DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master

Option: Professional

Course:

Agile Information System 2 (AIS-2)

Realized By:

Dr. Ouaar Hanane

Academic year 2024-2025

Chapter 1:
Need for agility

Chapter outline

1. **Introduction**
2. **Object Modeling Technique (OMT) – Software Engineerin**
3. **Dynamic Programming (DP)**
    - **2.1 Top-Down DP (Memorization)**
    - **2.2 Bottom-Up DP (Tabulation)**
4. **Process models**
    - **4.1 Definitions**
    - **4.2 Waterfall Model**
    - **4.3 Agile Model**
        a. **Scrum**
        b. **Crystal**
        c. **Dynamic Software Development Method(DSDM)**
        d. **Feature Driven Development(FDD)**
        e. **Lean Software Development**
        f. **eXtreme Programming(XP)**
    - **4.4 Iterative model**
    - **4.5 Incremental Model**
    - **4.6 Prototype Model**
5. **Need for agility**
6. **Conclusion**

1. Introduction

We use software and apps every day to plan rideshares, order food, and play games. But have you considered the time, effort, and resources it takes to build software from start to finish? **Software**, even at its most basic level, is **complex**. Successful software developers must use project management frameworks like Agile to streamline (تبسيط) the entire process and create the perfect app.

The term *agility* has been thrown a lot lately, across various fields to describe the ability to adapt to changes effectively and proactively (بشكل استباقي). When it comes to the rapidly evolving technology sector, agility has proven to be a vital cornerstone for businesses striving (السعي) to maintain a competitive edge.

Agility in software development refers to a development team's capacity to quickly respond to evolving requirements and consistently (systématiquement) deliver top-notch (excellente) products within defined timelines.

This chapter covers the needs about agile software development, it most features and the involved steps in their processes

2. Object Modeling Technique (OMT) – Software Engineerin

Object Modeling Technique (OMT) is a real-world modeling approach for software modeling and designing that is easy to draw and use.

2.1 OMT definition

- It was developed basically as a method to develop object-oriented systems and to support object-oriented programming.
- It describes the static structure of the system.
- It is used in many applications like telecommunication, transportation, compilers etc.
- It is also used in many real-world problems.
- OMT is one of the most popular object-oriented development techniques used nowadays.

2.2 Purpose of Object Modeling Technique

- To test physical entities before construction of them.
- To make communication easier with the customers.
- To present information in an alternative way i.e., visualization.
- To reduce the complexity of software.
- To solve the real-world problems.

2.3 Object Modeling Technique's Models

There are three main types of models that have been proposed by OMT:

2.3.1 Object Model

Object Model encompasses (englobe) the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistence. Object Model basically emphasizes on the *object* and *class*. Main concepts related with Object Model are classes and their association with attributes. Predefined relationships in object model are aggregation and generalization (multiple inheritance).

2.3.2 Dynamic Model

Dynamic Model involves states, events and state diagram (transition diagram) on the model. Main concepts related with Dynamic Model are states, transition between states and events to trigger the transitions. Predefined relationships in object model are aggregation (concurrency) and generalization.

2.3.3 Functional Model

Functional Model focuses on how data is flowing, where data is stored and different processes. Main concepts involved in Functional Model are data, data flow, data store, process and actors. Functional Model in OMT describes the whole processes and actions with the help of data flow diagram (DFD).

2.3.4 Phases of Object Modeling Technique

OMT has the following phases:

1. Analysis

This is the first phase of the object modeling technique. This phase involves the preparation of precise and correct modelling of the real world problems. Analysis phase starts with setting a goal i.e. finding the problem statement. Problem statement is further divided into above discussed three models i.e. object, dynamic and functional model.

2. System Design

This is the second phase of the object modeling technique and it comes after the analysis phase. It determines all system architecture, concurrent tasks and data storage. High level architecture of the system is designed during this phase.

3. Object Design

Object design is the third phase of the object modelling technique and after system design is over, this phase comes. Object design phase is concerned with classification of objects into different classes and about attributes and necessary operations needed. Different issues related with generalization and aggregation are checked.

4. Implementation

This is the last phase of the object modeling technique. It is all about converting prepared design into the software. Design phase will translated into the Implementation phase.

3. Dynamic programming (DP)

3.1 Definition

Dynamic programming (often abbreviated as DP) is a method for solving complex problems by breaking them down into simpler, smaller, more manageable parts. The results are saved, and the subproblems are optimized to obtain the best solution.

DP is particularly useful for problems where the solution can be **expressed recursively** and the **same subproblem appears multiple times.**

DP stores the solution to each of these smaller parts to avoid doing the same work over and over again

3.2 Practical Application: The Fibonacci Sequence

To really get a grip (prendre le contrôle) on dynamic programming, let's explore a classic example: The Fibonacci sequence.

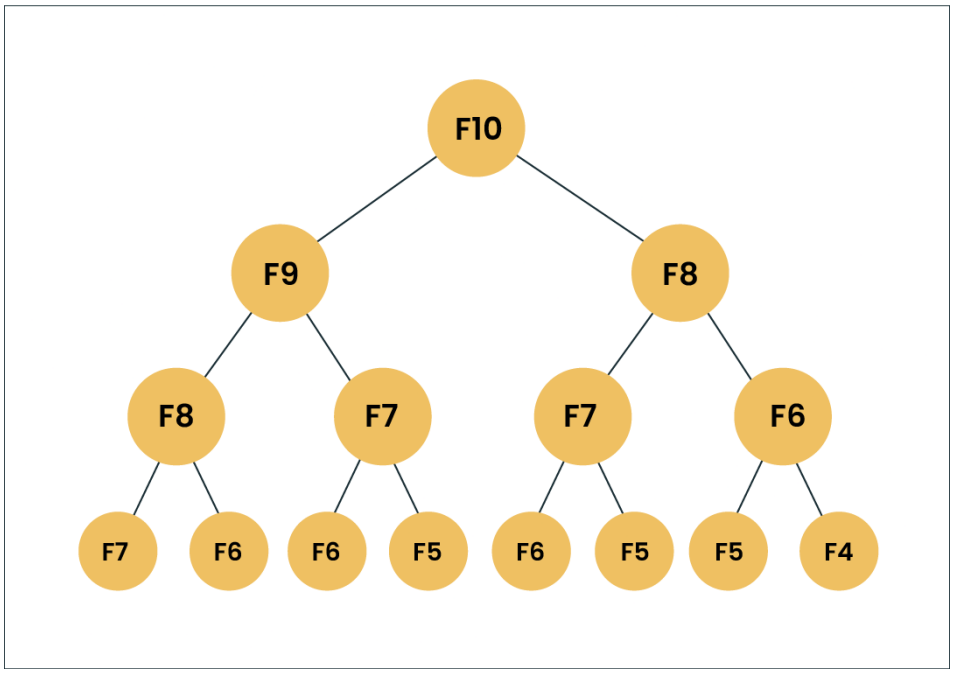
It is a series of numbers in which each number **is the sum of the two preceding ones**, usually starting with 0 and 1.

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…and so on.

Mathematically, we could write each term using the formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. And we'll follow the above relationship to calculate the other numbers. For example, F(6) is the sum of F(4) and F(5), which is equal to 8. Let's look at the diagram for better understanding.



Suppose we've to calculate F(10). Going by the formula, F(10) should be the sum of F(8) and F(9). Similarly, F(9) would also be the sum of the subproblems F(7) and F(8). As you can see, F(8) is an overlapping subproblem here.

In the above example, if we calculate the F(8) in the right subtree, then it would result in a increased usage of resources and reduce the overall performance.

The better solution would be to store the results of the already computed subproblems in an array. First, we'll solve F(6) and F(7) which will give us the solution to F(8) and we'll store that solution in an array and so on. Now when we calculate F(9), we already have the solutions to F(7) and F(8) stored in an array and we can just reuse them. F(10) can be solved using the solutions of F(8) and F(9), both of which are already stored in an array.

Similarly, at each iteration we store the solutions so we don't have to solve them again and again. This is the main attribute of dynamic programming.

If you try to compute this sequence with a straightforward (directe) recursive function, you'll end up doing a lot of unnecessary work.

Here's a simple Python implementation using DP to calculate a Fibonacci sequence:

```
def fibonacci(n):
    # Create an array of size (n+1) to store the computed values
    dp = [0, 1] + [0]*(n-1)
    for i in range(2, n + 1):
        # Compute the ith Fibonacci number
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
print(fibonacci(10))  # Outputs: 55
```

Step-by-Step Approach to DP

Let's explore how to implement dynamic programming step-by-step:

- Grasp (dirigée) the Problem
- Find the Overlapping (Chevauchement) Subproblems
- Compute and Store Solutions
- Construct the Solution to the Main Problem

3.3 Types of Dynamic Programming

Dynamic programming is divided into two main approaches: top-down (memorization) and bottom-up (tabulation). Both of these methods help in solving complex problems more efficiently by storing and reusing solutions of overlapping subproblems, but they differ in the way they go about it. Let's dive into these two approaches:

3.3.1 Top-Down DP (Memorization)

In the top-down approach, also known as memorization, we start with the original problem and break it down into subproblems. Think of it like starting at the top of a tree and working your way down to the leaves.

Here, problems are broken into smaller ones, and the answers are reused when needed. With every step, larger, more complex problems become tinier (plus petit), less complicated, and, thus, faster to solve, and the results of each subproblem are stored in a data structure like a dictionary or array to avoid recalculating them. The ‘memoization’ (a key technique in DP where you store and retrieve previously computed values) process is equivalent to adding the recursion (any function that calls itself again and again) and caching steps.

Some parts can be reused for the same problem and solved when requested, making them easier to debug. However, this approach results in more memory in the call **stack** being occupied, which can result in a reduction in overall performance and stack overflow.

Let's revisit the Fibonacci sequence example:

```
def fibonacci(n, memo = {}):
    if n <= 2:
        return 1
    elseif n in memo:
        return memo[n]
    else:
        memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
        return memo[n]
print(fibonacci(10))  # Outputs: 55
```

Here, memo is a dictionary that stores the previously computed numbers. Before we compute a new Fibonacci number, we first check if it's already in memo. If it is, we just return the stored value. If it's not, we compute it, store it in memo, and then return it.

3.3.2 Bottom-Up DP (Tabulation)

The bottom-up approach, also known as tabulation, takes the opposite direction. This approach solves problems by breaking them up into smaller ones, solving the problem with the smallest mathematical value, and then working up to the problem with the biggest value.

Solutions to its subproblems are compiled in a way that falls and loops back on itself. Users can opt to rewrite the problem by initially solving the smaller subproblems and then carrying those solutions for solving the larger subproblems.

Here, we fill up a table hence (ainsi) the name "tabulation" in a manner that uses the previously filled values in the table. This way, by the time we come to the problem at hand, we already have the solutions to the subproblems we need.

Let's use the Fibonacci sequence again to illustrate the bottom-up approach:

```
def fibonacci(n):
    fib_table = [0, 1] + [0]*(n-1)
    for i in range(2, n+1):
        fib_table[i] = fib_table[i-1] + fib_table[i-2]
    return fib_table[n]
print(fibonacci(10))  # Outputs: 55
```

In this case, fib_table is an array that stores the Fibonacci numbers in order. We start by filling in the first two numbers (0 and 1), and then we iteratively compute the rest from these initial numbers.

In contrast to the top-down approach, the bottom-up approach relies on eliminating recursion functions. There is no stack overflow, and memory space is saved with reduced timing

complexity, making it more efficient and preferred when the order of solving subproblems is not critical.

3.4 Criteria to choose an approach

Both top-down and bottom-up dynamic programming can be useful, and your choice depends on the problem at hand and the specific requirements of your program.

The top-down approach might be easier to understand because it follows the natural logic of the problem, but it can involve a lot of recursion and may have a larger memory footprint (empreinte) due to the call stack.

On the other hand, the bottom-up approach can be more efficient because it avoids recursion and uses a loop instead, but it might require a better understanding of the problem to build the solution iteratively.

4. Process models

4.1 Definitions

Process models is a visual representation that describes the operations and activities undertaken(تم التعهد به) by a firm. It provides insight (aperçu) into how the firm conducts its business, showcasing (présentation) the flow of processes and interactions within the organization. Its a visual representation that describes the operations and activities undertaken by a firm.

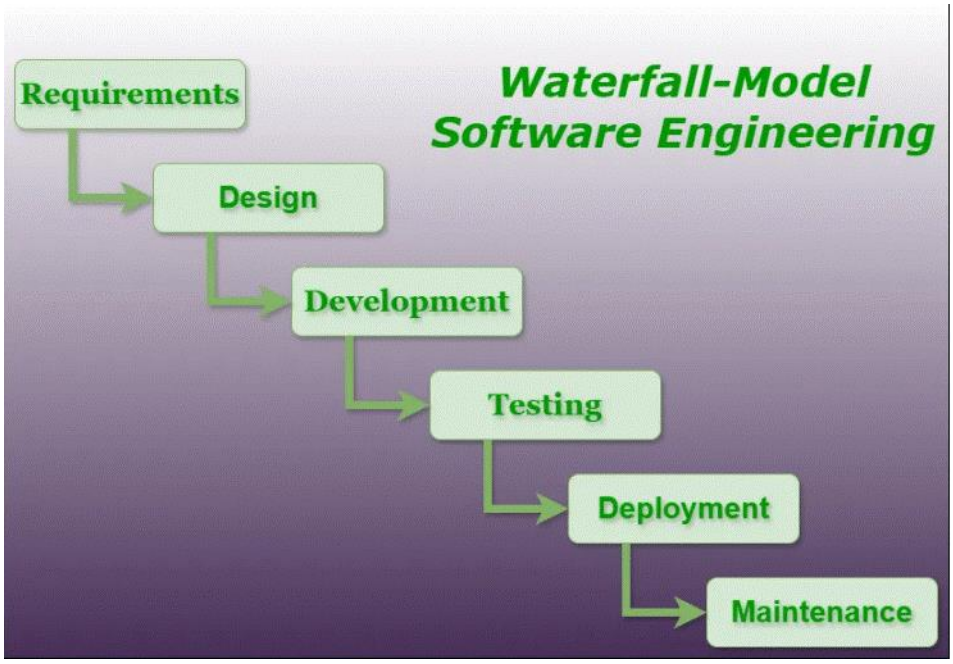
4.2 Waterfall Model

4.2.1 Principle

The classical waterfall model is the basic **software development life cycle** model (**SDLC**). It is very simple but idealistic. Earlier this model was very popular but nowadays it is not used. However, it is very important because all the other software development life cycle models are based on the classical waterfall model.

The waterfall model is a software development model used in the context of large, complex projects, typically in the field of information technology. It is characterized by a structured, sequential approach to project management and software development.

The waterfall model is useful in situations where the project requirements are **well-defined** and the project **goals are clear**. It is often used for large-scale projects with long timelines, where there is little room for error and the project stakeholders need to have a high level of confidence in the outcome.



Waterfall Model-Software Engineering

4.2.2 Importance of SDLC Waterfall Model

a. **Clarity and Simplicity:** The linear form of the Waterfall Model offers a simple and unambiguous foundation for project development.
b. **Clearly Defined Phases:** The Waterfall Model's phases each have unique inputs and outputs, guaranteeing a planned development with obvious (évidente) checkpoints.
c. **Documentation:** A focus on thorough documentation helps with software comprehension, upkeep (entretien), and future growth.
d. **Stability in Requirements:** Suitable for projects when the requirements are clear and steady, reducing modifications as the project progresses.
e. **Resource Optimization:** It encourages effective task-focused work without continuously changing contexts by allocating resources according to project phases.
f. **Relevance for Small Projects:** Economical for modest projects with simple specifications and minimal complexity.

4.2.3 Advantages of the SDLC Waterfall Model

The classical waterfall model is an idealistic model for software development. It is very simple, so it can be considered the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model.

- **Easy to Understand:** The Classical Waterfall Model is very simple and easy to understand.
- **Individual Processing:** Phases in the Classical Waterfall model are processed one at a time.
- **Properly Defined:** In the classical waterfall model, each stage in the model is clearly defined.
- **Clear Milestones:** The classical Waterfall model has very clear and well-understood milestones.
- **Properly Documented:** Processes, actions, and results are very well documented.
- **Reinforces Good Habits:** The Classical Waterfall Model reinforces good habits like define-before-design and design-before-code.
- **Working:** Classical Waterfall Model works well for smaller projects and projects where requirements are well understood.

4.2.4 Disadvantages of the SDLC Waterfall Model

The Classical Waterfall Model suffers from various shortcomings we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model. Below are some major drawbacks (Inconvénients) of this model.

- **No Feedback Path:** In the classical waterfall model evolution of software from one phase to another phase is like a waterfall. It assumes that no error is ever committed by developers during any phase. Therefore, it does not incorporate (intégrer) any mechanism for error correction.
- **Difficult to accommodate (يستوعب) Change Requests:** This model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but the customer's requirements keep on changing with time. It is difficult to accommodate any change requests after the requirements specification phase is complete.
- **No Overlapping (Chevauchement) of Phases:** This model recommends that a new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase efficiency and reduce cost, phases may overlap.
- **Limited Flexibility:** The Waterfall Model is a rigid and linear approach to software development, which means that it is not well-suited for projects with changing or uncertain requirements. Once a phase has been completed, it is difficult to make changes or go back to a previous phase.

- **Limited Stakeholder Involvement:** The Waterfall Model is a structured and sequential approach, which means that stakeholders are typically involved (impliqué) in the early phases of the project (requirements gathering and analysis) but may not be involved in the later phases (implementation, testing, and deployment).
- **Late Defect Detection:** In the Waterfall Model, testing is typically done toward the end of the development process. This means that defects may not be discovered until late in the development process, which can be expensive and time-consuming to fix.
- **Lengthy Development Cycle:** The Waterfall Model can result in a lengthy development cycle, as each phase must be completed before moving on to the next. This can result in delays and increased costs if requirements change or new issues arise.

4.2.5 Applications of SDLC Waterfall Model

- **Large-scale Software Development Projects:** The Waterfall Model is often used for large-scale software development projects, where a structured and sequential approach is necessary to ensure that the project is completed on time and within budget.
- **Safety-Critical Systems:** The Waterfall Model is often used in the development of safety-critical systems, such as aerospace or medical systems, where the consequences of errors or defects can be severe.
- **Government and Defense Projects:** The Waterfall Model is also commonly used in government and defense projects, where a rigorous and structured approach is necessary to ensure that the project meets all requirements and is delivered on time.
- **Projects with well-defined Requirements:** The Waterfall Model is best suited for projects with well-defined requirements, as the sequential nature of the model requires a clear understanding of the project objectives and scope.
- **Projects with Stable Requirements:** The Waterfall Model is also well-suited for projects with stable requirements, as the linear nature of the model does not allow for changes to be made once a phase has been completed.

4.3 Agile Model

The meaning of Agile is swift or versatile. "**Agile process model**" refers to a software development approach based on **iterative** development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down (وضعت) at the beginning of the development process. Plans regarding the **number of iterations**, the **duration** and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

The most useful agile Methods are:

- Scrum
- Crystal
- Dynamic Software Development Method(DSDM)
- Feature Driven Development(FDD)
- Lean Software Development
- eXtreme Programming(XP)

4.3.1 Scrum

SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.

There are three roles in it, and their responsibilities are:

- **Scrum Master:** The scrum can set up the master team, arrange the meeting and remove obstacles for the process
- **Product owner:** The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
- **Scrum Team:** The team manages its work and organizes the work to complete the sprint or cycle.

4.3.2 eXtreme Programming(XP)

This type of methodology is used when customers are constantly changing demands or requirements, or when they are not sure about the system's performance.

The XP framework normally involves 5 phases or stages of the development process that iterate continuously:

- **Planning,** the first stage, is when the customer meets the development team and presents the requirements in the form of **user stories** to describe the desired result. The team then **estimates** the stories and creates a release plan broken down into iterations needed to cover the required functionality part after part. If one or more of the stories can’t be estimated, so-called *spikes* (pointes ,المسامير) can be introduced which means that further research is needed.

- **Designing** is actually a part of the planning process, but can be set apart to emphasize its importance. It's related to one of the main XP values that we'll discuss below -- simplicity. A good design brings (apporte) logic and structure to the system and allows to avoid (éviter) unnecessary complexities and redundancies.
- **Coding** is the phase during which the actual code is created by implementing specific XP practices such as coding standards, pair programming, continuous integration, and collective code ownership (الملكية).
- **Testing** is the core of extreme programming. It is the regular activity that involves both unit tests (automated testing to determine if the developed feature works properly) and acceptance tests (customer testing to verify that the overall system is created according to the initial requirements).
- **Listening** is all about constant communication and feedback. The customers and project managers are involved to describe the business logic and value that is expected.

a. XP guidelines values

XP has simple rules that are based on 5 values to guide the teamwork:

- **Communication.** Everyone on a team works jointly (بشكل مشترك) at every stage of the project.
- **Simplicity.** Developers strive (يجتهد) to write simple code bringing more value to a product, as it saves time and effort.
- **Feedback.** Team members deliver software frequently, get feedback about it, and improve a product according to the new requirements.
- **Respect.** Every person assigned to a project contributes to a common goal.
- **Courage.** Programmers objectively evaluate their own results without making excuses and are always ready to respond to changes.

These values represent a specific mindset of motivated team players who do their best on the way to achieving a common goal. XP principles derive from these values and reflect them in more concrete ways.

b. Principles of extreme programming

Most researchers denote 5 XP principles as:

- **Rapid feedback.** Team members understand the given feedback and react to it right away.
- **Assumed simplicity.** Developers need to focus on the job that is important at the moment and follow **YAGNI** (You Aren't Gonna Need It), is a practice in software development which states that features should only be added when required. and **DRY**

(Don’t Repeat Yourself) principles, is a principle of software development aimed at reducing repetition of information which is likely to change.

- **Incremental changes.** Small changes made to a product step by step work better than big ones made at once.
- **Embracing change.** If a client thinks a product needs to be changed, programmers should support this decision and plan how to implement new requirements.
- **Quality work.** A team that works well, makes a valuable product and feels proud (fière) of it.

4.3.3 Crystal:

The Crystal agile framework is designed for software development. It emphasizes people rather than procedures, empowering teams to develop their solutions for each project rather than being constrained by pre-determined methodologies. (permettre aux équipes de développer leurs solutions pour chaque projet plutôt que d'être limitées par des méthodologies prédéterminées)

a. Principle:

At the heart of the crystal, a family is seven principles. The first three are compulsory (obligatoire) for all crystal approaches, but the rest are optional and can be adopted if appropriate:

- **Frequent delivery :**

You should deliver code regularly to your real users. Without this, you might be building a product nobody needs.

- **Reflective improvement**

Look back on what you've done, how you've done it, and why. As a team, reflect and decide how to improve it in the future.

- **Osmotic communication**

Co-location (having teams in the same physical space) is critical as it allows information to flow between team members, as if by osmosis (التناضح أو الخاصية الأسموزية).

- **Personal safety**

Team members should feel safe to discuss ideas openly, without fear of ridicule (سخرية). There are no wrong answers or bad suggestions in a crystal team.

- **Focus on work**

Team members should know what to work on next and be able to do it. This requires clear communication and documentation when required.

- **Access to subject matter experts and users**

Team members should be able to get feedback from real users and experts when required.

- **Technical tooling**

Development teams should have access to toolings (الأدوات) like continuous deployment, automated testing, and configuration management. This means errors and mistakes can be caught quickly without human intervention.

b. Crystal's strengths include:

- Allows teams to work the way they deem (considérer) most effective
- Facilitates direct team communication, transparency and accountability
- The adaptive approach lets teams respond well to changing requirements

c. Crystal's weaknesses include:

- Lack of pre-defined plans can lead to scope creep
- Lack of documentation can lead to confusion

4.3.4 Dynamic Software Development Method (DSDM):

DSDM is a rapid application development strategy for software development and gives an agile project distribution structure. The essential features of DSDM are that users must be actively connected, and teams have been given the right to make decisions. The techniques used in DSDM are:

1. Time Boxing
2. MoSCoW Rules
3. Prototyping

MoSCoW prioritization, also known as the MoSCoW method or MoSCoW analysis, is a popular prioritization technique for managing requirements provide the best return on investment (ROI) (العائد على الاستثمار).

The acronym MoSCoW represents four categories of initiatives: must-have, should-have, could-have, and won't-have, or will not have right now. Some companies also use the "W" in MoSCoW to mean "wish."

a. The DSDM project contains seven stages:

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation

7. Post-project

4.3.5 Feature Driven Development (FDD):

This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.

FDD is customer-centric, iterative, and incremental, with the goal of delivering tangible (ملموس) software results often and efficiently. FDD in Agile encourages status reporting at all levels, which helps to track progress and results.

FDD allows teams to update the project regularly and identify errors quickly. Plus, clients can be provided with information and substantial results at any time. FDD is a favorite method among development teams because it helps reduce two known morale-killers in the development world: Confusion (التباس) and rework.

Typically used in large-scale development projects, five basic activities exist during FDD:

- Develop overall model
- Build feature list
- Plan by feature
- Design by feature
- Build by feature

An overall model shape is formed during the first two steps, while the final three are repeated for each feature. The majority (roughly (تقريبا) 75%) of effort during FDD will be spent on the fourth and fifth steps – Design by Feature and Build by Feature.

4.3.6 Lean Software Development:

Lean software development methodology follows the principle "just in time production." The lean method indicates the increasing speed of software development and reducing costs. Lean development can be summarized in seven phases.

1. Eliminating Waste (Déchets)
2. Amplifying (التضخيم) learning
3. Defer commitment (تأجيل الالتزام , Différer l’engagement) (deciding as late as possible)
4. Early delivery
5. Empowering the team
6. Building Integrity
7. Optimize the whole (جميع)

4.3.7 Use the Agile Model

- When frequent changes are required.

- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

4.3.8 Advantage of Agile Method:

1. Frequent Delivery
2. Face-to-Face Communication with clients.
3. Efficient design and fulfils (accomplit) the business requirement.
4. Anytime changes are acceptable.
5. It reduces total development time.

4.3.9 Disadvantages of Agile Model:

1. Due to the shortage (نقص) of formal documents, it creates confusion and crucial decisions taken throughout (tout au long de) various phases can be misinterpreted (mal interprété) at any time by different team members.
2. Due to the lack of proper documentation, once the project completes and the developers allotted (attribué) to another project, maintenance of the finished project can become a difficulty.

4.4 Iterative Model

In this Model, you can start with some of the software specifications and develop the first version of the software. After the first version if there is a need to change the software, then a new version of the software is created with a new iteration. Every release of the Iterative Model finishes in an exact and fixed period that is called iteration.

The Iterative Model allows the accessing earlier phases, in which the variations made respectively. The final output of the project renewed (renouvelée) at the end of the Software Development.

4.4.1 Iterative model life cycle

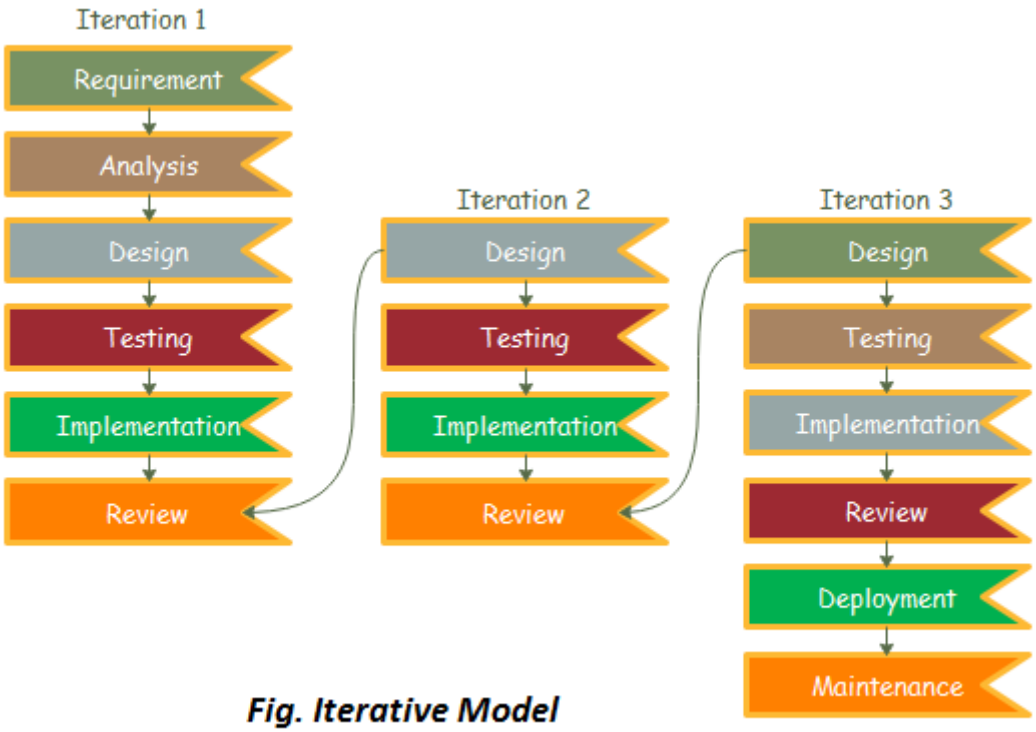


Fig. Iterative Model

The various phases of Iterative model are as follows:

- **Requirement gathering & analysis:** In this phase, requirements are gathered from customers and check by an analyst whether requirements will fulfil (remplir) or not. Analyst checks that need will achieve within budget or not. After all of this, the software team skips to the next phase.
- **Design:** In the design phase, team design the software by the different diagrams like Data Flow diagram, activity diagram, class diagram, state transition diagram, etc.
- **Implementation:** In the implementation, requirements are written in the coding language and transformed into computer programmes which are called Software.
- **Testing:** After completing the coding phase, software testing starts using different test methods. There are many test methods, but the most common are white box, black box, and grey box test methods.
- **Deployment:** After completing all the phases, software is deployed to its work environment.
- **Review:** In this phase, after the product deployment, review phase is performed to check the behaviour and validity of the developed product. And if there are any error found then the process starts again from the requirement gathering.

- **Maintenance:** In the maintenance phase, after deployment of the software in the working environment there may be some bugs, some errors or new updates are required. Maintenance involves debugging and new addition options.

Use the Iterative Model

- When requirements are defined clearly and easy to understand.
- When the software application is large.
- When there is a requirement of changes in future.

4.4.2 Advantage of Iterative Model

- Testing and debugging during smaller iteration is easy.
- A Parallel development can plan.
- It is easily acceptable to ever-changing needs of the project.
- Risks are identified and resolved during iteration.
- Limited time spent on documentation and extra time on designing.

4.4.3 Disadvantage of Iterative Model

- It is not suitable for smaller projects.
- More Resources may be required.
- Design can be changed again and again because of imperfect requirements.
- Requirement changes can cause over budget.
- Project completion (انتهاء) date not confirmed because of changing requirements.

4.5 Incremental Model

Incremental Model is a process of software development where requirements divided into multiple standalone (autonome) modules of the software development cycle. In this model, each module goes through the requirements, design, implementation and testing phases. Every subsequent release of the module adds function to the previous release. The process continues until the complete system achieved.

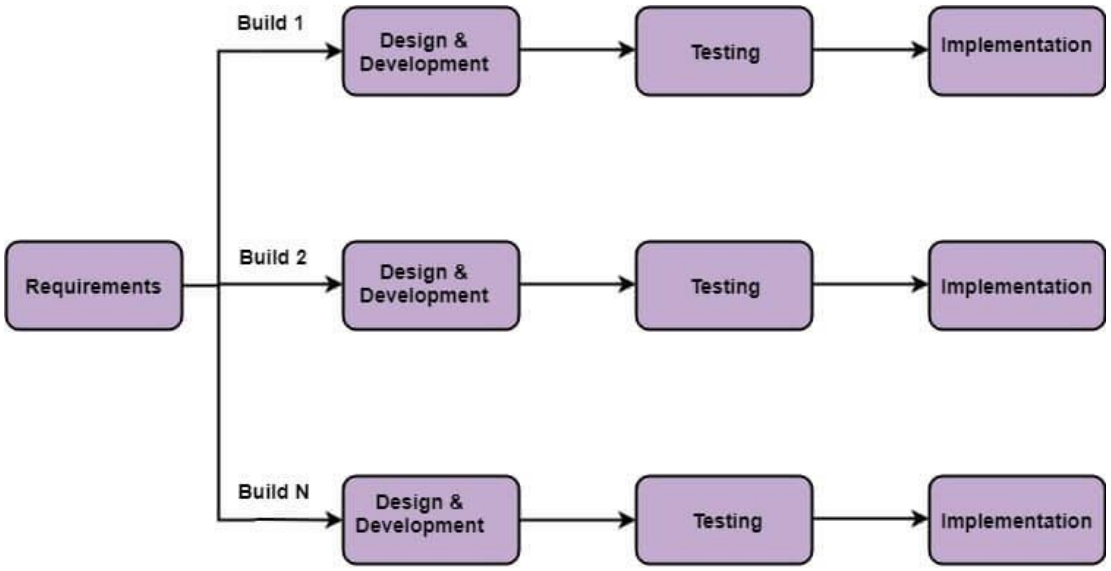


Fig: Incremental Model

4.5.1 Phases of incremental model

- **Requirement analysis:** In the first phase of the incremental model, the product analysis expertise identifies the requirements. And the system functional requirements are understood by the requirement analysis team. To develop the software under the incremental model, this phase performs a crucial role.
- **Design & Development:** In this phase of the Incremental model of SDLC, the design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.
- **Testing:** In the incremental model, the testing phase checks the performance of each existing function as well as additional functionality. In the testing phase, the various methods are used to test the behavior of each task.
- **Implementation:** Implementation phase enables the coding phase of the development system. It involves the final coding that design in the designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up (mis à niveau) to the final system product

4.5.2 Use the Incremental Model

- When the requirements are superior.
- A project has a lengthy development schedule (calendrier).
- When Software team are not very well skilled or trained.
- When the customer demands a quick release of the product.

- You can develop prioritized requirements first. (Vous pouvez d'abord développer des exigences prioritaires.)

4.5.3 Advantage of Incremental Model

- Errors are easy to be recognized.
- Easier to test and debug
- More flexible.
- Simple to manage risk because it handled during its iteration.
- The client gets important functionality early.

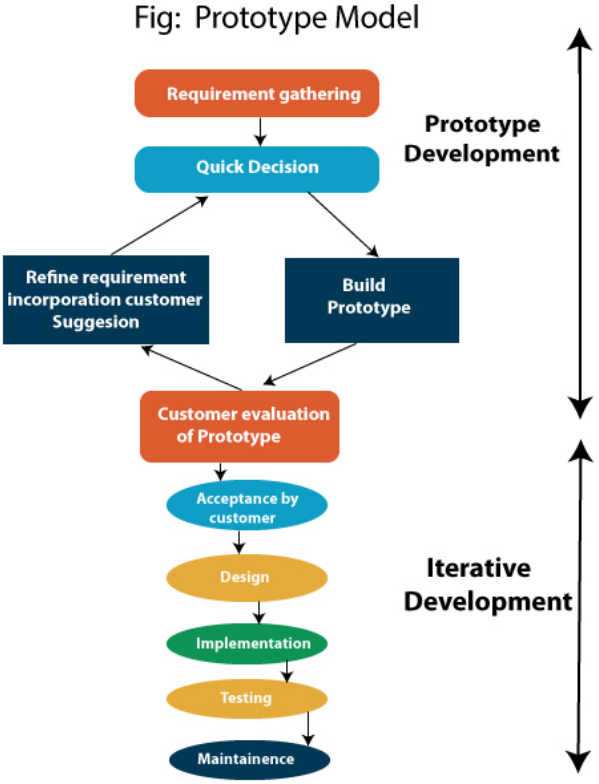
4.5.4 Disadvantage of Incremental Model

- Need for good planning
- Total cost is high.
- Well defined module interfaces are needed.

4.6 Prototype Model

The prototype model requires that before carrying out (تنفيذ) the development of actual software, a working prototype of the system should be built. A prototype is a toy (jouet) implementation of the system. A prototype usually turns out (تضح) to be a very crude (brute) version of the actual system, possible exhibiting (exposant) limited functional capabilities, low reliability(fiabilité), and inefficient performance as compared to actual software. In many instances, the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs, and the output requirement, the prototyping model may be employed.

Fig: Prototype Model



4.6.1 Steps of Prototype Model

- Requirement Gathering and Analyst
- Quick Decision
- Build a Prototype
- Assessment (judgment) or User Evaluation
- Prototype Refinement
- Engineer Product

4.6.2 Advantage of Prototype Model

- Reduce the risk of incorrect user requirement
- Good where requirement are changing/uncommitted (non engage)
- Regular visible process aids management (Gestion régulière et visible des aides au processus)
- Support early product marketing
- Reduce Maintenance cost.
- Errors can be detected much earlier as the system is made side by side.

4.6.3 Disadvantage of Prototype Model

- An unstable/badly implemented prototype often becomes the final product.
- Require extensive (شاسِع) customer collaboration
  - Costs customer money

- Needs committed (ملتزم) customer
- Difficult to finish if customer withdraw (retirer)
- May be too customer specific, no broad (large) market

- Difficult to know how long the project will last (durera).
- Easy to fall back (retomber) into the code and fix without proper requirement analysis, design, customer evaluation, and feedback.
- Prototyping tools are expensive.
- Special tools & techniques are required to build a prototype.
- It is a time-consuming (prend du temps) process.

5. Need for agility

The need of agility it has argued by several raisons as:

5.1 Adaptability to market changes: Business agility helps companies stay relevant (pertinente, مناسب) and competitive by allowing them to respond quickly to evolving market changes and customer demands.

For instance, think about the rise of online shopping – businesses that quickly adapted by offering e-commerce options gained an edge (a gagné un avantage) over those that didn't. Being agile in the face of change allows companies to spot trends, take emerging opportunities, and keep their customers happy.

5.2 Competitive advantage: Companies gain a competitive advantage through business agility, enabling them to deliver products and services faster, snag (تعلق) new opportunities, and effectively respond to customer feedback. With competition fiercer (أكثر شراسة) than ever, staying agile and ahead (devant, إلى الأمام) of the curve (courbe) means the difference between thriving (مزدهر) and fading (يتلاشى) into the background. Companies like Netflix and Amazon are prime illustrations of how agility can contribute to staying ahead in the market. Netflix swiftly transitioned from DVD rentals (locations) to online streaming services, becoming a major disruptor in the entertainment industry. Similarly, Amazon adapted to changing consumer behavior and expanded its portfolio to offer products, services, and platforms like Amazon Prime, Kindle, and Amazon Web Services (AWS).

5.3 Customer-centric approach: Adopting a customer-centric approach through business agility is essential for companies seeking (السعي) long-term success. Companies can foster a strong connection with the target audience by emphasizing the importance of understanding customer needs, continuously engaging with customers, and delivering incremental value.

This drive for customer satisfaction and loyalty means businesses can gather better insights (connaissances) into customer preferences (التفضيلات) – ultimately leading to a stronger, more sustainable market presence.

5.4 Enhanced innovation and experimentation: When it comes to business agility, innovation and experimentation go hand in hand. An agile mindset (عقلية) means you're open to new ideas and willing (راغب) to test different possibilities. This trial-and-error approach allows companies to find the best business solutions to improve their offerings (العروض).

Just think about all those famous tech companies – we're talking about the likes of Apple and Tesla. They didn't just stick (لصق) to the status quo (الوضع الراهن); they embraced their agility and kept innovating, resulting in groundbreaking (رائد, innovant) products and services.

5.5 Improved risk management: Companies that adopt business agility become better at figuring out potential problems (تصبح أفضل في اكتشاف المشكلة المحتملة) before they get out of hand. They can switch gears (engrenages), tweak their strategies (تعديل استراتيجياتهم), and tackle risks head-on (التعامل مع المخاطر بشكل مباشر). Things change fast – with market fluctuations (التقلبات), new competitors, or even global events – and agile companies stay ready for whatever life throws (lance) at them.

5.6 Employee satisfaction and retention (حفظ): Happy employees are more likely to stick around, and business agility plays a big role in keeping everyone content. Agility goes hand in hand with a trusting, autonomous, and empowering work environment. Employees who feel like they're part of a company that's moving forward and adapting to change are more likely to feel satisfied, motivated, and valued. Agile companies also tend to offer great opportunities for skill development – helping employees grow and flourish (ازدهر) in their careers (carrières).

6. Conclusion

In the fast-paced field of software development, agility is the ultimate game-changer (le changement de jeu ultime), propelling (الدفع) teams toward success and ensuring they stay ahead of the curve. It's the secret sauce that transforms good teams into great ones!

Agile development is important because it helps to ensure that development teams complete projects on time and within budget. It also helps to improve communication between the development team and the product owner. Additionally, Agile development methodology can help reduce the risks associated with complex projects. It allows for development teams to make changes quickly and easily without affecting the overall project timeline.

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module: SIA-2**
Teacher: Ouaar Hanane **Year: 2024/2025**

Hands-out N° 1: Object-oriented approach vs Process-oriented approach

Objective 1:

The main goal of this study is to find the most important differences between the object-oriented approach vs the process-oriented approach.

Details

ATMs, or automated teller machines, are banking outlets where you can withdraw cash without going into a branch of their bank. Some ATMs only dispense cash, while others allow transactions such as check deposits or balance transfers**.** Thus, a customer carrying a card must enter his secret code before having access to the attached account. ATMs provides two main business process:

*** Withdraw money** and *** Deposit money**

Tasks 1:Object-oriented approach

- Elaborate the : use case, class and activity diagrams

Tasks 2: Process-oriented approach

- Using **Camunda** as a software tool, draw the previous case study in PBMN modal.
- Study the parsing rules (actors, use cases, static and dynamic behaviors) as below:

	Object-oriented approach	**Process-oriented approach**
Actors		
Use cases		
Static behaviors		
Dynamic behaviors		

Objective 2: Dynamic Programing (DP)

- Select a recursive problem.
- Write tow programs to applying the two main approaches: top-down (memorization) and bottom-up (tabulation).
- Display the important screen shoots.
- Discus your opinion about the both codes in the conclusion.
- Edit the best (pdf) report that resumes the study contents.

Chapter 2:

Power of agility

Chapter outline

Introduction

1. **Unit Testing**
2. **Top 4 Unit Testing Tools**
   - **2.1 JUnit**
   - **2.2 NUnit**
   - **2.3 TestNG**
   - **2.4 PHPUnit**

3 Unit Testing Best Practices

4 Test-Driven Development (TDD) and Unit Testing

5 Agile Code Refactoring Techniques

- **6.1 Composing Method**
- **6.2 Red-Green Refactoring**
- **6.3 Refactoring by Abstraction**
- **6.4 Preparatory Refactoring**
- **6.5 Troubleshoot and Debug Separately**
- **6.6 Prioritize Code Deduplication**

7 Technical debt

8 Version control system

- **5.1 Distributed**
- **5.2 Centralized**
- **5.3 Lock-based**

9 Presentation of the best versions control system:

- **9.1 GIT**

9.2 Git Hub

9.3 CVS

9.4 GitLab

9.5 BitBucket

9.6 SVN

Conclusion

1. Introduction

Agility in software development refers to a development team's capacity to quickly respond to evolving requirements and consistently deliver top-notch products within defined timelines. The agile methodology emphasizes key principles such as flexibility, collaboration, and continuous improvement among development teams.

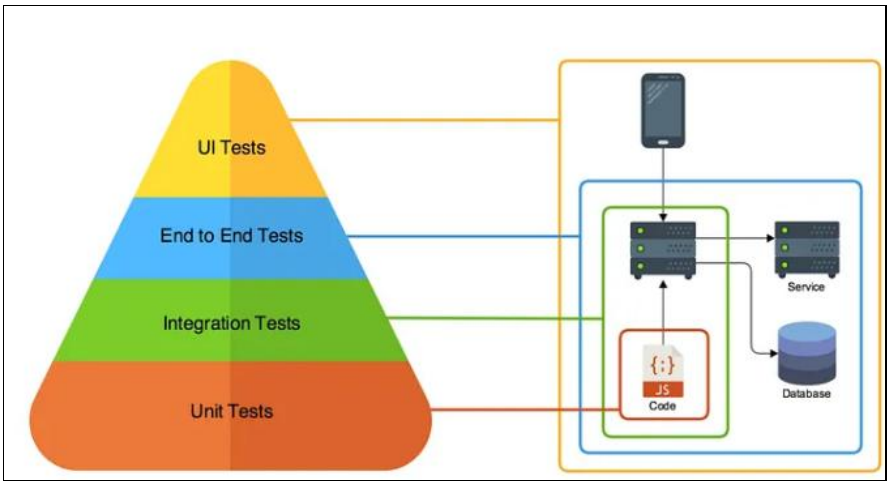
Given its dynamic nature, the **software development** industry, in particular, seems to be in a prime position to reap major benefits from the continuous advancements in agility. But how does software development actually tap into the power of agility and make the most of it?

2. Unit Testing

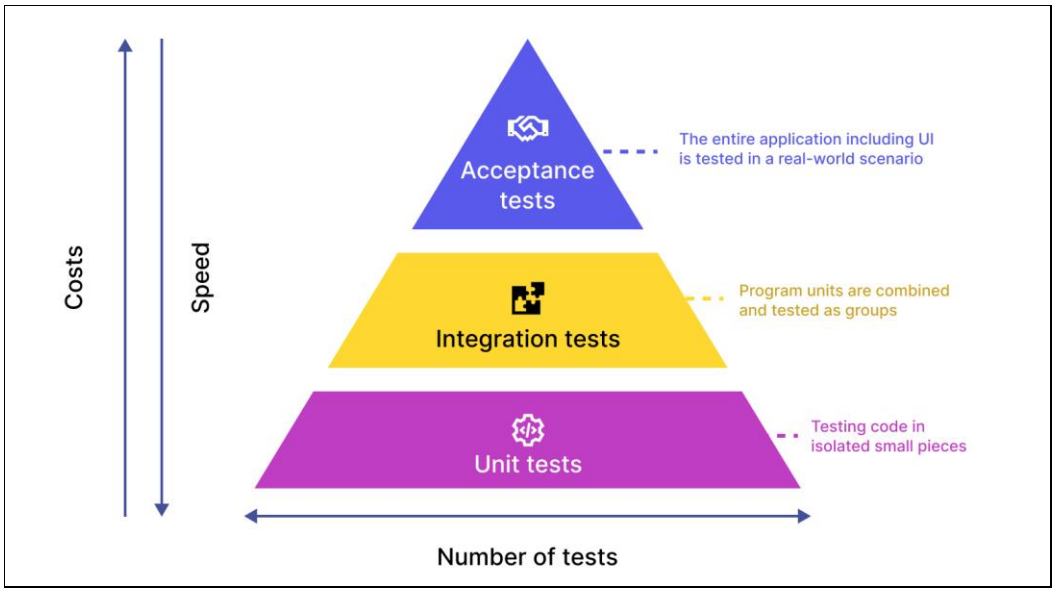
2.1 Definition

Unit testing is a software development practice that involves testing individual **units** or **components** of a software application in **isolation**. A unit is the **smallest** testable part of an application, usually a single function, method, procedure, module, or class.

Together these code units form a complete application, and if they don't work well individually, they definitely won't work well together. Unit testing ensures that each component of the software works correctly on its own before integrating it into the larger system.



Unit testing is usually the very first level of testing, done before integration testing. The **number** of tests to perform in each cycle is **huge** (very big), but the time it takes for each test is insignificant as these code units are relatively **simple**. Because of this, developers can **quickly** perform unit testing themselves.



In certain **teams**, developers don't want to allocate their limited bandwidth to do unit testing so that they can focus entirely on development. In these cases, **QA engineers (Quality Assurance)** will take over unit testing and integrate this activity into their test plan, leveraging (maximum) the existing testing tools to better execute and manage test results.

2.2 Purpose of Unit Testing

Unit testing is crucial to the **software testing** process for several reasons:

- Early bug detection
- Better code writing
- Simplifies debugging
- Provides documentation

2.3 Characteristics of a Good Unit Test

Unit tests are generally:

- **Fast:** These tests only check very simple and limited-in-scope units, so they can be executed in milliseconds. A mature project can have up to thousands of unit tests.
- **Isolated:** They should be executed in isolation from external dependencies to ensure the most accurate (دقيق) results.
- **Easily automated:** Due to their simple nature, unit tests are perfect candidates for **automated testing**. Developers can employ **leading (قيادة) testing tools** to help them run unit tests better.

2.4 How To Do Unit Testing

- **Identify the unit:** Determine the specific code unit to be tested: either a function, method, class, or any other isolated component. Read the code and brainstorm the logic

needed to test it. In this step developers should also have an idea of the cases they need to test for that unit to ensure high test coverage.

- **Choose the approach:** Similar to many other testing types, there are two major approaches to unit testing:
    - **Manual testing:** Developers manually run the code and perform the necessary interactions to see if the code works well.
    - **Automated testing:** Developers write a script that automates the interactions with the code.
- **Prepare the test environment:** Set up the mock (wrong, damaged) objects, prepare test data, configure the dependencies, as well as any other required preconditions. A confident developer would isolate the function for a more rigorous testing process. This practice involves copying and pasting the code into a dedicated testing environment, separate from its original context. By isolating the code, unnecessary dependencies between the code being tested and other units or data spaces in the product are uncovered.
- **Write and execute test case:** If the developer chooses the automated approach, they'll start writing the test case, usually with a Unit Test Framework. This framework (or a test runner) can be used to execute the test and produce results (whether it passed or failed).
- **Debug, fix, and confirm:** If a test case fails, developers must debug it to identify the root cause, fix the issues (problem), then rerun the tests to confirm that the bugs have indeed been fixed.

2.5 Unit Testing Techniques

There are several unit testing techniques commonly used to ensure thorough test coverage, including:

- **Black box testing:** The internal structure and implementation details of the unit under test are not considered (similar to how the internals of a black box is not known). The tests focus on the external behavior and functionality of the unit. Test cases are designed based on the expected inputs, outputs, and specifications of the unit.
- **White box testing: T**he internal structure, logic, and implementation of the unit is taken into account, which contrasts with black box testing. Test cases are designed to explore different paths within the unit, ensuring that all code branches and segments are tested.

3. Top 4 Unit Testing Tools

3.1 JUnit

JUnit is an open-source unit testing tool in Java. It does not require the creation of class objects or the definition of the main method to run tests. It has an assertion library for evaluating test results. Annotations in JUnit are used to execute test methods. JUnit is commonly used to run automation suites with multiple test cases.

Key features:

- Supports test-driven development.
- Integrates with Maven and Gradle.
- Executes tests in groups.
- Compatible with popular IDEs like NetBeans, Eclipse, IntelliJ, etc.
- Fixture feature provides an environment for running repeated tests.
- Using the @RunWith and @Suite annotations, we can run unit test cases as test suites.
- Provides test runners for executing test cases.

Example

```
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

private int x = 1;
private int y = 1;

public void testAddition() {
int z = x + y;
assertEquals(2, z);
}
}
```

3.2 NUnit

NUnit, an open-source unit testing framework based on .NET, inherits many of its features directly from JUnit. Like JUnit, NUnit offers robust support for Test-Driven Development (TDD) and shares similar functionalities. NUnit enables the execution of automated tests in batches through its console runner.

Key features:

- A console runner provided by NUnit enables batch test execution.
- NUnit facilitates parallel test execution.
- Multiple assemblies are supported.
- Various attributes allow tests to be run with different parameters.
- Extensive support for Assertions is available.

- Data-driven testing is supported.
- Microsoft family languages such as .NET Core and Xamarin forms are supported.

3.3 TestNG

TestNG, short for Test Next Generation, is a robust framework that offers comprehensive control over the testing and execution of unit test cases. It incorporates features from both JUnit and NUnit, providing support for various test categories such as unit, functional, and integration testing. TestNG stands out as one of the most powerful unit testing tools due to its user-friendly functionalities.

Key features:

- Ability to execute test cases in parallel.
- Built-in exception handling mechanism.
- Support for testing integrated classes.
- Generation of HTML reports and logs.
- Capability to retrieve keywords/data from logs.
- Support for multi-threaded execution.
- Complete object-oriented nature with convenient annotations.
- XML-based configuration for all test settings.

3.4 PHPUnit

PHPUnit is a programmer-oriented unit testing framework specifically designed for PHP. It adheres (يلتصق) to the xUnit architecture commonly utilized by unit testing frameworks such as NUnit and JUnit. PHPUnit operates exclusively through command-line execution and does not have direct compatibility with web browsers.

Key features:

- Comprehensive code coverage analysis and the ability to simulate mock objects.
- Facilitation of test-driven development practices.
- Integration with the xUnit library to enable logging functionalities.
- Support for object mocking.
- Introduction of new assertions such as assertXMLFileTag() and assertXMLNotTag().
- Incorporation of error handler support in the existing version.
- Flexibility in extending test cases according to the programmer's specific requirements.
- Generation of multiple test reports.

4. Unit Testing Best Practices

- **Unit tests should be fast:** Usually unit tests are huge (enormous) in quantity, and if they require a lot of time to execute, developers will be hesitant (متردد) in taking on this

task. The goal of having unit tests is to boost the developers' confidence in the existing code so that they can proceed with the next features, so they should be short and straight to the point.

- **Unit tests should be simple:** Each unit test should focus on verifying a specific behavior or functionality (follow the "**one assertion per test**" rule). Structure your test on the **AAA pattern** to maintain clarity and readability in your unit tests. Choose descriptive, meaningful, but simple names for your test methods so that you have an easier time managing thousands of them.
    - The AAA (**Arrange-Act-Assert**) pattern has become almost a standard across the industry. It suggests that you should divide your test method into three sections: arrange, act and assert. Each one of them only responsible for the part in which they are named after.

So the Arrange section you only have code required to setup that specific test. Here objects would be created, mocks setup (if you are using one) and potentially expectations would be set. Then there is the Act, which should be the invocation of the method being tested. And on Assert you would simply check whether the expectations were met. More info can be found HERE.

Following this pattern does make the code quite well structured and easy to understand.

In general lines, it would look like this:

```
// arrange
var repository = Substitute.For<IClientRepository>();
var client = new Client(repository);// act
client.Save();// assert
mock.Received.SomeMethod();
```

- **Unit tests should be executed in isolation:** Code isolation is a highly recommended practice to eliminate any external influences. Test input data should also be controlled, so try to avoid using dynamically generated data that may influence test results. Also ensure that you have reset the state for each unit test run, so there can be no interference with previous tests.
- **Test results should be highly consistent (متناسق):** The more deterministic your unit tests are, the better. In other words, their results should always be consistent, no matter what changes were made to the code or what order you run your tests in.

- **Regularly refactor unit tests:** Treat your unit test code with the same care and attention as your production code. Refactor tests when necessary to improve readability, maintainability, and adherence to best practices.
- **Continuous integration and test automation:** Incorporate unit tests and automate their execution. This ensures that tests are run regularly, providing timely feedback on the health of your codebase.

4.1 Challenges of Unit Testing

Unit testing comes with a host (multitude) of challenges for developers:

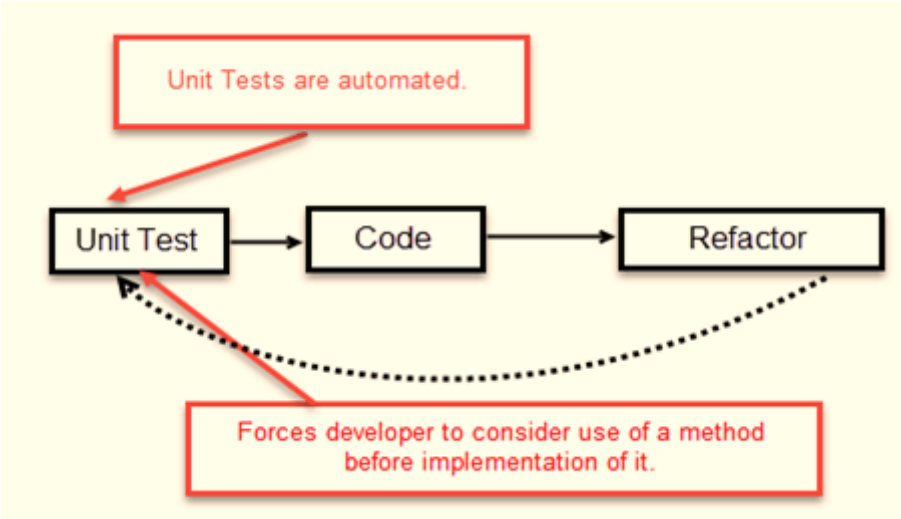
- Managing thousands of unit tests without a dedicated test management tool is a resource-intensive task.
- Writing test scripts and maintaining them across code updates is also time-consuming.
- Setting up test environment for a wide variety of tests requires effort.
- Unit testing activities need to be seamlessly integrated into the development workflow.

5. Test-Driven Development and Unit Testing

Test-Driven Development (TDD) and unit testing are two connected practices. The process of TDD involves writing automated unit tests **prior** to writing the code. These tests will surely **fail**, since there is no code written yet. After that, they will use the results from these tests to **guide** their code writing. Once they have developed the feature, they'll re-execute the previously failed tests to confirm that their code indeed (بالفعل) delivers the intended functionality.

TDD is a systematic development approach that consistently offers feedback, facilitating quick bug detection and debugging. Imagine a situation where many frustrated (محبط) users complain about a major problem that makes the app extremely slow. In an effort to fix this issue, your team quickly releases a patch. Unfortunately, this rushed (متسرع) solution introduces an even bigger problem, resulting in a widespread system failure.

TDD entails writing an automated unit test before the code itself. According to this approach, every piece of code must pass the test to be released. So, software engineers thereby focus on writing code that can accomplish the needed function. That's the way TDD allows programmers to use immediate feedback to produce reliable software.



5.1 Basic Steps for writing unit tests using TDD

- Write a test
- Fail compile
- Write stub
- Fail test
- Write implementation
- Pass test
- Refactor
- Pass test
- Commit

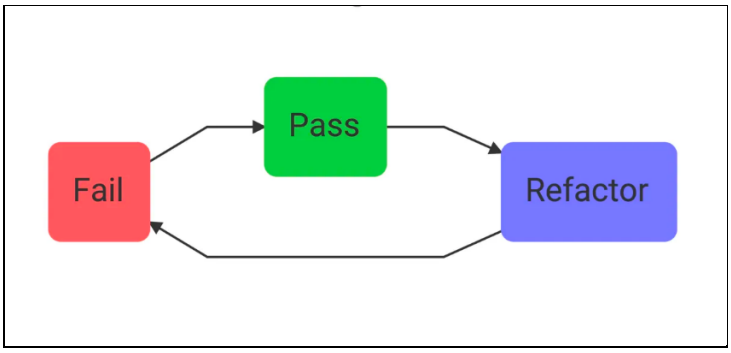
5.2 Importance of failure

Failing test cases can provide very important feedback. Some of the important characteristics of failures in TDD are:

- A failing test is the road to a passing test
- A failing test provides a safety net
- A failing test gives us assurance
- A failing test provides vital information

With Test-Driven Development, you can effectively prevent (يمنع) such incidents. Generally, developers that follow the TDD approach will go through a three-step process:

1. **Fail:** Write unit tests that will surely fail because no code is written.
2. **Pass:** Write code until those tests pass.
3. **Refactor:** Improve the code, then continue to run unit tests for the next features.



5.3 Red, Green, Refactor

This approach helps developers focus their efforts into three specific phases:

Red

- Write test for a behavior to be implemented
- Analyze the requirements without the implementation
- Behavior of the interface as *expected* by the caller
- Should be clean and consumable

Green

- Write just enough production code to pass the test
- Don't worry about algorithms or its performance
- Less code is better

Refactor

- Rework the code to make it more efficient and optimized

5.4 A Simple Scenario

Example 1: As a developer I want to implement code so that it prints the numbers from 1 to 100.

Given: an input of numbers from 1–100

When:

A number is a multiple of '3' return "Fizz"

A number is a of '5' return "Buzz"

A number is a of both '3' and '5' return "FizzBuzz"

A number is not divisible by '3' or '5' return the number itself

Then: print "Fizz", "Buzz", "FizzBuzz" or the number accordingly

Expected output: 1, 2, Fizz, 4, Buzz, ……, 14, FizzBuzz, 16, …

1. Write the initial test(s), divisible by 3

```
//FizzBuzzTest.java
// this should test if a number is a multiple of 3 or not
```

```
Public void test_multipleOf3(3) { ...
// should test for "1 2 Fizz"
...
}
```

2. Write the implementation

```
//FizzBuzz.java
Public String FizzBuzz(int num){
for(int count = 1; count <= num; count++){
    if (count % 3 == 0)
    ...
  }}
```

3. Update tests with additional requirements, divisible by 5

```
//FizzBuzzTest.java
// this will test if a number is a multiple of 5 or not
Public void test_multipleOf5(5) {{ ...
// should test for "1 2 3 4 Buzz"
 ...
}
```

4. Update implementation

```
//FizzBuzz.java
Public String FizzBuzz(int num)
{for(int count = 1; count <= num; count++)
{    if (count % 5== 0)
    ...
   }}
```

5. Update tests with additional requirements, divisible by 3 and 5

```
//FizzBuzzTest.java:
// this will test if a number is a multiple of both 3 and 5 or not
Public void test_multipleOf3And5(15) {
// should test for "1 2 .... 14 FizzBuzz.."
...
}
```

6. Update implementation

```
//FizzBuzz.java
```

```
Public String FizzBuzz(int num)
{for(int count = 1; count <= num; count++)
{   if (count % 15 == 0)
 ...
 ...
}}
```

7. Refactor — Update the implementation, optimize and bring all the code together

```
//FizzBuzz.java
Public String FizzBuzz(int num)
{for ( int idx = 1; idx <= count; idx ++)
{  switch(idx % 15)
{    case 0:
print "FizzBuzz ";
    …
case 3:
print "Fizz ";
 …
case 5:
 print "Buzz ";
   …
default:
 …
print number  } }}
```

Finally, run all the tests and make sure they succeed.

Example 2 : Here in this Test Driven Development example, we will define a class password. For this class, we will try to satisfy following conditions.

A condition for Password acceptance:

- The password should be between 5 to 10 characters.

First in this TDD example, we write the code that fulfills all the above requirements.

```
package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
  @Test
  public void TestPasswordLength() {
      PasswordValidator pv = new PasswordValidator();
      Assert.assertEquals(true, pv.isValid("Abc123"));

  }
}
```

Needed for TestNG

We can not run test because this class is not created yet

This is main validation test

Scenario 1: To run the test, we create class PasswordValidator ();

```
package Prac;

public class PasswordValidator {
 public boolean isValid(String Password)
 {
     if (Password.length()>=5 && Password.length()<=10)
     {
         return true;
     }
     else
         return false;
 }
}
```

This is main condition checking length of password. If meets return true otherwise false.

We will run above class TestPassword ();

Output is PASSED as shown below;

Output:

```
<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 2:10:22 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--571370159\testng-customsuite.xml

PASSED: TestPasswordLength

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 202 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 63 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 78 ms
[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@5a01ccaa: 2 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 1 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@2b80d80f: 10 ms
```

Result of test as Passed

Scenario 2: Here we can see in method TestPasswordLength () there is no need of creating an instance of class PasswordValidator. Instance means creating an object of class to refer the members (variables/methods) of that class.

```
package Prac;

import org.testng.Assert;

public class TestPassword {
  @Test
  public void TestPasswordLength() {

      PasswordValidator pv = new PasswordValidator();

      Assert.assertEquals(true, pv.isValid("Abc123"));

  }
}
```

We will remove it.

We will remove class PasswordValidator pv = new PasswordValidator () from the code. We can call the **isValid ()** method directly by **PasswordValidator. IsValid ("Abc123")**. (See image below)

So we Refactor (change code) as below:

```
package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
  @Test
  public void TestPasswordLength() {

      Assert.assertEquals(true, PasswordValidator.isValid("Abc123"));

  }
}
```

Re factor code as there is no need of creating instance of class PasswordValidator().

Scenario 3: After refactoring the output shows failed status (see image below) this is because we have removed the instance. So there is no reference to non –static method **isValid ().**

```
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--157192639\testng-customsuite.xml

FAILED: TestPasswordLength
java.lang.Error: Unresolved compilation problem:
        Cannot make a static reference to the non-static method isValid(String) from the type PasswordValidator

        at Prac.TestPassword.TestPasswordLength(TestPassword.java:10)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at org.testng.internal.MethodInvocationHelper.invokeMethod(MethodI
        at org.testng.internal.Invoker.invokeMethod(Invoker.java:639)
        at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:821)
```

If we removed instance creation statement compiler will give error. As we do not create instance it becomes non static method and there is no any reference to this method. Test results in Fail. To remove this error we have to make isValid() method of class PasswordValidator as static.

So we need to change this method by adding "static" word before Boolean as public static boolean isValid (String password). Refactoring Class PasswordValidator () to remove above error to pass the test.

```
package Prac;

public class PasswordValidator {
 public static boolean isValid(String Password)
 {
     if (Password.length()>=5 && Password.length()<=10)
     {
         return true;
     }
     else
         return false;
 }
}
```

Re factor : Added static word to pass test.

Output:

After making changes to class PassValidator () if we run the test then the output will be PASSED as shown below.

```
<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 3:02:16 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--1385484104\testng-customsuite.xml

PASSED: TestPasswordLength

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 19 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 10 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 34 ms
```

Test results passed as we changed code in class PasswordValidator().

5.5 Benefits of Test Driven Development

Test-driven development has a lot to offer developers and testers. Here are some of the benefits of test-driven development:

- **Improved code quality**

Test-driven development helps enhance code quality by promoting clean, modular and well-structured code. As tests are written before the actual code, developers can consider the detailed requirements and design accordingly, leading to more thoughtful and deliberate coding practices. Each unit code is validated as it is developed, ensuring it meets the desired functionality and performance standards from the outset.

- **Easier maintenance and refactoring**

Another major advantage of TDD is that it can simplify maintenance and refactoring. Since every piece of functionality is covered by tests, developers can confidently make changes and improvements to the codebase without the fear of introducing new bugs. This safety net encourages regular refactoring, leading to more maintainable and adaptable code over time.

Enhanced collaboration between developers and testers

TDD integrates testing into the development process, instilling better collaboration between developers and testers. Testers can write tests while developers implement the corresponding code, ensuring that both teams are aligned on the expected functionality and quality standards. This collaborative approach helps bridge communication gaps.

- **Early bug detection**

By writing tests first, TDD allows for early bug detection, reducing the cost and effort required to solve issues later in the development cycle. This approach ensures that problems are detected and addressed immediately, leading to more stable and reliable software. Early bug detection

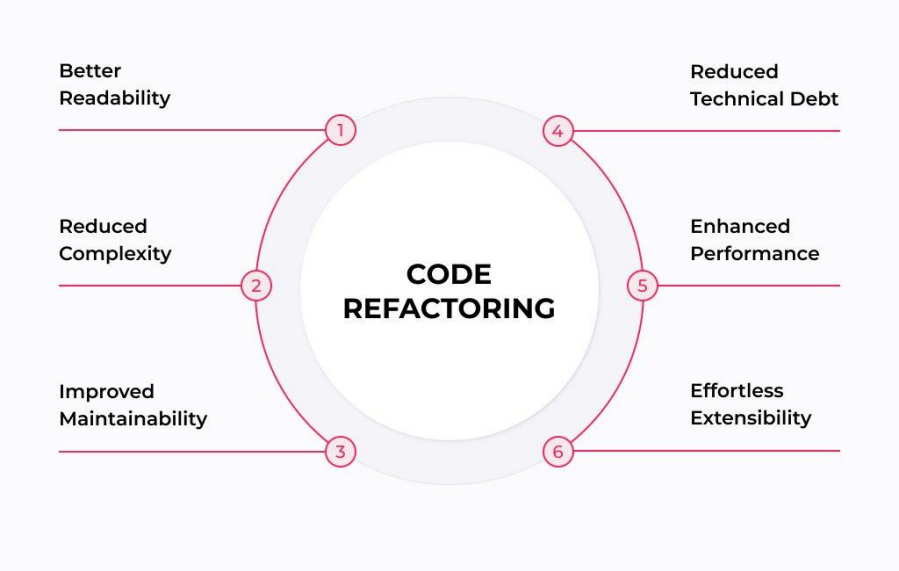
also contributes to a smoother software development process, with fewer disruptions and less time spent on debugging and troubleshooting.

6. Agile Code Refactoring

Every software development project looks the same at first glance (coup d'oeil). You gather requirements and then write the code to turn them into useful functionalities to which a software solution provides access. The development process can take many months, and the application will be used for years or even decades after the launch. **During this period, the source code needs to be changed** countless times to add new features or fix bugs, for example. When developers **revise** projects they weren't involved in for quite some time, it may be hard to figure out what their code is doing immediately. Imagine how hard it can be to work with the results of somebody else's work, especially if there's no possibility of contacting the author to receive some clarification. In this, and many other cases, code refactoring may come to the **rescue**. It helps make the app's source code more **efficient** and **maintainable**.

This part, considers the main techniques of code refactoring and look at the benefits it can bring.

6.1 Code Refactoring principle



Code refactoring, means that the **software development company adjusts the code** of an application **without making any changes to how it functions**. In this scenario, even if the app works according to the requirements, it doesn't mean that there's no room for improvement.

Usually, complex apps are built by more than one developer, which means that all the preferences of involved parties will be reflected in the code. As a result, you can get an optimal working app with the code not structured in the best possible way under the hood (capot).

One of the main benefits of code refactoring is that it helps to **reduce technical debt**. The term refers to the **cost** of maintaining and **supporting** software over time. As codebases grow, they can become complex and challenging to understand, which increases technical debt.

By refactoring code, developers can reduce complexity, making it easier to maintain and improve in the future. It reduces technical debt, ultimately saving time and money in the long run.

Another benefit of code refactoring is that it can improve the **performance** of the software. Refactoring can identify and remove **bottlenecks** in the code, which can make it faster and more efficient. This is especially important in high-performance applications where even minor performance improvements can significantly impact user experience.

Code refactoring can also enhance code quality by improving its **readability** and making it easier to understand. By **removing unnecessary** parts, developers can simplify the codebase, making it easier to maintain and reducing the **probability** of bugs. This ultimately makes it easier for developers to work with the codebase, which saves time and effort.

6.2 When to Perform Code Refactoring

You should consider software refactoring in the following situations:

- **Low readability and maintainability:** If the code is difficult to understand or modify due to poor organization, inconsistent naming conventions, or lack of comments, refactoring can help improve its clarity and ease of maintenance.
- **Updating to newer technologies or libraries:** When migrating to a new technology stack or upgrading libraries, refactoring can help ensure compatibility and smooth (سلس) integration.
- **Improving performance:** When you identify areas of the code with inefficient algorithms or data structures, refactoring can help optimize the code and enhance its performance.
- **Code smells (odeurs):** If you identify recurring patterns or issues in the code that indicate poor design or implementation, such as duplicated code, large classes, long methods, or excessive use of global variables, refactoring should be considered.
- **Enhancing modularity and reusability:** If the code has tightly coupled (étroitement couple, مقترنة بإحكام) components or lacks modular design, refactoring can help break it into smaller, more manageable pieces with well-defined interfaces, promoting reusability and easier testing.

- **Preparing for new features or changes:** Before implementing new functionality or making significant changes to the existing codebase, refactoring can help create a cleaner, more adaptable foundation.
- **Simplifying the codebase:** If the code contains redundant or overly complex logic, refactoring can help streamline the code and make it easier to manage.

Remark: It's essential to approach refactoring carefully, as it **can introduce new bugs** or issues if not done correctly. Make sure to have proper test coverage and version control in place before starting the refactoring process.

6.3 Common Code Refactoring Techniques

6.3.1 Composing Method: This refactoring technique focuses on breaking down **large** classes or methods into **smaller**, more manageable components. The primary goal is to improve code readability, maintainability, and testability by making sure each component has a single responsibility.

This approach promotes a modular and organized codebase, making it easier to understand, modify, and extend in the future. Composing encourages a **clean separation** of concerns, which results in higher-quality and more reliable software.

There's a lot of approaches to implement this technique, such as the **extract** method. For example, let's look at this code below:

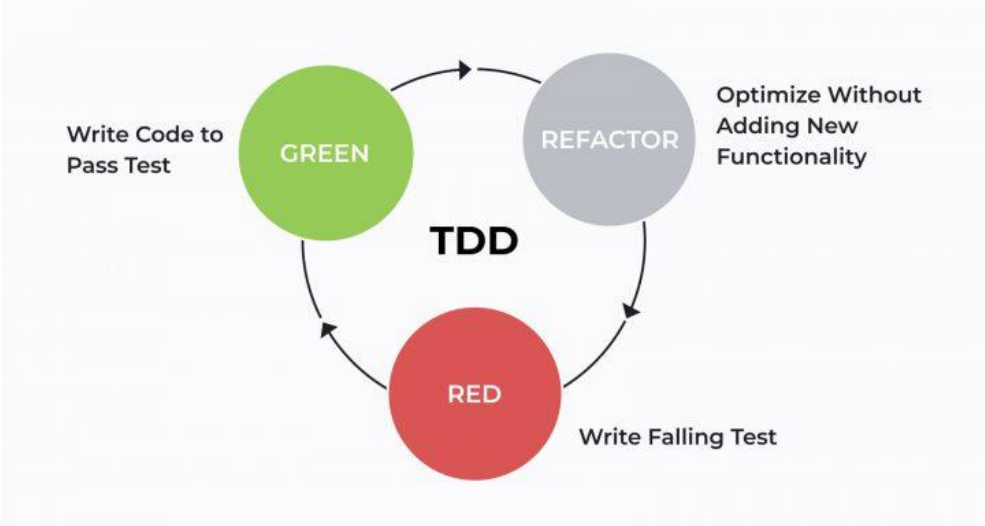
```
function calculateTotalPrice() {
    let total = 0;
    for (let item of cart) {
        total += item.price * item.quantity;
    }
    return total;
}
```

Here is an example of how to use extract method in this case:

```
function calculateTotalPrice() {
    let total = calculateSubtotal();
    return total;
}

function calculateSubtotal() {
    let subtotal = 0;
    for (let item of cart) {
        subtotal += item.price * item.quantity;
    }
    return subtotal;
}
```

6.3.2 Red-Green Refactoring: This technique is an integral part of test-driven development (TDD), a software development methodology that emphasizes writing tests before writing the actual code. The process consists of three main steps:



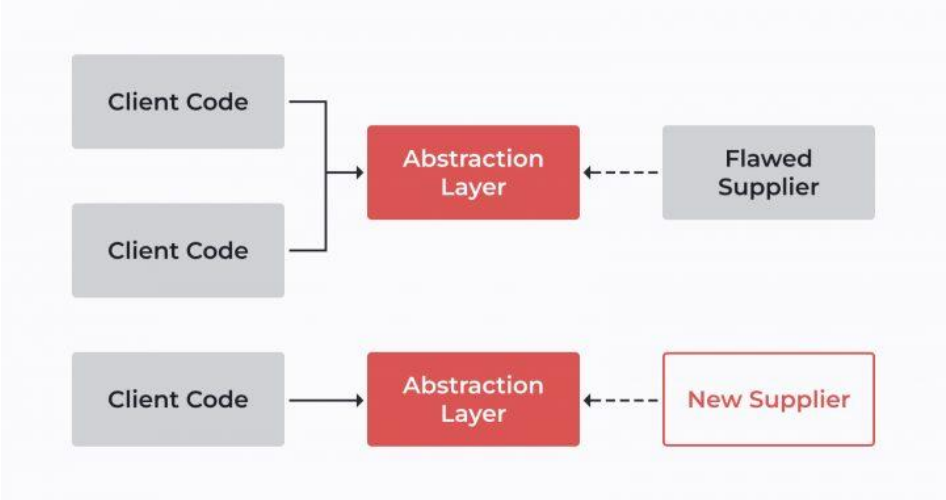
- **Red:** Write a failing test that exposes the issue or required functionality. The goal is to create a test that will fail because the desired functionality has not been implemented yet. This step ensures that the test is valid and can detect the absence of the intended feature.
- **Green:** Write the **minimum** code **necessary** to pass the test. In this step, you focus on implementing the functionality needed to make the test pass, without worrying about the quality or maintainability of the code. The primary goal is to make the test pass as quickly as possible.
- **Refactor:** Improve the code while keeping the test green (i.e., without breaking the functionality). Once the test passes, you can refactor the code to make it cleaner, more efficient, and maintainable. The test serves as a safety net during the refactoring process, ensuring that the external behavior remains unchanged.

This cycle of red-green-refactor is repeated for each new feature or bug fix, promoting a development process where the code is continuously tested and refactored, resulting in higher-quality and more reliable software.

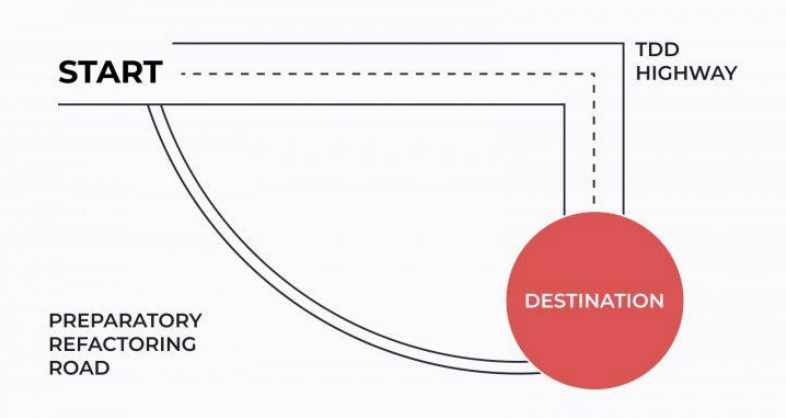
6.3.3 Refactoring by Abstraction: Refactoring by abstraction is a technique where you identify **common functionality** shared by multiple classes or methods and extract it into a **separate**, **abstract class or interface**. This process helps reduce code duplication, promotes reusability, and makes it easier to manage and maintain the shared functionality.

For example, consider two classes with similar methods that perform the same calculations. By extracting the shared logic into an abstract class or an interface, you can create a single implementation that both classes can **inherit** or implement. This abstraction reduces the amount of duplicated code and makes it easier to update or modify the shared functionality in the future.

This is a good option for really big software development projects where you deal (treat) with tons of code. (Flawed = deficient)



6.3.4 **Preparatory Refactoring**: This approach implies refactoring the source code before making significant changes or adding new features. It is used when preparing the codebase for new features or significant changes. The objective is to create a cleaner, more adaptable foundation for future development by **restructuring** code, updating deprecated APIs or libraries, or simplifying complex logic. Preparatory refactoring helps ensure the codebase remains maintainable, testable, and scalable as new functionality is added.



It ensures that the codebase is stable, making adding new features easier without causing issues or introducing bugs. Preparatory refactoring must come hand-in-hand (accompany) with all the updates you make during the software system life cycle to help you avoid possible tech debt.

Moving Features Between Objects. This technique involves **creating new classes** and moving features from one class to another. It can be used to eliminate duplication or to make the code **more modular**. Not all classes are equally valuable for software projects. Some of them are useless, allowing development teams to make these adjustments without harm

(damage, wrong). By moving features between objects, developers can improve the overall (total) **design** of the codebase. Suppose we have code that looks like this:

```
class Reports {
  // ...
  sendReports(): void {
    let nextDay: Date = new Date(previousEnd.getYear(),
      previousEnd.getMonth(), previousEnd.getDate() + 1);
    // ...
  }
}
```

In examples like this, the class may not contain the method you desire to use and adding it is not an option. As a solution, you can add such a method to a client class. Then, nothing will limit you from passing an object of the class you're working with to it as an argument:

```
class Reports {
  // ...
  sendReports() {
    let newStart: Date = nextDay(previousEnd);
    // ...
  }
  private static nextDay(arg: Date): Date {
    return new Date(arg.getFullYear(), arg.getMonth(), arg.getDate() + 1);
  }
}
```

6.3.5 Simplifying Methods: This refactoring technique aims to make methods more understandable and maintainable by reducing their complexity. It involves various approaches that simplify the code, such as reducing the number of method parameters, replacing complex conditional logic with simpler constructs, or breaking down long methods into smaller, focused methods.

Original	Refactored
public void processOrder (String customerName, String productCode, int quantity, boolean isPriority) { // method implementation }	public void processOrder(Order order) { // method implementation using order object }

6.3.6 Moving Features Between Objects : This technique focuses on reassigning (réaffectation) responsibilities or methods between classes to improve cohesion (how closely a class's responsibilities are related) and reduce coupling (the degree to which one class depends on another). By redistributing code or functionality among classes, you create a more balanced and logical design that is easier to maintain and extend.

Original	Refactored

class SourceClass {
int movedField;
}class TargetClass {
// … other fields int movedField;
}
	class SourceClass {
	// … other fields
	}class TargetClass {
	int movedField;
	}

6.4 Best practices for refactoring

a. Collaborate with Testers

Engaging the quality assurance (QA) team during the refactoring process ensures that the changes made do not introduce new bugs or negatively impact the software's functionality. The QA team can help validate that the refactored code behaves as expected, maintaining the software's quality and reliability.

b. Automate the Process

Utilizing automated tools for code analysis, testing, and refactoring can help streamline the process and minimize the risk of introducing errors. Tools like static code analyzers, linters, and automated testing frameworks can help identify code smells, enforce coding standards, and ensure the refactored code meets the required quality criteria.

c. Refactor in Small Steps

Making small, incremental changes during the refactoring process reduces the likelihood (probabilité) of introducing bugs or breaking the software. By focusing on one improvement at a time, you can more easily identify and fix issues, making it easier to maintain a stable codebase throughout the refactoring process.

d. Troubleshoot and Debug Separately

Refactoring should be kept separate from bug fixing and troubleshooting. Mixing the two can lead to confusion and complicate the process of identifying the root causes of issues. Address known bugs before refactoring, and treat any new issues that arise during refactoring as separate tasks to maintain clarity and focus.

e. Prioritize Code Deduplication

One of the primary goals of refactoring is to eliminate duplicated code, as it can lead to inconsistencies, increase maintenance complexity, and make the codebase more error-prone (عرضة للخطأ). Focusing on identifying and extracting common functionality into reusable components, such as abstract classes, interfaces, or utility methods, can greatly improve the maintainability and readability of the code.

6.5 What is the difference between code optimization and code refactoring

Code Optimization

- Improves resource efficiency, execution time, and performance of the code.
- Focuses on improving the memory or processing time.
- May change the original code to achieve performance gains.
- Involves changes to algorithmic or low-level implementation details.
- May sacrifice code readability to achieve performance targets.

Code Refactoring

- Aims to improve the structure, readability, and maintainability of the code.
- Looks to enhance code organization without altering external behavior.
- Retains intrinsic (nature) functionality while enhancing the code quality.
- Involves holistic restructuring, simplifying, and clarifying code.
- Usually, doesn't consider performance improvements as the main goal.

7. Technical debt

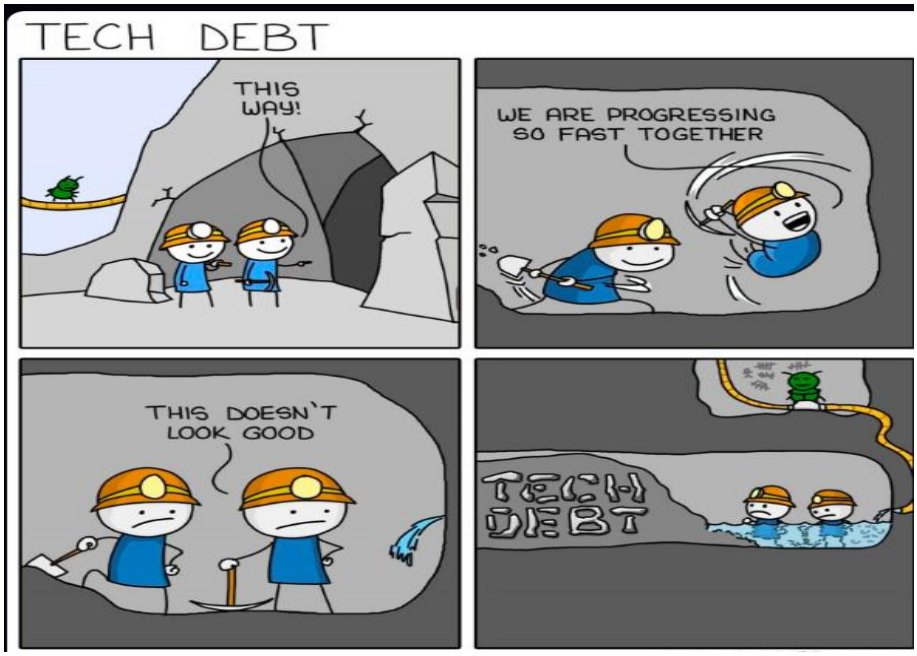
"Technical debt is the cost of reworking code in the future because of decisions made in the present"

7.1 Technical Debt in Agile

In the rush (speed, hurry) to meet project deadlines and deliver products, engineers tend to take **shortcuts** to help development process ship (boat) **faster**, that accrue (accumulate) **technical debt**.

Tech debt, or **code debt**, is a virtually **inevitable** (certain) part of the agile software development process. It's not always a bad thing either

Technical debt is the **result** of trade-offs (accord, contract) made to expedite (**accelerate**) the software development process.



The process of writing code is based on a programmer's **understanding** of the problem at the **time** they write it. This might seem obvious (evident), but it has huge implications. Their understanding can evolve **rapidly**, leaving their code **outdated** and no longer **appropriate** for the problem at hand. This is particularly true for **legacy** (heritage) code and in **high-growth** environments.

Short-term solutions, such as **refactoring**, can temporarily address the issue, but this just adds more **cruft** to the code and increases technical debt.

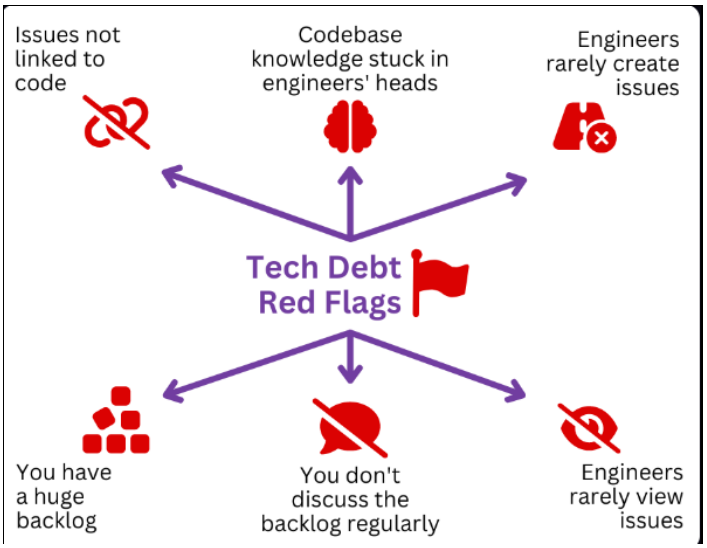
The trade-offs that lead to technical debt often involve **shortcuts** that compromise (discus) the **quality** of the code. These shortcuts might include hard-**coding**, lack of **testing**, or other code quality **issues**. Tech debt can accumulate rapidly and have a significant **impact (effect)** on software development.

Technical debt can **cost** your company **delayed** (retard) product launches, **countless** lost hours of engineering time, and developer **productivity**. And tech debt in Agile environments can be especially **troublesome**.

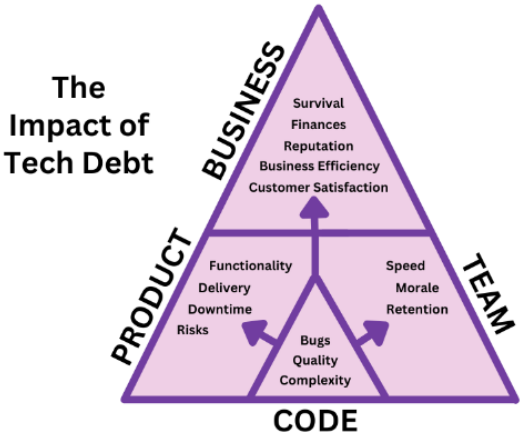
Agile and Scrum's focus on **iterative** development and **rapid** delivery can lead to choices that accumulate a **backlog** of problems over subsequent (following) **sprints**.

7.2 Recognising the symptoms of tech debt

Very few engineers say their teams are good at tracking (pursuit, تتبع) technical debt. Tackling tech debt has to start with **tracking** it. If you haven't tracked it, you can't do anything useful. It can't be managed or prioritised, and you can't prevent (stop) the spread (propagation) or stop the rot (العفن). These are the red flags that might signal to you that you need to work on your tech debt strategy:



When tech debt isn't managed properly, it has the capacity to **destroy** business productivity at every level



7.3 How Agile Scrum Masters Can Reduce Technical Debt

To effectively manage Scrum technical debt, consider these approaches:

- **Regular code reviews**: Conduct prudent code reviews to identify and address areas of code debt.
- **Enhanced code reviews**: Use language learning models (LLMs) to help conduct thorough code reviews, even in globally distributed teams. Machine learning algorithms and transformer models can identify potential issues or inconsistencies in code, ensuring that you detect unintentional debt early and address it before it accumulates.
- **Automated documentation**: Employ language learning models to facilitate the creation of comprehensive and standardized documentation. This includes automatically generating code comments, annotations, and explanations, ensuring that the codebase

remains well-documented, and reducing the chances of technical debt arising from unclear or undocumented code.

- **Prioritize refactoring**: Dedicate (Consacrer) a portion of each sprint to refactoring, a practice involving restructuring current code to enhance its quality. This ensures the development process remains iterative and adaptable while consistently striving (السعي) for improved code integrity and decreased product backlog.
- **Automated testing**: Implement automated testing to catch regressions early and ensure modifications don't introduce new problems.
- **Predictive analysis**: Use machine learning models to predict areas of potential technical debt by analyzing patterns in the development process. Deep learning algorithms can forecast (prévision) where future technical debt might accumulate by studying historical data on code changes, bug fixes, and performance improvements. This proactive insight (رؤية استباقية) enables Agile teams to allocate resources for timely refactoring, preventing the compounding (composition) of debt over multiple sprints.
- **Incremental improvements**: Focus on gradual improvements instead of massive overhauls to prevent disrupting the development cycle. (Concentrez-vous sur des améliorations progressives plutôt que sur des révisions massives pour éviter de perturber le cycle de développement.)
- **Collaborative culture**: Encourage open communication with team members and product owners about identified tech debt and strategies to manage it.

Incorporating these practices into the development process helps the team proactively address technical debt, ensuring a more adaptive and sustainable software development journey. And incorporating language learning models and machine learning makes the process faster and more efficient.

7.4 Technical Debt Ratio

Technical Debt Ratio is a **metric** that provides insights (رؤى) into the quality of code in a software project by comparing the cost to fix the technical debt against the total cost of developing the code. It helps teams and management to understand the proportion of the development effort that is consumed by technical debt and provides a high-level overview of the codebase's health.

7.4.1 measure Technical Debt Ratio

Formula for measuring Technical Debt Ratio:

TDR=(Cost to Fix Technical Debt/Total Development Cost)×100%

Components:

- **Cost to Fix Technical Debt:** The estimated effort required to fix all known issues and code smells in the codebase, often measured in person-days or person-hours.
- **Total Development Cost:** The total effort spent on developing the code, which can include designing, coding, testing, and deploying the software.

7.4.2 Use Technical Debt Ratio Metric

- **Continuous Monitoring:** TDR should be monitored regularly to track the evolution of technical debt over time and to identify trends or issues that may need to be addressed.
- **Strategic Planning:** It can be used in strategic planning and backlog prioritization to ensure that technical debt is addressed in a timely and cost-effective manner.
- **Communication:** TDR can be used to communicate the impact and importance of technical debt to stakeholders and to justify investments in code quality and refactoring.

7.4.3 Case study - Technical Debt Ratio x Payment processing app

To illustrate the concept of measuring technical debt ratio in a fintech app, let's consider a hypothetical example.

a. Identifying Technical Debt Components:

- **Codebase Analysis:** The development team reviews the codebase and identifies areas with outdated libraries, redundant code, and inefficient algorithms. For instance, they find that the payment processing module uses an outdated encryption library.
- **Architecture Review:** The team notices that the app's microservices architecture is not optimally designed, leading to frequent downtime during high traffic.
- **Documentation and Code Comments:** The documentation is outdated, and there are very few code comments, making it difficult for new developers to understand the code.

b. Quantifying Technical Debt:

- **Time Estimation for Refactoring:** The team estimates that it would take approximately 200 hours to update the encryption library, 500 hours to optimize the microservices architecture, and 100 hours to update documentation and add necessary code comments.
- **Financial Cost:** They calculate the cost based on the hourly rate of developers. Suppose the rate is $50/hour, the total cost for refactoring would be: $(200+500+100)\times 50=\$40,000$

b. Calculating Technical Debt Ratio:

- **Cost:** Assume the overall cost of building the fintech app was $500,000.
- **Technical Debt Ratio Calculation:** The technical debt ratio is calculated by dividing the cost of resolving technical debt by the total project cost. So, in this case, it would be: 40,000/500,000=0.08 or 8%.

This **8%** technical debt ratio indicates that 8% of the project's budget would be required to pay off the existing technical debt. This metric helps in understanding the scale of the debt and in making informed decisions about whether and when to address it.

8. Version control system

A version control system (VCS) **tracks** every alteration to a file or set of files, enabling developers to **journey** (trajet) back to earlier versions and collaborate **seamlessly** (transparence). **Centralized** version control systems (CVCS) streamline this process by housing **all file versions on a single server**. Developers borrow (take, يستعير) a file to tweak (adjust), then return it with updates, all neatly (cleverly, بدقة) stored and cataloged by the server. This method shines in its simplicity, offering a straightforward (directe) path for managing changes.

Yet, as teams grow and projects become more intricate (complex), the **distributed** version control systems (DVCS) such as Git step into the spotlight. DVCS doesn't just centralize files; it democratizes them. Every developer holds the entire project history locally, empowering offline work and facilitating a tapestry (نسيج) of branching and merging strategies. This flexibility is a boon (نعمة) for dynamic teams aiming to weave (نسج) together multiple project threads without tangling (تشابك) them.

Whether centralized or distributed, version control is the cornerstone of efficient, cohesive (متماسك) software development. It safeguards (guarantee) progress, clarifies the past, and smooths (ينعم) the path forward, ensuring that every team member can contribute their best work towards crafting (fabrication) stellar (excellent) software.

8.1 Types of version control systems

The two most popular types of version or revision control systems are centralized and distributed. Centralized version control systems store all the files in a central repository, while distributed version control systems store files across multiple repositories. Other less common types include lock-based.

8.1.1 Distributed

A distributed version control system (DVCS) allows users to access a repository from multiple locations. DVCSs are often used by developers who need to work on projects from multiple computers or who need to collaborate with other developers remotely.

8.1.2 Centralized

A centralized version control system (CVCS) is a type of VCS where all users are working with the same central repository. This central repository can be located on a server or on a developer's local machine. Centralized version control systems are typically used in software development projects where a team of developers needs to share code and track changes.

8.1.3 Lock-based

A lock-based version control system uses file locking (verrouillage) to manage concurrent access to files and resources. File locking prevents two or more users from making conflicting changes to the same file or resource.

8.2 Presentation of the best versions control system: GIT, Git Hub, CVS, GitLab, BitBucket

8.2.1 Git : is a version control system that intelligently tracks changes in files. Git is particularly useful when you and a group of people are all making changes to the same files at the same time.

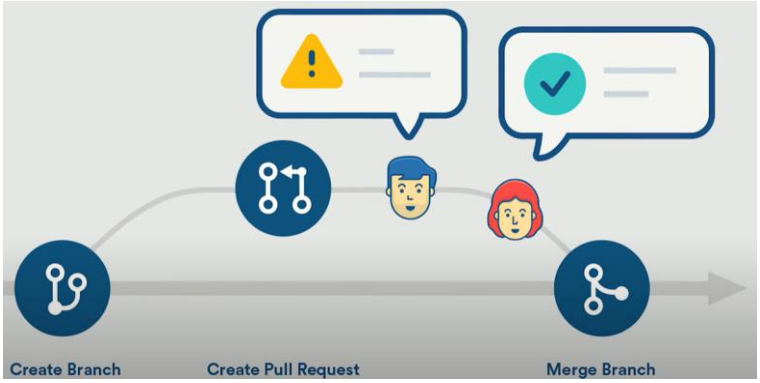
- Is distributed version control system
- Is free and open source
- GitHub is source for project and source [**GitLab, BitBucket**]
- GitHub simplify using Git
- You can use Git without GitHub
- Git has GUI

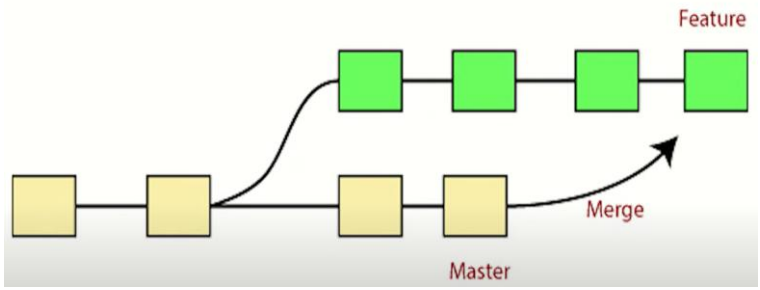
8.2.2 Some Git's benefits

- Developers contribute the same project
- You can revert (return, go back) changes
- You can elaborate to fixe issues
- You can elaborate to create new feature
- You can solve conflicts
- You can organize features

Typically, to do this in a Git-based workflow, you would:

- **Create a branch** off from the main copy of files that you (and your collaborators) are working on.
- **Make edits** to the files independently and safely on your own personal branch.
- Let Git intelligently **merge** your specific changes back into the main copy of files, so that your changes don't impact other people's updates.
- Let Git **keep track** of your and other people's changes, so you all stay working on the most up-to-date (update) version of the project.





8.2.3 **GitHub** is a web-based interface that uses Git, the open source version control software that lets multiple people make separate changes to web pages at the same time. As Carpenter notes, because it allows for real-time collaboration, GitHub encourages teams to work together to build and edit their site content.

- Is one of the most popular resources for developers to share code and work on projects together. It’s free, easy to use, and has become central in the movement toward open-source software.
- It makes it easy for developers to share code files and collaborate with fellow developers on open-source projects. GitHub also serves as a social networking site where developers can openly network, collaborate, and pitch (launch, throw) their work.

8.2.4 How do Git and GitHub work together

When you upload files to GitHub, you'll store them in a "Git repository." This means that when you make changes (or "commits") to your files in GitHub, Git will automatically start to track and manage your changes.

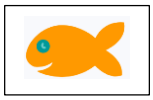
There are plenty (multitude) of Git-related actions that you can complete on GitHub directly in your browser, such as creating a Git repository, creating branches, and uploading and editing files.

However, most people work on their files locally (on their own computer), then continually sync these local changes—and all the related Git data—with the central "remote" repository on GitHub. There are plenty of tools that you can use to do this, such as GitHub Desktop.

Once you start to collaborate with others and all need to work on the same repository at the same time, you'll continually:

- **Pull** all the latest (most recent) changes made by your collaborators from the remote repository on GitHub.
- **Push** back your own changes to the same remote repository on GitHub.

Git figures out how to intelligently merge this flow of changes, and GitHub helps you manage the flow through features such as "pull requests."

8.2.5 CVS :  Concurrent Versions System (CVS) is a program that lets a code developer save and retrieve different development versions of **source code**. It also lets a team of developers share control of different versions of files in a common repository of files. This kind of program is sometimes known as a ***version control system*** . CVS was created in the UNIX operating system environment and is available in both Free Software Foundation and commercial versions. It is a popular tool for programmers working on Linux and other UNIX-based systems.

CVS works not by keeping track of multiple copies of source code files, but by maintaining a single copy and a record of all the changes. When a developer specifies a particular version, CVS can reconstruct that version from the recorded changes. CVS is typically used to keep track of each developer's work individually in a separate working directory. When desired, the work of a team of developers can be merged in a common repository. Changes from individual team members can be added to the repository through a "commit" command.

CVS uses another program, Revision Control System (RCS), to do the actual revision management - that is, keeping the record of changes that go with each source code file. The writers of the most popular CVS Frequently Asked Questions document are careful to emphasize that CVS is not a *build system*, a code configuration management system, or a

substitute for other good development practices, but simply a way to control the versions of the pieces of a program as they are developed

8.2.6 SVN

Subversion software, also called SVN, is an open source version control system. Subversion (SVN) allows teams to look at previous file versions and track their changes over time.

Subversion software is used for maintaining current and historical versions of projects. Subversion is an open source centralized version control system. It's also referred (reference) to as a software version and revision control system.

Subversion software used to be one of the most popular systems. But SVN's popularity is **waning (**تناقص). Many businesses chose SVN to save costs. Subversion was initially appealing since it is open source and was able to work with these businesses' original scale (échelle) and project needs.

Subversion is actually a **centralized** version control system. SVN is different from **distributed** systems, like **Git**.

8.2.7 GitLab

a web-based Git repository that provides free open and private repositories, issue-following capabilities, and wikis. It is a complete DevOps platform that enables professionals to perform all the tasks in a project—from project planning and source code management to monitoring and security. Additionally, it allows teams to collaborate and build better software. GitLab helps teams reduce product lifecycles and increase productivity, which in turn creates value for customers. The application doesn't require users to manage authorizations for each tool. If permissions are set once, then everyone in the organization has access to every component. Customers can opt for the paid version of GitLab if they want to access more functionalities. For example, the Premium version costs $19 per user/month.

8.2.8 BitBucket

a Git-based source code repository hosting service owned by Atlassian. Bitbucket offers both commercial plans and free accounts with an unlimited number of private repositories.

Bitbucket Cloud is a Git based code hosting and collaboration tool, built for teams. Bitbucket's best-in-class Jira and Trello integrations are designed to bring the entire software team together to execute on a project. We provide one place for your team to collaborate on code from concept to Cloud, build quality code through automated testing, and deploy code with confidence.

While GitHub comes with a lot of features and allows you to create your own workflows, Bitbucket arguably has more flexibility built-in. Bitbucket can also import from Git, CodePlex, Google Code, SourceForge, and SVN.

8.3 Importance of versions control system

- Collaboration -Branching and merging -Track Changes -Enables easy tracking of code changes -Managing and protecting the source code -Traceability -Enhanced project - development speed -Conflict resolution –Efficiency -Detailed change history -Loss of data Reduce error and duplication -Storing versions -Version control enables concurrent

- **Application of versions control system**

* Software development *Web development *System management * Documentation

8.4 Difference between Git, CVS and SVN

- Git is distributed version control system unlike centralized systems such as SVN and CVS. This results in better reliability, performance and true support for merging parallel development branches.
- SVN and Git are both powerful version control systems that each use a different approach to managing and merging code changes. Git uses a distributed model, whereas SVN uses a centralized model. Which VCS that you choose largely depends on your software development project's requirements
- For personal code versioning, Git is widely recommended due to its distributed version control system, flexibility, and robust community support. Git allows for efficient branching, merging, and collaboration on projects. CVS and SVN are older systems with fewer features compared to Git
- The Git- SVN tool is an interface between a local Git repository and a remote SVN repository. Git- SVN lets developers write code and create commits locally with Git, then push them up to a central SVN repository with SVN commit-style behavior.
- SVN vs CVS are both version control files. They are mostly used by teams that are collaborating on a single project. SVN stands for SubVersioN, and CVS stands for Concurrent Versions System. They allow the team members to keep track of all changes made and know who is developing what.
- The main distinction between CVS and SVN is that CVS is a free and client-based version control system, whereas SVN is the newest and most advanced version control system.

8.5 Wrapping up

In the fast-paced field of software development, agility is the ultimate game-changer, propelling teams toward success and ensuring they stay ahead of the curve. It's the secret sauce that transforms good teams into great ones!

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module : SIA-2**
Teacher: Ouaar Hanane **Year : 2024/2025**

Hands-out N° 2-1: JUnit + TDD + Agile Code refactoring

For the present homework, your are asked to;

- Display the important screen shoots
- Discuss your opinion in the conclusion
- Edit the best (pdf) report that resumes the requested content

Task N° 1: JUnit

- For this assignment, you will be adding **JUnit** unit tests to your **Booking.class**. This way, the methods of the class can be tested automatically.
- Use **JUnit of Java (eclipse) or JACOCO environment tools.**
- Adopt the AAA (**Arrange-Act-Assert**) pattern unit test

Details

You have been hired (rented, تم التعاقد معه) to test our new calendar app!

This program allows users to book meetings, adding those meetings to calendars maintained (keep) for rooms and employees. It will actively prevent (stop) multiple bookings, and will manage the busy and open status for employees and rooms.

The system enables the following high-level functions:

- Booking a meeting
- Booking vacation time
- Checking availability for a room
- Checking availability for a person
- Printing the agenda for a room
- Printing the agenda for a person

Normally, actions are conducted through the driver provided by the **main** method in the **PlannerInterface class.** As a tester, you - of course - **have full access to the source cod**e to employ in testing the system. You are to do the following:

1. Formulate a test plan.
   - Given the above features and the code documentation, plan out a series of test cases to ensure that these features can be performed without error.
   - Think about what the "testable units" are.
     * Your tests may use any of the classes in the system, and may be at the

method, class, or system level.

- Make sure you think about both the normal execution and illegal inputs and actions that could be performed.
    * Think of as many things that could go wrong as you can! For instance, you will probably be able to add a normal meeting, but can you add a meeting for February 35th? Try it out.

2. Write tests in the JUnit framework.
    - If a test is supposed to cause an exception to be thrown. Make sure you check for that exception.
    - Make sure that your expected output is detailed enough to ensure that - if something is supposed to fail - that it fails for the correct reasons.

Submitting

For this lab, you should submit the **BookingTest.java** file with your unit testing for the **Booking.class.**

Task N° 2 : TDD

- Reuse the existing previous case study program code, you need now to improve it code.
- Write suitable JUnit assertions that enables testing non-existent methods as: "**to_string ()**", "**add_meting ()**" that have the same name of another one, "**find_meting ()"** in an empty list. **"Count_meting()"** that enables number of meting by: day/week/month/and year
- Propose these new methods by applying the TDD principle.

Task N° 3 : Code Refactoring

- In this part, your are asked to improve your program after task 2 in suitable code parts.
- by adopting **Code Refactoring** techniques as:
    - **Composing Method** (**extract** method)
    - **Red-Green Refactoring**.
    - **Refactoring by Abstraction**.
    - **Preparatory Refactoring**
    - **Simplifying Methods**
    - **Moving Features Between Objects**
- Adjusts the code of this application, using this technics without making any changes to how it functions.

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module: SIA-2**

Teacher: Ouaar Hanane **Year: 2024/2025**

Hands-out N° 2-2: Project management + version control systems Git/Git Hub

---

Purpose: Software tools of a collaborative project management

Using Jira Jira Software or ClickUp as a powerful agile project management tools that offers a wide range of features to help you plan, track and manage your projects.

- Choose a collaborative case study, **argument** your choice
- You have to use the **agile scrum** method
- Edit **Gantt diagram** in order to organize your collaborative task
- Specify the main **Backlogs** and its **sprints**

Requested work: reporting

- Fill the table every week (you have 4 weeks)
- Explain the main idea of the **analysis phase**
- Edit the best diagram of the **design phase**
- Generate the report and dashboard's documentation of **coding** and **testing** phases
- Display the important screen shoots by adding suitable commands (GitHub)
- Draw the global diagram describing **collaboration features branches**
- Discus your opinion in the conclusion
- Edit the best (pdf) report that resumes the study's content

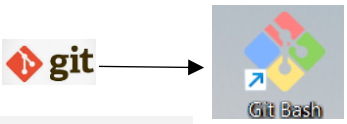
---

Application name:

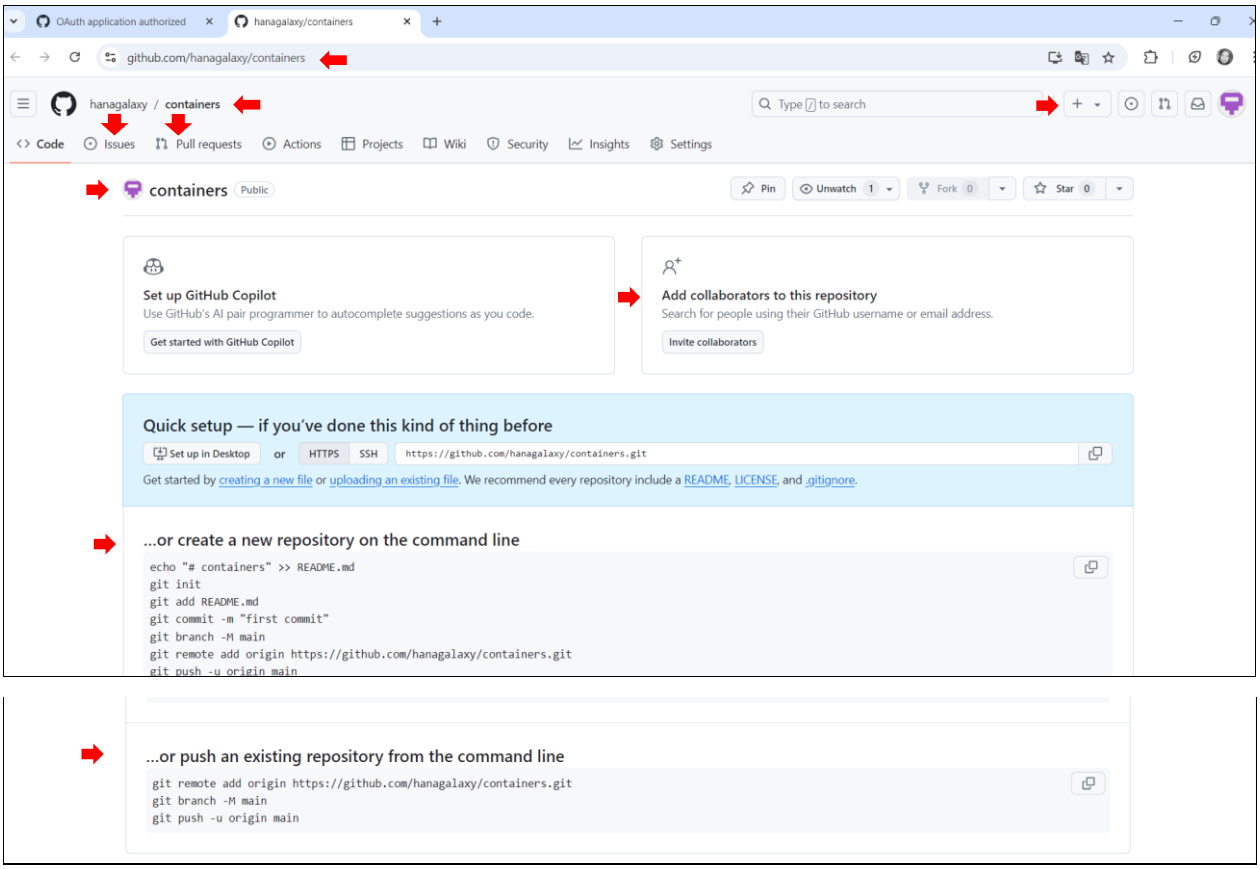
Weeks	Backlog	Sprint	Scrum master	Scrum team	Admin task	End user task
Week 1	Backlog 1	Sprint 1				
		Sprint 2				
		Sprint 3				
	Backlog 2					
	Backlog 3					
Week 2						
Week 3						
Week 4						

Application of version control system: Git and Git-hub

- Install the **Git** tool **:**

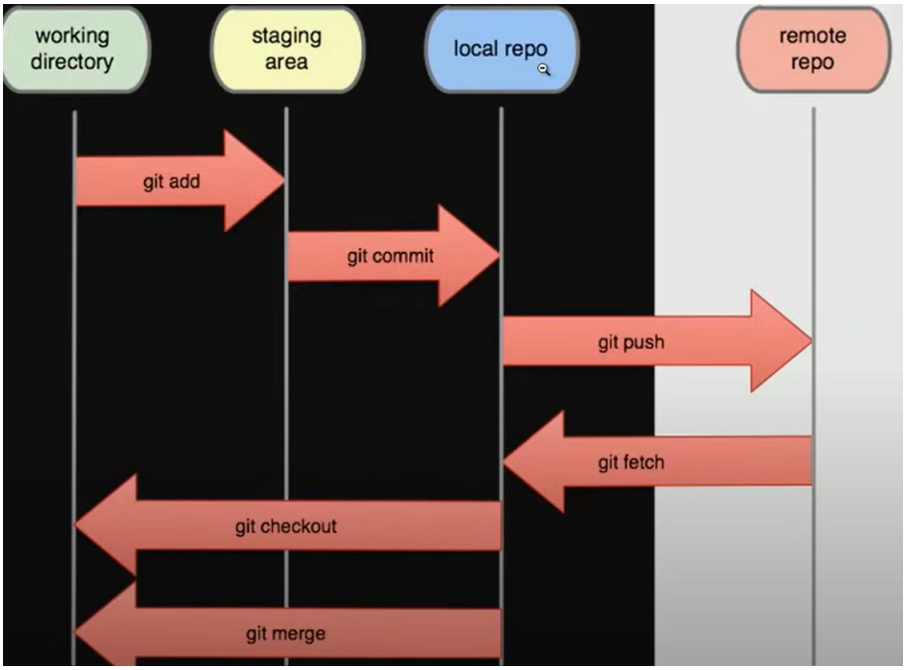


- Go to the **GitHub** platform **:**



- Create an **account** ……. Create a new **repository** > **public or private**
- Is recommended to write in the **read me** file the main goals of your application (.md = Markdown language)

Resume: Manipulated commands:



- **Local poste :** in your PC create a **new directory** that will contained your Git application's files, you can create it either in **windows** or type this command: **mkdir** name_dir, **cd** name_dir
- Type **git init** in order to initialize an **empty repo** in this new location (directory), you get **(master -> origin)** in your command line, it means that it **exists a repository** in this directory.
- **Local poste** :in your directory, Clone the new repo by copying it **link** from GitHub by:

  git clone the GitHub link
- **Working directory:** Create your application's **folders** "css/main.css" and "index.html" (**git status)**
- **Staging area**: Add this folders in your **stage** area **(git add)** is **the tracking, (git status**)
- **You can use :(git reset head)** file name.xxx in order to delete it from the staging area
- **Local repo: (git commit –m** "massage or description"), (git status)
- **Remote repo**: locale branch, **git branch** → Master
- To know your remove name write **git remove-v** → origin
- To publish your locale commit toward the remote GitHub write :

  git push remote_name branch_name

  => **Git push origin master**

  => **login (id, pw)** if the repo is private

- Go to your GitHub GUI, **refresh** your page before, you find the published commit, and check now your files.

- To apply to revers operation (from the remote repo to their locale), go to menu: **settings/collaborators** and **add** the name of your collaborators, => **you receive an e-mail as an invitation to accept it**

- **Change** some program code in files by **another** collaborator, **refresh** the page

- **Copy** the link of the repo, **go** to the **local Git,** type **Git clone** the repo link

- Type **git pull** origin, it gets the new change from remote repo to you, it takes a copy from the branch in the remote repo to you **+ merge** it, **git status**

Configuration: you can change setting at any time like:

Type git config –l or git help config

Example: git config --globla user.email "aaa@bbb.cc"

To show it write: **git config --globla user.email**

Change setting via an editor :**git config --globla --edit**

- Discus and share some issues

Annex :Git Commands

git config

Usage: git config –global user.name "[name]"

Usage: git config –global user.email "[email address]"

This command sets the author name and email address respectively to be used with your commits.

git init

Usage: git init [repository name]

This command is used to start a new repository.

git clone

Usage: git clone [url]

This command is used to obtain a repository from an existing URL.

git add

Usage: git add [file]

This command adds a file to the staging area.

Usage: git add *

This command adds one or more to the staging area.

git commit

Usage: git commit -m "[Type in the commit message]"

This command records or snapshots the file permanently in the version history.

Usage: git commit -a

This command commits any files you've added with the git add command and also commits any files you've changed since then.

git diff

Usage: `git diff`

This command shows the file differences which are not yet staged.

`Usage: git diff –staged`

This command shows the differences between the files in the staging area and the latest version present.

Usage: `git diff [first branch] [second branch]`

This command shows the differences between the two branches mentioned.

git reset

Usage: `git reset [file]`

This command unstages the file, but it preserves the file contents.

Usage: `git reset [commit]`

This command undoes all the commits after the specified commit and preserves the changes locally.

Usage: `git reset –hard [commit]` This command discards all history and goes back to the specified commit.

Learn how to connect Git secrets with a Jenkins pipeline.

git status

Usage: `git status`

This command lists all the files that have to be committed.

git rm

Usage: `git rm [file]`

This command deletes the file from your working directory and stages the deletion.

git log

Usage: `git log`

This command is used to list the version history for the current branch.

Usage: `git log –follow[file]`

This command lists version history for a file, including the renaming of files also.

git show

Usage: `git show [commit]`

This command shows the metadata and content changes of the specified commit.

git tag

Usage: `git tag [commitID]`

This command is used to give tags to the specified commit.

git branch

Usage: `git branch`

This command lists all the local branches in the current repository.

Usage: `git branch [branch name]`

This command creates a new branch.

Usage: `git branch -d [branch name]`

This command deletes the feature branch.

git checkout

Usage: `git checkout [branch name]`

This command is used to switch from one branch to another.

Usage: `git checkout -b [branch name]`

This command creates a new branch and also switches to it.

git merge

Usage: `git merge [branch name]`

This command merges the specified branch's history into the current branch.

git remote

Usage: `git remote add [variable name] [Remote Server Link]`

This command is used to connect your local repository to the remote server.

git push

Usage: `git push [variable name] master`

This command sends the committed changes of master branch to your remote repository.

Usage: `git push [variable name] [branch]`

This command sends the branch commits to your remote repository.

Usage: `git push –all [variable name]`

This command pushes all branches to your remote repository.

Usage: `git push [variable name] :[branch name]`

This command deletes a branch on your remote repository.

git pull

Usage: `git pull [Repository Link]`

This command fetches and merges changes on the remote server to your working directory.

git stash

Usage: `git stash save`

This command temporarily stores all the modified tracked files.

Usage: `git stash pop`

This command restores the most recently stashed files.

Usage: `git stash list`

This command lists all stashed changesets.

Usage: `git stash drop`

This command discards the most recently stashed changeset.

Chapter 3:

The return of patterns

Chapter outline

Introduction

1. **Gangs of Four (GoF) Design Patterns**
2. **Classification of Design patterns**
   - **2.1Creational Design Patterns**
   - **2.2Structural Design Patterns**
   - **2.3Behavior Design Patterns**
3. **Design Patterns objectifs**
4. **Design patterns principles**
   - **4.1 Encapsulation**
   - **4.2 Abstraction**
   - **4.3 Inheritance**
   - **4.4 Polymorphism**
5. **J2EE design patterns**
   - **5.1 Design Patterns in J2EE**
   - **5.2 Integration with J2EE Technologies and Frameworks**
6. **Best Practices and Implementation Guidelines**
7. **Design patterns benefits**
8. **Criticism**

Conclusion

1. Introduction

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns can speed up the development process by providing tested, proven (مؤكد) development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems.

Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs

2. Gangs of Four (GoF) Design Patterns

2.1 Historic:

Gangs of Four Design Patterns is the collection of 23 design patterns from the book “Design Patterns: Elements of Reusable Object-Oriented Software”. This book was first published in 1994 and it’s one of the most popular books to learn design patterns. The book was authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. It got nicknamed as Gangs of Four design patterns because of four authors. Furthermore, it got a shorter name as “GoF Design Patterns”. The book was published over 20 years ago, it still continues to be an Amazon best seller.

The GoF wrote the book in a C++ context but it still remains very relevant to Java programming. C++ and Java are both object-oriented languages. The GoF authors, through their experience in coding large scale enterprise systems using C++, saw common patterns emerge. These design patterns are not unique to C++. The design patterns can be applied in any **object oriented** language.

As a Java developer using the Spring Framework to develop enterprise class applications, you will encounter the GoF Design Patterns on a daily basis

2.2 Definition

- Design patterns in software engineering are typical solutions to common problems in software design. They represent best practices, evolved over time, and are a toolkit for software developers to solve common problems efficiently.
- Design patterns are reusable solutions to common problems in software design. They represent best practices used by experienced object-oriented software developers. Design patterns provide a standard terminology and are specific to particular scenarios.

2.3 Motivation

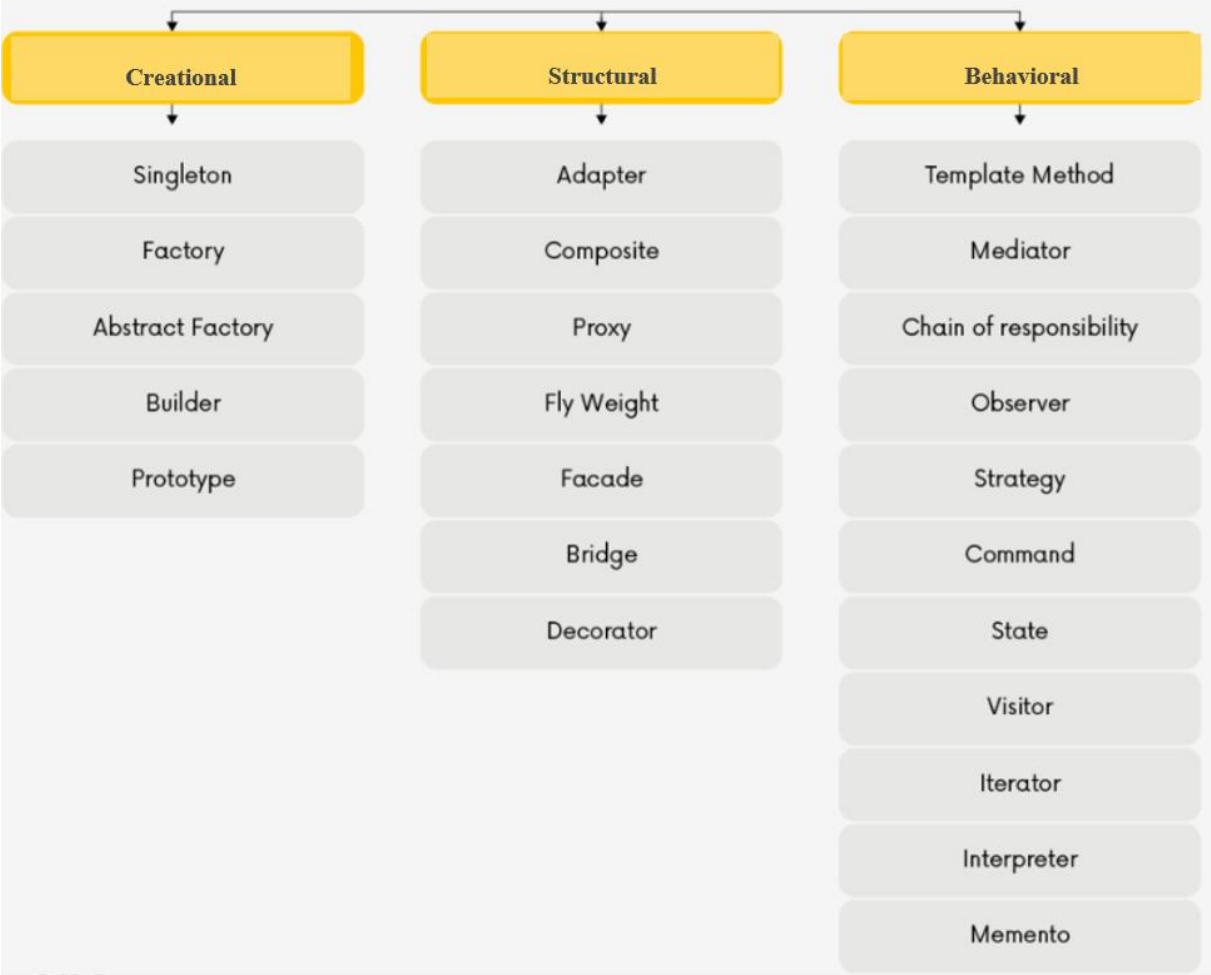
There are several reasons to learn design patterns:

- **Improve Code Quality**: Design patterns help in creating code that is easier to understand, maintain, and extend. They promote best practices and provide solutions that have been tested and proven effective.
- **Enhance Problem-Solving Skills**: Learning design patterns equips developers with a book of standard solutions to common problems. This enables them to quickly and effectively address similar challenges in various projects.
- **Promote Reusability and Efficiency:** By applying design patterns, developers can create reusable components that can be used across multiple projects. This reduces redundancy and saves development time.
- **Learn from Experts**: Design patterns are derived from the collective experience of skilled developers and architects. Learning these patterns allows developers to benefit from the wisdom (حكمة) and insights (رؤى) of industry experts.

3 Classification of Design patterns:

The GoF Design Patterns are broken into three categories: Creational Patterns for the creation of objects; Structural Patterns to provide relationship between objects; and finally, Behavioral Patterns to help define how objects interact, also:

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.



3.1Creational Design Patterns

- **Abstract Factory.** Allows the creation of objects without specifying their concrete type.
- **Builder.** Uses to create complex objects.
- **Factory Method.** Creates objects without specifying the exact class to create.
- **Prototype**. Creates a new object from an existing object.
- **Singleton**. Ensures only one instance of an object is created.

3.2Structural Design Patterns

- **Adapter**. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
- **Bridge**. Decouples an abstraction so two classes can vary independently.
- **Composite**. Takes a group of objects into a single object.
- **Decorator**. Allows for an object's behavior to be extended dynamically at run time.
- **Facade**. Provides a simple interface to a more complex underlying object.
- **Flyweight**. Reduces the cost of complex object models.

- **Proxy**. Provides a placeholder interface (espace réservé) to an underlying object to control access, reduce cost, or reduce complexity.

3.3Behavior Design Patterns

- **Chain of Responsibility.** Delegates commands to a chain of processing objects.
- **Command**. Creates objects which encapsulate actions and parameters.
- **Interpreter**. Implements a specialized language.
- **Iterator**. Accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator**. Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento**. Provides the ability to restore an object to its previous state.
- **Observer**. Is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State**. Allows an object to alter its behavior when its internal state changes.
- **Strategy**. Allows one of a family of algorithms to be selected on-the-fly at run-time.
- **Template Method.** Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
- **Visitor**. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

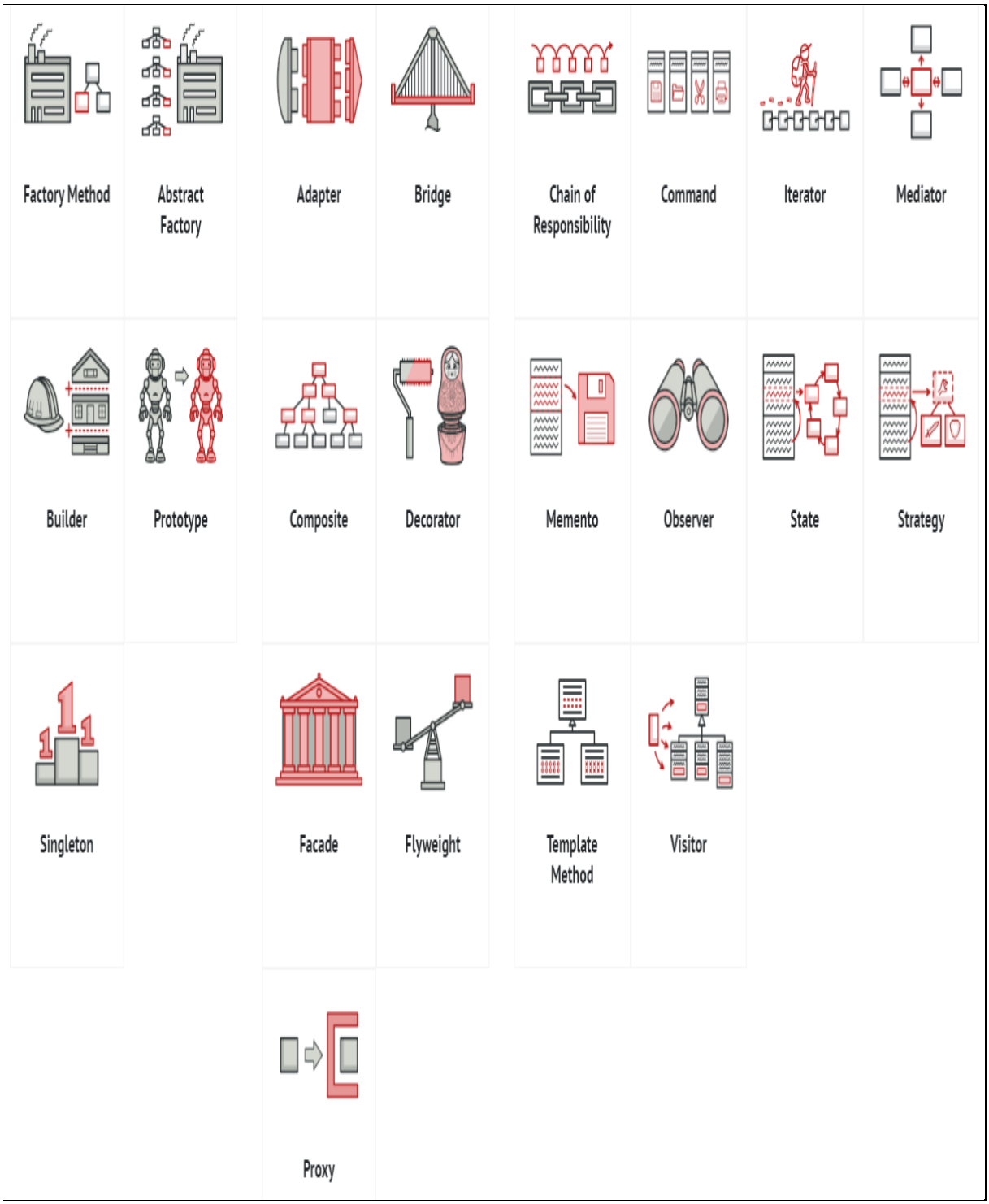


Figure 1: Common Design Patterns

Examples: visit this links **https://refactoring.guru/design-patterns/java**

https://medium.com/@saygiligozde/design-patterns-in-java-5251032ca244

4 Design Patterns objectifs

- **Solving Recurring Problems**: Design patterns offer solutions to problems that software developers encounter repeatedly. Instead (بدلاً من) of reinventing the wheel for each project, engineers can use these patterns as blueprints (المخططات)for solving common design issues.
- **Facilitating Communication**: Design patterns provide a shared language for software developers. When a developer mentions a specific pattern like 'Singleton' or 'Observer,' their peers immediately understand the design concept being discussed. This shared vocabulary facilitates clearer and more efficient communication among team members.
- **Enhancing Code Reusability and Maintainability**: By using established patterns, developers create code that is more modular and easier to maintain. Patterns encourage the principle of 'Don't Repeat Yourself' (DRY), which leads to less redundant code and easier updates and maintenance.
- **Improving Scalability**: Design patterns often incorporate principles that make it easier to scale applications. For instance, patterns that emphasize modularity and loose coupling allow systems to grow without significant rework.
- **Facilitating Best Practices**: Design patterns encapsulate best practices derived from experienced developers and designers. They guide less experienced developers in using methods that have proven effective over time.

4.1 Examples

Examples 1: For example, the Singleton pattern ensures that a class has only one instance and provides a global point of access to it, optimizing resource usage and consistency.

In this example, the `Singleton` class overrides the `_ _new_ _` method, which is responsible for creating a new instance of the class. The first time the `Singleton` class is instantiated, it creates a new instance. On subsequent instantiations, it returns the already created instance. This ensures that there's only one instance of the `Singleton` class at any given time, thereby adhering (ملتصق) to the Singleton pattern.

```
class Singleton:
    _instance = None

    def _ _new_ _(cls):
        if cls._instance is None:
            print("Creating the instance")
            cls._instance = super(Singleton, cls)._ _new_ _(cls)
        return cls._instance
```

```
# Usage:
singleton1 = Singleton()
print("Instance 1:", singleton1)

singleton2 = Singleton()
print("Instance 2:", singleton2)

# Both instances will be the same
assert singleton1 is singleton2
```

Example 2: Consider an application that requires different types of notifications to be sent out, such as email, SMS, and push notifications. Implementing this without a design pattern might lead to a complex if-else or switch-case block that selects and creates the appropriate notification type. This approach, however, is not scalable and violates the Open-Closed Principle (software entities should be open for extension but closed for modification). We could apply — "**Factory Method Pattern**" to solve the above problem. The Factory Method Pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.

```
class Notification:
    def send(self):
        pass

class EmailNotification(Notification):
    def send(self):
        return "Sending Email Notification"

class SMSNotification(Notification):
    def send(self):
        return "Sending SMS Notification"

class NotificationFactory:
    @staticmethod
    def create_notification(type):
        if type == 'email':
            return EmailNotification()
        elif type == 'sms':
            return SMSNotification()
        else:
            raise ValueError('Notification type not supported')

# Client Code
notification_type = 'email'  # This can be dynamically determined
notification =
NotificationFactory.create_notification(notification_type)
print(notification.send())
```

In this example, the `NotificationFactory` class encapsulates the logic for creating different types of notification objects. This makes the code more modular, easier to maintain, and extend. For instance, adding a new notification type like 'PushNotification' would only require adding a new subclass and updating the factory method, without modifying the existing client code.

5 Design patterns principles

The four key principles central to design patterns — **encapsulation**, **abstraction**, **inheritance**, and **polymorphism** — are foundational concepts in object-oriented programming (OOP). Each plays a crucial role in creating efficient, reusable, and maintainable software. Let's explore each principle with examples:

5.1 Encapsulation:

Encapsulation is the bundling of data and methods that operate on that data within one unit, typically a class in OOP. It restricts direct access to some of an object's components, which is a means of preventing (منع) accidental interference and misuse (سوء الاستخدام) of the methods and data.

Example: Consider a class `BankAccount`. Encapsulation allows us to hide the account balance from direct access, exposing only methods to `deposit` and `withdraw`.

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # private variable

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return self.__balance

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return self.__balance

# Usage
account = BankAccount()
account.deposit(100)
print(account.withdraw(50))  # Output: 50
```

In this example, `__balance` is a private attribute. It cannot be accessed directly from outside the class, which prevents unauthorized operations on the account balance.

5.2 Abstraction

Abstraction involves hiding complex implementation details and showing only the necessary features of the object. It helps in reducing programming complexity and effort.

Example: Imagine a class `Car` that exposes a method `drive`. The `drive` method abstracts away the complexities of the internal workings of the car, like fuel (carburant) injection, gear shifting (changement de vitesse), etc.

```
class Car:
    def drive(self):
        self.start_engine()
        self.change_gears()
        # More complex steps
        print("Car is moving")

    def start_engine(self):
        # Complex engine start procedures
        pass

    def change_gears(self):
        # Complex gear changing procedures
        pass

# Usage
my_car = Car()
my_car.drive()  # Output: Car is moving
```

Users of the `Car` class need not understand how the engine starts or gears change; they just need to know how to call the `drive` method.

5.3 Inheritance

Inheritance allows a class to inherit properties and methods from another class. It promotes code reuse and establishes a relationship between classes (parent and child classes).

Example: Let's say we have a parent classé `Vehicle` and two child classes `Car` and `Motorcycle`. Both child classes inherit common properties from `Vehicle`.

```
class Vehicle:
    def has_wheels(self):
        return True

class Car(Vehicle):
    def number_of_wheels(self):
        return 4
```

```
class Motorcycle(Vehicle):
    def number_of_wheels(self):
        return 2

# Usage
car = Car()
motorcycle = Motorcycle()
print(car.has_wheels())  # Output: True
print(motorcycle.number_of_wheels())  # Output: 2
```

Both `Car` and `Motorcycle` inherit the `has_wheels` method from `Vehicle`.

5.4 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is the ability to present the same interface for differing underlying forms (data types).

Example: Using the previous example of `Vehicle`, `Car`, and `Motorcycle`, we can demonstrate polymorphism by writing a function that operates on any `Vehicle` object.

```
def vehicle_info(vehicle):
    print("Has wheels:", vehicle.has_wheels())
    print("Number of wheels:", vehicle.number_of_wheels())

vehicle_info(car)  # Works with a Car object
vehicle_info(motorcycle)  # Works with a Motorcycle object
```

Here, `vehicle_info` function can work with any object of `Vehicle` or its subclasses, demonstrating polymorphism

Remark: Each of these principles plays a vital role in effective software design, contributing to the development of systems that are more robust, flexible, and maintainable. By adhering (ملتصق) to these principles, developers can create software that effectively addresses the needs of users while remaining adaptable to changing requirements.

6 J2EE (Java 2 Enterprise Edition) design patterns

6.1 Principle

J2EE Design Patterns are reusable solutions to common design problems encountered when developing enterprise-level Java applications using the Java 2 Platform, Enterprise Edition (J2EE). These patterns provide developers with proven techniques and best practices for designing scalable, maintainable, and robust software systems.

Version <=6 is called **J2EE (by oracle)**

Version >=7 is called **java EE (by eclipse)**

6.2 Design Patterns in J2EE (Java 2 Enterprise Edition)

In Java 2 Enterprise Edition (J2EE, now Jakarta EE), design patterns play a crucial role in designing robust and maintainable enterprise applications. They provide proven solutions to common problems encountered in software design. Here's an overview of some key design patterns used in J2EE:

- **Model-View-Controller (MVC)**:
  - MVC is a design pattern used to separate the concerns of an application into three main components: Model (business logic and data), View (presentation layer), and Controller (handles user input).
  - In J2EE, frameworks like Struts and JSF (JavaServer Faces) implement the MVC pattern.
- **Data Access Object (DAO)**:
  - The DAO pattern is used to separate the data access logic from the business logic.
  - It provides an abstract interface to access data from a data source (e.g., database) without exposing the underlying details.
  - J2EE applications commonly use DAOs to interact with databases using JDBC or JPA.
- **Factory Method:**
  - The Factory Method pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
  - In J2EE, this pattern is often used to create instances of classes that implement business logic, such as session beans or entity beans.
- **Singleton:**
  - The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

- In J2EE, Singletons are commonly used for managing resources that need to be shared across the application, such as database connections or configuration settings.

- **Session Facade:**
  - The Session Facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes it easier to use the subsystem and reduces the complexity of client code.
  - In J2EE, session beans are often used to implement the Session Facade pattern.
- **Front Controller:**
  - The Front Controller pattern is used to centralize request handling in a web application.
  - In J2EE, frameworks like Struts and Spring MVC provide implementations of the Front Controller pattern, where a single servlet or controller handles incoming requests and dispatches them to the appropriate handlers.
- **Dependency Injection (DI):**
  - Dependency Injection is a design pattern used to remove hard-coded dependencies between components, making the system more flexible and easier to test.
  - In J2EE, frameworks like Spring provide DI capabilities, allowing developers to inject dependencies into their components using annotations or XML configuration.

These are just a few examples of the design patterns commonly used in J2EE. Each pattern addresses specific design problems and helps developers build more maintainable and scalable enterprise applications.

6.3 Integration Technologies and Frameworks with J2EE

Integration with J2EE technologies and frameworks is crucial for building enterprise applications. Here are some key integration points and commonly used technologies and frameworks:

- **Database Integration**
  - **JDBC (Java Database Connectivity):** Standard API for connecting Java applications to databases. Use JDBC to interact with relational databases.
  - **JPA (Java Persistence API):** Standard API for object-relational mapping in Java applications. Use JPA with ORM (Object-Relational Mapping) frameworks like Hibernate or EclipseLink for database integration.

- **Web Services**
    - **JAX-RS (Java API for RESTful Web Services):** API for building RESTful web services in Java. Use JAX-RS implementations like Jersey or RESTEasy for web service integration.
    - **JAX-WS (Java API for XML Web Services):** API for building SOAP-based web services in Java. Use JAX-WS implementations like Apache CXF or Metro for SOAP web service integration.
- **Messaging**
    - **JMS (Java Message Service):** API for asynchronous messaging between applications. Use JMS with messaging brokers like Apache ActiveMQ or RabbitMQ for messaging integration.
    - **Java EE Connector Architecture (JCA):** Provides a standard architecture for integrating Java EE applications with heterogeneous EIS (Enterprise Information System) providers such as ERP systems or databases.
- **Security**
    - **JAAS (Java Authentication and Authorization Service):** API for implementing authentication and authorization in Java applications. Use JAAS for security integration.
    - **Java EE Security:** Use Java EE security features such as declarative security, SSL/TLS, and encryption for securing your application.
- **Dependency Injection**
    - **CDI (Contexts and Dependency Injection):** Standard for dependency injection in Java EE applications. Use CDI for managing dependencies and integrating with other Java EE components.
- **Web Frameworks**
    - **JSF (JavaServer Faces):** Component-based web framework for building user interfaces. Use JSF for web application integration.
    - **Struts:** MVC framework for building web applications. Use Struts for web application integration.
- **Integration Frameworks:**
    - **Spring Framework:** Comprehensive framework for building Java applications. Use Spring for dependency injection, transaction management, and other integration aspects.

- **Apache Camel:** Integration framework for routing and mediation rules. Use Camel for integrating systems using different protocols and data formats.

These technologies and frameworks provide various integration points for building robust and scalable enterprise applications in the J2EE ecosystem

7 Best Practices and Implementation Guidelines

Implementing enterprise applications involves a mix of technology, architecture, and development practices. Here are some best practices and guidelines to consider:

- **Use Design Patterns:** Apply design patterns appropriately to address common design problems and improve the maintainability of your code.
- **Follow SOLID Principles:** Ensure your code is designed to be:
  - S: Single Responsibility Principle
  - O: Open/Closed Principle
  - L: Liskov Substitution Principle
  - I: Interface Segregation Principle
  - D: Dependency Inversion Principle
- **Separation of Concerns:** Keep different aspects of your application (e.g., presentation, business logic, data access) separate to improve maintainability and testability.
- **Unit Testing:** Write unit tests for your code to ensure its correctness and help with refactoring. Consider using tools like JUnit, NUnit, or MSTest.
- **Integration Testing:** Test the interaction between different components of your application to ensure they work together as expected. Use tools like Selenium for web applications or Postman for APIs.
- **Continuous Integration/Continuous Deployment (CI/CD):** Automate the build, test, and deployment processes to improve efficiency and reduce the risk of errors. These best practices and guidelines can help you build robust, maintainable, and scalable enterprise applications. However, it's important to adapt them to fit the specific requirements and constraints of your project

8 Design patterns benefits

Design patterns are a valuable tool in software development, and they offer various benefits and uses, some of them are explained below :

- **Enhancing Maintainability**:
  - Design patterns help organize code in a structured and consistent way. This makes it easier to maintain, update, and extend the codebase. Developers familiar with the patterns can quickly understand and work on the code.

- **Promoting Code Reusability**:
    - Design patterns encapsulate solutions to recurring design problems. By using these patterns, we can create reusable templates for solving specific problems in different parts of your application.
- **Simplifying Complex Problems**:
    - Complex software problems can be broken down into smaller, more manageable components using design patterns. This simplifies development by addressing one problem at a time and, in turn, makes the code more maintainable.
- **Improving Scalability**:
    - Design patterns, particularly structural patterns, allow us to create a flexible and extensible architecture, making it easier to add new features or components.
- **Improving Testability**:
    - Code designed with patterns in mind is often more modular and easier to test. we can write unit tests for individual components or classes, leading to more reliable and robust software.
- **Supporting Cross-Platform Development**:
    - Design patterns are not tied to a specific programming language or platform. They are general guidelines that can be applied across different technologies, making it easier to adapt your code to different environments.
- **Enhancing Collaboration**:
    - Design patterns provide a common language and a shared understanding among team members. They enable developers to communicate effectively and collaborate on software projects by referring to well-known design solutions.

9 Criticism

The concept of design patterns has been criticized by some in the field of computer science.

- ***Targets the wrong problem :***

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely (فقط) referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay **Revenge of the Nerds**.

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

- ***Lacks formal foundations***

The study of design patterns has been excessively (بشكل مفرط) ad hoc (مخصصة), and some have argued that the concept sorely (بشدة) needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial (محاكمة), in which they were "charged" with numerous crimes against computer science. They were "convicted"(مدان) by ⅔ of the "jurors" (المحلفون) who attended the trial.

- ***Leads to inefficient solutions***

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

- ***Does not differ significantly from other abstractions***

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

*** Are Design Patterns Still Relevant:** Despite the evolution of technology and the emergence of new programming paradigms, the core problems that design patterns address, such as modularity, maintainability, and code reusability, remain. While the specific implementations of these patterns might evolve with new technologies, the underlying principles continue to be relevant. For instance, in microservices architecture, patterns like Proxy, Circuit Breaker, or API Gateway are crucial for handling distributed system concerns.

10. Application :

10.1 Creational Design Patterns

Example: Factory Method Pattern (Java)

Purpose

The **Factory Method Pattern** defines an interface for creating objects, but lets **subclasses decide** which class to instantiate.

It helps you **avoid direct use of `new`**, making your code more **flexible and extensible**.

```java
// Step 1: Create an interface (or abstract product)
interface Shape {
    void draw();
}

// Step 2: Create concrete classes implementing the same interface
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

// Step 3: Create a Factory class to generate objects
class ShapeFactory {

    // Factory Method
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
```

```java
// Step 4: Use the factory to create objects instead of using new directly
public class Main {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        Shape shape1 = factory.getShape("CIRCLE");
        shape1.draw(); // Output: Drawing a Circle

        Shape shape2 = factory.getShape("SQUARE");
        shape2.draw(); // Output: Drawing a Square
    }
}
```

Explanation

Step	Description
1	`Shape` is the **interface** (the product).
2	`Circle` and `Square` are **concrete products** implementing `Shape`.
3	`ShapeFactory` is the **factory** responsible for object creation.
4	The client (`Main`) calls the **factory method** instead of using `new`.

This makes it easy to add new shapes (like `Triangle`) without changing existing code.

Advantages

- **Encapsulates object creation logic** in one place.
- **Reduces code duplication**.
- **Makes code more flexible** — you can add new product types easily.

Disadvantages

- May require more classes.
- Sometimes adds slight complexity if overused.

10.2 Structural Design Patterns

Example: Decorator Pattern (Java)

Purpose

The **Decorator Pattern** allows you to **add new behaviors** or **responsibilities** to objects **dynamically**, **without modifying their original code**.

Think of it like **decorating a cake** — you start with a basic cake and then add layers, icing, and toppings, each enhancing it without changing the cake itself.

```java
// Step 1: Component interface
interface Coffee {
    String getDescription();
    double getCost();
}

// Step 2: Concrete Component
class SimpleCoffee implements Coffee {
    public String getDescription() {
        return "Simple Coffee";
    }

    public double getCost() {
        return 2.0;
    }
}

// Step 3: Abstract Decorator (implements the same interface)
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    public double getCost() {
        return decoratedCoffee.getCost();
```

```
// Step 4: Concrete Decorators (add extra features)
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

```java
// Step 5: Client code
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " → $" + coffee.getCost());

        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " → $" + coffee.getCost());

        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " → $" + coffee.getCost());
    }
}
```

Output

```
Simple Coffee → $2.0
Simple Coffee, Milk → $2.5
Simple Coffee, Milk, Sugar → $2.7
```

Explanation

Step	Description
1	`Coffee` is the **component interface**, defining base behavior.
2	`SimpleCoffee` is the **concrete component** (the base object).
3	`CoffeeDecorator` is the **abstract decorator**, holding a reference to a `Coffee` object.
4	`MilkDecorator` and `SugarDecorator` are **concrete decorators** that add new behavior.
5	The **client** (in `Main`) wraps objects dynamically to add features at runtime.

Advantages

- Adds functionality **without modifying existing code**.
- Promotes **code reuse** and follows **Open/Closed Principle**.
- You can **combine multiple decorators** flexibly.

Disadvantages

- Can lead to **many small classes**.
- Debugging can be harder due to **many layers of wrapping**.

10.3 Behavior Design Patterns

Example: Strategy Pattern (Java)

💡 Purpose

The **Strategy Pattern** defines a **family of algorithms**, encapsulates each one, and makes them **interchangeable**.
This pattern lets the algorithm vary **independently** from the clients that use it.

It's like having multiple **strategies** for solving a problem — and being able to **choose which one** to use at runtime.

Java Code Example

```java
// Step 1: Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Step 2: Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(int amount) {
        System.out.println("💳 Paid " + amount + " using Credit Card: " + cardNumber);
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    public void pay(int amount) {
        System.out.println("💻 Paid " + amount + " using PayPal account: " + email);
    }
}

// Step 3: Context class (uses a PaymentStrategy)
class ShoppingCart {
    private PaymentStrategy paymentStrategy;
```

```
    // The strategy can be changed at runtime
    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        if (paymentStrategy == null) {
            System.out.println("⚠️ No payment method selected!");
        } else {
            paymentStrategy.pay(amount);
        }
    }
}

// Step 4: Test the Strategy Pattern
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        // Using Credit Card
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9999"));
        cart.checkout(100);

        // Switching to PayPal
        cart.setPaymentStrategy(new PayPalPayment("user@example.com"));
        cart.checkout(200);
    }
}
```

Output

```
💳 Paid 100 using Credit Card: 1234-5678-9999
💻 Paid 200 using PayPal account: user@example.com
```

Explanation

Step	Description
1	`PaymentStrategy` defines a **common interface** for all payment methods.
2	`CreditCardPayment` and `PayPalPayment` are **concrete strategies**, each implementing the algorithm differently.
3	`ShoppingCart` is the **context** — it uses a `PaymentStrategy` and can switch strategies at runtime.
4	The **client (Main)** can easily choose or change the payment method **without modifying** the cart class.

Advantages

- Promotes **flexibility and reusability** — new strategies can be added easily.
- Follows the **Open/Closed Principle** (open for extension, closed for modification).
- Makes the algorithm **interchangeable** at runtime.

Disadvantages

- Slightly increases the number of classes.
- The client must understand **which strategy to choose**.

11. Conclusion

Design patterns are like smart and efficient recipes for coding in the world of software development. They help us solve common problems and build software that works well, is easy to update, and can handle changes without breaking. The Gangs of Four design patterns lay the foundation of core design patterns in programming. There are many other design patterns built on top of these patterns for specific requirements.

Building enterprise applications in the J2EE (Java 2 Enterprise Edition, now Jakarta EE) ecosystem involves leveraging a variety of technologies, frameworks, and design patterns. By following best practices, such as using design patterns to address common challenges, following SOLID principles, and implementing robust testing and deployment strategies, developers can create scalable, maintainable, and secure applications.

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module: SIA-2**

Teacher: Ouaar Hanane **Year: 2024/2025**

Hands-out N° 3.1: Design Patterns applications

1- Creational Design Patterns

Example: Singleton Pattern

Purpose

The **Singleton pattern** ensures that **only one instance** of a class exists in the entire application and provides a **global point of access** to it.

It's useful for things like:

- Database connections
- Logging
- Configuration management

Advantages

- Ensures **controlled access** to a single instance.
- Saves **memory** by avoiding multiple copies of the same object.
- Provides a **global point of access**.
- Ensures **only one object** is created.
- Useful for shared resources (e.g., logging, configuration, database connections).

Drawback

- Can make **unit testing** harder since it introduces **global state**.
- May introduce **tight coupling** if overused.

```java
// Singleton class
public class DatabaseConnection {
    // Step 1: Create a private static instance of the class
    private static DatabaseConnection instance;

    // Step 2: Make the constructor private to prevent external instantiation
    private DatabaseConnection() {
        System.out.println("Database Connection created!");
    }

    // Step 3: Provide a public static method to get the single instance
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection(); // create only once
        }
        return instance;
    }

    // Example method
    public void connect() {
        System.out.println("Connected to database.");
    }
}
```

```java
// Test class
public class Main {
    public static void main(String[] args) {
        DatabaseConnection db1 = DatabaseConnection.getInstance();
        DatabaseConnection db2 = DatabaseConnection.getInstance();

        db1.connect();

        // Checking if both references point to the same object
        System.out.println(db1 == db2); // Output: true
    }
}
```

Explanation

Step	Description
1	The static variable `instance` holds the single object.
2	The constructor is **private**, preventing other classes from creating new objects.
3	The method `getInstance()` checks if an instance exists — if not, it creates one, otherwise it returns the existing instance.
4	Both `db1` and `db2` point to the **same object**, proving only one instance exists.

Example: Adapter Pattern (Java)

Purpose

The **Adapter Pattern** allows **incompatible interfaces** to work together.
It acts as a **bridge** between two classes that couldn't otherwise communicate.

Think of it like a **power plug adapter** — it adapts one type of plug to fit another socket.

```java
// Step 1: Target interface (the interface the client expects)
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Step 2: Adaptee class (has a different interface)
class AdvancedMediaPlayer {
    void playMp4(String fileName) {
        System.out.println("Playing MP4 file: " + fileName);
    }

    void playVlc(String fileName) {
        System.out.println("Playing VLC file: " + fileName);
    }
}

// Step 3: Adapter class (makes AdvancedMediaPlayer compatible with MediaPlayer)
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedPlayer = new AdvancedMediaPlayer();

    @Override
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(fileName);
        } else if (audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer.playVlc(fileName);
        } else {
            System.out.println("Invalid media type: " + audioType);
        }
    }
}
```

```java
// Step 4: Concrete class using the adapter
class AudioPlayer implements MediaPlayer {
    private MediaAdapter adapter;

    @Override
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing MP3 file: " + fileName);
        } else {
            adapter = new MediaAdapter();
            adapter.play(audioType, fileName);
        }
    }
}

// Step 5: Test the Adapter pattern
public class Main {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "song1.mp3");
        audioPlayer.play("mp4", "movie.mp4");
        audioPlayer.play("vlc", "clip.vlc");
        audioPlayer.play("avi", "trailer.avi"); // unsupported
    }
}
```

Explanation

Step	Description
1	`MediaPlayer` is the **target interface** expected by the client.
2	`AdvancedMediaPlayer` is the **adaptee** with a different interface.
3	`MediaAdapter` acts as a **bridge**, adapting `AdvancedMediaPlayer` to `MediaPlayer`.
4	`AudioPlayer` uses the adapter when it needs to play non-MP3 formats.
5	The client code (`Main`) interacts only with `AudioPlayer` — it doesn't need to know about adapters or formats.

Advantages

- Allows **reuse of existing classes** with incompatible interfaces.
- **Decouples** client code from implementation details.
- Follows the **Open/Closed Principle** (you can add new adapters without changing existing code).

Disadvantages

- Can add **extra complexity** with many adapters.
- May increase **indirection**, slightly impacting performance.

3- Behavioral Design Pattern

Example: Observer Pattern (Java)

Purpose

The **Observer Pattern** defines a **one-to-many relationship** between objects, so that when **one object (the subject)** changes its state, **all its dependents (observers)** are notified automatically.

It's used in systems where objects need to **react to events** — for example, GUI buttons, event listeners, or live data updates.

Java Code Example

```java
import java.util.ArrayList;
import java.util.List;

// Step 1: Subject (the one being observed)
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private String state;

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

    public void setState(String state) {
        this.state = state;
        notifyAllObservers();
    }

    public String getState() {
        return state;
    }

    private void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}
```

```java
interface Observer {
    void update(String newState);
}

// Step 3: Concrete Observers
class EmailObserver implements Observer {
    public void update(String newState) {
        System.out.println("📧 Email Notification: Subject changed to " + newState);
    }
}

class SMSObserver implements Observer {
    public void update(String newState) {
        System.out.println("📱 SMS Notification: Subject changed to " + newState);
    }
}

// Step 4: Test the Observer Pattern
public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();

        Observer email = new EmailObserver();
        Observer sms = new SMSObserver();

        subject.attach(email);
        subject.attach(sms);

        System.out.println("🔹 Changing state to 'ACTIVE'");
        subject.setState("ACTIVE");

        System.out.println("\n🔹 Changing state to 'INACTIVE'");
        subject.setState("INACTIVE");
    }
```

Output

```
🔹 Changing state to 'ACTIVE'
📧 Email Notification: Subject changed to ACTIVE
📱 SMS Notification: Subject changed to ACTIVE

🔹 Changing state to 'INACTIVE'
📧 Email Notification: Subject changed to INACTIVE
📱 SMS Notification: Subject changed to INACTIVE
```

Explanation

Step	Description
1	`Subject` is the **observable object** that keeps a list of observers and notifies them when its state changes.
2	`Observer` is an **interface** defining the `update()` method.
3	`EmailObserver` and `SMSObserver` are **concrete observers** that react differently to updates.
4	The **client (Main)** changes the subject's state, triggering notifications to all observers.

Advantages

- **Loose coupling** between subject and observers.
- Easy to **add or remove observers** at runtime.
- Promotes **event-driven programming**.

Disadvantages

- If there are many observers, **updates may be slow**.
- Observers need to handle **unexpected update orders** carefully.

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module: SIA-2**

Teacher: Ouaar Hanane **Year: 2024/2025**

Hands-out N° 3.2: Design Patterns

Requested Task 1: J2EE

1. Design Patterns in J2EE (Java 2 Enterprise Edition)

- Using J2EE design patterns tool
- Choose an example (a package and its classes) on Java
- Select the design pattern the most suitable to develop your example:

Design pattern	Argument	Example (code)
Model-View-Controller (MVC)		
Data Access Object (DAO)		
Data Access Object (DAO)		
Singleton		
Session Façade		
Front Controller		
Dependency Injection (DI)		

2. Integration Technologies and Frameworks with J2EE

- Complete your example by using the best framework

Technologies/ Frameworks	Choice	Example (code)
Database Integration (JDBC/ JPA)		
Web Services (JAX-RS/ JAX-WS)		
Messaging (JMS/ JCA)		
Security (JAAS/ Java EE Security)		
Dependency Injection (CDI)		
Web Frameworks (JSF/ Struts)		
Integration Frameworks (Spring Framework/ Apache Camel)		

- Fill the tow tables
- Argument every choice
- Implement all application parts
- Edit the report that resuming your work

Application: Design patterns

Here you will find code examples on how to use design patterns in the software development process.

Example 1:

In the first case, we will use four types of software design patterns to solve the problem of creating different vehicle objects with specific behaviors.

• The Builder design pattern obtained with the @SuperBuilder annotation is used to create Builder methods during inheritance. The @Builder pattern from Lombok does not apply here.

• The Factory design pattern used in the VehicleFactory class to produce object types based on the name.

• The Factory methods (manufacturing methods) defined with the VehicleCreator classes contain details about the manufacturing of individual vehicles.

• The Strategic design pattern used in the form of EconomicDriving and AggressiveDriving defines the driving styles of different vehicles.

An example of using 4 patterns to solve the problem of creating different Vehicle objects with specific behavior. **(use the below link code on GitHub)**

https://github.com/vmplacademy/java-design-patterns/tree/develop/src/main/java/pl/vm/javaguild/designpatterns/pattern/behavioral/strategy

```
package org.example;

import lombok.Getter;
import lombok.experimental.SuperBuilder;

//Builder pattern using Lombok in class hierarchy
@SuperBuilder
@Getter
abstract class Vehicle {
	private int wheels;
	private int seats;
	private String engine;

	private VehicleBehavior behavior;
	public void drive(){
		behavior.perform();
	};
}

@SuperBuilder
class Truck extends Vehicle {
	@Override
	public void drive() {
		System.out.print(getEngine() + " truck ");
		super.drive();
	}
}

@SuperBuilder
class Bus extends Vehicle {
	@Override
	public void drive() {
		System.out.print(getEngine() + " bus ");
		super.drive();
	}
}


@SuperBuilder
class Car extends Vehicle {
	@Override
	public void drive() {
		System.out.print(getEngine() + " car ");
		super.drive();
	}
}

//Factory pattern
class VehicleFactory {
	public static Vehicle getVehicle(String type) {
		switch (type) {
			case "truck":
				return new TruckCreator().buildVehicle();
			case "bus":
				return new BusCreator().buildVehicle();
```

After launching the program, you will see the following screen:

- 24.0L V12 diesel truck economic driving
- 6.0L V6 diesel bus economic driving
- 2.0L R4 petrol car aggressive driving

Example 2:

In the second example, we will use the Memento as a behavioral design pattern (it allows to save and restore the previous state of an object without revealing the details of its implementation) in a program like Paint to obtain the possibility of a 10-level "Undo" or "Backtrack" function.

We can draw shapes on the Canvas and fill it with colors. Each time we press Save, Caretaker creates and saves the current state of the canvas as a CanvasMemento object. If we press Undo, we return to the previous state.

After running the program twice, save the changes, change them again, and then use the "undo" function twice to restore the previous values of the canvas.

Memento design pattern as an example of architectural pattern**. (use the below link code on GitHub)**

https://github.com/vmplacademy/java-design-patterns/tree/develop/src/main/java/pl/vm/javaguild/designpatterns/pattern/behavioral/memento

```
package org.example;

import java.util.LinkedList;
import java.util.List;

// Memento Class
class CanvasMemento {
	private final String canvasState;

	public CanvasMemento(String state) {
		this.canvasState = state;
	}

	public String getState() {
		return canvasState;
	}
}

// Originator Class
class Canvas {
	private String content = "";

	public void draw(String shape) {
		content += " draw:" + shape;
	}

	public void fill(String color) {
		content += " fill:" + color;
	}

	// Creates a new Memento with a new state
	public CanvasMemento save() {
		return new CanvasMemento(content);
	}


	// Creates a new Memento with a new state
	public CanvasMemento save() {
		return new CanvasMemento(content);
	}

	// Restores the state from the Memento
	public void undoToLastSave(Object obj) {
		CanvasMemento memento = (CanvasMemento) obj;
		content = memento.getState();
	}

	@Override
	public String toString() {
		return content;
	}
}

// Caretaker Class
class Caretaker {
	private final List<CanvasMemento> saveStates = new LinkedList<>();
	private final Canvas canvas;

	public Caretaker(Canvas canvas) {
		this.canvas = canvas;
```

On the screen, we obtain:

• Canvas after all operations: draw:Circle fill:Red draw:Triangle

• Canvas after undo draw triangle: draw:Circle fill:Red

• Canvas after undo fill red: draw:Circle

Example 3:

The third example is the application of the Visitor design pattern to a building inspector, who is responsible for performing actions known only to him when visiting the successive rooms of a house.

Once the inspector is accepted and admitted to the house (house.accept(inspector)), BuildingInspector starts inspecting the rooms defined when the house was built. It performs specific actions in each room and finally checks the structure of the entire house when we call "visitor.visit(this)" in the accept method of the House class.

Visitor pattern separates algorithms from the objects on which they operate-better object composition **(use the below link code on GitHub)**

https://github.com/vmplacademy/java-design-patterns/tree/develop/src/main/java/pl/vm/javaguild/designpatterns/pattern/behavioral/visitor

```
package org.example;

public class BuildingInspection {
	interface BuildingPart {
		void accept(BuildingPartVisitor visitor);
	}

	static class Kitchen implements BuildingPart {
		public void accept(BuildingPartVisitor visitor) {
			visitor.visit(this);
		}
	}

	static class LivingRoom implements BuildingPart {
		public void accept(BuildingPartVisitor visitor) {
			visitor.visit(this);
		}
	}

	static class Bathroom implements BuildingPart {
		public void accept(BuildingPartVisitor visitor) {
			visitor.visit(this);
		}
	}

	static class House implements BuildingPart {
		BuildingPart[] parts;

		public House() {
			parts = new BuildingPart[] { new Kitchen(), new
LivingRoom(), new Bathroom() };
		}
```

```
        public void accept(BuildingPartVisitor visitor) {
            for (BuildingPart part : parts) {
                part.accept(visitor);
            }
            visitor.visit(this); // Visiting the house itself
        }
    }

    interface BuildingPartVisitor {
        void visit(Kitchen kitchen);

        void visit(LivingRoom livingRoom);

        void visit(Bathroom bathroom);

        void visit(House house);
    }

    static class BuildingInspector implements BuildingPartVisitor {
        public void visit(Kitchen kitchen) {
```

On the screen, after completing the program according to the order of the parts when creating the house:

1. Inspecting the quality of kitchen appliances and safety.

2. Checking living room space and ventilation.

3. Inspecting plumbing and hygiene conditions in the bathroom.

4. Performing an overall structural integrity check of the house.

Application 2:

Your are invited to visit this links for applying more examples

Link1: **https://refactoring.guru/design-patterns/java**

Link 2: **https://medium.com/@saygiligozde/design-patterns-in-java-5251032ca244**

Chapter 4:

Agile technologies

Chapter outline

Introduction

1. **Agile modeling**
   - **1.1 Agile Modeling Values**
   - **1.2 Agile Modeling's Principles**
   - **1.3 Agile Modeling Best Practices**
2. **Agile technologies (software tools)**
   - ***2.1*** **MetaUML**
   - **2.2 CrossCheck**
   - **3.2 Archetypes**
   - **3.3 Metapatterns**
3. **Agile implementation**
   - **3.1 UML to Java and Java to UML**
   - **3.2 Code snippets & unit tests**
4. **Agile Technologies**
   - **5.1 Aspect-oriented programming (AOP)**
   - **5.2 AspectJ Softawr tool**
   - **5.3 MDA**
   - **5.4 XUML Executable UML (xtUML or xUML)**
   - **5.5 BPML**

Conclusion

1. Introduction

Our growing digital world has an increasing appetite for more increasingly complex and numerous applications. Unfortunately, this demand collides (يصطدم) with high failure rates in software development. System development failure rates (taux d'échec) often reach 75 percent. Developers turn to tools like agile modeling to combat high failure rates and release quality applications for an ever-growing audience (public). Today, we are exploring in this chapter the world of agile technologies. Agile has a lot to offer the development world, so let's get acquainted with it.

2. Agile modeling:

a. Definition

When you want the ultimate definition of any concept, you can't do better than going right to the source. Agile modeling is defines as "…a practice-based methodology for
effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and lightweight manner."

The modeling adds to existing Agile methodologies such as the Rational Unified Process (RUP) or extreme programming (XP). Agile modeling helps developers create a customized software development process that fulfills (accomplit, يفي) their development needs yet is flexible enough to adjust to future situations.

b. Agile Modeling Values

Agile Modeling Embraces Five Values :

- **Communication**: Agile modeling fosters (favorise) communication between team members, developers, and stakeholders.
- **Simplicity**: Models help simplify both the software and the software development process. Drawing a diagram that illustrates a concept or plan and the related growth can eliminate hours of unnecessary work and manual coding.
- **Feedback**: Similar to the "communication" step, team members who use diagrams to communicate their ideas enable stakeholders to give fast feedback, which then cuts the project turnaround time (ce qui réduit ensuite le délai d'exécution du projet).
- **Courage**: Fortune favors the bold (audace), and you need the courage to make the difficult decisions and change course, even if your team has already spent much time and resources on the work.
- **Humility**: Although some iterations of Agile modeling values stop at four, other models include this fifth one. Humility (التواضع) shows that everyone on the team is essential

and has equal value. Sometimes we can even be wrong! Humility, in this case, means respect for others' ideas and suggestions and acknowledging the value of others' contributions.

c. Agile Modeling's Principles

The modeling embraces 11 core principles. You will notice many of the principles reference the five values previously discussed.

1. **Model With a Purpose**: Ask why you're developing the models and who you are developing them for.

2. **Adopt Simplicity:** Keep the models as straightforward (واضح وصريح) and uncomplicated as possible, and believe the simplest solution is also the best solution.

3. **Embrace Change**: The more your understanding of a project grows, the more likely it will change. Instead of fighting change, accept them and have the courage to readjust and rebuild.

4. **Enabling the Next Effort:** Your successors (خلفاء) might have to improve or enhance (améliorer ou renforcer) your project after you depart. Leave them enough documentation and models to expedite possible changes or improvements.

5. **Incremental Change:** It's rare for a model to be complete on the first try. Models evolve as the project grows and develops. You can cushion against the shock (Vous pouvez amortir les chocs) of change by making minor changes to the models as needed.

6. **Maximize Stakeholder Investment:** The team must make the best effort to develop software that meets the stakeholder's needs. Bear in mind (Gardez à l'esprit), the whole purpose of producing the software is to maximize the return for the client.

7. **Remember the Existence of Multiple Models:** There are many modeling solutions available, so pick the ones that fit the current situation best. Additionally, there are many different methods of software delivery.

8. **Produce Quality Work:** Nobody wants careless (مهمل), rushed (متسرع) work. The developer doesn't like it because they know it's not something they can be proud of deep down. The teams that come later to check the work don't like it because sloppy (غامض) work is challenging to understand and means more hours spent fixing it. And

finally, the end-users won't like the sub-par (inférieur à la moyenne) work because it most likely won't function properly or doesn't meet their expectations.

9. **Provide Rapid Feedback:** Receiving timely feedback (reponse, reaction) on the model closes the model's loop of understanding. Model a small portion—show it to the appropriate parties for review—then model again.

10**. Make Working Software Your Primary Goal:** Models are just a means to the end, which is building great software for your customer. Make sure that documentation and modeling directly support the goal of your software development project.

11. **Travel Light:** Traveling light is another way of saying that you have sufficient documentation about the models you're developing, but no more than that. If you have too little documentation, the developing team might lose its way—if you have too much, the development team may forget that the primary goal is not writing documentation but instead building software and the right models!

d. Agile Modeling Best Practices

Here are the ten most common best practices.

- **Active Stakeholder Participation :**Stakeholders must provide information, make timely product decisions, and be as actively involved in the development process as possible, using inclusive tools and techniques.
- **Architecture Envisioning**: The team must do some initial, high-level architectural modeling at the beginning of the project to identify a viable (قابلة للحياة) technical strategy for creating a solution.
- **Iteration Modeling**: At the start of each iteration, you must do some modeling as part of your planning activities.
- **Just Barely (بالكاد)Good Enough (JBGE) Artifacts:** A model or document must be enough for the current situation and no more.
- **Look-ahead Modeling:** Sometimes you need to look ahead (إلى الأمام) to reduce overall risk.
- **Model Storming**: During an iteration, you will sometimes need to model storms on a Just in Time (JIT) basis for a little while. This time investment helps the team explore the details behind a requirement or work through a design issue.

  Just-in-time, or JIT, is an inventory management method in which goods are received from suppliers only as they are needed. The main objective of this method is to reduce inventory holding costs and increase inventory turnover. (Le juste-à-temps, ou JAT, est une méthode

de gestion des stocks dans laquelle les marchandises sont reçues des fournisseurs uniquement au fur et à mesure des besoins. L'objectif principal de cette méthode est de réduire les coûts de stockage et d'augmenter la rotation des stocks.)

- **Multiple Models:** Each model has its strengths and weaknesses. A good developer needs a range (gamme) of models in their repertoire to apply the right model in the best way for the current situation.
- **Prioritized Requirements:** Agile teams implement requirements in a priority order defined by their stakeholders. This order provides the greatest possible return on investment (ROI).
- **Requirements Envisioning**: At the start of an Agile project, you need to invest a bit of time identifying the project's scope and creating the initial prioritized requirements stack.
- **Test-Driven Development (TDD):** Write a single test aimed at either the design or requirements or design level and just enough code to run the test. TDD is a JIT (Just in Time) approach to specifying detailed requirements and a confirmatory testing approach.

e. The Pros and Cons of Agile Modeling

The modeling brings advantages and disadvantages to the table.

2.5. 1 Advantages

- Facilitates effective communication between teams and clients
- Enhances project flexibility, easily handling sudden (فجأة) changes anytime
- Cuts overall development time
- Increases customer satisfaction via rapid, continuous delivery of a workable product
- Delivers functioning software frequently, in weeks instead of months

2.5.2 Disadvantages

- Confusion between teams may develop because documentation wasn't emphasized. This uncertainty can lead to difficult transitions between phases.
- It is sometimes difficult to gauge how much effort will be needed to start the development life cycle of larger software deliverables.
- If stakeholders project ladder are not on the same page, the project will derail.
- The modeling isn't for newbies. The sort of decisions involved in Agile require people with experience and solid developer and programming skills.

f. Agile Modeling : tool /example

Agile Modeling does not impose specific tools, but several tools support **lightweight, collaborative, and iterative modeling**, which fits AM philosophy.

Tool	Description	Why it fits Agile Modeling
Modelio	Open-source UML/BPMN tool	Simple diagrams, fast modeling, supports iterations
StarUML	Lightweight UML tool	Quick diagramming, easy to modify
Lucidchart / Draw.io (Diagrams.net)	Online diagramming tools	Real-time collaboration, simple visuals
Camunda Modeler	BPMN and DMN	Good for agile process modeling
PlantUML	Text-based UML diagrams	Fast creation/editing, version control friendly
Enterprise Architect	Full modeling suite	Supports iterative modeling and teamwork
Miro / Mural	Visual whiteboards	Brainstorming, collaborative modeling

3. Agile technologies (software tools)

a. MetaUML : UML for LaTeX/MetaPost

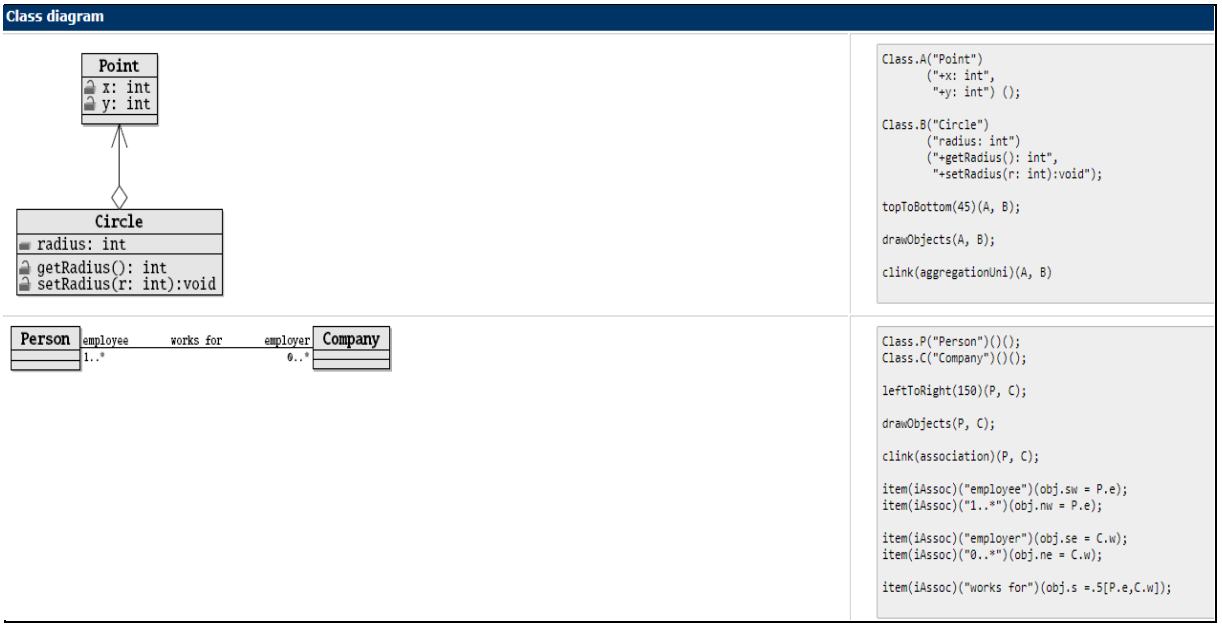
Metamodel: is a model of a model, and metamodeling is the process of generating such metamodels. Thus metamodeling or meta-modeling is the analysis, construction, and development of the frames, rules, constraints, models, and theories applicable and useful for modeling a predefined class of problems. As its name implies, this concept applies the notions of meta- and modeling in software engineering and systems engineering. Metamodels are of many types and have diverse applications

MetaUML is a METAPOST library for typesetting UML diagrams, which provides a usable, human-friendly textual notation for UML, offering now support for class, package, activity, state, and use case diagrams.

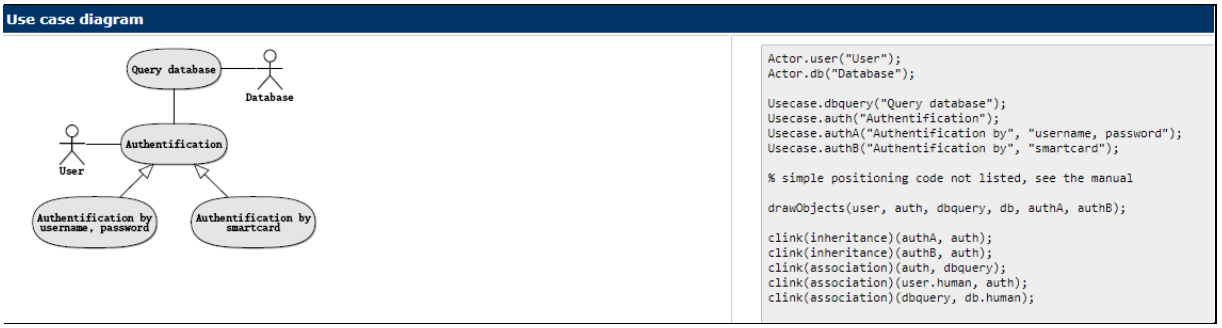
MetaUML offers support for class diagrams, package diagrams, activity diagrams, state machine diagrams use case diagrams and component diagrams.

MetaUML GitHub https://github.com/ogheorghies/MetaUML

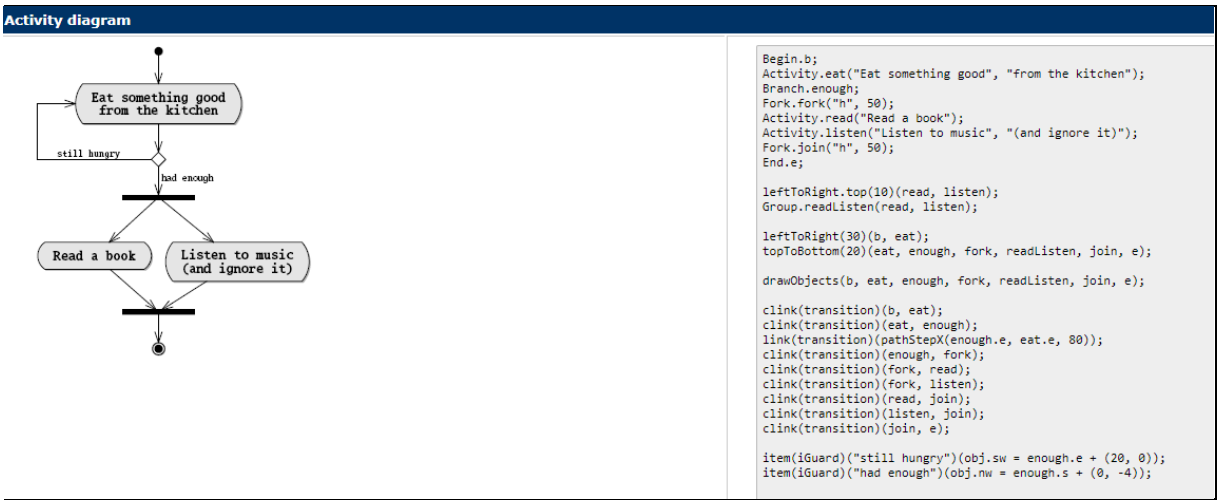
i. MetaUML: Class diagram



3.1.2 MetaUML: use case diagram



3.1.3 MetaUML: activity diagram



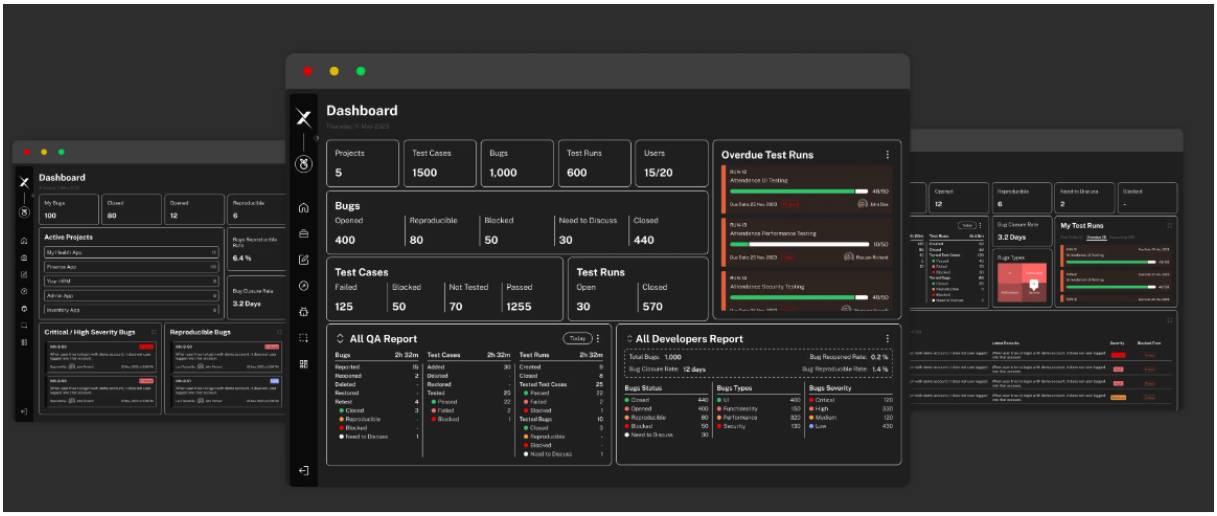
MetaUML is a GNU GPL library for typesetting UML diagrams, particularly useful in a TeX or LaTeX environment; see Knuth (1986), Lamport (1994). It provides an easy to use, human

readable API. The code of a diagram typeset in MetaUML appears clearer than the corresponding code in uml.sty, pst-uml.sty, umldoc or even XMI; see Gjelstad (2001), Diamantini (1998), Palmer (1999), OMG (2003). It is the next best thing to using a visual tool, while having the free- 78 MAPS 33 Ovidiu Gheorghie¸s dom of not becoming technologically dependent of any particular visual tool. The util infrastructure of MetaUML offers means of defining and using "objects", which may recommend it for other typesetting projects, unrelated to UML.

We mention here a few of its benefits: the ability to stack and align text in a visually pleasing way; a fine degree of control of how elements are laid out; the ability to group objects while having access to the properties of inner elements; a design pattern and syntactic sugar for writing modern-looking, reusable MetaPost code. With this infrastructure in place, it should be possible to extend MetaUML until it offers complete UML 2.0 support.

b. **CrossCheck** [redacted]rosscheck is a JavaScript **unit-testing** framework capable of emulating multiple browser environments.

We created a centralized test management tool that covers test case management, test runs, bug reporting, and tracking, complete with insightful dashboards and a free DevTools Extension. By fostering Agile and Quality Driven Development principles, we ensure everything is cross-checked during software testing in one platform.



3.2.1 Using Crosscheck

crosscheck.jar is a standalone program for running crosscheck tests. It includes all dependencies, so there's no need for mucking about with a classpath to get it to run.

Usage: java -jar crosscheck.jar [options] [test-file | test-directory]*
Options:

-help print this message
-hosts colon-separated list of hosts (ie-6, moz-1.7, moz-1.8)
e.g.
java -jar crosscheck.jar tests/
java -jar crosscheck.jar -hosts=moz-1.7 tests/test.jst

For those working on Java projects who may already have other versions of the libraries that Crosscheck depends upon, the distribution also contains just-crosscheck.jar. As the name implies,
this file contains only crosscheck class files and resources. In order to use this version of Crosscheck, you'll need to include just-crosscheck.jar and all the files in the "lib" directory in your classpath.

Example: java -classpath just-crosscheck.jar:lib/commons-lang.jar:lib/tagsoup.jar:lib/js.jar net.thefrontside.crosscheck.framework.ConsoleRunner test/net/thefrontside/crosscheck/hosts/

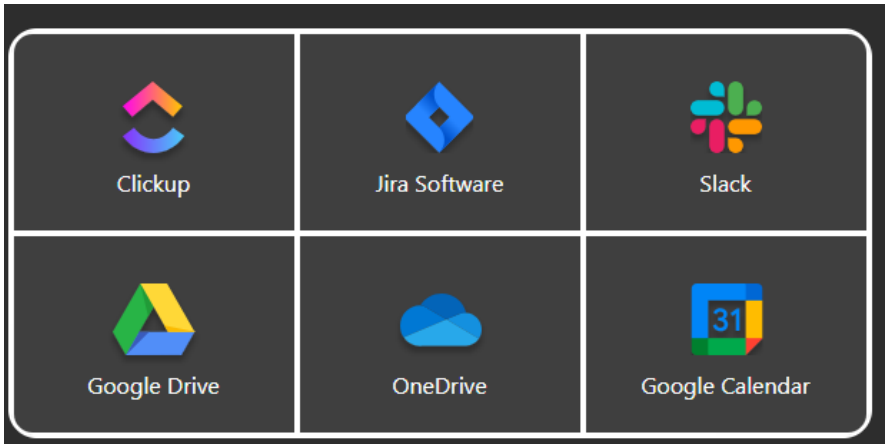
3.2.2 About Crosscheck

Crosscheck is a free software published in the Browser Tools list of programs, part of Network & Internet.

This Browser Tools program is available in English. It was last updated on 22 April, 2024. Crosscheck is compatible with the following operating systems: Linux, Mac, Windows. The company that develops Crosscheck is cowboyd. The latest version released by its developer is 0.2.1. This version was rated by 7 users of our site and has an average rating of 4.1. The download we have available for Crosscheck has a file size of 4.81 MB. Just click the green Download button above to start the downloading process. The program is listed on our website since 2007-02-05 and was downloaded 5,714 times. We have already checked if the download link is safe, however for your own protection we recommend that you scan the downloaded software with your antivirus. Your antivirus may detect the Crosscheck as malware if the download link is broken.

3.2.3 Integrate with the tools your team already uses

Crosscheck offers integration with industry-standard tools like Jira, ClickUp, Slack, and cloud storage providers to empower your QA team to collaborate, manage tests, and deliver exceptional results. Achieve more with Crosscheck software testing tool.



3.2.3 Explore Key Features

Test Case Management

Bugs Reporting

Test Run

Dashboards

Feedback Widget

Crosscheck Extension

3.3 Archetypes

3.3.1 Using an Archetype

Archetype is a Maven project templating toolkit. An archetype is defined as *an original pattern or model from which all other things of the same kind are made*. The name fits as we are trying to provide a system that provides a consistent means of generating Maven projects. Archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates.

Using archetypes provides a great way to enable developers work quickly in a way consistent with best practices employed by your project or organization. Within the Maven project, we use archetypes to try and get our users up and running as quickly as possible by providing a sample project that demonstrates many of the features of Maven, while introducing new users to the best practices employed by Maven.

In a matter of seconds, a new user can have a working Maven project to use as a jumping board for investigating more of the features in Maven. We have also tried to make the Archetype mechanism additive, and by that we mean allowing portions of a project to be captured in an archetype so that pieces or aspects of a project can be added to existing projects.

A good example of this is the Maven site archetype. If, for example, you have used the quick start archetype to generate a working project, you can then quickly create a site for that project by using the site archetype within that existing project. You can do anything like this with archetypes.

You may want to standardize J2EE development within your organization, so you may want to provide archetypes for EJBs, or WARs, or for your web services. Once these archetypes are created and deployed in your organization's repository, they are available for use by all developers within your organization.

Archetypes are packaged up in a JAR and they consist of the archetype metadata which describes the contents of archetype, and a set of Velocity templates which make up the prototype project. If you would like to know how to make your own archetypes.

To create a new project based on an Archetype, you need to call **mvn archetype: generate** goal. **mvn archetype : generate**

3.3.2 Provided Archetypes

Maven provides several Archetype artifacts:

Archetype ArtifactIds	Description
maven-archetype-archetype	An archetype to generate a sample archetype project.
maven-archetype-j2ee-simple	An archetype to generate a simplifed sample J2EE application.
maven-archetype-mojo	An archetype to generate a sample a sample Maven plugin.
maven-archetype-plugin	An archetype to generate a sample Maven plugin.
maven-archetype-plugin-site	An archetype to generate a sample Maven plugin site.
maven-archetype-portlet	An archetype to generate a sample JSR-268 Portlet.
maven-archetype-quickstart	An archetype to generate a sample Maven project.
maven-archetype-simple	An archetype to generate a simple Maven project.
maven-archetype-site	An archetype to generate a sample Maven site which demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.
maven-archetype-site-simple	An archetype to generate a sample Maven site.
maven-archetype-webapp	An archetype to generate a sample Maven Webapp project.

3.4 Metapatterns

Design patterns for Python implemented with decorators and classes. Currently only one design pattern is implemented: `Listener`.

Listener Pattern

The `Listener` pattern (otherwise known as the `Observer` or `Publish-Subscribe` pattern) is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing (source).

Its use is demonstrated here:

```
from metapatterns.listener import Listenable, listenable
```

```
class Subject(Listenable):
    @listenable
    def myfunc(self, arg1):
        """
        @listenable indicates this function can be 'listened in on'.
        It allows Listeners to hook into it (see MyListener)
        """
        print("myfunc called with arg", arg1)
        return "Hoozah"

    def myfunc2(self, arg1):
        print("myfunc called with arg", arg1)


class MyListener(Subject.Listener):
    """
    Identify this class as a listener of `Subject` through inheritance.
    This makes it so not all listenable methods need to be implemented (they have a default empty implementation in `Subject.Listener`).
    """
    def on_myfunc(self, subject, arg1):
        print("listened in on call to myfunc with arg", arg1)

    def on_myfunc_finished(self, subject, result, arg1):
        print("listened in on result of myfunc with arg", arg1, "and result", result)

    # This cannot be defined because myfunc2 is not a listenable function in Subject
    #def on_myfunc2(self, arg1):
        #pass
```

We can run this as follows:

```
if __name__ == "__main__":
    subject = Subject()
    print("# Calling myfunc without listener")
    subject.myfunc(3)

    listener = MyListener()
    subject.add_listener(listener)

    print("\n# Calling myfunc with listener")
    subject.myfunc(5)

    print("\n# Calling myfunc2 with listener")
    subject.myfunc2(7)

    subject.remove_listener(listener)
```

```
print("\n# Calling myfunc again with listener removed")
subject.myfunc(5)
```

which gives the output:

```
# Calling myfunc without listener
myfunc called with arg 3

# Calling myfunc with listener
listened in on call to myfunc with arg 5
myfunc called with arg 5
listened in on result of myfunc with arg 5 and result Hoozah

# Calling myfunc2 with listener
myfunc called with arg 7

# Calling myfunc again with listener removed
myfunc called with arg 5
```

Subclassing from `Subject.Listener` has the advantage of raising a `TypeError` when an `on_` function has no matching counterpart in `Subject`. This can help detect difficult to find problems that arise when changing the name of a function in `Subject`. Our method includes this check to guard against some issues that come with the loose coupling of the `Listener` pattern

4. Agile implementation

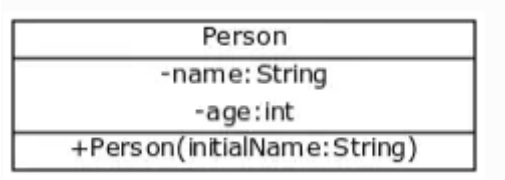
a. UML to Java and Java to UML

Object-oriented software development has become very popular. Also, UML has been accepted as the standard design language. We discuss use of UML to arrive at a design solution. Skeletal java code generation from UML diagrams will be discussed. Design patterns are reusable solutions. These are good solutions to typical programming problems, that can be understood and applied in a specific design situation to improve the overall design and reduce design iterations.

A class diagram is a diagram used in designing and modeling software to describe classes and their relationships. Class diagrams enable us to model software in a high level of abstraction and without having to look at the source code.

Classes in a class diagram correspond with classes in the source code. The diagram shows the names and attributes of the classes, connections between the classes, and sometimes also the methods of the classes.

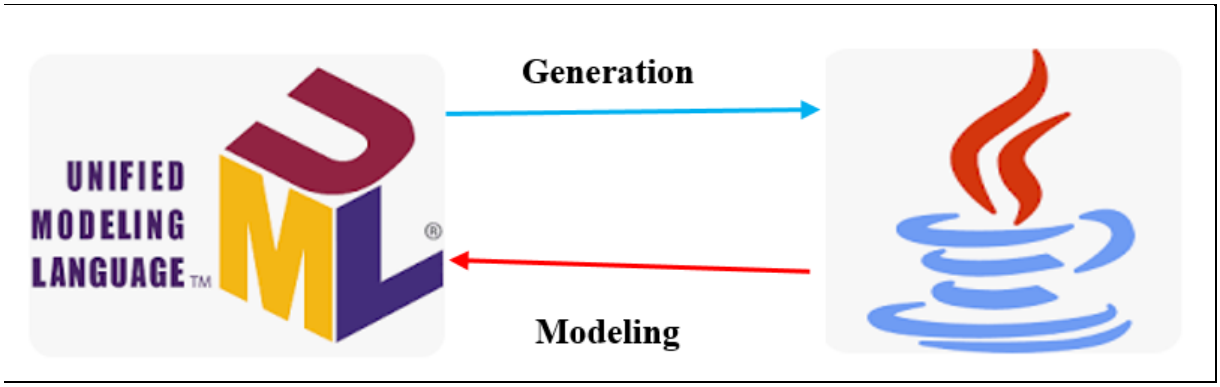
In a class diagram, a class is represented by a rectangle with the name of the class written on top. A line below the name of the class divides the name from the list of attributes (names and types of the class variables). The attributes are written one attribute per line.
In a class diagram, class attributes are written "attributeName: attributeType". A + before the attribute name means the attribute is public, and a - means the attribute is private.
In a class diagram, we list the constructor (and all other methods) below the attributes. A line below the attributes list separates it from the method list. Methods are written with +/- (depending on the visibility of the method), method name, parameters, and their types. The constructor above is written +Person(initialName:String)
The parameters are written the same way class attributes are — "parameterName: parameterType".




```java
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.name = initialName;
        this.age = 0;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " +   this.age + " years");
    }
}
```



b. Code snippets & unit tests

i. Java Snippet

In programming, the **snippet** is a piece of code that resolves a bunch of problems with a few lines of code. Also, reduces the line of code and make programmer more knowledgeable. In this section, we will discuss **what is a snippet in Java, its uses,** and **example.** Also, we will focus on the **Java Snippet class, methods,** and **nested subclasses**.

```
public static void printList(int[] list) {
        System.out.println("index, value");
        for(int i = 0; i < list.length; i++) {
        System.out.printin(i + " , " + list[i];
        }
}
```

Fig: Java Code Snippet

Before moving to the **snippet in Java**, first, we have to understand JShell (Java Shell).

JShell :JShell is abbreviated for the **Java Shell tool**. It is an interactive tool that helps programmer to learn Java programming language and prototyping of the Java code. It is a **Read-Eval-Print Loop** (REPL) that evaluates declarations, statements, and expressions as they are entered and immediately shows the results. The tool can be run from the command line.

4.2.2 Uses of JShell :J Shell tool allows programmers to enter program elements one at a time and at the same time one can see the result. Also, allows us to make adjustments accordingly. Typically, a general Java program development involves the following process.

- Write any Java program
- Compile it, fix errors (if any)
- Run the program
- Getting correct output. If no, figures out what is wrong with it
- Edit it
- Repeat the above process

Apart from the above, it helps the programmer to try out code easily and also explore possible options for our program. Another benefit of using JShell is that it allows us to test an individual statement, try out various versions of a function, and more.

Once we develop the program, paste code into the JShell tool to try it out. After that, paste the working code into the program editor or IDE from JShell tool.

As we discussed above JShell accepts statements, variables, methods, expressions, class definitions, and imports. These fragments (part or pieces) of Java code are called **snippets**. In general, the snippet is a section or piece of Java source code that save in XML format.

4.2.3 Example of Snippet

Suppose, we have to create a Java program to calculate the factorial of a number. Instead of writing complete Java source code, we can write the logic (main section of the source code) section, as follows.

The following code snippet depicts the recursive function for calculating the factorial of a number.

```
1. int factorial(int n) {
2.  if (n <= 1)
3.  return 1;
4.  else
5.  return (n * factorial(n - 1));
6. }
```

Note: Snippet and pseudo-code can be used in the same context but both are different in meaning. Pseudocode just represents principle, but doesn't use real, usable syntax. While snippet uses proper syntax.

4.2.4 Components of a Snippet

Each snippet contains the following **four** components:

1. **Name of the snippet:** Snippet name make it unique from different snippets.
2. **Prefix:** The keyword which generates current snippets in the program.
3. **Body:** It represents the actual code (logic) that we bind into snippets contained in the body.
4. **Description**: Information about the snippet contains in a snippet.

Format of the Snippet

The snippet of particular code implemented in the **java.json** file that uses JSON format.

```
1. "Name_of_the_snippet ":
2. {
3. "prefix": "prefix_of_the_snippet",
4. "body": [
5.  // Actual code of the snippet
6.  ],
7. "description": "description_about_the_snippet"
8. }
```

4.2.5 Creating Snippets in Java

There are the following two steps involved in creating a snippet of Java code.

- Create a Java class for which you want to create a snippet.
- Create a snippet of this Java class.

Create a .java file. In our case, we have created a file in which we have calculated factorial.

FactorialExample.java

```
1.  public class FactorialExample
2.  {
3.  static int factorial(int n)
4.  {
5.  if (n == 0)
6.  return 1;
7.  else
8.  return(n * factorial(n-1));
9.  }
10. public static void main(String args[])
11. {
12. int i,fact=1;
13. int number=4;//It is the number to calculate factorial
14. fact = factorial(number);
15. System.out.println("Factorial of "+number+" is: "+fact);
16. }
17. }
```

Output: Factorial of 4 is: 24

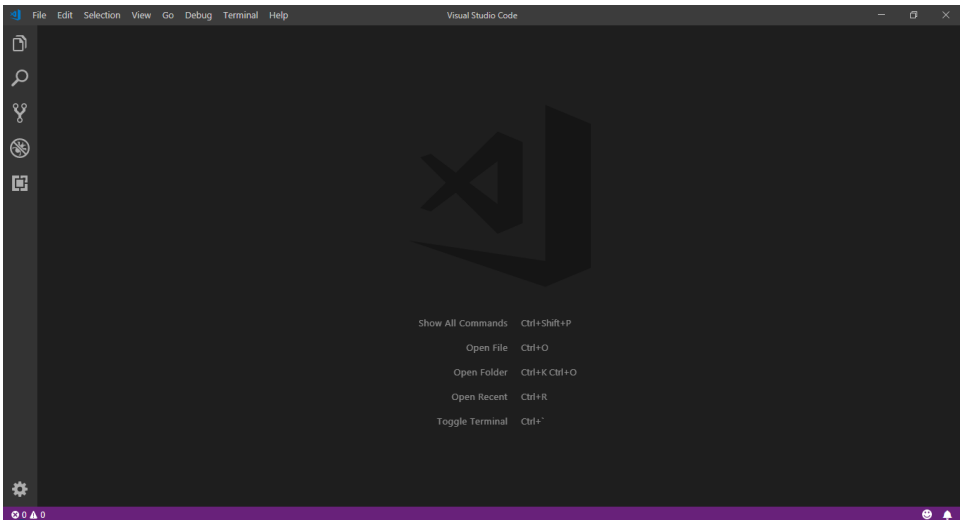
Create a Snippet of the Java File

In order to generate snippets, we will use VS Code text editor. The editor supports development operations and a version control system.

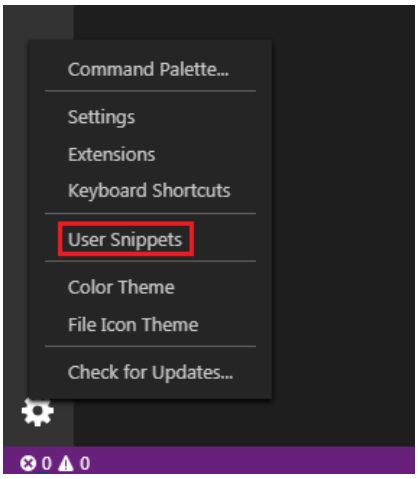
Note: Moving ahead, ensure that VS Code editor is properly installed in your system.

Follow the steps given below:

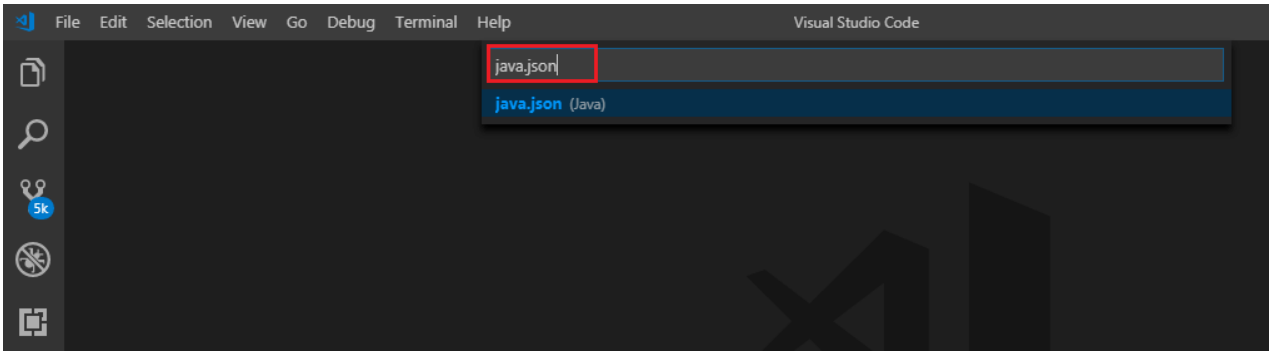
Step 1: Open VS Code in a folder to be created.



Step 2: Click on the **Setting** icon placed at the bottom left corner. It shows a pop-up menu. From the menu, click on the **User Snippets** option.



Step 3: In the search box, type **java.json** and press enter.



After hitting the **Enter** key, it opens the **java.json** file. It looks like the following.

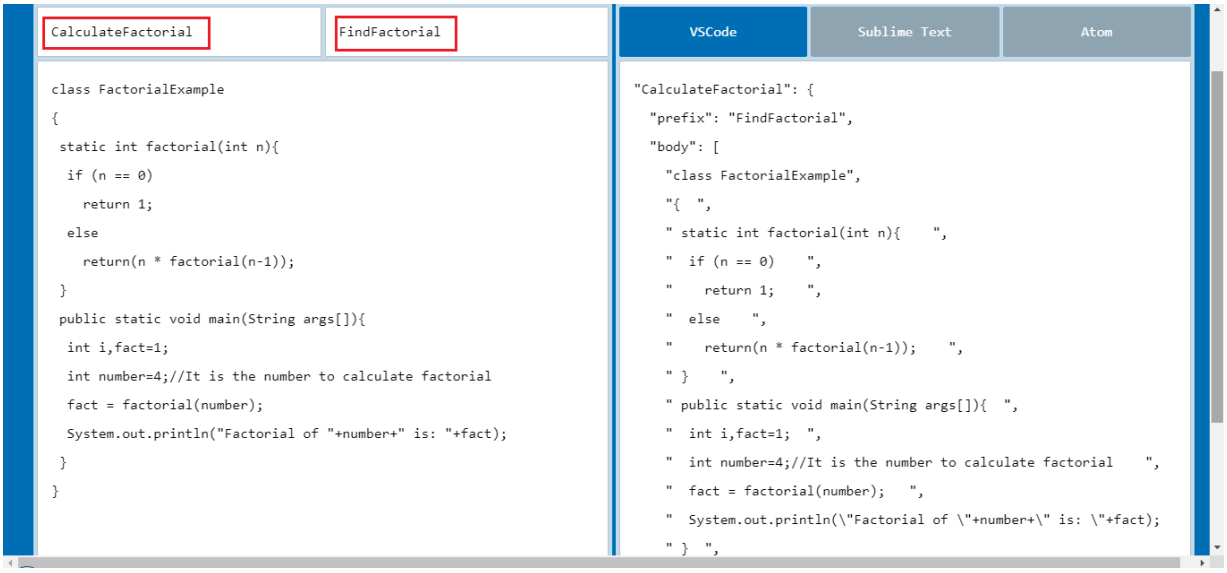
Step 4: Now, go to Google and search **Snippet generator**. It is a tool that converts Java code into snippets. The tool looks like the following.

Step 5: Paste your Java code into the left block. In our case, we have pasted Java code that calculates the factorial of a number.

FactorialExample.java

```
1.  class FactorialExample
2.  {
3.   static int factorial(int n){
4.    if (n == 0)
5.      return 1;
6.    else
7.      return(n * factorial(n-1));
8.   }
9.   public static void main(String args[]){
10.   int i,fact=1;
11.   int number=4;//It is the number to calculate factorial
12.   fact = factorial(number);
13.   System.out.println("Factorial of "+number+" is: "+fact);
14.  }
15. }
```

After pasting the Java code, the window looks like the following.



Also note, in the above window there are the boxes one is for **Description** and the second for **Tab trigger**. In the description box, we have written **CalculateFactorial** which denotes the name of the snippet**.** In the **Tab trigger** box, we have written **FindFactorial** which represents keywords to generate.

Step 5: At last, copy the snippet of the Java code by clicking on the **Copy snippet** button and paste it into the **java.json** file (that we have searched in Step 3) in VS code.

java.json

```
1.  "CalculateFactorial": {
2.   "prefix": "FindFactorial",
3.   "body": [
4.    "class FactorialExample",
5.    "{ ",
6.    " static int factorial(int n){ ",
7.    " if (n == 0) ",
8.    " return 1; ",
9.    " else ",
10.   " return(n * factorial(n-1)); ",
11.   " } ",
12.   " public static void main(String args[]){ ",
13.   " int i,fact=1; ",
14.   " int number=4;//It is the number to calculate factorial ",
15.   " fact = factorial(number); ",
16.   " System.out.println(\"Factorial of \"+number+\" is: \"+fact); ",
17.   " } ",
18.   "} "
19.  ],
20.  "description": "CalculateFactorial"
21. }
```

In the same way, we can also generate snippets for other Java source codes.

5 Agile Technologies

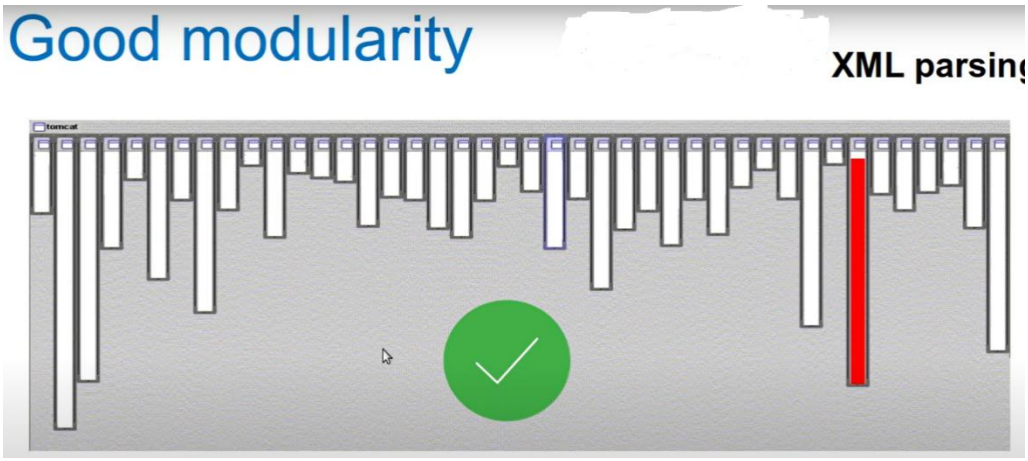
5.1 Aspect-oriented programming (AOP)

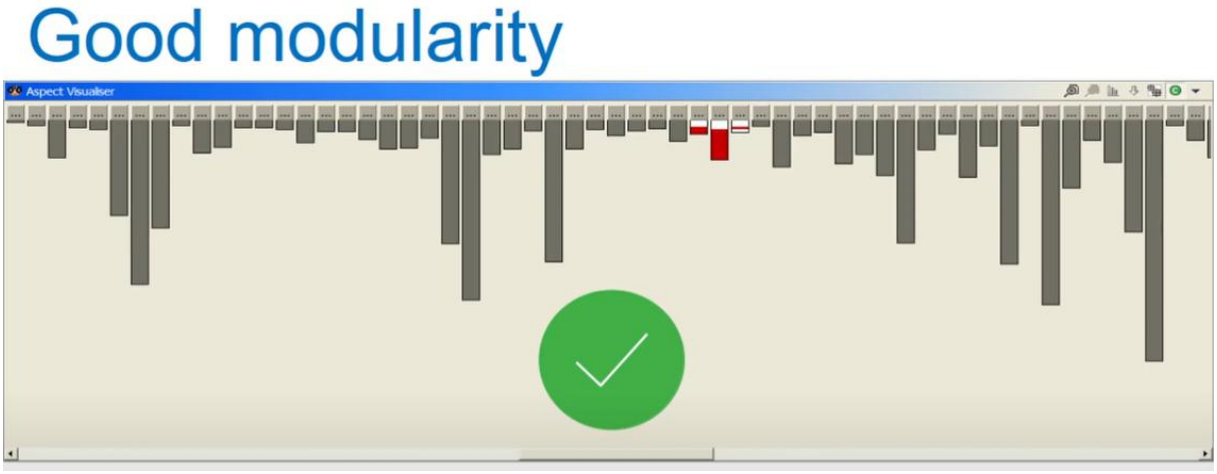
5.2.1 Principle

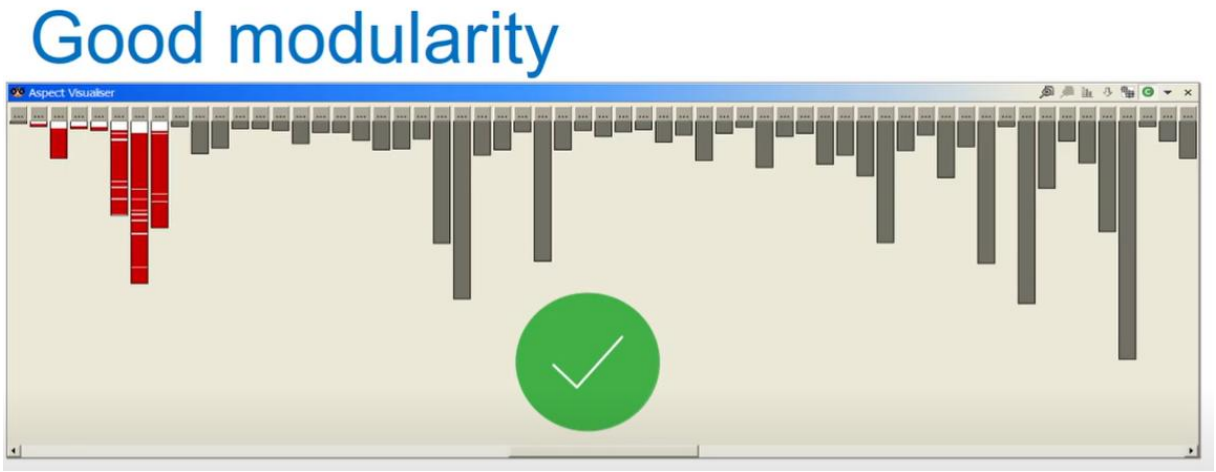
Aspect-oriented programming (AOP) is a programming paradigm designed to improve and increase **modularity** by enabling the separation of cross-cutting concerns. It makes it easier to add code to pre-existing programmes – by extracting code into manageable sections known as 'aspects' – without changing the code itself.

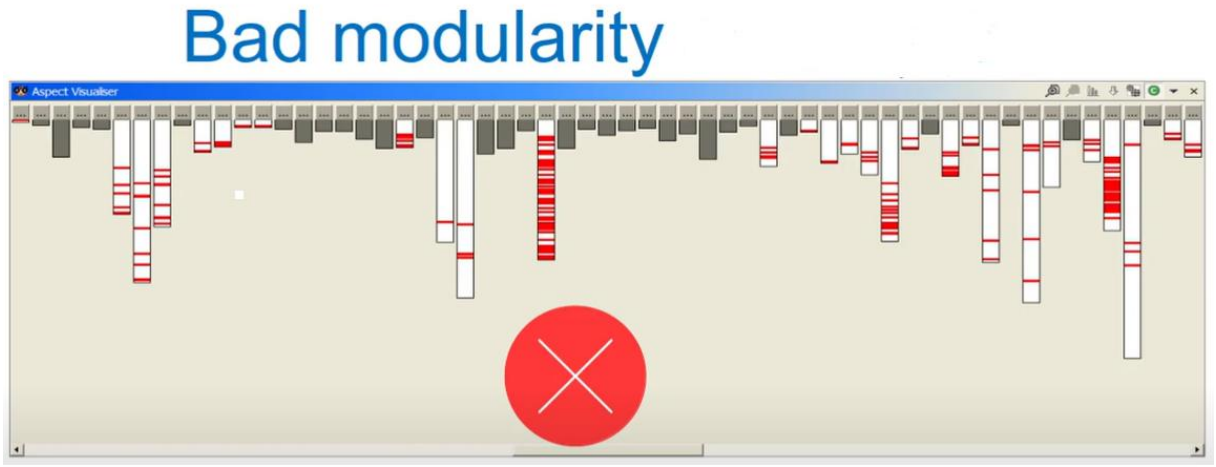
cross-cutting: identify area of code where common functionality exits

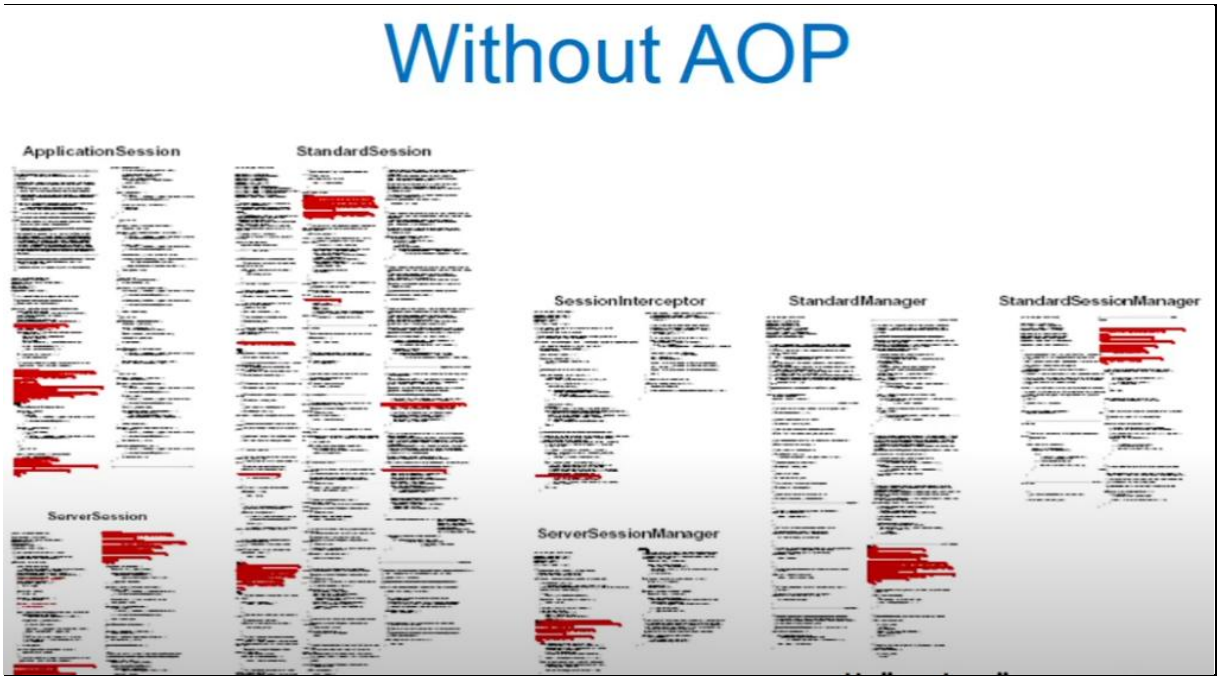
Cross-cutting concerns refer to aspects of a programme affecting other concerns that – in both design and implementation – cannot be easily, or cleanly, decomposed from a system. Cross-cutting concerns can cause code duplication (known as 'scattering') and critical dependencies between systems (known as 'tangling'), as concerns can be found across multiple classes and components. Common cross-cutting concerns include transaction processing, security authentication, data validation, caching, format data, error handling, debugging, and logging.





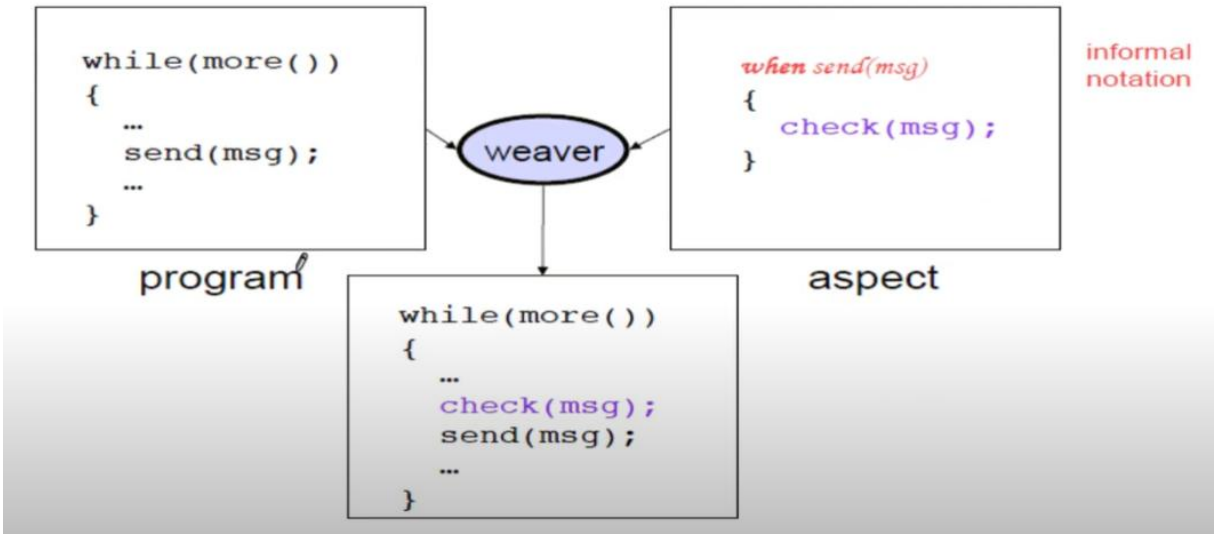

Good modularity
Aspect Visualiser

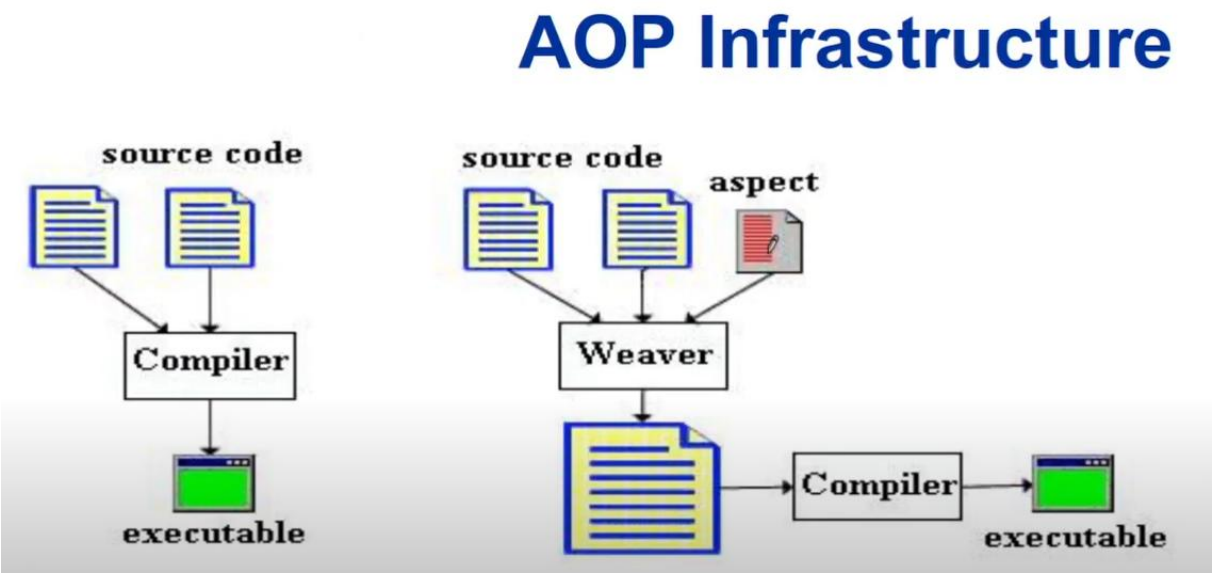

Bad modularity
Aspect Visualiser


Without AOP
ApplicationSession
StandardSession
SessionInterceptor
StandardManager
StandardSessionManager
ServerSession
ServerSessionManager

with AOP

AOP example





5.1.2 Crosscutting examples:

Tracking and profiling

Logging

Configuration management

Exception handling

Security

Synchronization

Verifying correctness

Visualization

Here is a simplified walk-through of how AOP works in practice:

1. **Identify cross-cutting concerns** (for example, logging)
2. **Define aspects** (the modules that encapsulate the concerns)
3. **Weaving** (use an AspectJ compiler or weaver to combine aspects with main business logic at compile-time, load-time, or run-time)
4. **Application of aspects** (apply aspects to specific 'join points', where functionality is integrated into the programme)
5. **Isolation of concerns** (which enables modularization).

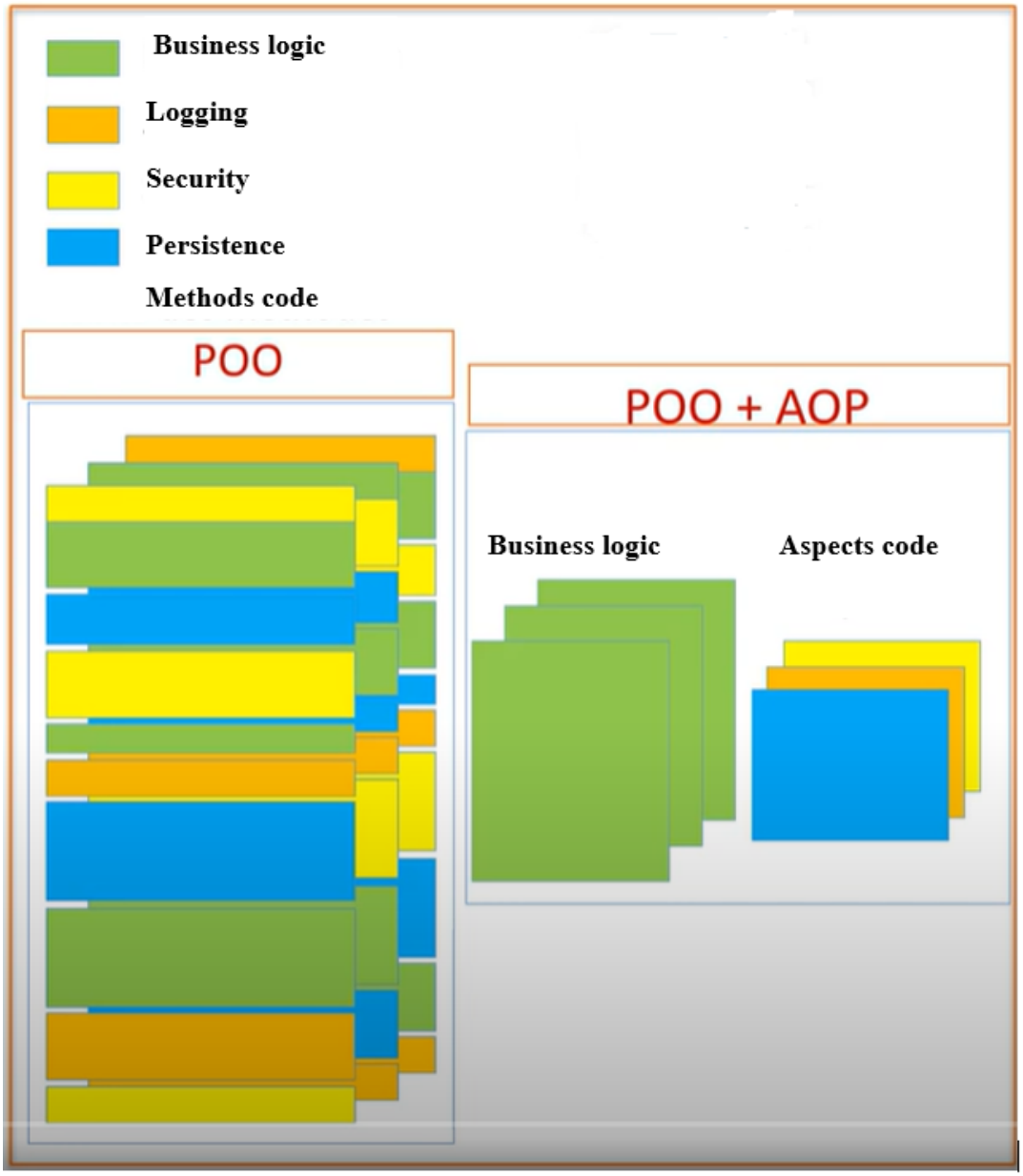
The AOP framework is supported by a number of popular programming languages and platforms, including Java (and Java Spring Framework/Spring AOP), C#, Python, .NET, Perl, XML, and Ruby.

How is aspect-oriented programming different from object-oriented programming?

Both AOP and object-oriented programming (OOP) are programming paradigms.

The goal of OOP is to organise code into 'objects' (instances of classes) that encapsulate data and behaviour. It uses four principles to model real-world entities: encapsulation (binding data), abstraction (using simple classes to represent complexity), inheritance (enabling classes to inherit features of others), and polymorphism (using different objects to reply to one form, and interact with the same interface).

While AOP and OOP focus on different aspects of software development and design – the former on separation of concerns and the latter on data encapsulation and behaviour – they can be used to complement one another.



5.1.3 Advantages of AOP

AOP's ability to provide declarative enterprise services – such as declarative transaction management/annotations – for a specific software system or organisation has contributed to its popularity and widespread use. It also allows users to implement custom elements and add additional functionalities and features that were not initially present in the software.

In addition, aspect-oriented software development provides a number of other strategic advantages:

- **Modularity**. Improvements to code modularity make everything simpler to understand. AOP modularises and separates cross-cutting concerns from core business logic, allowing programmers and developers to handle concerns separately.
- **Maintainability**. AOP makes it possible to modify or update specific code functionalities – without affecting the wider source code – making software changes more manageable for programmers and developers. It also helps to reduce undesirable or unintended side effects.
- **Code reusability**. The aspects that are inherent in AOP encapsulate common functionalities and, therefore, promote the reuse of code across different parts of the application.
- **Centralised management**. AOP makes it easier to implement modifications and changes uniformly across the entire application and, as a result, manage cross-cutting concerns such as transactions and security.
- **Readability**. Isolating and specifying non-functional requirements improves the readability of central business logic. As such, software engineers can focus on core functionality, free from the distractions of unrelated concerns.
- **Scalability**. As the codebase grows, the effective management of concerns becomes more important, as well as more complex. AOP supports a cleaner, more organised code structure – which, in turn, supports scalability.
- **Better testing functionality**. Concerns that have been separated/isolated make it much easier to conduct independent testing. This is useful in terms of promoting better overall software quality and implementing a more effective testing methodology and strategy.

5.1.4 Disadvantages of AOP

While there are many benefits to using the AOP framework, it's not perfect. As a result, computer programmers, engineers and developers should take time to understand whether AOP aligns well with the specific characteristics and requirements of a particular software program or project.

Some of the disadvantages of AOP include:

- **Issues with debugging**. Debugging can become more of a challenge. Aspects applied at different points in the programme can affect the flow of control and increase the level of complexity, making it more difficult to identify issues.
- **Complexity**. Codebases can suffer from greater levels of complexity as more aspects interact with the central business logic. This creates challenges for developers who may not be as comfortable, or familiar, working with AOP frameworks. There are also knock-on effects for testing; additional work may be required to ensure that isolated aspects behave as intended.
- **Portability**. Portability between different platforms, applications and programming languages may cause issues. As such, AOP can limit the capacity to reuse code in diverse environments.
- **Support systems**. Integrated development environments (IDE) and tooling may be limited in range when it comes to AOP frameworks. In contrast to OOP, which benefits from a more extensive range of tooling support, AOP can encounter issues working with certain aspects in certain environments.

5.2 AspectJ Softawr tool

AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code without modification of the code itself. Instead, we declare separately which code to modify.

AspectJ implements both concerns and the weaving of crosscutting concerns using extensions of Java programming language.

All valid Java programs are also valid AspectJ programs, but AspectJ lets programmers define special constructs called aspects. Aspects can contain several entities unavailable to standard classes.

AspectJ is an aspect-oriented programming (AOP) extension for the Java programming language, created at PARC. It is available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse. AspectJ has become a widely used de facto standard for AOP by emphasizing simplicity and usability for end users. It uses Java-like syntax, and included IDE integrations for displaying crosscutting structure since its initial public release in 2001. (See Handout N°5)

5.3 MDA:

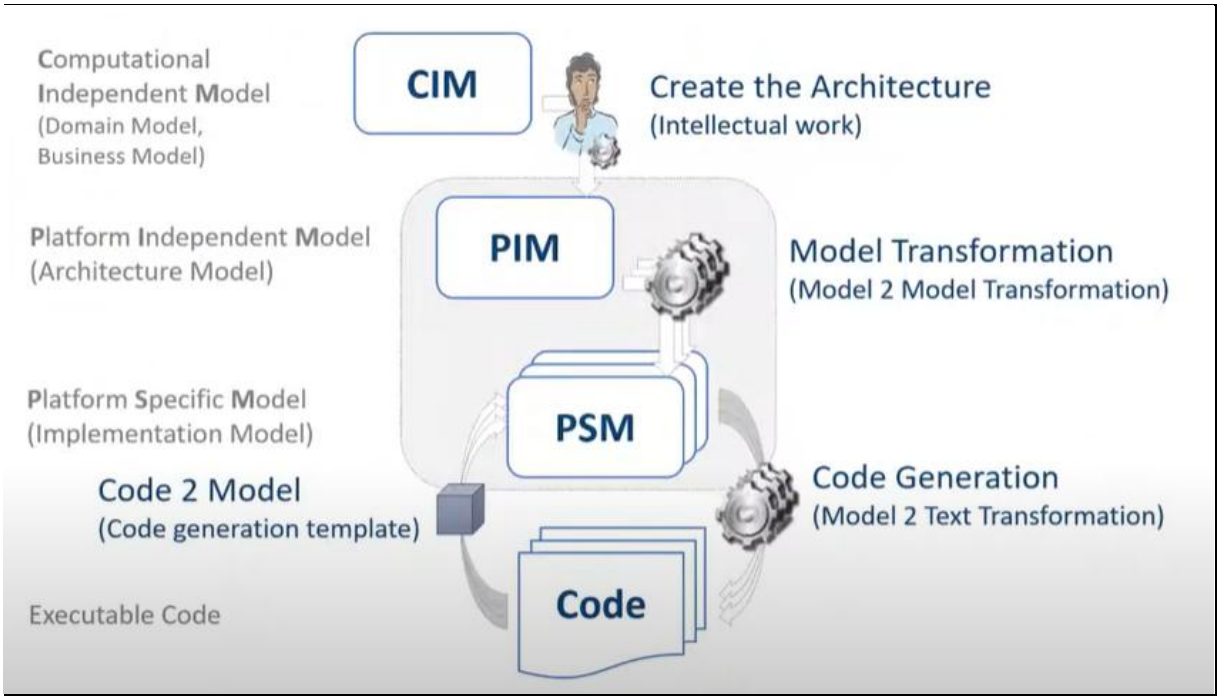
- Model Driven Architecture **MDA**, is an approach to software design, development and implementation spearheaded by the the **OMG** (**Object Management Group**), defines an

approach to IT system specification that separates the specification of system functionalities from the specification of the implementation of these functionalities on a particular technological platform.

- **Model-driven architecture** (**MDA**) is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Model Driven Architecture is a kind of domain engineering, and supports model-driven engineering of software systems. It was launched by the Object Management Group (OMG) in 2001

This approach places the emphasis on models, provides a higher level of abstraction during development, and enables significant decoupling between platform-independent models (**PIM**) and platform-specific models (**PSM**).

Of particular importance to **model-driven architecture** are the notions of **metamodel** and **model transformation**. Metamodels are defined at the OMG using the **MOF** (**Meta Object Facility**) standard. A specific standard language for model transformation called QVT has been defined by OMG. The OMG has also defined a model interchange mechanism based on XML called **XMI**.



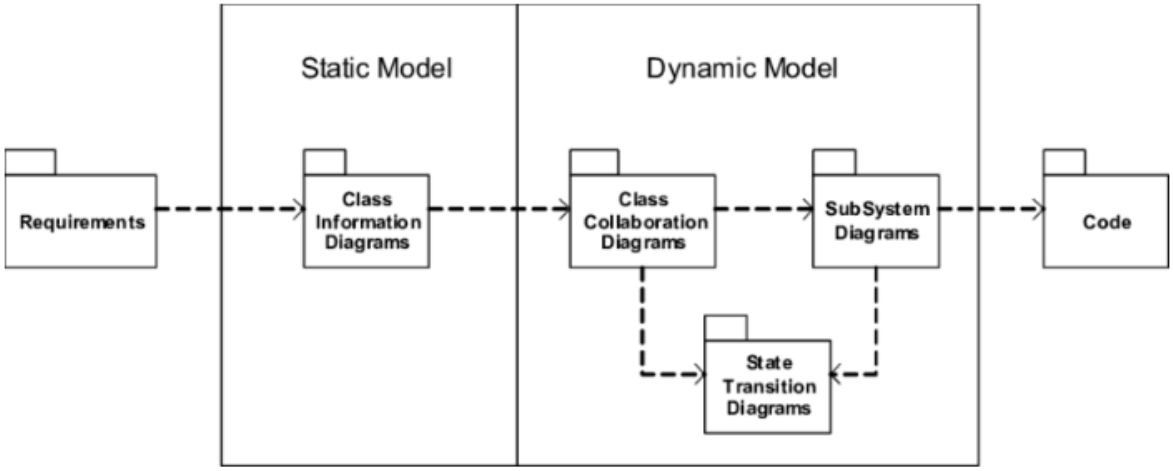
5.4 XUML Executable UML (**xtUML** or **xUML**) for MDA

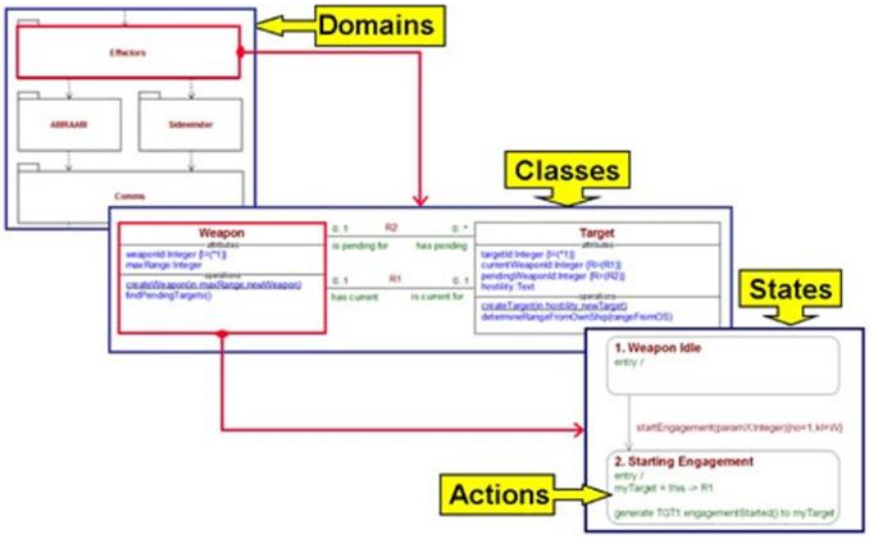
Is both a software development method and a highly abstract software language. It was described for the first time in 2002 in the book "Executable UML: A Foundation for Model-

Driven Architecture". The language "combines a subset of the UML (Unified Modeling Language) graphical notation with executable semantics and timing rules." The Executable UML method is the successor to the Shlaer–Mellor method.

Executable UML models "can be run, tested, debugged, and measured for performance.", and can be compiled into a less abstract programming language to target a specific implementation. Executable UML supports model-driven architecture (MDA) through specification of platform-independent models, and the compilation of the platform-independent models into platform-specific models.

The primary modelling constructs of xUML are illustrated in the figure below, which shows that:





1. Each system is partitioned into domains, representing areas of expertise;
2. Each domain is partitioned into classes, which together will fulfil the data and processing requirements of each domain;

3. Each class can have a state machine, which processes asynchronous signals directed to that class by executing state actions;

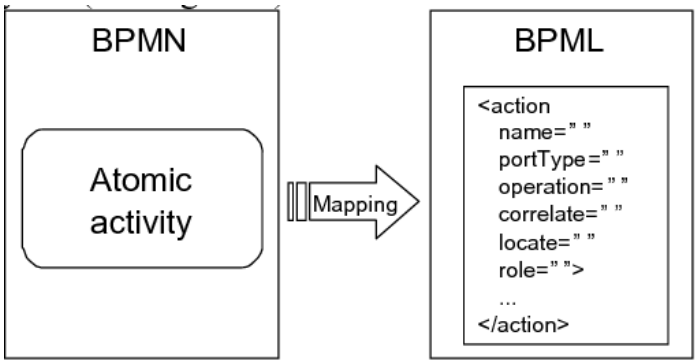
4. Each class can have operations, which perform synchronous processing. The state actions and operation methods are specified using a UML Action Language to preserve platform independence.

5.5 BPML

Business Process Modeling Language (**BPML**) is an XML-based language for business process modeling. It was maintained by the Business Process Management Initiative (BPMI) until June 2005 when BPMI and Object Management Group announced the merger of their respective business process management activities to form the Business Modeling and Integration Domain Task Force. It is **deprecated** (Dépassé, périmé, abandoné). since 2008. BPML was useful to OMG in order to enrich UML with process notation

BPML was designed as a formally complete language, able to model any process, and, via a business process management system, deployed as an executable software process **without generation of any software code**

Business Process Modeling Language (BPML) is an XML-based language used to describe (model) and run business processes. Each business process is defined by a single, unique BPML document known as a business process model (.bpml or .bp file). Each business process model is the definition of the process as it will be run in Sterling B2B Integrator.



BPML code includes *activities* and *elements* that work together in a business process.

An activity is a step in a business process, and may be comprised of multiple elements. Elements are defined components of code that provide structure and instructions regarding the activity they embody. BPML refers to entities outside of the business process as *participants*. An example of a participant is an inventory system.

- A simple activity is a single step in a business process. For examples, see Simple BPML Activities.
- A complex activity is an activity that comprises a set of steps in a business process. For examples, see Complex BPML Activities.

Activities within BPML code correspond to the icons you include in your business process models when you create them using the GPM. However, while an icon displays as a single, contained, unit, viewing the related BPML code shows the several elements that comprise the activity. You may refer to any service or other business process model component as an activity, but in the context of BPML, an activity may also be a BPML construct used to define the structure and progress of a business process flow.

The *operation activity* is a good example of this difference. The operation activity is the BPML component used to call a service within a business process.

Some BPML activities are represented by icons in the GPM (such as the Sequence icons and choice icons), while others are included within the service icons you can select in the GPM. For example, the operation activity is the BPML component used to call a service within a business process. If you use a text editor to write a business process model in BPML code, you include the operation activity, along with related elements, to call a service. If you create a business process model using the GPM, simply including the appropriate icon for a service automatically builds the operation activity into the BPML source code for the process model.

For example, the following figure shows the BPML for a business process; within it, the operation activity is

```
<process name="OverdueAckCheck">
  <sequence>
    <operation>
      <participant name="OverdueAck"/>
      <output message="Xout">
        <assign to="." from="*"></assign>
      </output>
      <input message="Xin">
        <assign to="." from="*"></assign>
      </input>
    </operation>

  </sequence>
</process>
```

name element for the operation

Total activity, comprised of elements

denoted by the <operation> name element:

The elements between <operation> and </operation> define the particulars of the activity, including calling the EDI Overdue Acknowledgment Check service. The </operation> element indicates the conclusion of the operation activity.

If you create business process models by direct-coding, you are responsible for including all of the appropriate elements that make up activities. When you use the GPM to create your process models, the icons you include automatically create the required BPML element components (although you may also need to configure service parameters).

You can relate the example in the first figure to the GPM display in the following way:

- The Start icon provides the BPML code for the process name saved with the business process model.
- The Sequence Start icon provides the BPML for the sequence element.
- The participant name, output message, assign, and input message elements are provided by parameters associated with the EDI Overdue Acknowledgment Check service.
- **Process Element**
- The process element defines an activity and is the root element of a business process model.

Process Element

Last Updated: 2022-05-11

The process element defines an activity and is the root element of a business process model.

- A process activity consists of exactly one simple or complex activity. The process completes after this activity completes.
- A process element begins and ends every business process model and can contain only one complex activity or one simple activity. The complex activity must contain all other activities required for the business process model in question. A process element uses the following syntax:

```
<process name>
<rule name/>*
( simpleActivity | complexActivity )
</process>
```

- A process element has a name attribute to indicate the name of the business process. The name attribute references any namespaces used in the business process.

```
<process name="ProcessCustomerOrder">
.
.</process>
```

Simple BPML Activities

Last Updated: 2022-05-11

Simple BPML activities enable a business process to communicate with a participant.

There are three types of simple activities:

- Produce – Sends messages to participants asynchronously
- Consume – Receives messages from participants asynchronously
- Operation – Exchanges messages with participants synchronously
- **Produce and Consume Activities**

  The produce and consume activities enable business processes to communicate. Produce and consume are used together.
- **Operation Activity**

  The operation activity invokes an action against a participant. Using an operation activity is the only way that you can call a service.

  Complex BPML Activities

  Last Updated: 2022-05-11

  A complex activity is a combination of simple activities that are child activities to the parent complex activity. The child activities can also be simple or complex.

  A process element can contain only one parent complex activity. The parent complex activity contains all of the simple and complex activities necessary to complete the business process.

  There are three kinds of complex activities:
- Sequence (serial)
- Choice (conditional)
- All (parallel)
- **Sequence Activity**

  A sequence activity runs a series of child activities in the order in which it lists them. When a process runs, all of the child activities are run. The sequence activity finishes only after the last child activity is finished.
  - A sequence activity runs a series of child activities in the order in which it lists them. When a process runs, all of the child activities are run. The sequence activity finishes only after the last child activity is finished.
  - The following example shows a sequence activity that contains two child activities: Check Inventory and Verify Credit Card. When the process runs, Sterling B2B Integrator runs Check Inventory first and Verify Credit Card second because that is the order in which the activities are listed in the sequence. The name attribute in the sequence element is optional.

```
<process name="ProcessCustomerOrder">
  <sequence>
    <operation name='Check Inventory'>
```

```
        <participant name='InventoryService' />
        <output message='checkStockRequest'>
          <assign to='ISBN'>1-56592-488-6</assign>
        </output>
        <input message name='checkStockResponse'>
          <assign to='foundBook' from='InStock' />
        </input>
      </operation>
  <operation name='Verify Credit Card'> </operation>
    </sequence>
  </process>
```

- **Choice Activity**

  The Choice activity makes decisions in the business process model and runs only one of the child activities it contains. Conditions determine which child activity runs.

  The Choice activity makes decisions in the business process model and runs only one of the child activities it contains. Conditions determine which child activity runs.

  You define the rules for the choice conditions. The only child activity to run is the one tied to the first case statement in the Choice activity that matches a rule. If none of the case statements in the Choice activity match a rule, the process continues to the next activity after the choice.

- **Branching**

  The choice activity makes it possible to model process branching. To model process branching, the process must evaluate one or more rules to reach a decision, and then specify which activity to run as a result of that decision.

- **All Activity**

  The all activity contains two or more complex child activities and runs all of them simultaneously. The all activity finishes only after the child activities are finished.

  The all activity contains two or more complex child activities and runs all of them simultaneously. The all activity finishes only after the child activities are finished.

  The following example contains three child activity sequences: Seq_1, Seq_2, and Seq_3. Sterling B2B Integrator begins these sequences at the same time—that is, the first operations in the sequences (operations A, C, and E) start simultaneously. The sequences continue to run independently until the last operation in each is completed. The all activity finishes when all three child activity sequences finish.

```
<all>
  <sequence name='Seq_1'>
    <operation name='A'> … </operation>
    <operation name='B'> … </operation>
  </sequence>

  <sequence name='Seq_2'>
    <operation name='C'> … </operation>
    <operation name='D'> … </operation>
  </sequence>

  <sequence name='Seq_3'>
    <operation name='E'> … </operation>
    <operation name='F'> … </operation>
    <operation name='G'> … </operation>
  </sequence>
</all>
```

- **Rule Element**

  The rule element defines a rule, the conditions by which the rule is met, and dependency on other rules.

The rule element defines a rule, the conditions by which the rule is met, and dependency on other rules.

- The condition element formulates an expression. The condition is met if the expression evaluates to true, or if the expression evaluates to false when the negative attribute is true.
- Multiple conditions inside a rule are listed in logical order.
- The rule element defines a rule that is referenced in a choice activity, used in an input element, or dependent on another rule. Dependencies are not affected by the order that rules are listed.

- **Condition Element**

  The contents of the condition element must correspond to an XPath expression.

The contents of the condition element must correspond to an XPath expression.

Sterling B2B Integrator expects an XPath expression to evaluate a condition.

Conclusion

Agile was invented in the software industry but nowadays businesses in all industry sectors make use of it.

Agility is the ability of a business as a whole to respond quickly to changes, especially external changes. For example, by adapting business processes or changing customer experiences.

Agility is a crucial factor for businesses that are in a digital transformation. There are many ways to achieve agility, including digital technologies, innovative product design, process agility, and culture shifting.

Agility is the ability to rapidly change body direction, accelerate, or decelerate. It is influenced by balance, strength, coordination, and skill level. Agility can be improved by first developing an adequate base of strength and conditioning that is appropriate for the difficulty level of the athlete.

Agility for software projects can be defined as the ability to quickly adapt to changing conditions or user requirements. This requires effective collaboration among stakeholders and having an organized team which encourages collaboration and communication among all who serve it and is in complete control of the tasks completed. This also requires customers to be drawn to the team and performing swift incremental delivery of the software.

An agile software process must be adaptable, manage unpredictability and adapt incrementally.

Mohamed Khider University – BISKRA

Computer science department

Level: 2nd Master (Professional) **Module: SIA-2**

Teacher: Ouaar Hanane **Year: 2024/2025**

Hands-out N° 4: Agile software tools

Main goal: Edit the best report of the blow tree tasks.

Task 1: Agile technologies

1. **Study the goal of each software tools using clear examples, show the way to install and to run all this tools:**
   - **crossCheck**
   - Goals
   - Example

   - **Archetypes (maven)**
   - Goals
   - Example

   - **metapatterns (python)**
   - Goals
   - Example

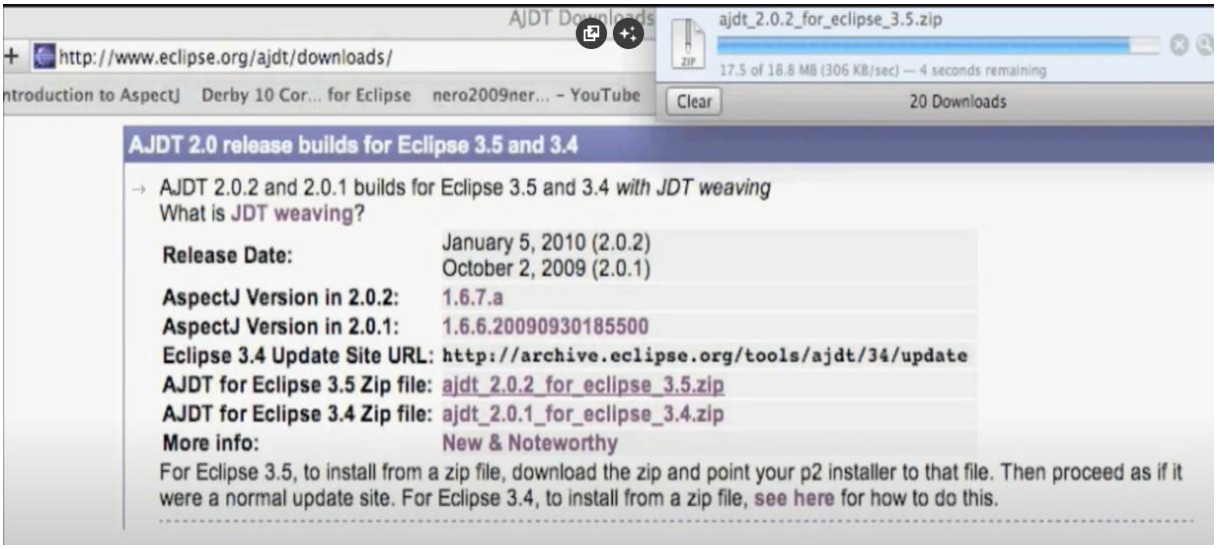
2. **Study the parsing Java to UML**
   - Tool
   - Example

3. **Study the generation from UML to JAVA**
   - Tool
   - Example
4. **Study the following transformation cases using examples**
   - UML to Design patterns
   - Design Patterns to UML

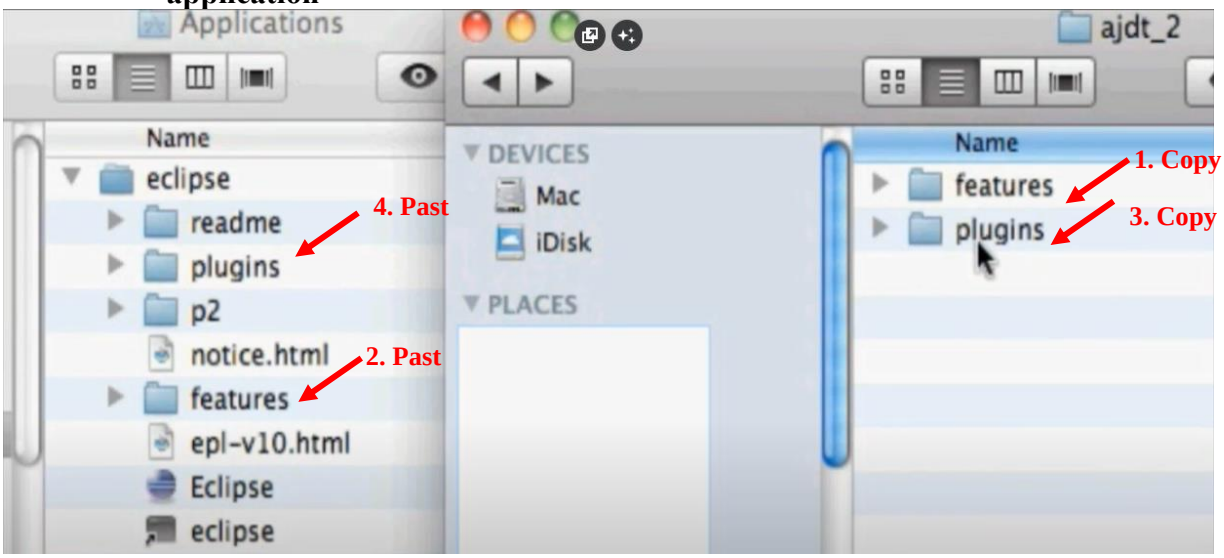
Task 2: Building POO vs AOP code using AspectJ

In order to study the use of AOP inside the POO environment:

- **Download and install the AspectJ tool**

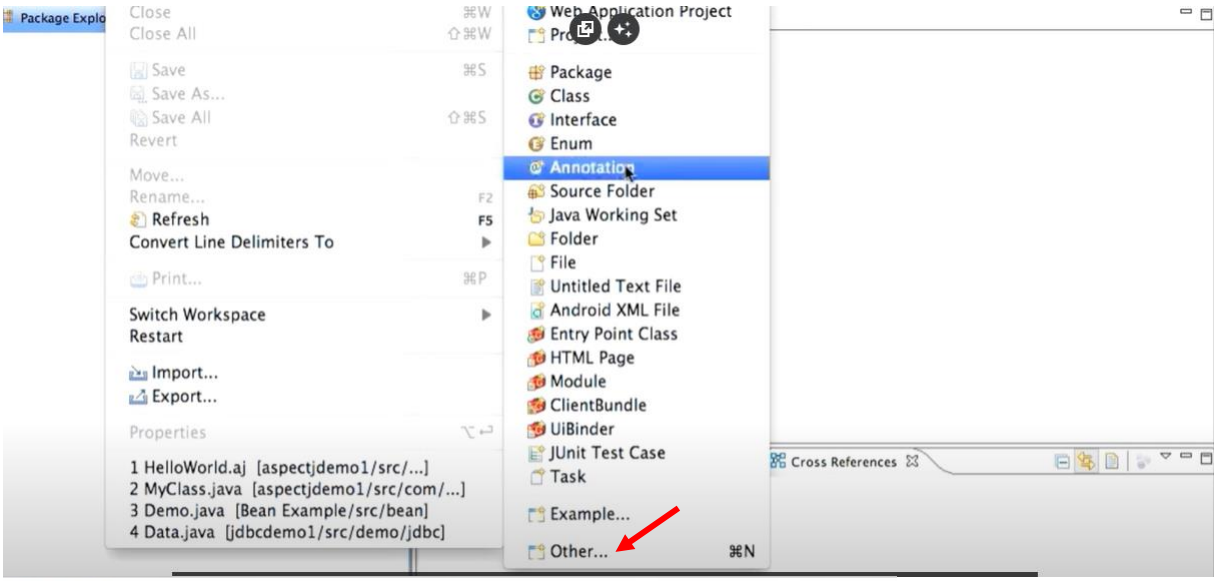


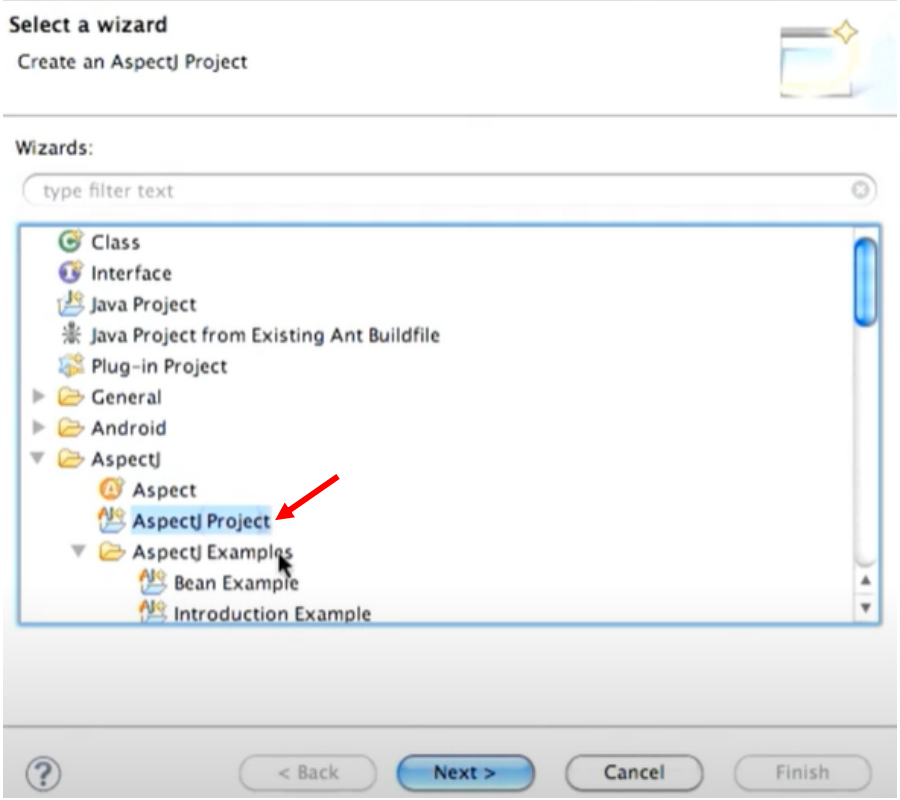
- **Copy all files of features and plugins in 'adjdt_2' and past them in 'application'**



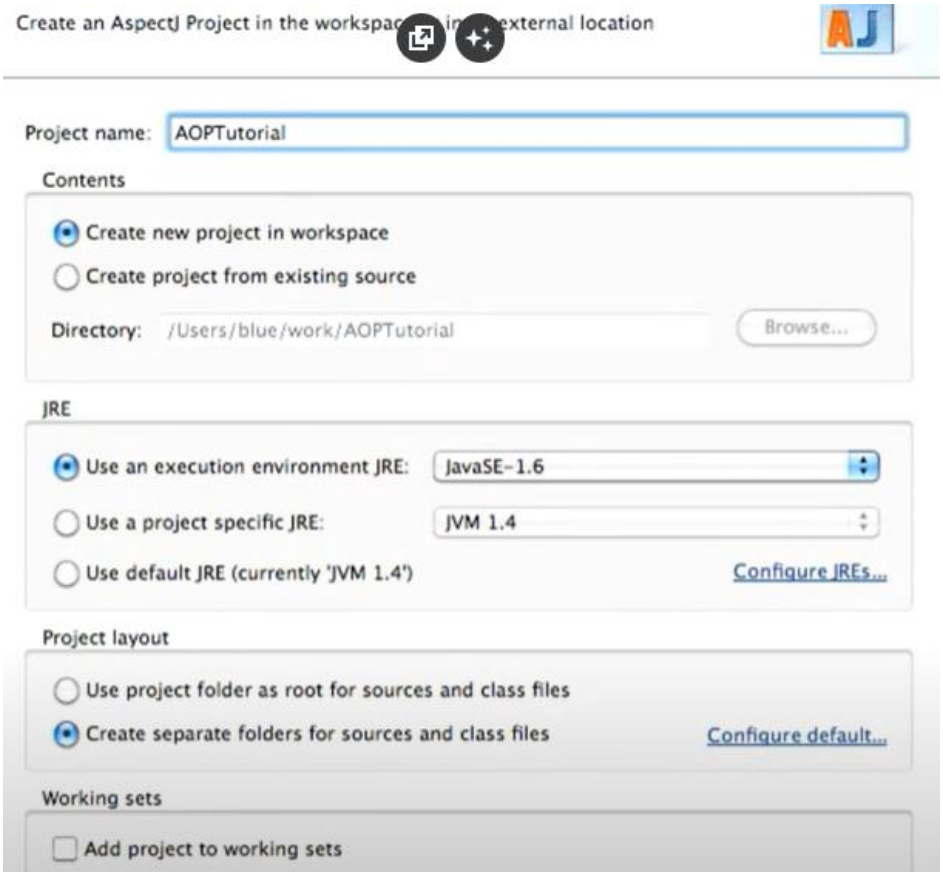
Create an AOP project in eclipse

- **Create and run Java application that use AspectJ**
  *** Open java editor (eclipse)**
  *** Create a new aspect project**

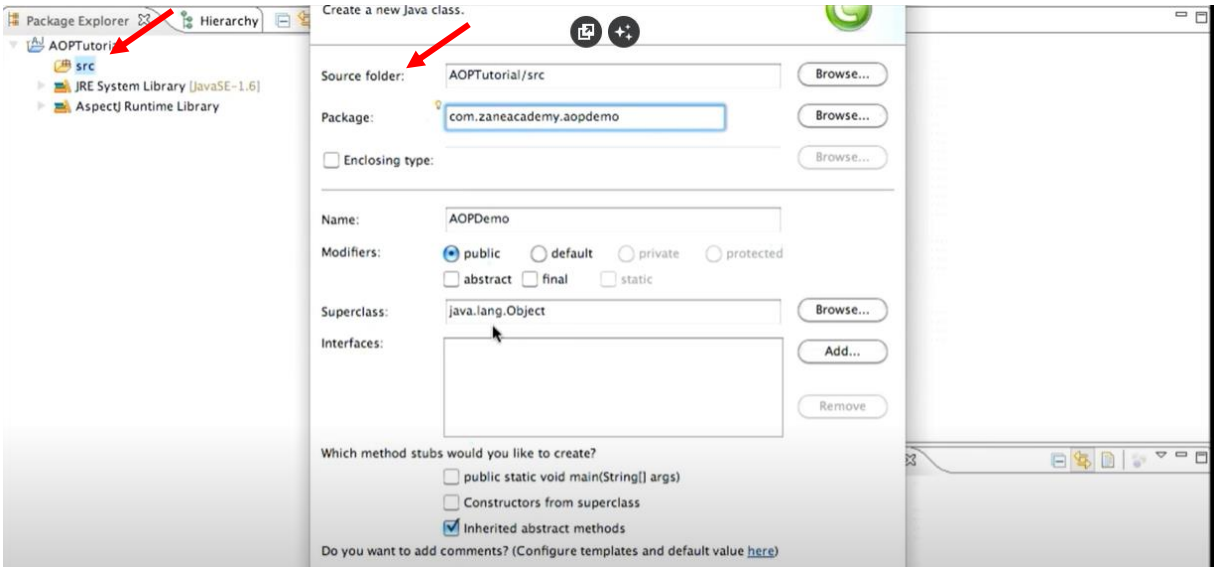


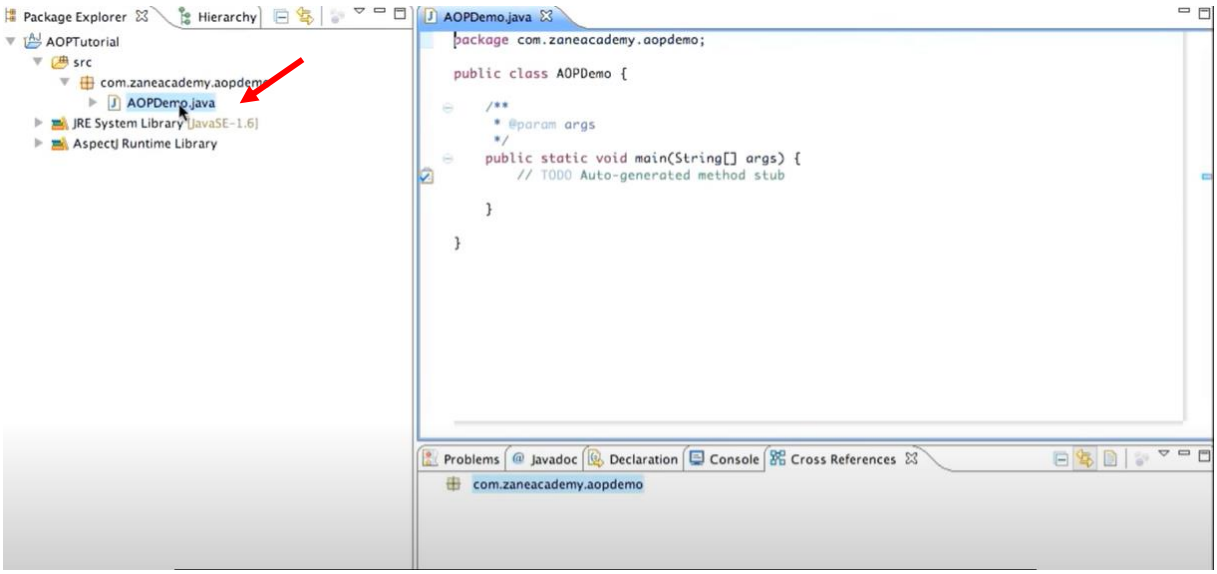


*** Full the all information**

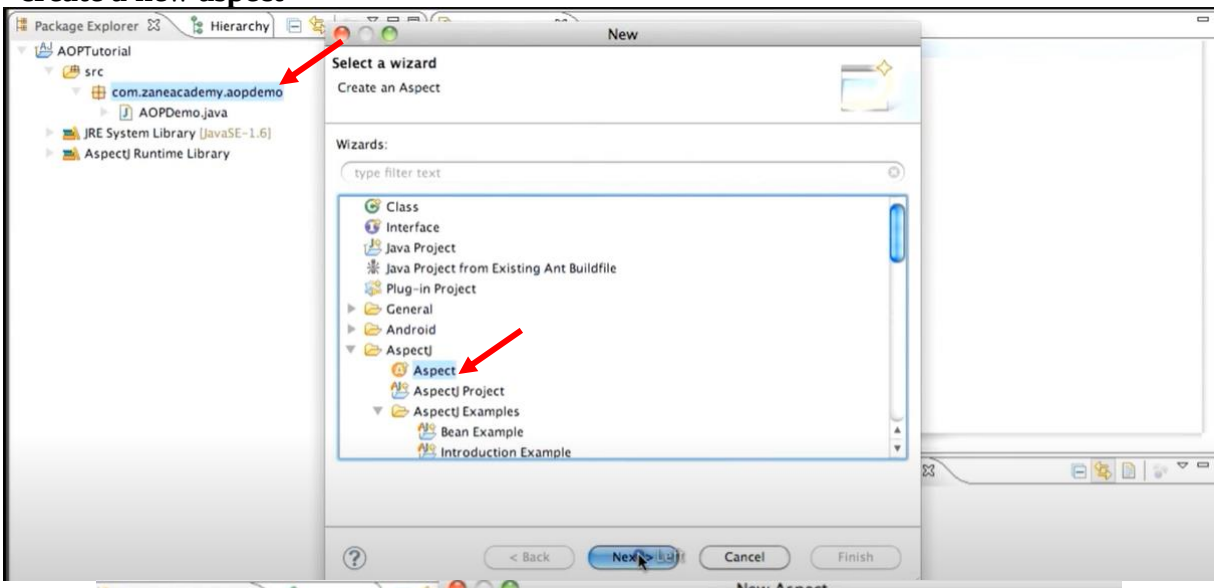

Create an AspectJ Project in the workspace in external location
Project name: AOPTutorial
Contents
Create new project in workspace
Create project from existing source
Directory: /Users/blue/work/AOPTutorial
Browse...
JRE
Use an execution environment JRE: JavaSE-1.6
Use a project specific JRE: JVM 1.4
Use default JRE (currently 'JVM 1.4')
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets

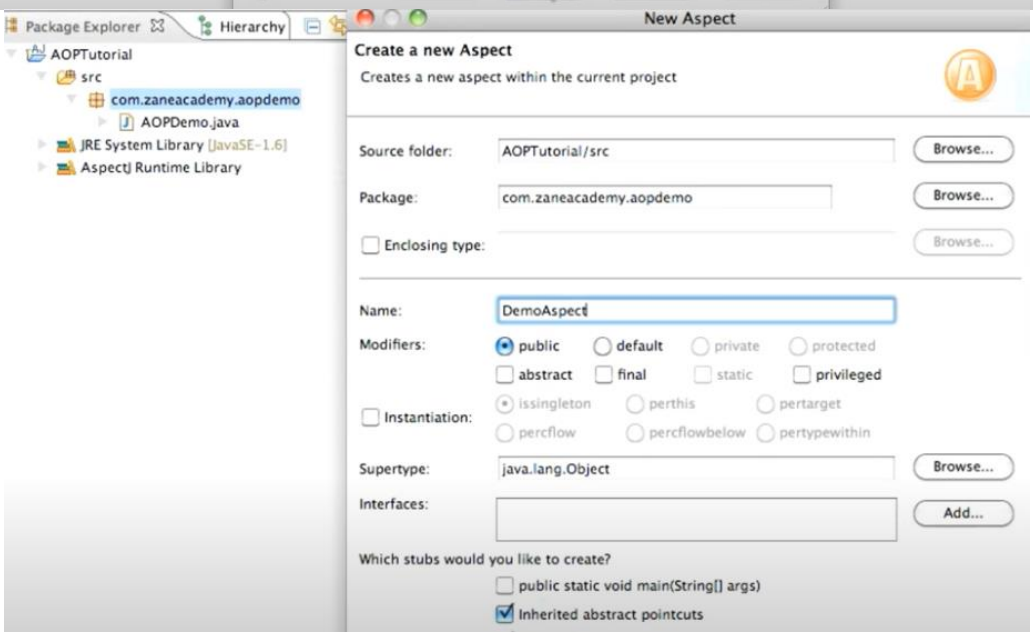
*** New/ class**

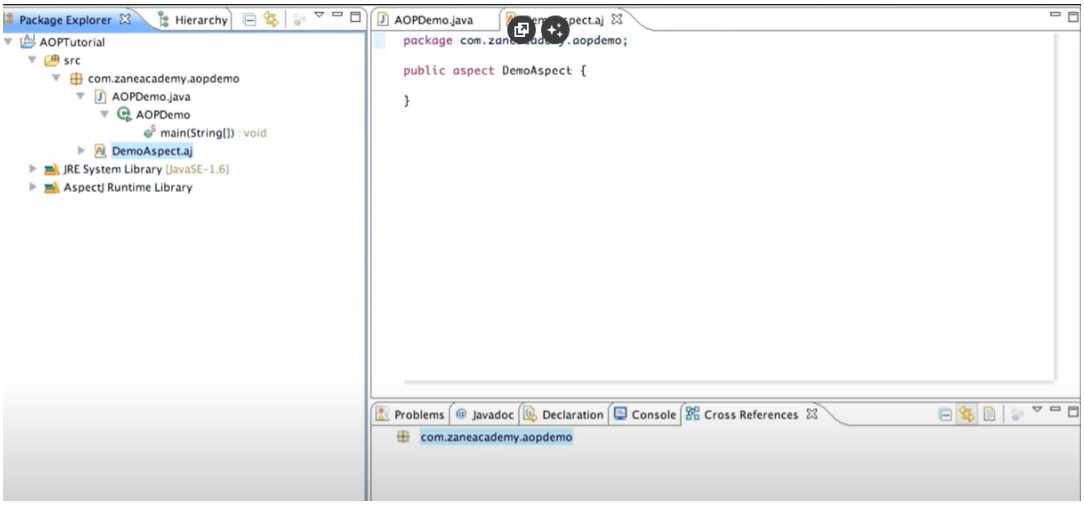

Package Explorer
Hierarchy
AOPTutorial
src
JRE System Library [JavaSE-1.6]
AspectJ Runtime Library
Create a new Java class.
Source folder: AOPTutorial/src
Package: com.zaneacademy.aopdemo
Enclosing type:
Name: AOPDemo
Modifiers: public default private protected
abstract final static
Superclass: java.lang.Object
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)



***create a new aspect**

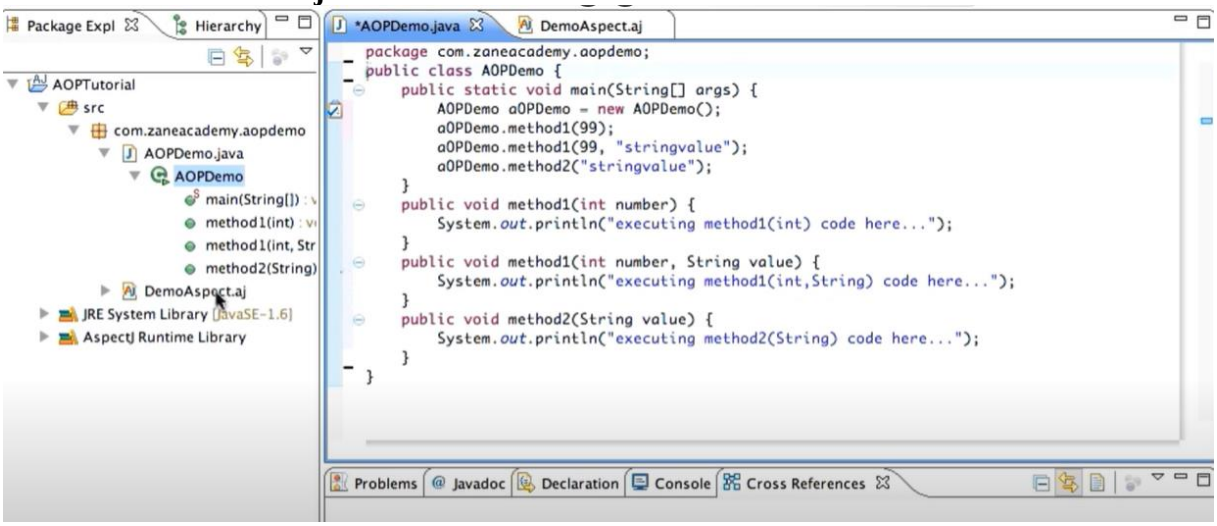


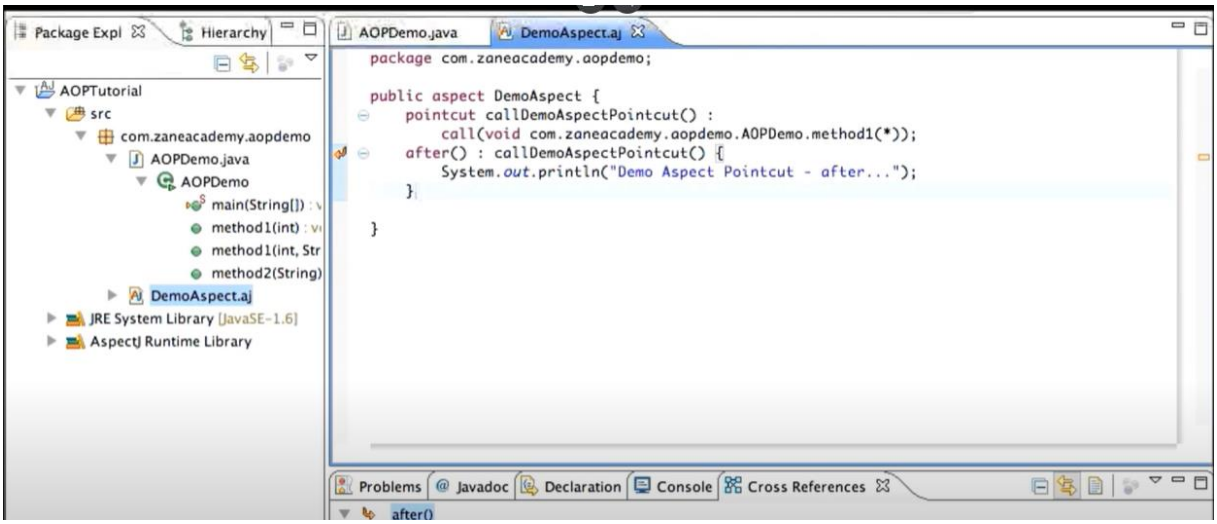




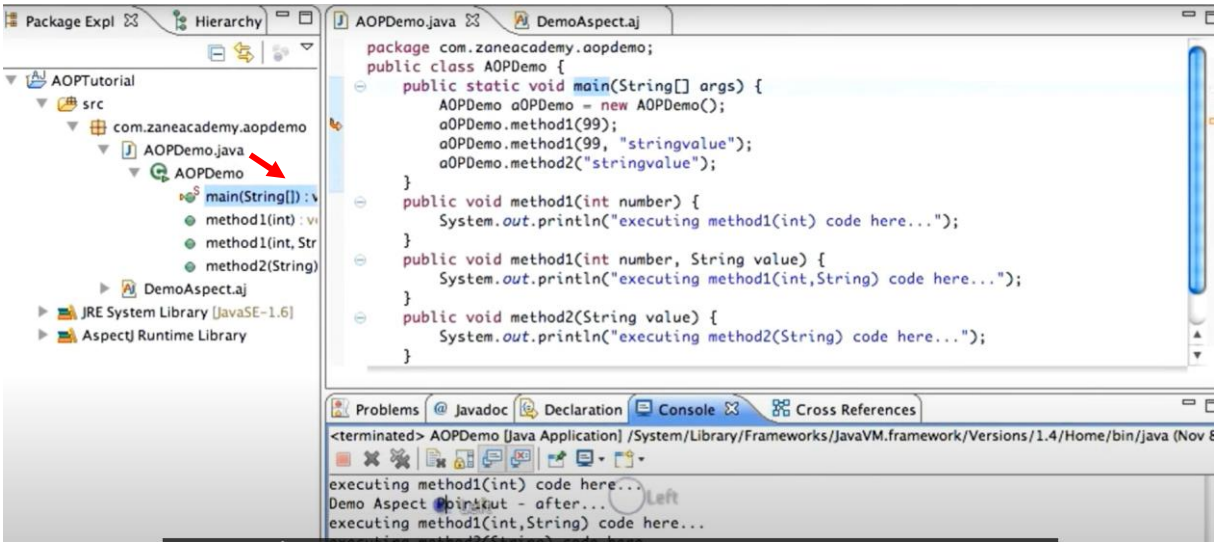
*** Create methods**

*** Create object of this class**

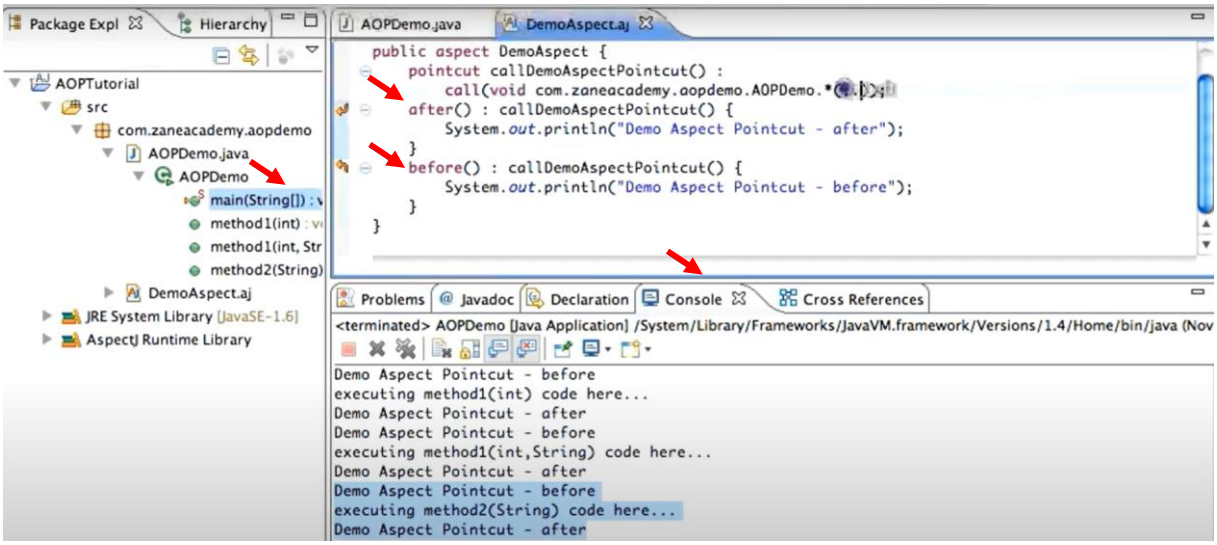




- **Run the main() of your class**



- **Before and after functions**
  *** Run the main () class**



Applying this example and try another one for your choice

Task 3: MDA transformation modal using XUML

- **Download one of the XUML tool**
- **Select a simple use case**
- **Propose the PIM modal**
- **Applying transformation toward a PSM modal**
- **Generate the associate code**
- **Edit the best pdf report**