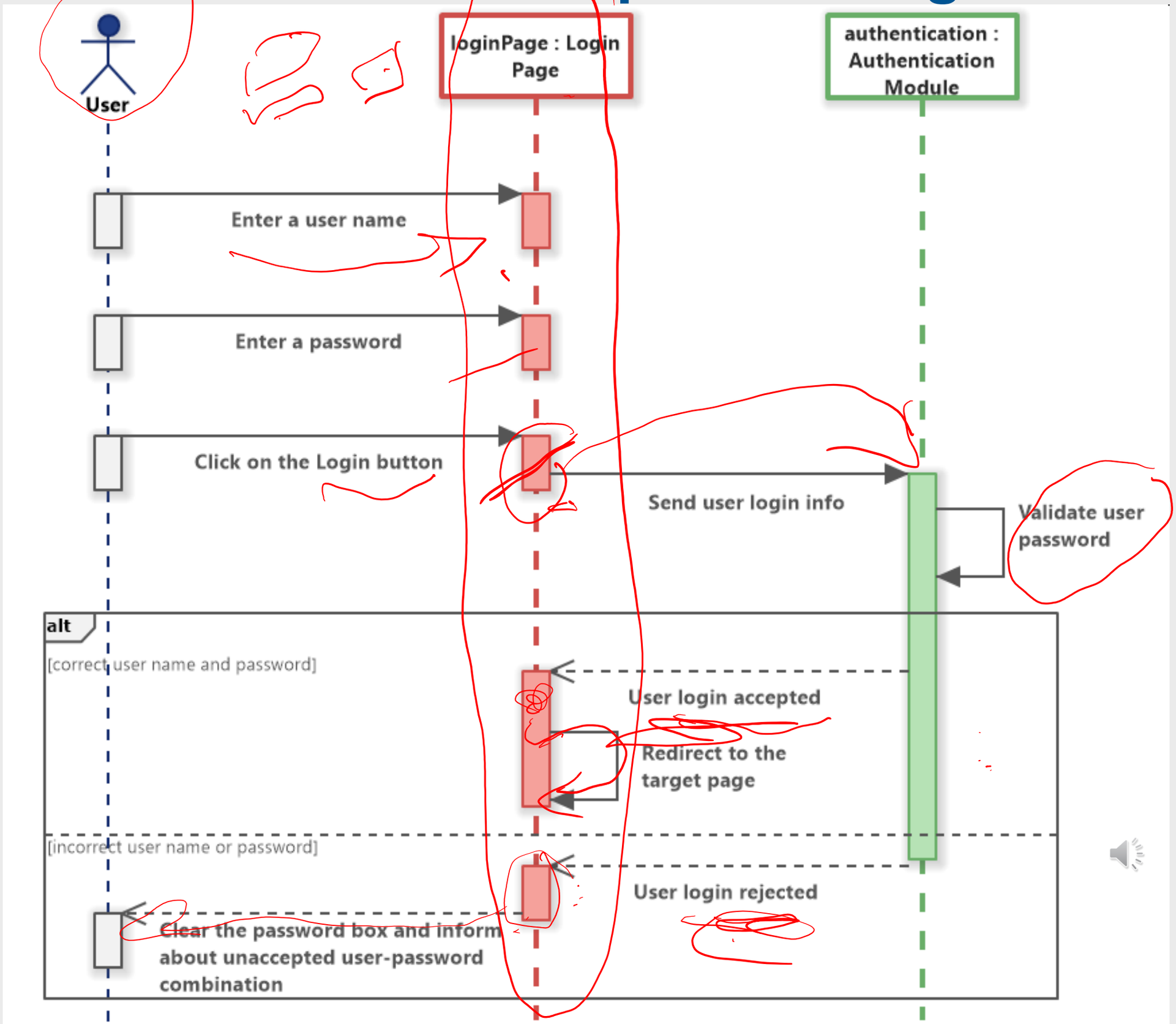
- When you are calling an external service, you should always check that the return code of the called service indicates that it has operated successfully.

- You should, also, if possible, check the validity of the result of the service call as you cannot be certain that the external service has carried out its computation correctly.

Software Diagrams

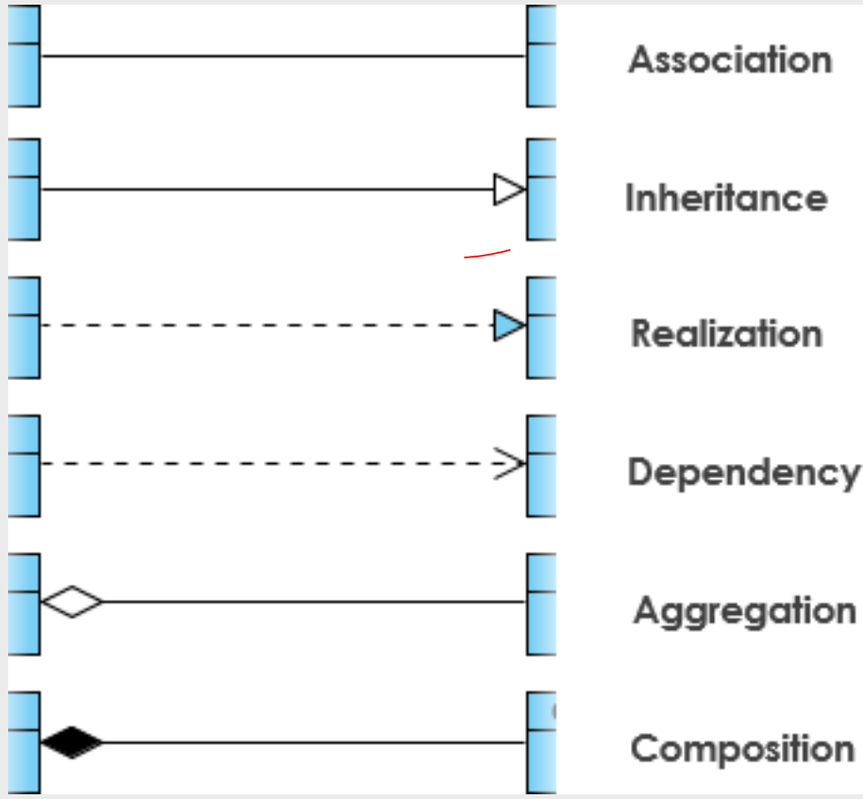
- Why?

Quick Formalities: Sequence Diagram

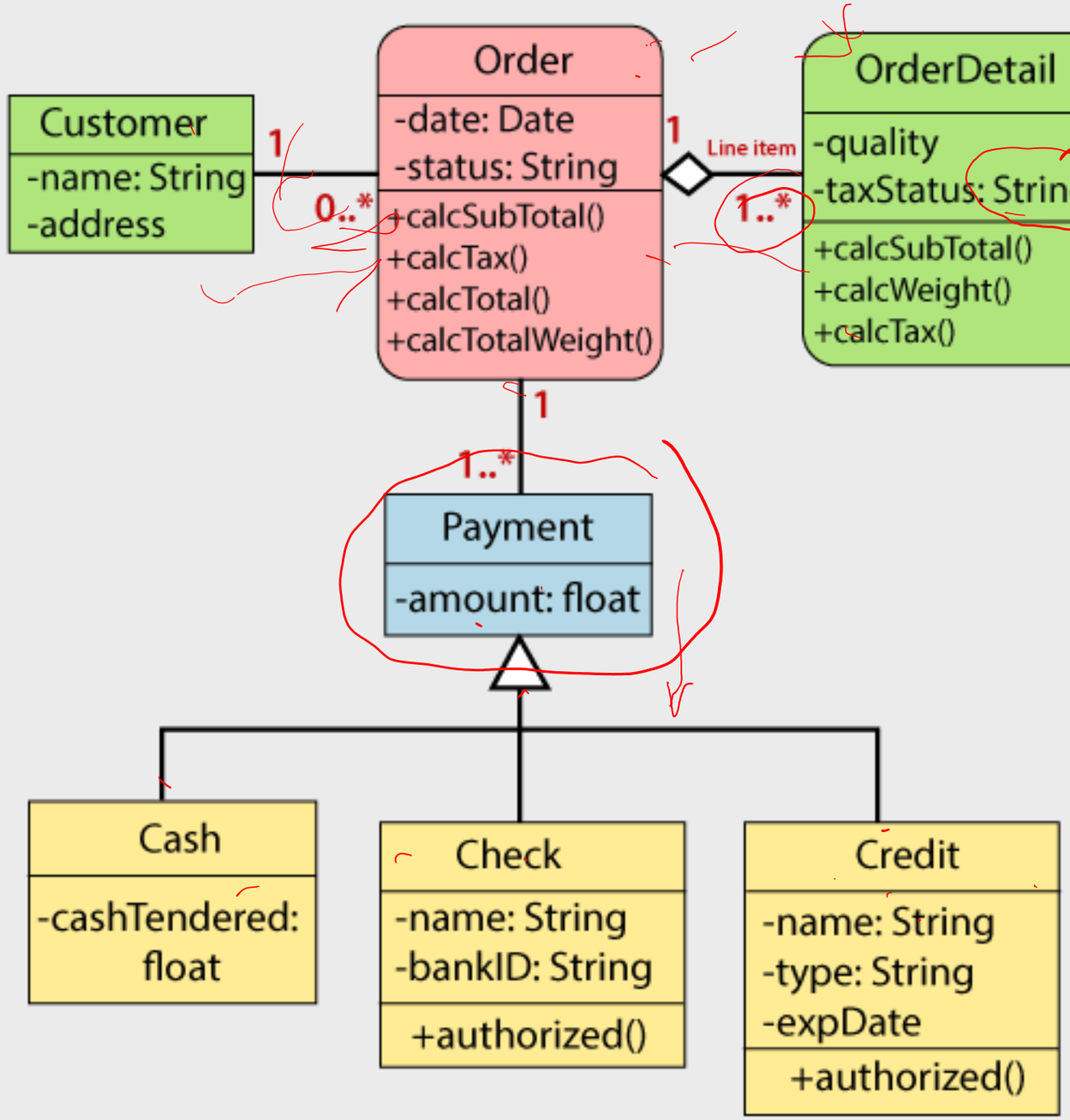


Quick Formalities: UML Class Diagram

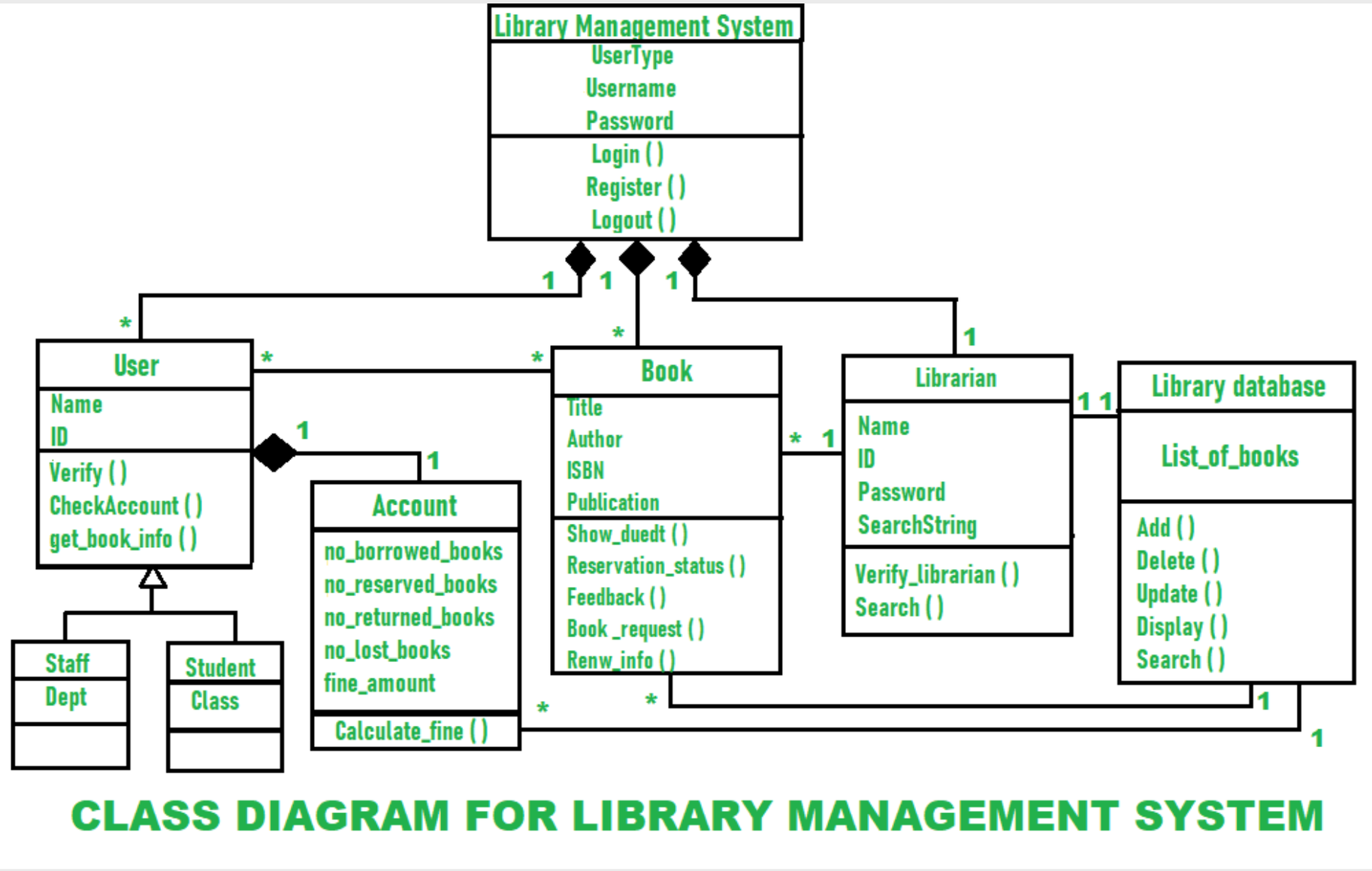
- Dependency: An object of one class might use an object of another class in the code of a method.
- Realization: A class implements an interface



Quick Formalities: UML Class Diagram

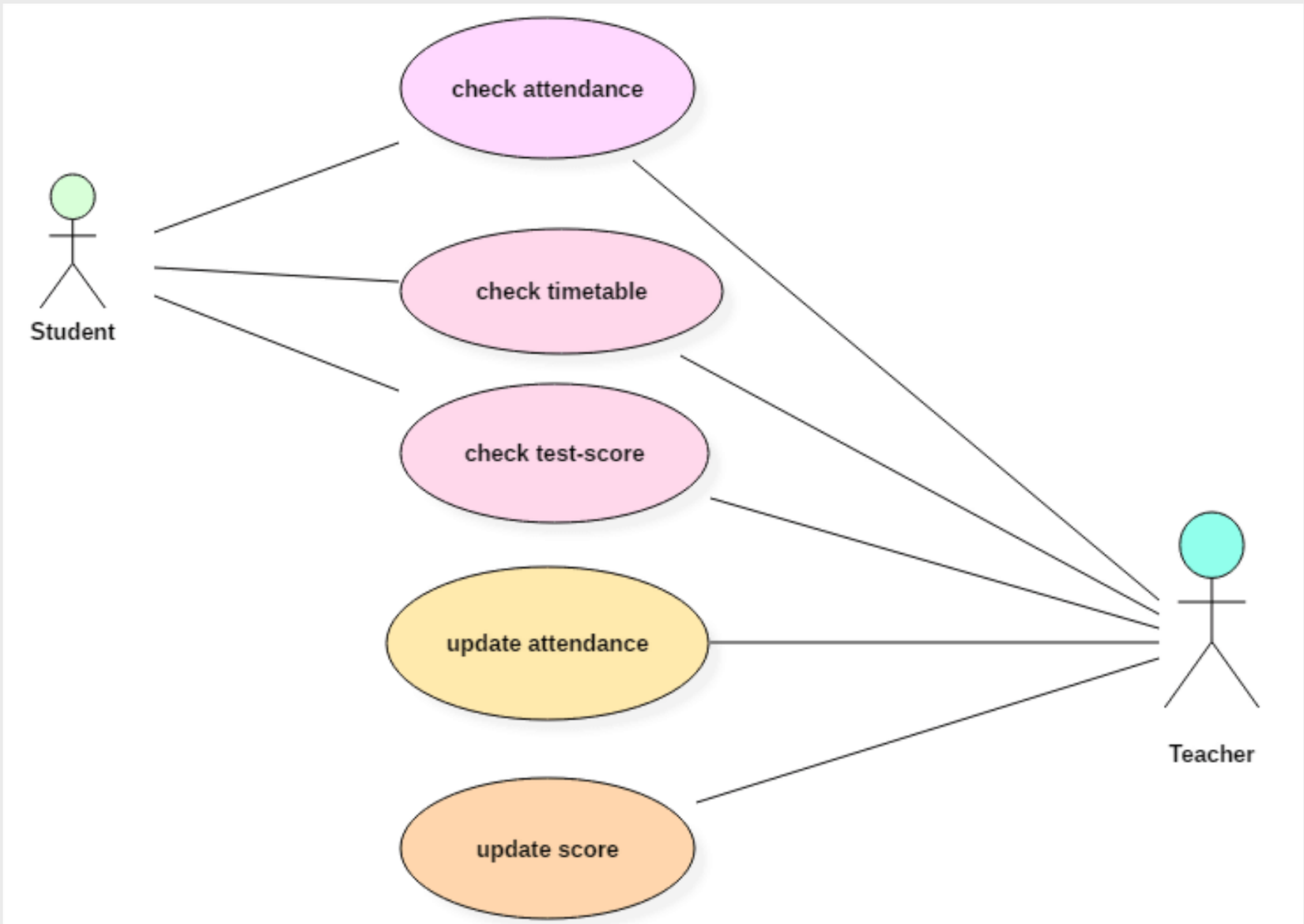


Quick Formalities: UML Class Diagram



Quick Formalities: Use Case Diagram

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams.



https://www.guru99.com/use-case-diagrams-example.html

Key points 1

- The most important quality attributes for most software products are reliability, security, availability, usability, responsiveness and maintainability.
- To avoid introducing faults into your program, you should use programming practices that reduce the probability that you will make mistakes.
- You should always aim to minimize complexity in your programs. Complexity makes programs harder to understand. It increases the chances of programmer errors and makes the program more difficult to change.
- Design patterns are tried and tested solutions to commonly occurring problems. Using patterns is an effective way of reducing program complexity.
- Refactoring is the process of reducing the complexity of an existing program without changing its functionality. It is good practice to refactor your program regularly to make it easier to read and understand.
- Input validation involves checking all user inputs to ensure that they are in the format that is expected by your program. Input validation helps avoid the introduction of malicious code into your system and traps user errors that can pollute your database.

Key points 2

- You should assume that your program may fail and to manage these failures so that they have minimal impact on the user.
- You should log user updates and maintain user data snapshots as your program executes. In the event of a failure, you can use these to recover the work that the user has done. You should also include ways of recognizing and recovering from external service failures.
- Diagrams: UML Class Diagram, Use-Case Diagram, Sequence Diagram