SYSTEM ANALYSIS AND DESIGN

INTRODUCTION

Overview of System Analysis and Design Process

System analysis and design is a structured approach used to develop or improve information systems that address specific business needs or problems. The process involves understanding the requirements of the system, designing its structure, and planning its implementation.

1. Understanding the Problem/Opportunity
2. Feasibility Study
3. Requirement Gathering and Analysis
4. System Design
5. Prototyping
6. Development
7. Testing
8. Implementation
9. Maintenance and Evaluation
10. Documentation

Understanding the Problem/Opportunity

The first step is to identify and define the problem or opportunity that the new system should address. This involves discussions with stakeholders, end-users, and other relevant parties to gather information and understand the current workflow.

Feasibility Study

Before proceeding, a feasibility study is conducted to assess whether it is technically, economically, and operationally viable to develop the system. Factors such as cost, time, resources, and potential risks are analyzed.

Requirement Gathering and Analysis

This stage involves gathering detailed requirements for the new system. Analysts interact with stakeholders, conduct interviews, surveys, and workshops to gather information about functional and non-functional requirements.

System Design

Once the requirements are gathered, the system design phase begins. It includes creating a blueprint for the new system, specifying how various components will interact, and defining the overall system architecture.

Prototyping

In some cases, a prototype of the system may be created to give stakeholders a tangible representation of the proposed solution. This helps in gathering feedback early in the process and refining the design.

Development

The actual development of the system takes place in this phase. Programmers and developers work to build the software, and database designers create the database structure.

Testing

The system is thoroughly tested to ensure that it meets the specified requirements and is free from defects. Different types of testing, such as unit testing, integration testing, and user acceptance testing, are performed

Implementation

The system is deployed and implemented in the organization. This may involve data migration, training of end-users, and a transition plan from the old system to the new one.

Maintenance and Evaluation

Once the system is operational, it requires ongoing maintenance to fix bugs, make improvements, and adapt to changing requirements. Additionally, regular evaluations are conducted to measure the system's performance and effectiveness

Documentation

Throughout the entire process, documentation is crucial. It includes requirements documents, design specifications, user manuals, and maintenance guides. Proper documentation helps in the smooth operation and maintenance of the system.

Importance of Effective System Analysis and Design

Effective system analysis and design is important for several reasons:

- ✓ Meets Business Requirements
- ✓ Reduces Development Costs and Time
- ✓ Enhances System Performance
- ✓ Improves User Satisfaction
- ✓ Reduces Risks and Errors
- ✓ Optimizes Resource Utilization
- ✓ Facilitates System Integration
- ✓ Supports Scalability and Flexibility
- ✓ Eases System Maintenance and Upgrades
- ✓ Enables Decision Making
- ✓ Increases Competitive Advantage

Introduction to SDLC and its phases

Introduction to SDLC and its phases

SDLC stands for Software Development Life Cycle, which is a structured and systematic approach to developing software applications.

It provides a framework for guiding the processes involved in software development from the initial concept to its deployment and maintenance.

The SDLC aims to deliver high-quality software that meets the requirements of stakeholders while adhering to timelines and budgets.

Each phase in the SDLC has specific objectives, deliverables, and activities that contribute to the overall development process.

The typical SDLC consists of the following phases

1. Requirements Gathering and Analysis
2. System Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

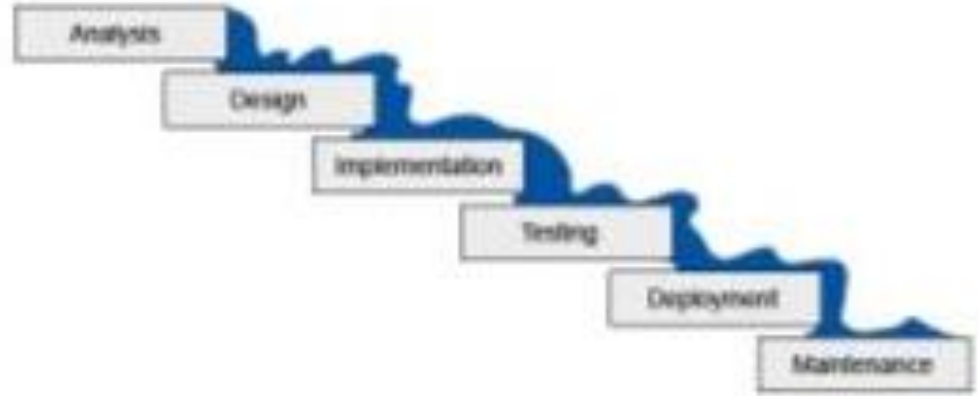
Introduction to SDLC and its phases

It's important to note that the SDLC is not a strictly linear process. It often follows an iterative or incremental approach, where some phases might be revisited or repeated based on feedback and evolving needs. This allows for flexibility and continuous improvement throughout the software development process.

The SDLC provides a systematic way to manage the complexities of software development, ensuring that the final product is of high quality, meets user needs, and is delivered on time and within budget. By following the SDLC, development teams can minimize risks, improve communication between stakeholders, and produce successful software applications.

Waterfall vs. Agile Methodologies

Waterfall



Agile

DevOps

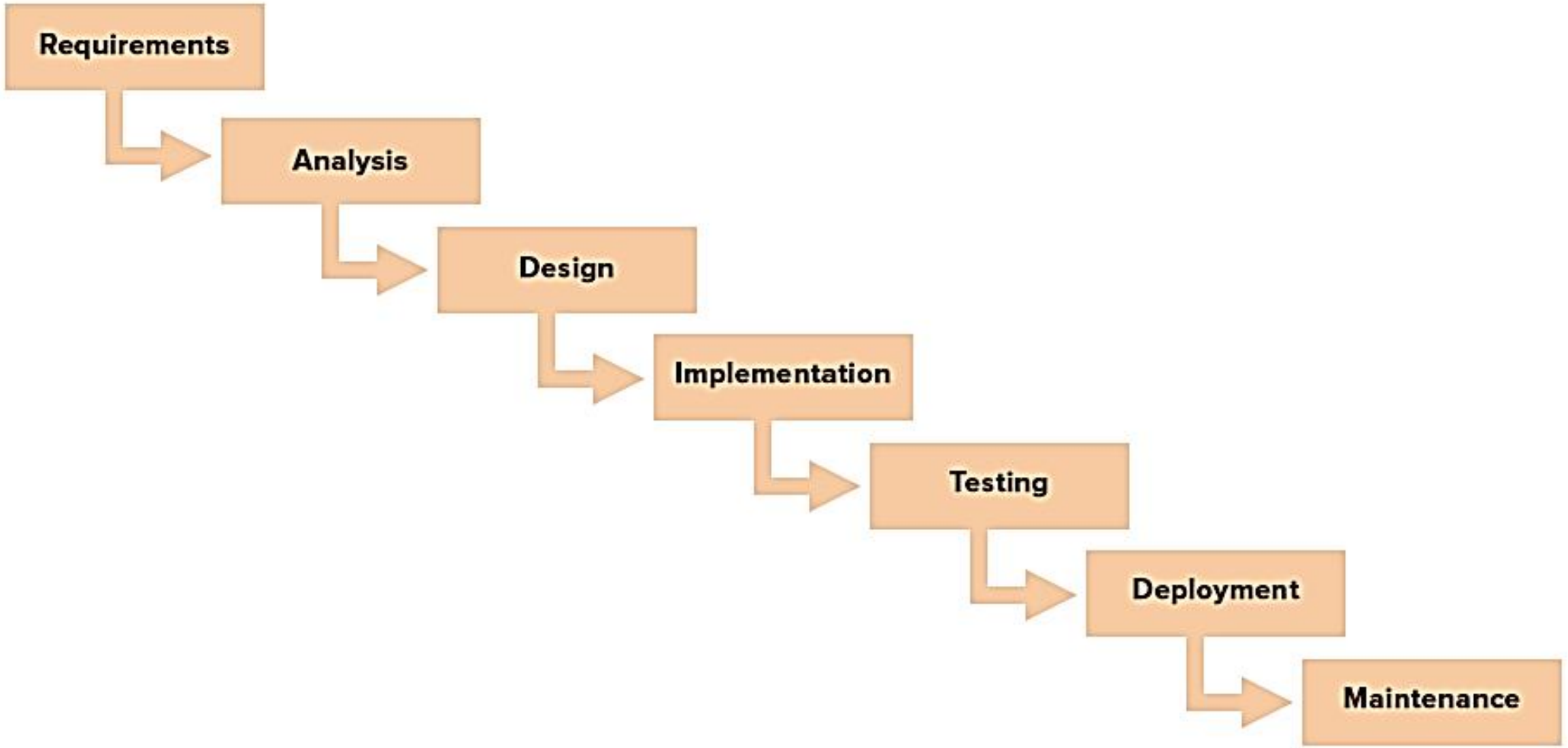
Waterfall

Agile

DevOps

WATERFALL

SDLC MODELS


Requirements
Analysis
Design
Implementation
Testing
Deployment
Maintenance

WATERFALL MODELS CHARACTERISTICS

- Plan-driven approach.
- Linear/Sequential approach to software development.
- Results of each phase is one or more documents that are signed-off.
- Following phase should not start until the previous phase has finished.
- Must plan and schedule all the process activities before starting work on them.

THE WATERFALL METHOD HAS SEVERAL ADVANTAGES, SUCH AS:

- Ease of use: This model is easy to understand and use. The division between stages is intuitive and easy to grasp regardless of prior experience.
- Structure: The rigidity of the Waterfall method is a liability, but can also be a strength. The clear demarcation between stages helps organize and divide work. Since you can't go back, you have to be "perfect" in each stage, which often produces better results.
- Documentation: The sharp focus on gathering and understanding requirements makes the Waterfall model heavily reliant on documentation. This makes it easy for new resources to move in and work on the project when needed.

DISADVANTAGES

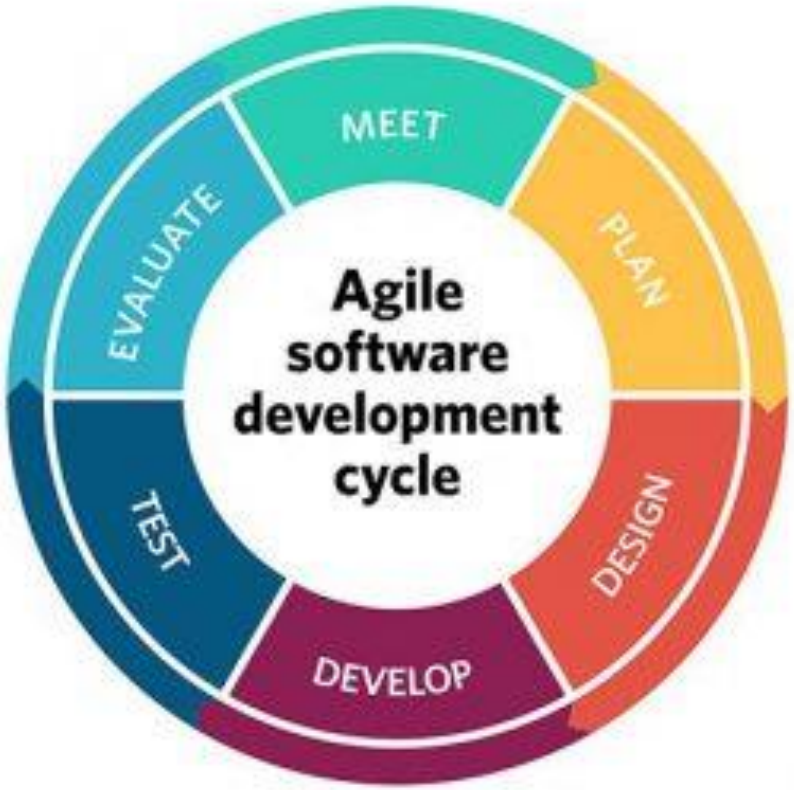
- Higher risk: The rigidity of this methodology means that if you find an error or need to change something, you have to essentially start the project from the beginning. This substantially increases the risk of project failure.
- Front-heavy: The entire Waterfall approach depends heavily on your understanding and analyzing requirements correctly. Should you fail to do that - or should the requirements change - you have to start over. This lack of flexibility makes it a poor choice for long and complex projects.

IT WORKS BEST FOR THE FOLLOWING PROJECT TYPES:

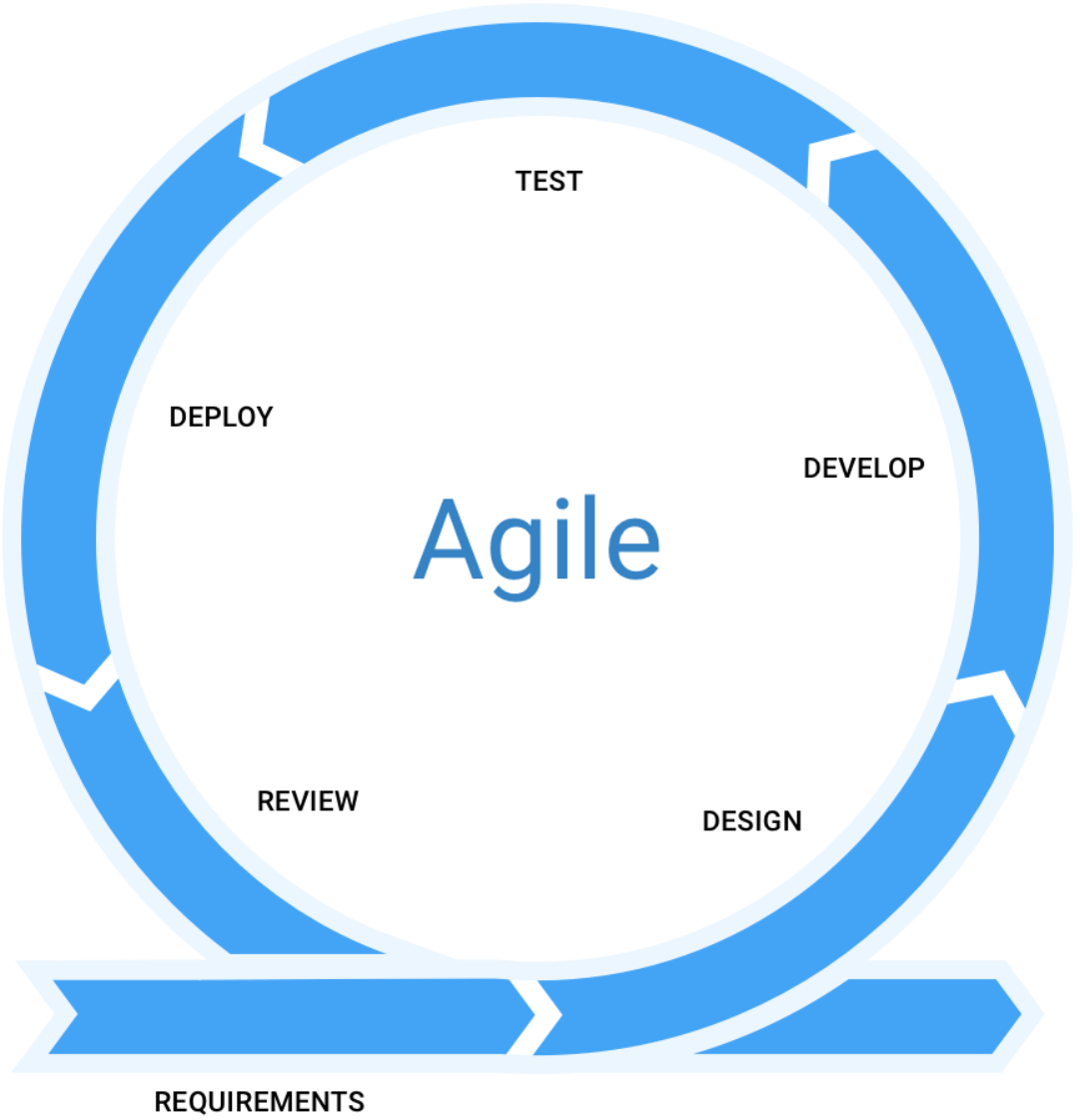
- Short, simple projects
- Projects with clear and fixed requirements
- Projects with changing resources that depend on in-depth documentation

AGILE

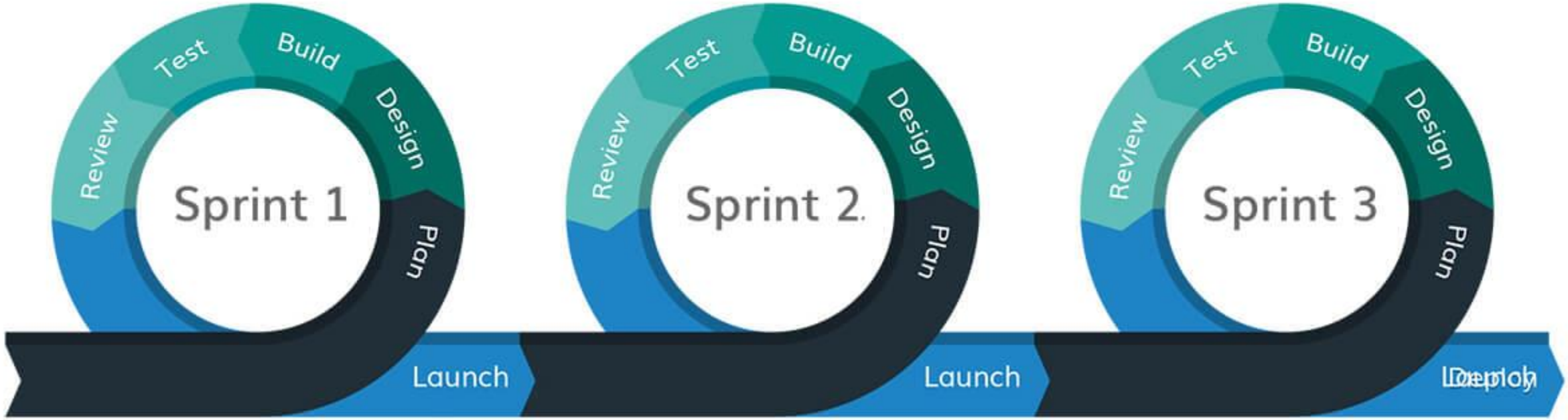
SDLC MODELS


MEET
PLAN
DESIGN
DEVELOP
TEST
EVALUATE
Agile
software
development
cycle

Agile model


TEST
DEPLOY
DEVELOP
Agile
REVIEW
DESIGN
REQUIREMENTS

Agile Methodology



CHARACTERISTICS

- Focus on satisfying the customer through early and frequent releases of valuable software integrating the customer’s prioritized functionalities.
- Welcome changing requirements even late in the development of the product.
- Frequently delivering working software to customers.
- Encourage collaboration between development teams, business people and customers.
- Working software is the primary measure of progress.

ADVANTAGES:

- Flexibility and freedom: Since there are no fixed stages or focus on requirements, it gives your resources much more freedom to experiment and make incremental changes. This makes it particularly well-suited for creative projects.
- Lower risk: With Agile management, you get regular feedback from stakeholders and make changes accordingly. This drastically reduces the risk of project failure since the stakeholders are involved at every step.

DISADVANTAGES:

- No fixed plan: The Agile approach emphasizes responding to changes as they occur. This lack of any fixed plan makes resource management and scheduling harder. You will constantly have to juggle resources, bringing them on/off on an ad-hoc basis.
- Collaboration-heavy: The lack of a fixed plan means all involved departments - including stakeholders and sponsors - will have to work closely to deliver results. The feedback-focused approach also means that stakeholders have to be willing (and available) to offer feedback quickly.

THE FLEXIBILITY OF THE AGILE APPROACH MEANS THAT YOU CAN ADAPT IT TO DIFFERENT TYPES OF PROJECTS.

THAT SAID, THIS METHODOLOGY WORKS BEST FOR:

- When you don't have a fixed end in mind but have a general idea of a product.
- When the project needs to accommodate quick changes.
- If collaboration and communication are your key strengths (and planning isn't)

Role of System Analysts and Designers in SDLC

System Analyst

- ✓Requirements Gathering
- ✓Requirements Analysis
- ✓System Modeling
- ✓Collaboration with Stakeholders

System Designer

- ✓System Architecture
- ✓Detailed Design
- ✓Technology Selection
- ✓Design Validation
- ✓Integration Planning

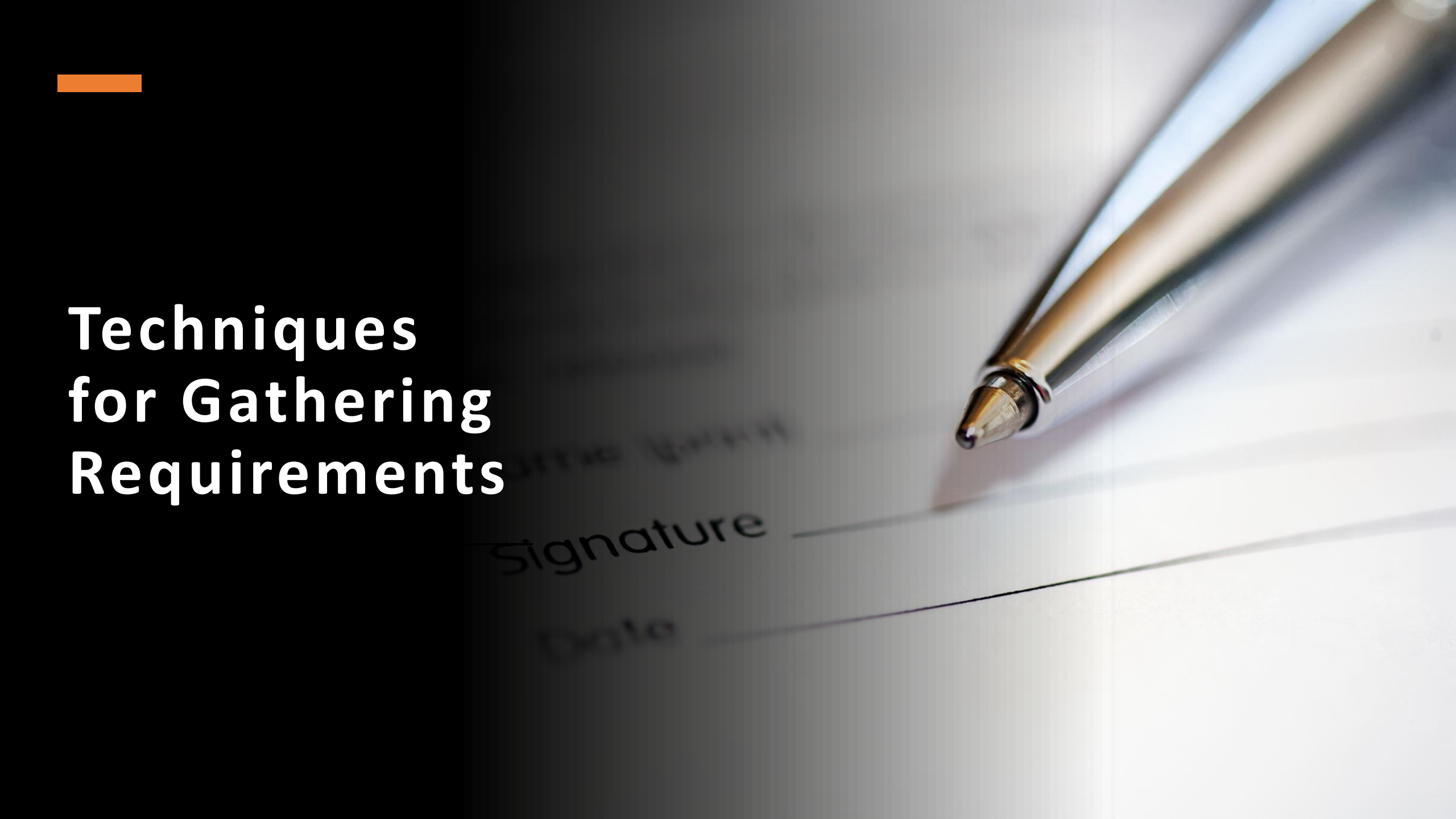
Understanding User Requirements and System Needs

Understanding user requirements and system needs is a critical step in the Software Development Life Cycle (SDLC) that lays the foundation for a successful software project.

Here are some instructions to effectively gather and understand user requirements and system needs:

1. Identify Stakeholders
2. Conduct Interviews and Workshops
3. Ask Open-Ended Questions
4. Create User Personas and Scenarios
5. Document Requirements
6. Prioritize Requirements
7. Differentiate between Functional and Non-Functional Requirements
8. Validate and Verify Requirements
9. Engage in Prototyping (if applicable)
10. Involve End-Users Throughout the Process
11. Collaborate with System Analysts and Designers
12. Manage Changes

Techniques for Gathering Requirements



The choice of techniques depends on factors such as the project's complexity, the nature of the system, the number of stakeholders involved, and the available resources. Here are some commonly used techniques for gathering requirements:

1. Interviews
2. Questionnaires and Surveys
3. Workshops and Focus Groups
4. Observation
5. Prototyping
6. Use Case Modeling
7. Data Gathering Techniques
8. Brainstorming
9. Document Analysis
10. Joint Application Development (JAD)
11. Benchmarking
12. Contextual Inquiry
13. Storyboarding

Overview of Use Case Modeling

Use case modeling is a popular and effective technique used in system analysis and design to capture, analyze, and document the functional requirements of a software system.

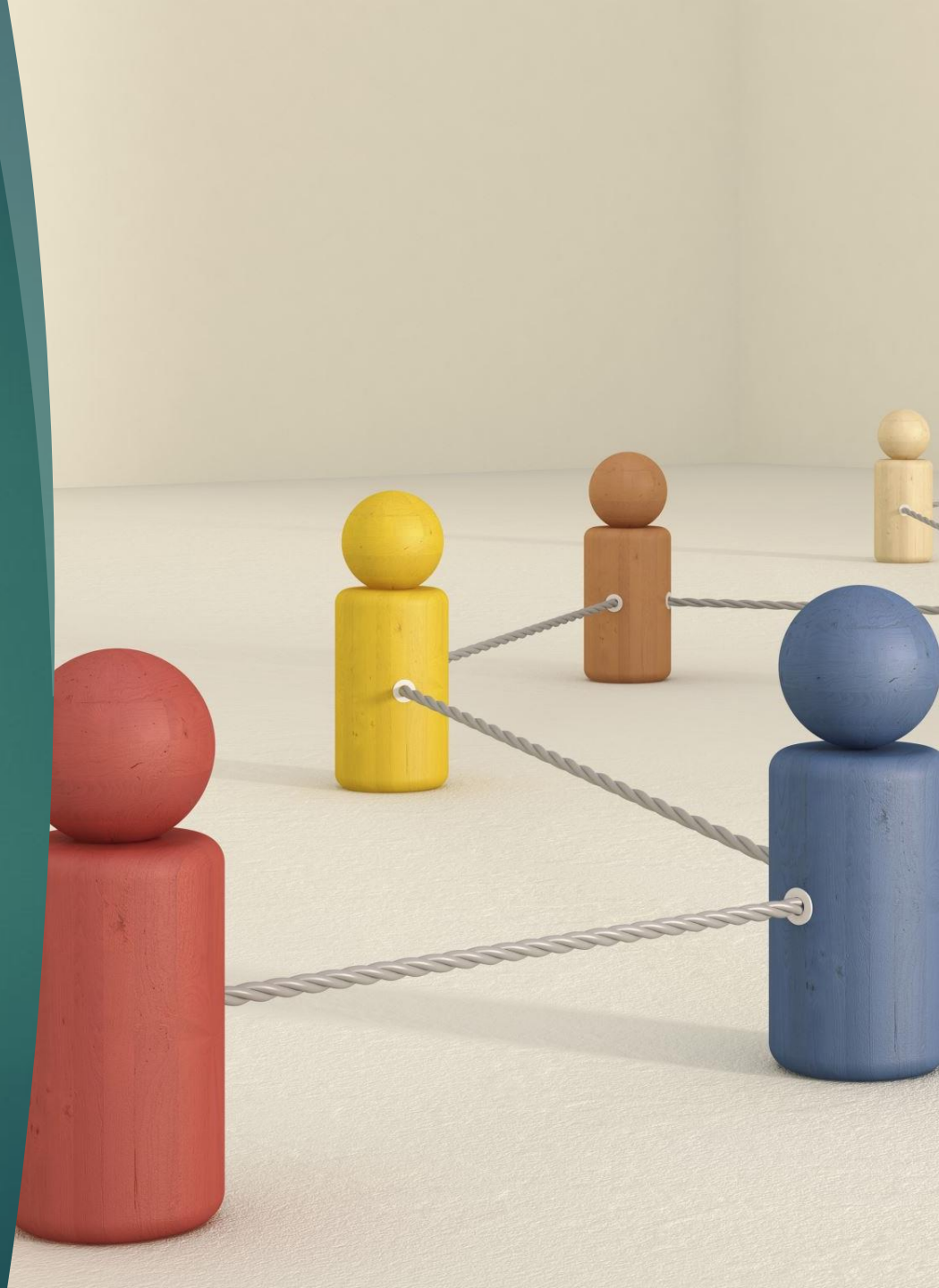
It focuses on describing how users interact with the system to achieve specific goals or tasks. Use case modeling helps in understanding the system's behavior from the end-user's perspective, ensuring that the software meets their needs and expectations.

1. Use Case:

A use case represents a specific user goal or task that the system needs to support. It describes a sequence of interactions between the user (actor) and the system, resulting in a valuable outcome for the user. Use cases are written in plain language and should be easily understandable by all stakeholders.

2. Actors:

Actors are external entities (users or other systems) that interact with the system to accomplish a task. They initiate the use cases and are usually represented as stick figures in use case diagrams. Actors can be primary (directly interacting with the system) or secondary (indirectly affecting the system).



3. Use Case Diagram:

A use case diagram provides a visual representation of the system's functionality and the interactions between actors and use cases. It shows the relationships between actors and use cases, helping to provide an overview of the system's scope.

4. Use Case Description:

A use case description provides a detailed narrative of each use case. It includes the preconditions (conditions that must be true before the use case starts), the main flow of events (the steps the user takes to achieve the goal), alternative flows (alternate paths for handling exceptions), and postconditions (the state of the system after the use case is complete).

5. Relationships between Use Cases:

Use cases can be related to each other through various relationships, such as "include," "extend," and "generalization." These relationships help in understanding how use cases are interconnected and how they share common functionality.

6. Scenario Modeling:

Scenario modeling involves creating specific instances of a use case to illustrate how the system will behave in real-life situations. Scenarios help in identifying additional requirements and validating the behavior of the system.

7. Iterative and Incremental Approach:

Use case modeling is often performed iteratively and incrementally. As the analysis and design process progresses, use cases are refined, modified, and extended based on feedback from stakeholders and new insights gained.

8. Traceability and Requirements Validation:

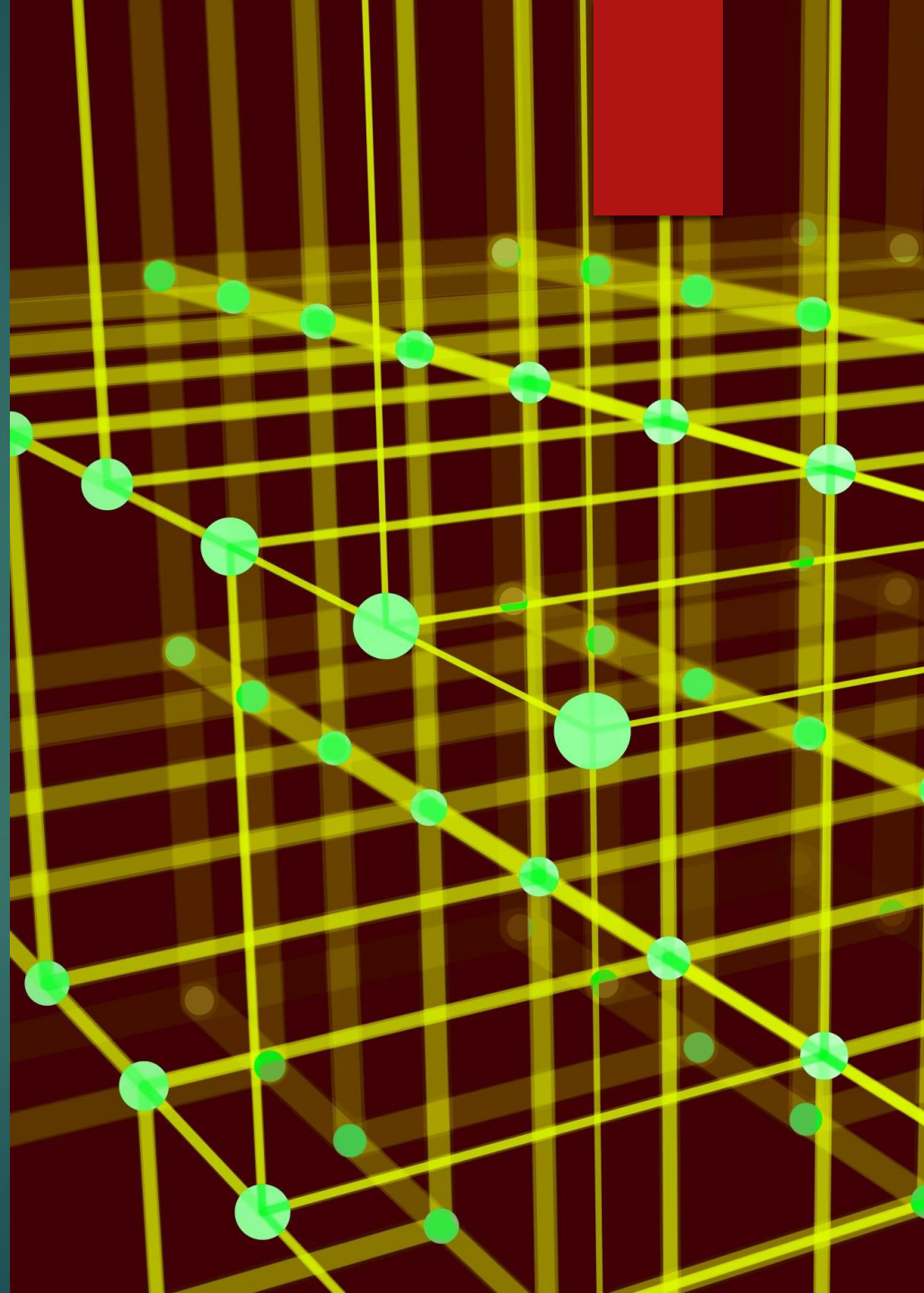
Use case modeling facilitates traceability between requirements and system behavior. Each use case corresponds to a specific requirement, enabling better validation and verification of requirements throughout the development process.

Creating Use Case Diagrams and Scenarios

- UML stands for Unified Modeling Language. It is a standardized visual modeling language used in software engineering to design, specify, visualize, and document software systems.

- Use case diagrams in UML are used during the requirements phase of software development – they usually put the system requirements into a diagram format, and it's easy to see what actions a system must support immediately.

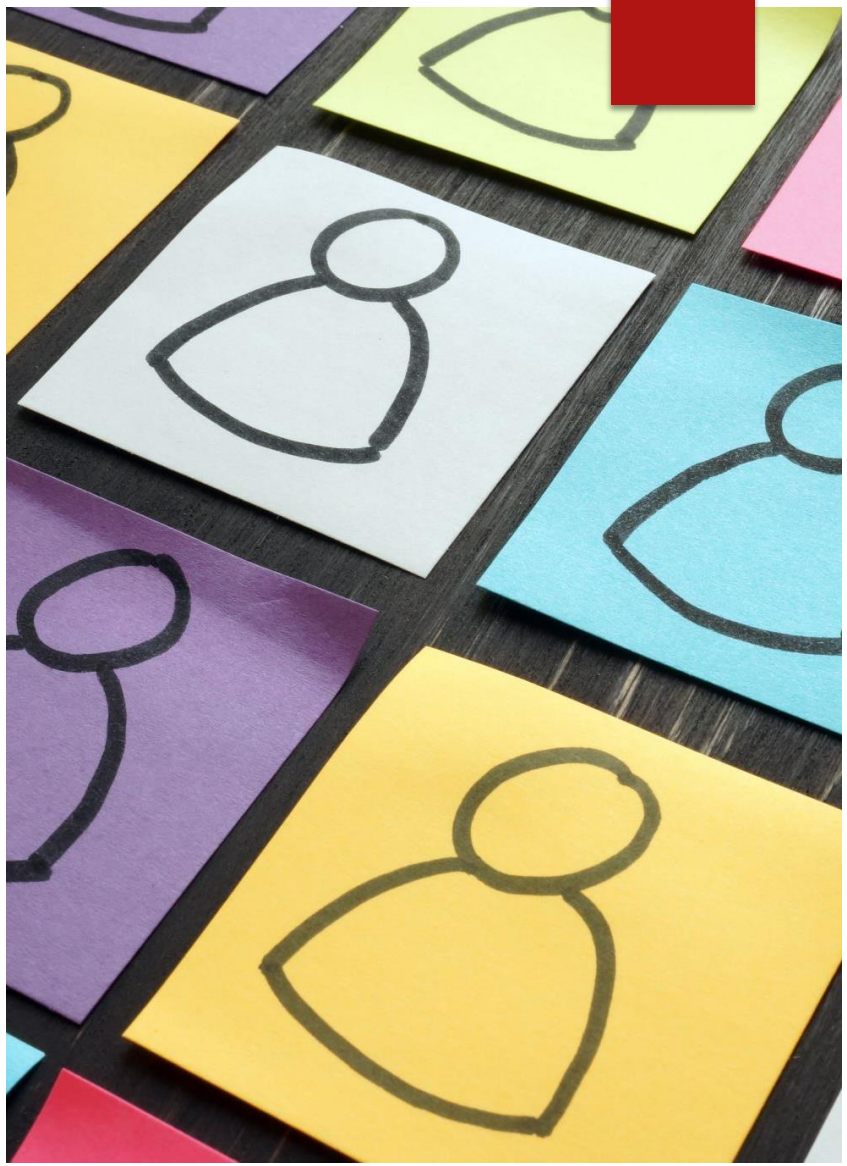
- Along with other UML diagrams, such as activity, sequence and component diagrams, use case diagrams help you to visualize your software and requirements, before jumping in and starting to program.

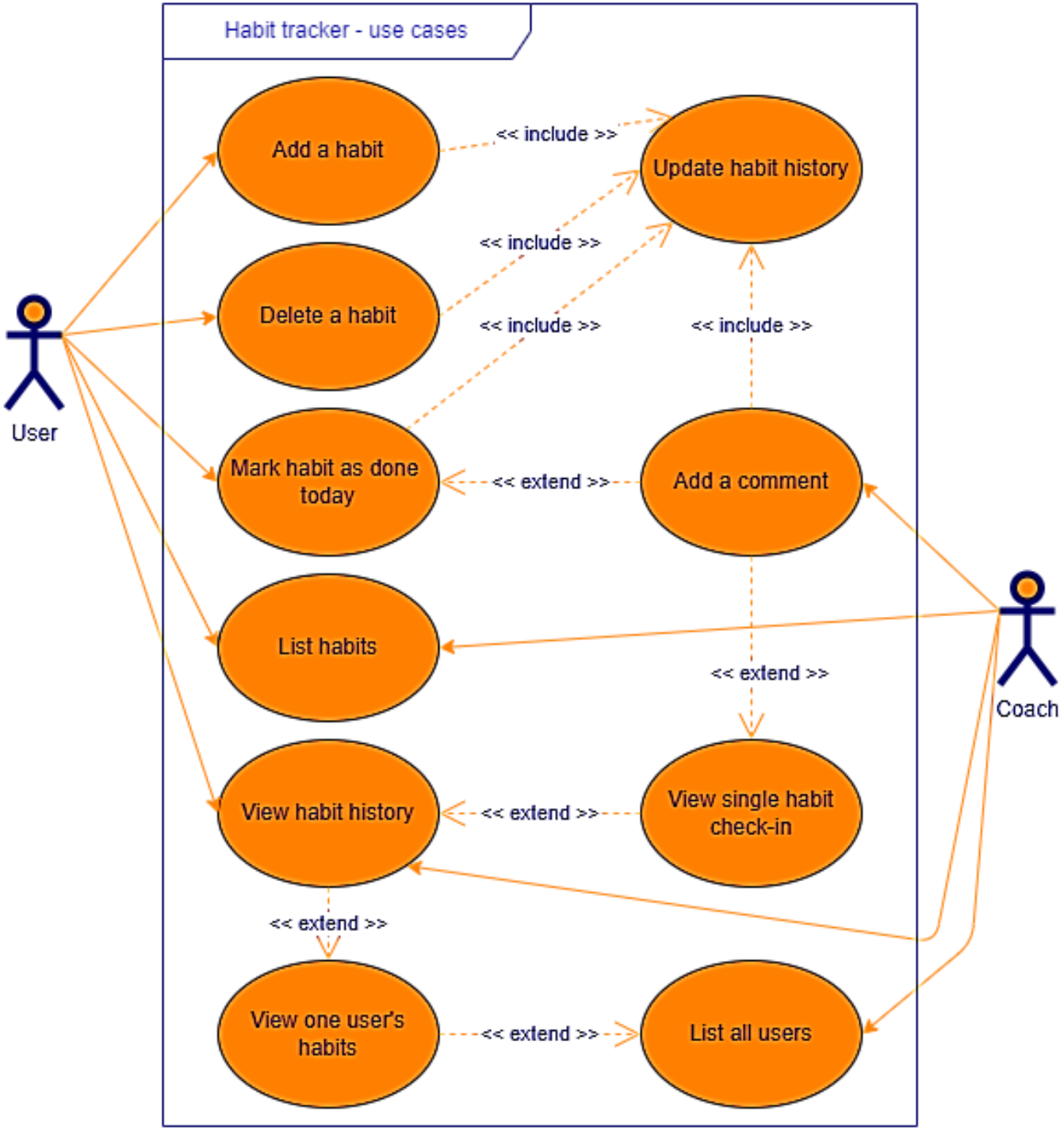


Create a use case diagram with draw.io

- In use case diagrams there are external actors (which may be users or processes that interact with your system). These are represented by stick figures. In the practical example shown – a habit tracking app – there are two external actors, a user and a coach.

- The actions that the actors take and their goals are represented by ovals with a solid arrow pointing to them. Not all actions can be directly done by an actor – some are triggered by other actions (represented by dotted lines, with the arrow indicating which action triggers the other).




Habit tracker - use cases
Add a habit
Update habit history
Delete a habit
Mark habit as done today
Add a comment
List habits
View habit history
View single habit check-in
View one user's habits
List all users
<< include >>
<< include >>
<< include >>
<< include >>
<< extend >>
<< extend >>
<< extend >>
<< extend >>
<< extend >>
User
Coach

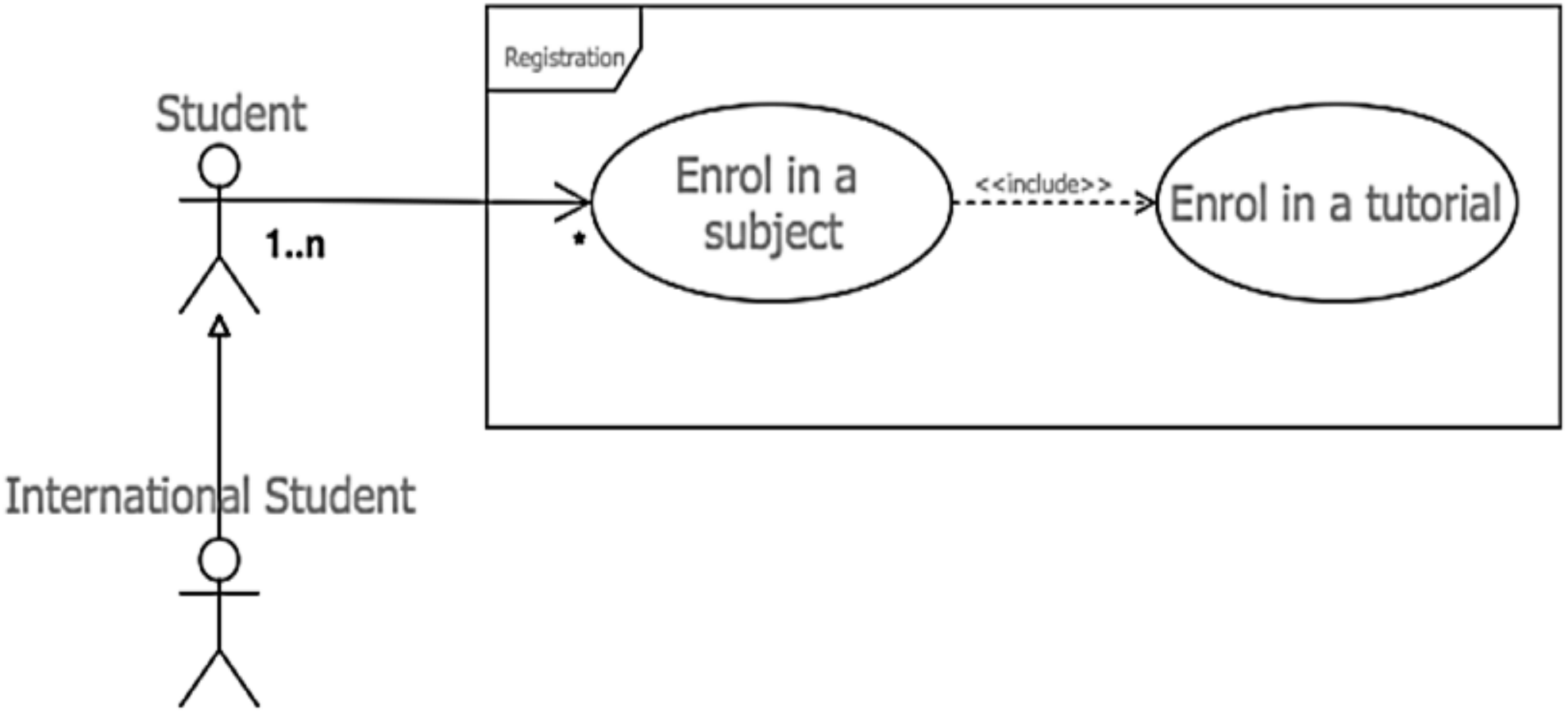
You can show specific relationships between the actions (or classes and methods) with include and extend.

- When one action is dependent on another action, use an include relationship. For example when you Add a comment in the habit tracker app, you must Update the habit history. But you wouldn't update the habit history without one of the four actions: Add a habit, Delete a habit, Mark a habit as done or Add a comment. So, the Update the habit history action is dependent on the other actions.

- When one action is an extension of another action (or a more specific version of that action), use extend to show that relationship. In my example, you can Mark a habit as done, or Mark a habit as done AND add a comment in the same action.

You can also show inheritance on both actions and actors with the standard UML notation of a connector with an open arrowhead.

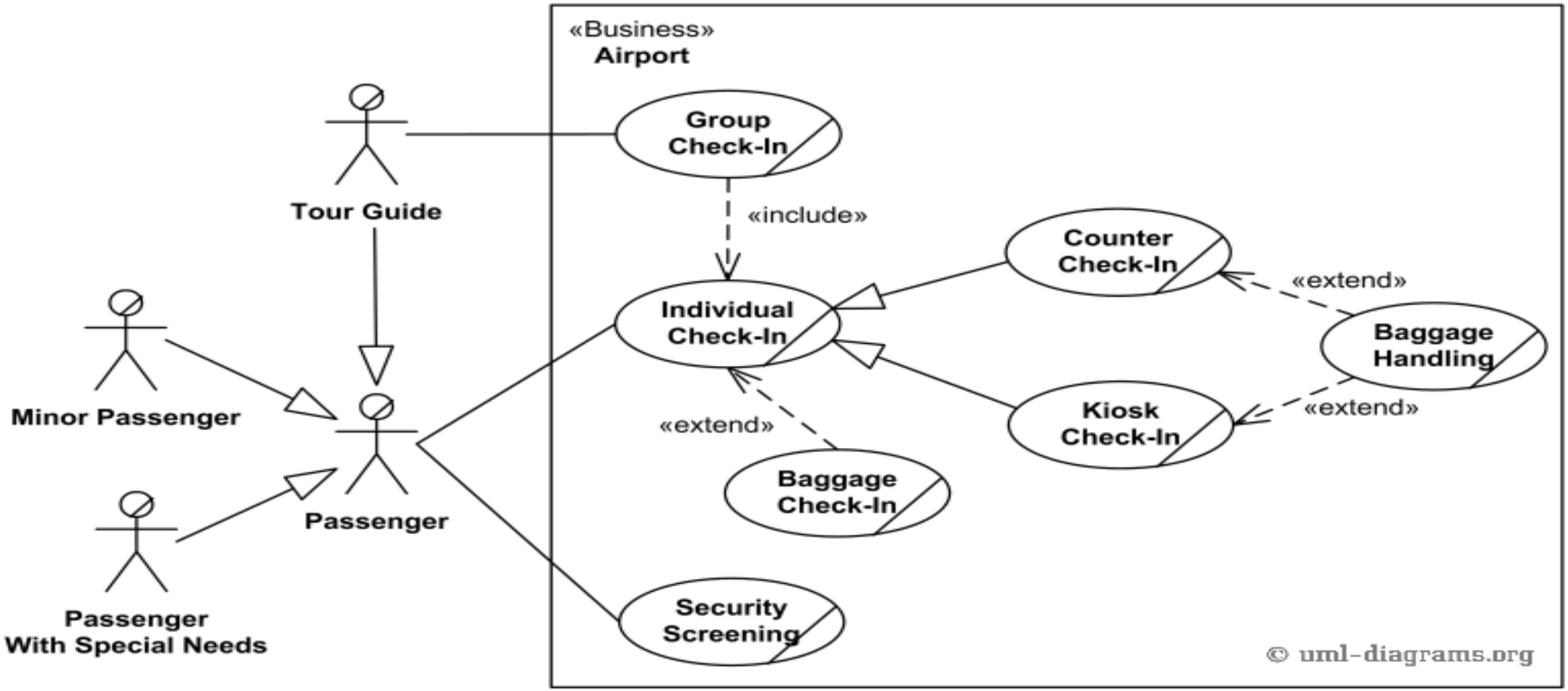
It's optional, but sometimes useful to add the standard UML notation for multiplicity: 1, 1..n, or *. If the relationship between the diagram elements is one-to-one, then simply leave it blank.



A simple use case diagram showing inheritance (actors) and multiplicity.

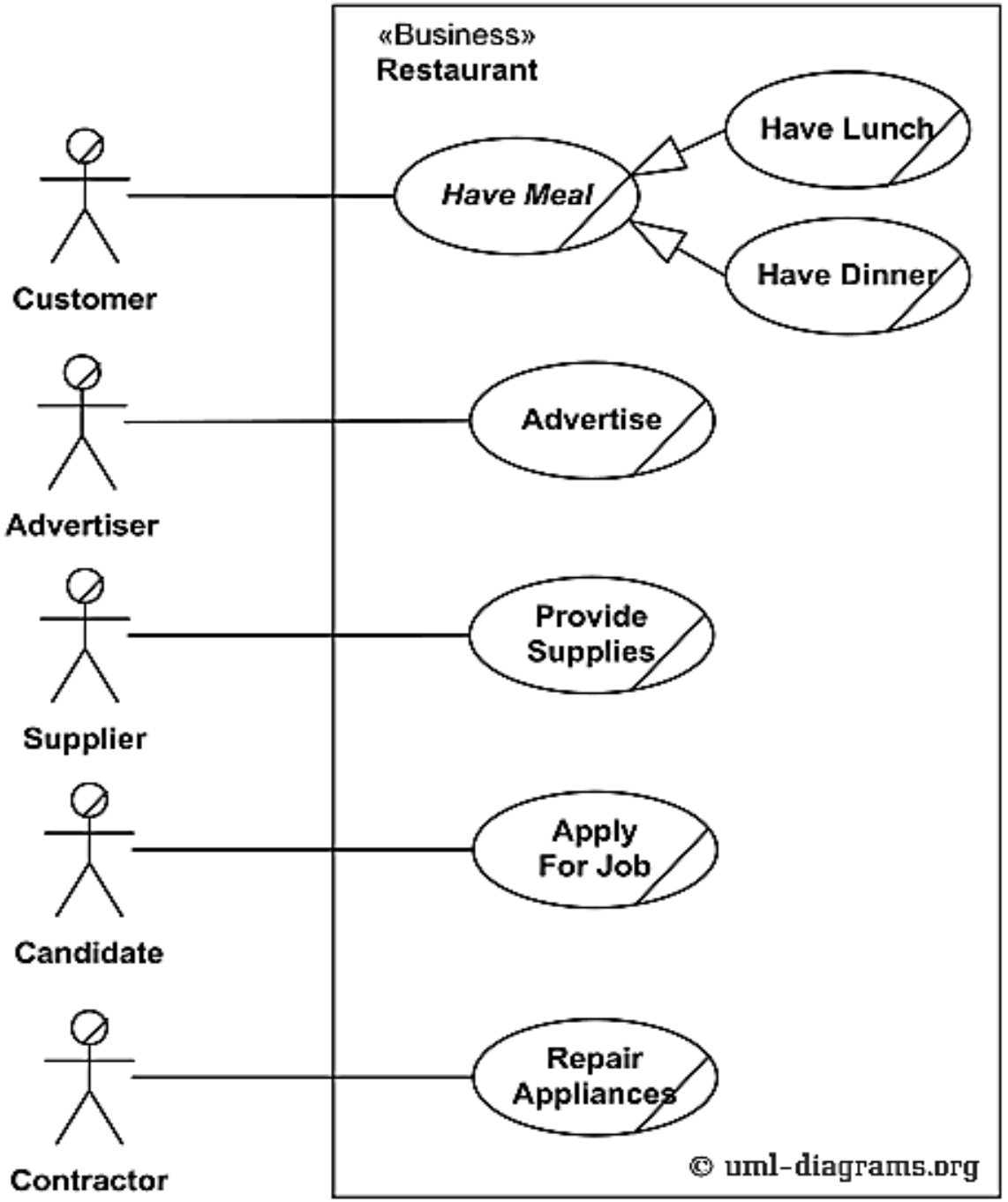
Airport check-in and security screening business model

- Purpose: An example of a business use case diagram for airport check-in and security screening.
- Summary: Business use cases are Individual Check-In, Group Check-In (for groups of tourists), Security Screening, etc. - representing business functions or processes taking place in an airport and serving needs of passengers.



Restaurant business model

- Purpose: Two alternative examples of business use case diagram for a Restaurant - external and internal business views of a restaurant.
- Summary: Several business actors having some needs and goals as related to the restaurant and business use cases expressing expectations of the actors from the business.

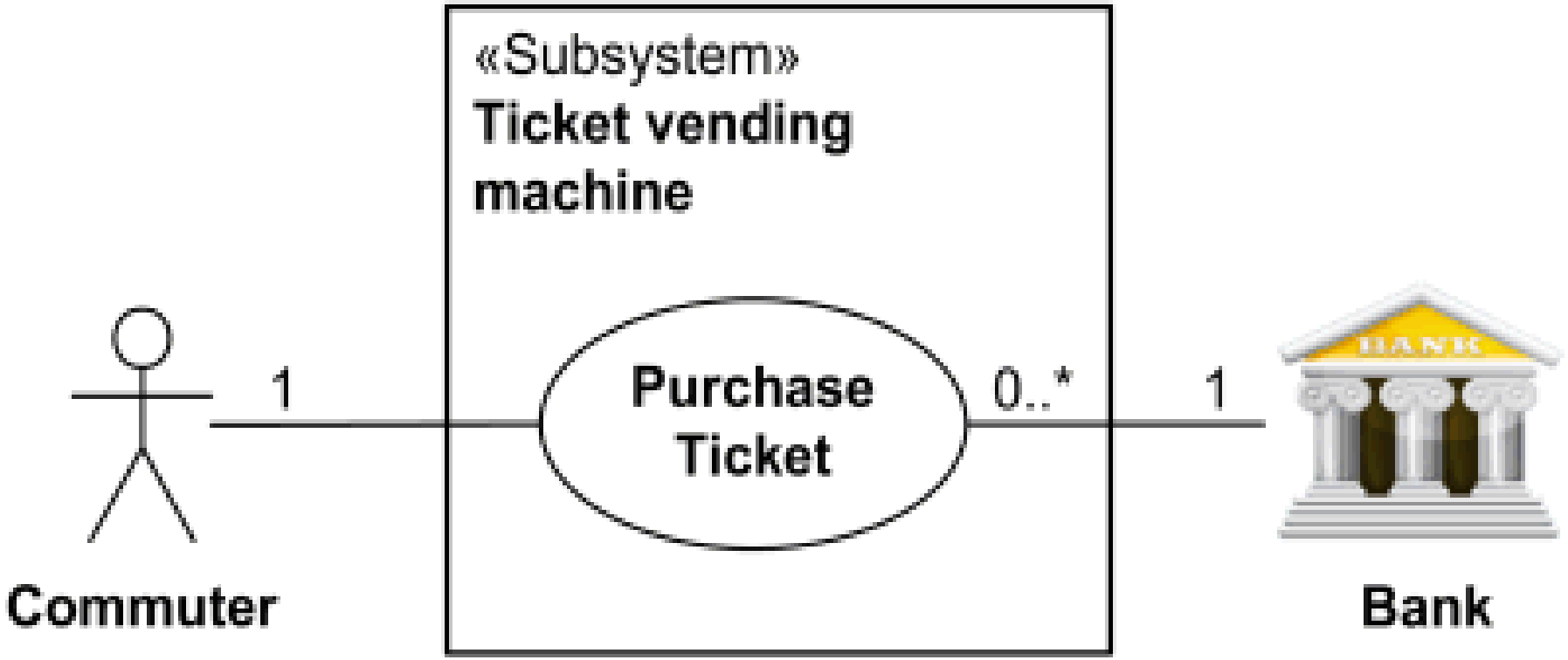


Examples of business use case diagrams

Examples of system use case diagrams

Ticket vending machine

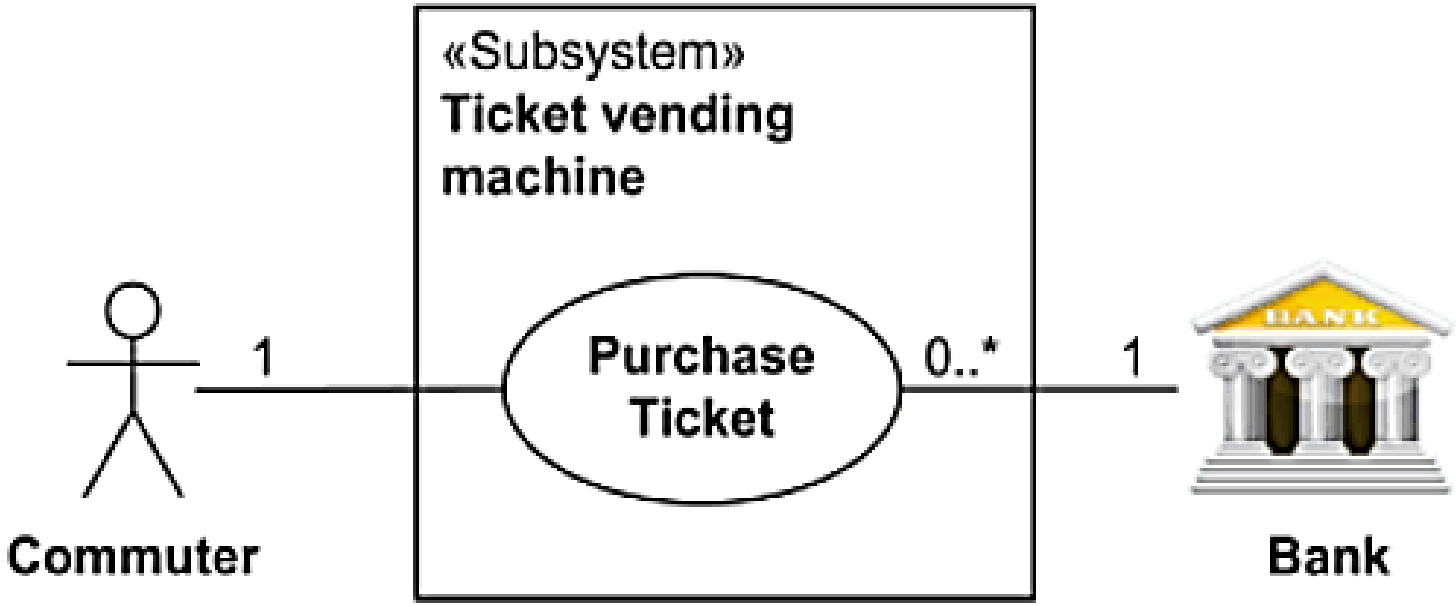
- Purpose: Show that ticket vending machine allows commuters to buy tickets.
- Summary: The ultimate goal of a Commuter in relation to our ticket vending machine is to buy a ticket. We have a single Purchase Ticket use case, as this vending machine is not providing any other services. Ticket vending machine is a subject of the example use case diagram. Commuter and Bank are our actors, both participating in the Purchase Ticket use case.



Examples of system use case diagrams

Ticket vending machine

- Purpose: Show that ticket vending machine allows commuters to buy tickets.
- Summary: The ultimate goal of a Commuter in relation to our ticket vending machine is to buy a ticket. We have a single Purchase Ticket use case, as this vending machine is not providing any other services. Ticket vending machine is a subject of the example use case diagram. Commuter and Bank are our actors, both participating in the Purchase Ticket use case.



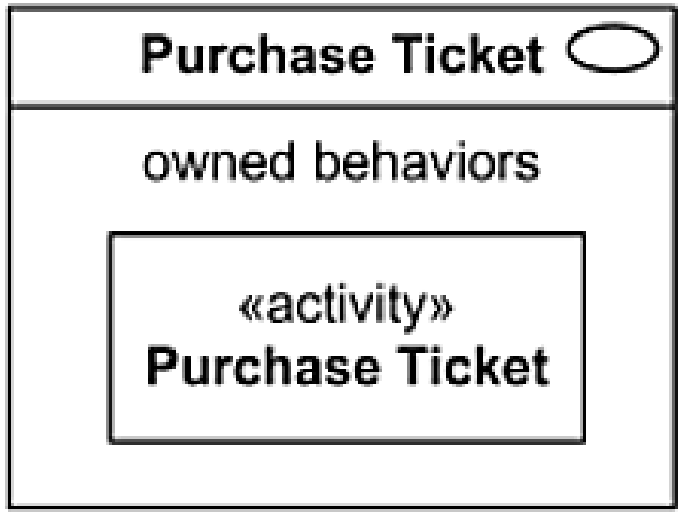
Ticket vending machine provides Purchase Ticket use case for the Commuter and Bank actors.

Examples of system use case diagrams

The ultimate goal of the Commuter in relation to our ticket vending machine is to buy a ticket.

So we have Purchase Ticket use case. Purchasing ticket might involve a bank, if payment is to be made using a debit or credit card. So we are also adding another actor - Bank. Both actors participating in the use case are connected to the use case by association.

Use case behaviors may be described in a natural language text (opaque behavior), which is current common practice, or by using UML behavior diagrams. UML tools should allow binding behaviors to the described use cases. Example of such binding of the Purchase Ticket use case to the behavior represented by activity is shown below using UML 2.5 notation.



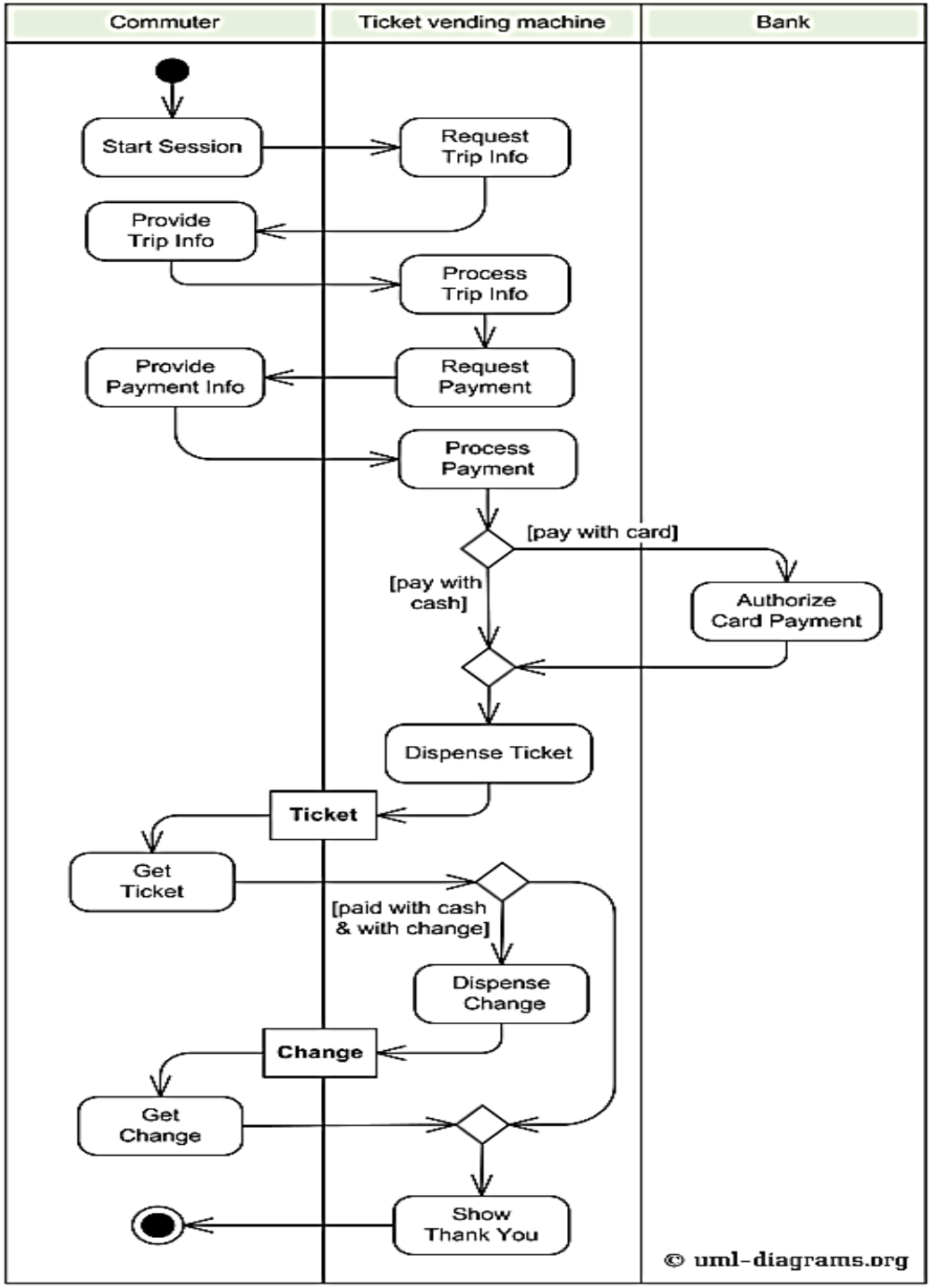
Purchase Ticket use case owns behavior represented by Purchase Ticket activity.

Examples of system use case diagrams

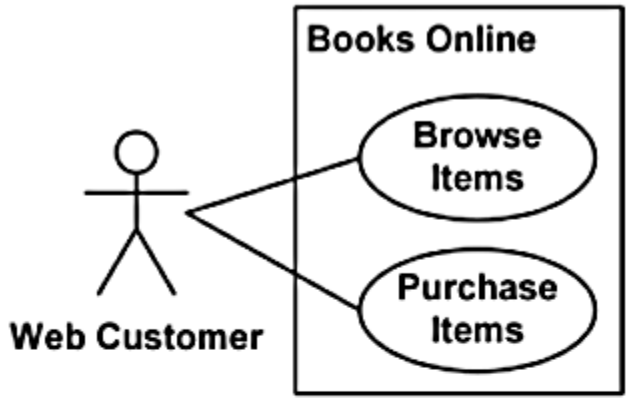
This is an example of UML activity diagram describing behavior of the Purchase Ticket use case.

Activity is started by Commuter actor who needs to buy a ticket. Ticket vending machine will request trip information from Commuter. This information will include number and type of tickets, e.g. whether it is a monthly pass, one way or round ticket, route number, destination or zone number, etc.

Based on the provided trip info ticket vending machine will calculate payment due and request payment options. Those options include payment by cash, or by credit or debit card. If payment by card was selected by Commuter, another actor, Bank will participate in the activity by authorizing the payment.



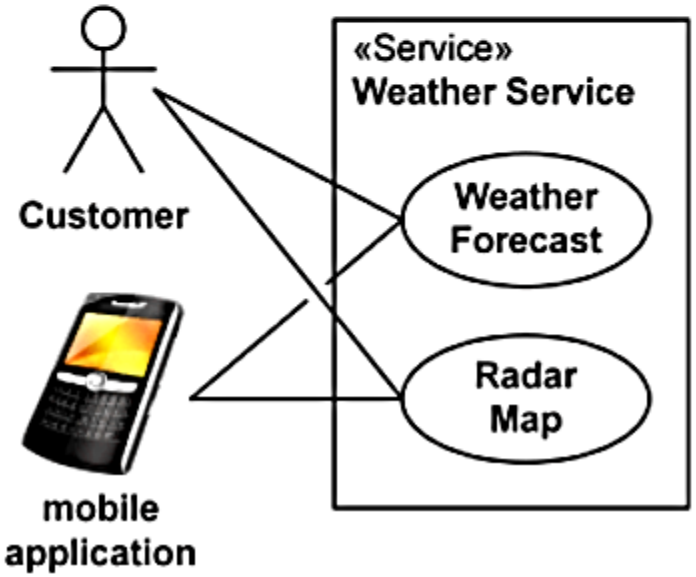
Subject



Books Online (subject) with applicable use cases and Web Customer actor.

The **subject** (of use cases) is the **system under design** or consideration to which a set of **use cases apply**. The subject could be a physical system, software program, or smaller element that may have **behavior**, e.g. **subsystem**, **component**, or even **class**.

Subject is presented by a rectangle with subject name in upper corner with **applicable use cases** inside the rectangle and **actors** - outside of the system boundaries.



Standard **UML stereotypes** and **keywords** for the subject are:

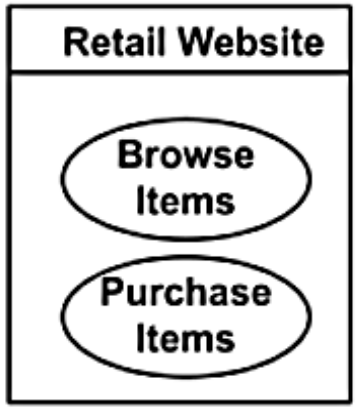
- «Subsystem»
- «Process»
- «Service»
- «Component»

Applicability of Use Cases

Use cases Browse Items and Buy Items are applicable to Retail Website subject.

Use cases visually located inside the system boundaries are the use cases **applicable** to the **subject** (but not necessarily **owned** by the subject).

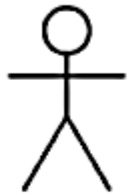
Ownership of Use Cases



*Retail Website **subject owns** use cases.*

The **nesting (ownership)** of a use case by a classifier is represented using the standard notation for nested classifiers.

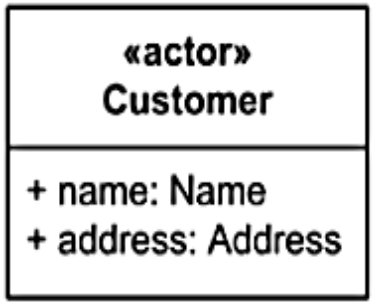
Actor



Student

Student ***actor.***

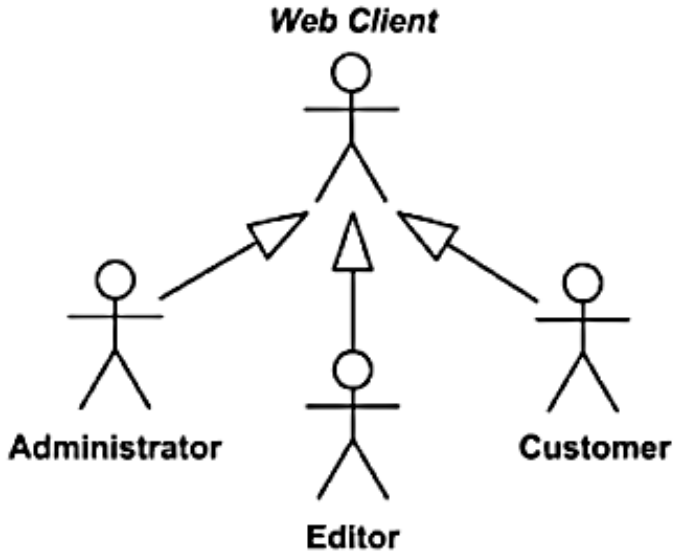
Standard UML icon for actor is "stick man" icon with the name of the actor above or below the icon. Actor names should follow the capitalization and punctuation guidelines for **classes**. The names of abstract actors should be shown in italics. All actors must have names.



Customer ***actor*** *as Class.*

An actor may also be shown as a **class** rectangle with the keyword **«actor»**, having usual notation for class compartments, if needed.

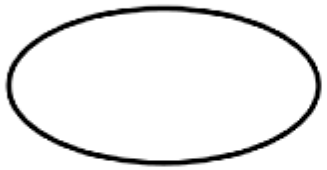
Generalization between actors

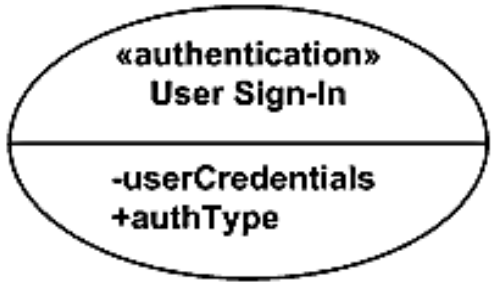


Web Client actor is abstract superclass for Administrator, Editor and Customer.

Generalization between actors is rendered as a solid directed line with a large arrowhead (the same as for generalization between classes).

Use Case

 *User Registration* ***Use Case***.	Every use case must have a name. Use case is shown as an ellipse containing the name of the use case.
 **Transfer Funds** *Transfer Funds* ***Use Case***.	A use case could be shown as an ellipse with the name of the use case placed below the ellipse.
 ***Business use case*** *Individual Check-In*.	**Business use case** was introduced in **Rational Unified Process** to support Business Modeling - to represent business function, process, or activity performed in the modeled **business**. **Business use case** is represented in RUP with use case oval icon and a line crossing it down right.

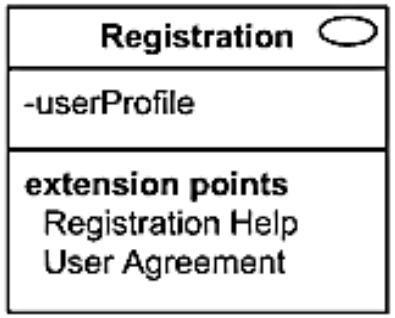


***Use Case** User Sign-In stereotyped as «authentication»*

An optional stereotype keyword may be placed above the name and a list of properties - operations and attributes - included below the name in a compartment.

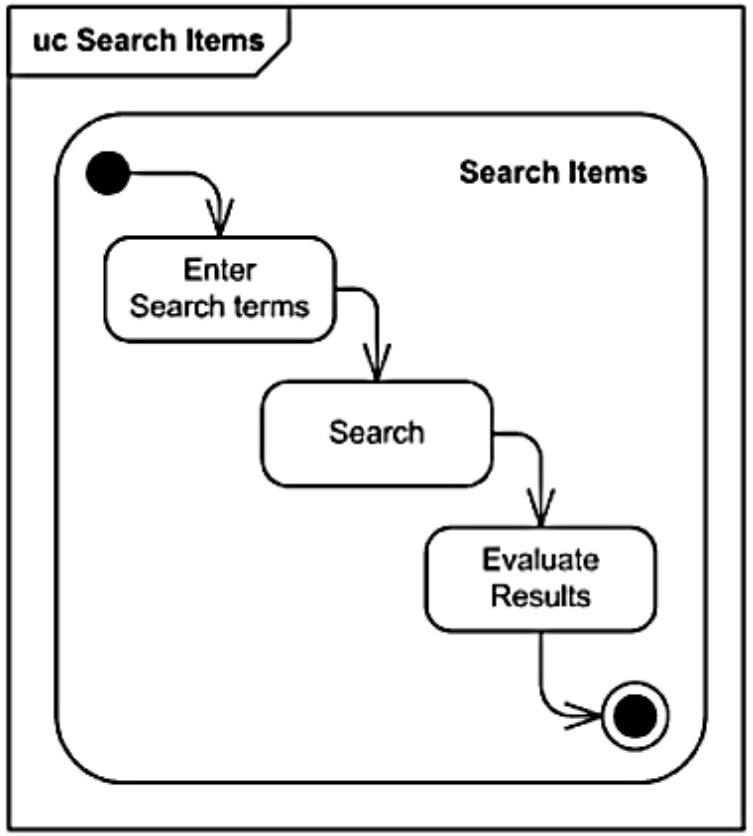
*Registration **Use Case** with **extension points** Registration Help and User Agreement.*

Use case with **extension points** may be listed in a compartment of the use case with the heading **extension points**.



*Registration **Use Case** shown using the standard rectangle notation for **classifiers**.*

A use case can also be shown using the standard rectangle notation for classifiers with an **ellipse icon** in the upper righthand corner of the rectangle and with optional separate list compartments for its features.

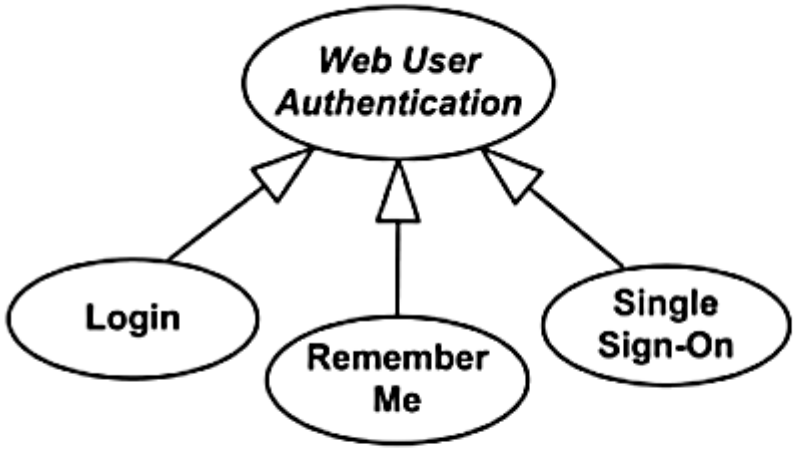


Use case** Search Items rendered as frame with associated Search Items **activity diagram

Use case could be rendered in the frame labeled as **use case** or in abbreviated form as **uc**.

The content area of the frame could contain different kinds of UML diagrams. For example, use case could be described with activity diagram or state machine.

Generalization between use cases

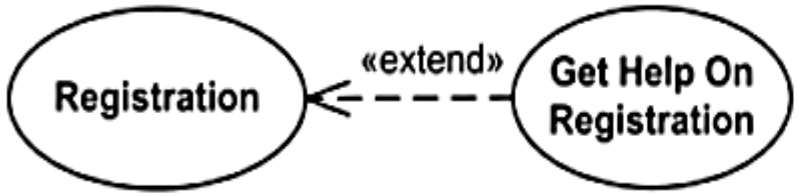


*Web User Authentication use case is **abstract use case** specialized by Login, Remember Me and Single Sign-On use cases.*

Generalization between use cases is similar to generalization between classes – child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

It is rendered as a solid directed line with a large arrowhead, the same as for **generalization** between classifiers.

Extend

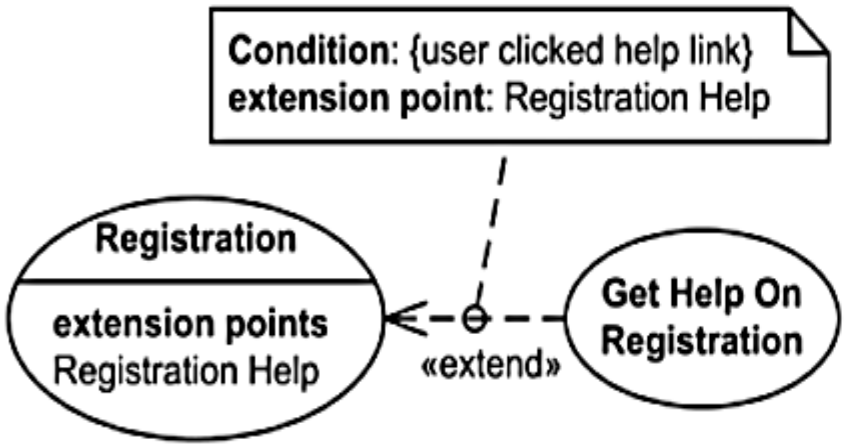


***Registration** use case is complete and meaningful on its own. It could be extended with optional **Get Help On Registration** use case.*

Extend is a **directed relationship** that specifies how and when the behavior defined in usually supplementary (optional) **extending use case** can be inserted into the behavior defined in the **extended use case**.

Extended use case is meaningful on its own, it is **independent** of the extending use case. **Extending** use case typically defines **optional** behavior that is not necessarily meaningful by itself.

Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the **extending use case** to the **extended (base) use case**. The arrow is labeled with the keyword **«extend»**.



Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help.

The **condition** of the extend relationship as well as the references to the **extension points** are optionally shown in a **comment** note attached to the corresponding extend relationship.

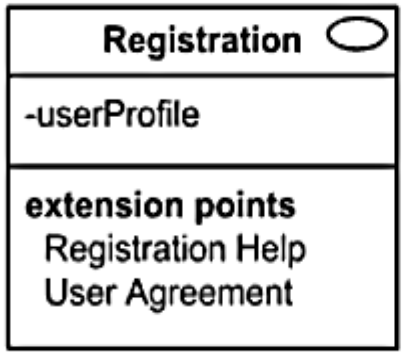
Extension Point

Registration ***Use Case*** *with* ***extension points*** *Registration Help and User Agreement.*

An **extension point** is a **feature** of a **use case** which identifies (references) a point in the behavior of the use case where that behavior can be extended by some other (extending) use case, as specified by **extend** relationship.

Extension points may be shown in a compartment of the use case oval symbol under the heading **extension points**. Each extension point must have a **name**, unique within a use case. Extension points are shown as a text string according to the syntax:

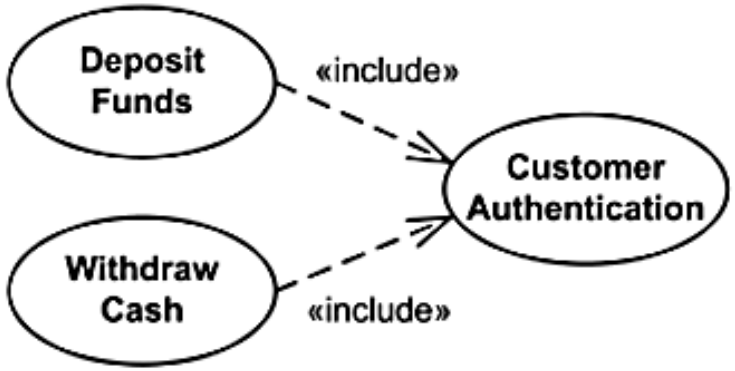
extension point ::= ***name*** [: ***explanation***]



Extension points of the Registration ***use case*** *shown using the rectangle notation.*

Extension points may be shown in a compartment of the use case rectangle with **ellipse icon** under the heading **extension points**.

Include



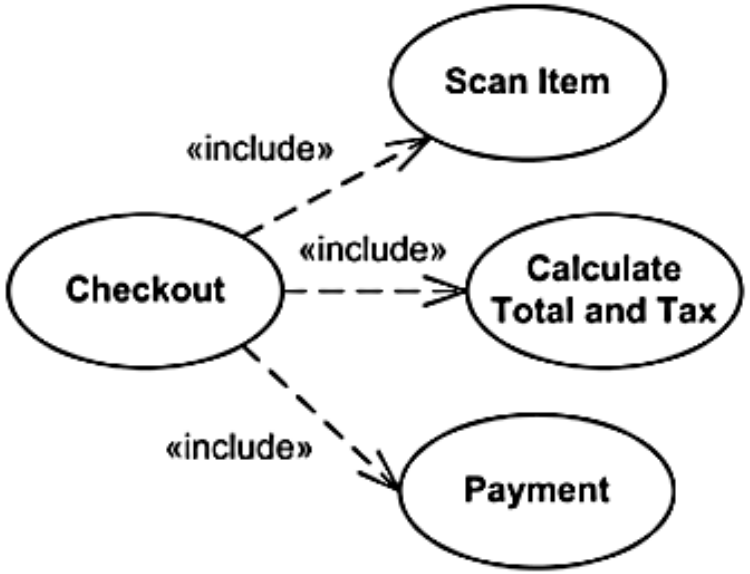
***Deposit Funds** and **Withdraw Cash** use cases include **Customer Authentication** use case.*

An **include** relationship is a **directed relationship** between two **use cases** when required, not optional behavior of the **included** use case is inserted into the behavior of the **including** (base) use case.

The **include** relationship is analogous to a subroutine call or macro and could be used:

- when there are **common parts** of the behavior of two or more use cases,
- to simplify large use case by splitting it into several use cases.

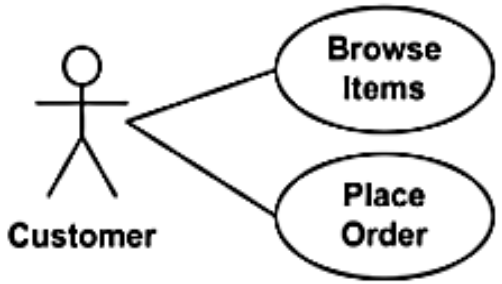
An **include** relationship between use cases is shown by a dashed arrow with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword **«include»**.



Checkout use case includes several use cases - Scan Item, Calculate Total and Tax, and Payment

Large and complex use case could be simplified by splitting it into several use cases each describing some logical unit of behavior. Note, that including use case becomes incomplete by itself and requires included use cases to be complete.

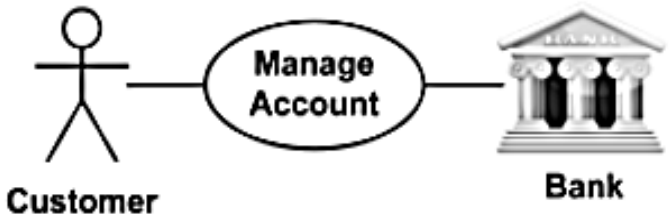
Association



***Actor** Customer associated with two **use cases**.*

An **association** between an actor and a use case indicates that the actor and the use case **communicate** with each other.

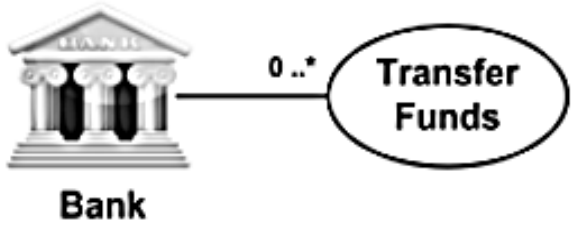
An actor could be associated to one or several use cases.



***Use case** Manage Account associated with Customer and Bank **actors**.*

Use case may have one or several associated actors.

It may not be obvious from use case diagram which actor **initiates** the use case, i.e. is "**primary actor**". (In non-standard UML, **primary actors** are those using system services, and **supporting actors** are actors providing services to the system.)

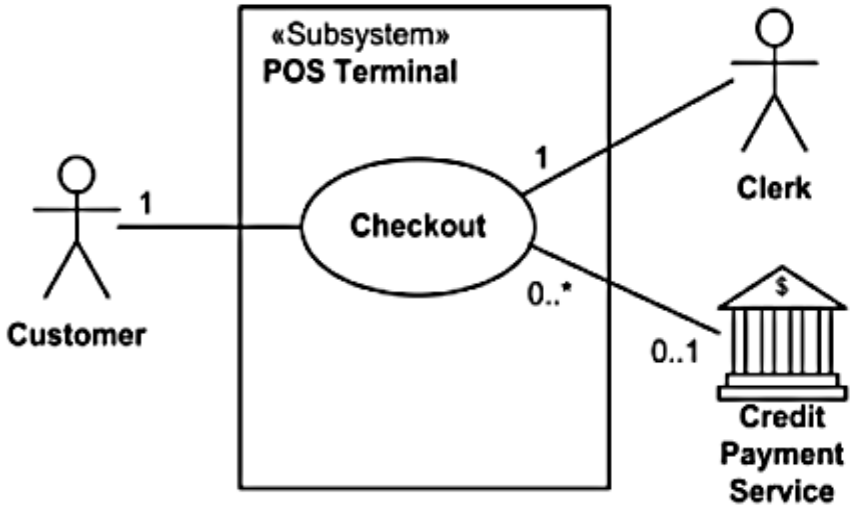


***Actor** Bank is involved in multiple **Use cases** Transfer Funds.*

When an actor has an **association** to a use case with a **multiplicity** that is greater than one at the **use case** end, it means that a given actor can be involved in **multiple use cases** of that type. The specific nature of this multiple involvement is **not defined** in the **UML 2.2**.

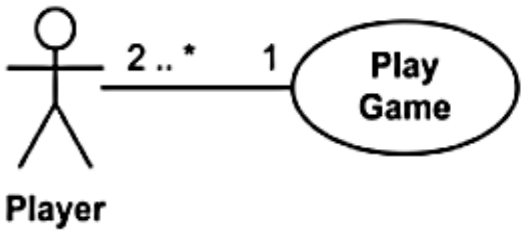
Use case multiplicity could mean that an actor initiates multiple use cases:

- in **parallel** (concurrently), or
- at different points in time, or
- **mutually exclusive** in time.



Checkout use case requires Customer actor, hence the 1 multiplicity of Customer. The use case may not need Credit Payment Service (for example, if payment is in cash), thus the 0..1 multiplicity.

Required actor may be explicitly denoted using multiplicity 1 or greater. UML 2.5 also allows actor to be optional. Multiplicity **0..1** of actor means that the actor is not required by any of the associated use cases.



Two or more Player actors are involved in Play Game use case. Each Player participates in one Play Game.

When a use case has an **association** to an actor with a multiplicity that is greater than one at the **actor** end, it means that **more than one actor** instance is involved in **initiating** the use case. The manner in which **multiple actors** participate in the use case is **not defined** in the **UML 2.x** up to the latest UML 2.5.

For instance, actor's multiplicity could mean that:

- a particular use case might require **simultaneous** (concurrent) action by two separate actors (e.g., in launching a nuclear missile), or
- it might require complementary and **successive** actions by the actors (e.g., one actor starting something and the other one stopping it).

System Design and Implementation

Introduction to Data Modeling Concepts

Data modeling is the process of creating a model of data. A data model is a simplified representation of the data that is used in a system. It is used to understand the data, to design new systems, and to communicate the design of systems to others.

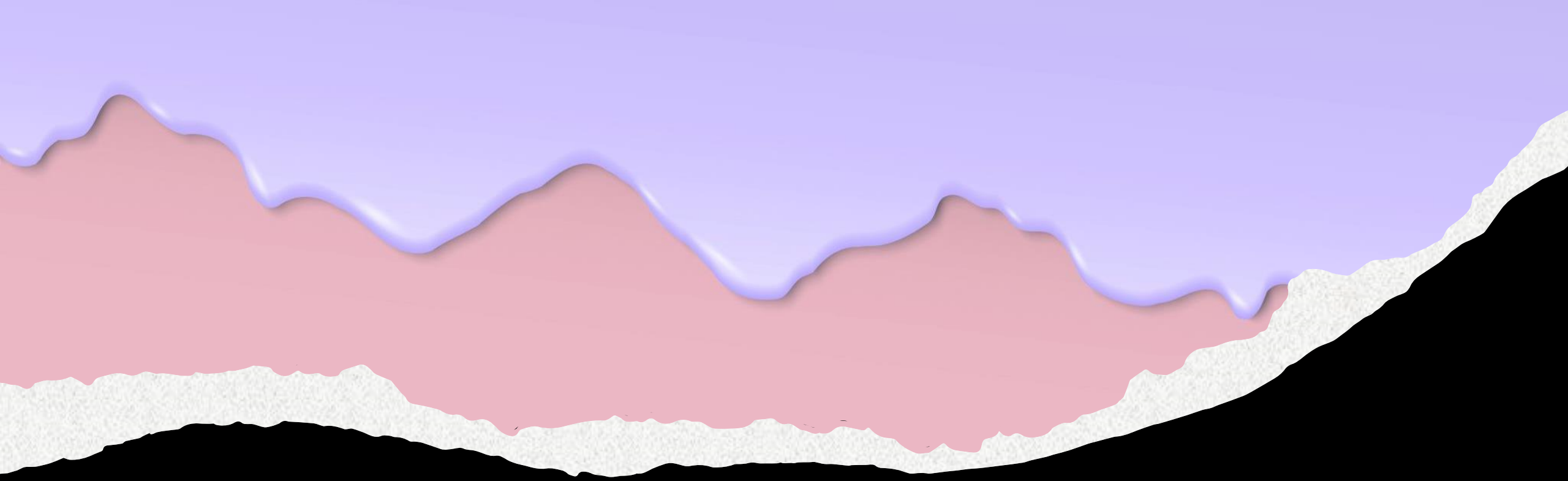
Data modeling

- Entities: An entity is a real-world object or concept that is represented in a data model. For example, a customer, an order, or a product might be entities in a data model.
- Attributes: An attribute is a characteristic of an entity. For example, a customer might have attributes such as name, address, and phone number.
- Relationships: A relationship is a connection between two entities. For example, a customer might have a relationship with an order, where the customer placed the order.
- Constraints: A constraint is a rule that restricts the values that can be stored in a data model. For example, a customer's name might be constrained to be unique.

Data model elements

- Conceptual data modeling: Conceptual data modeling is the highest level of data modeling. It is concerned with the overall structure of the data.
- Logical data modeling: Logical data modeling is the middle level of data modeling. It is concerned with the logical structure of the data.
- Physical data modeling: Physical data modeling is the lowest level of data modeling. It is concerned with the physical structure of the data.

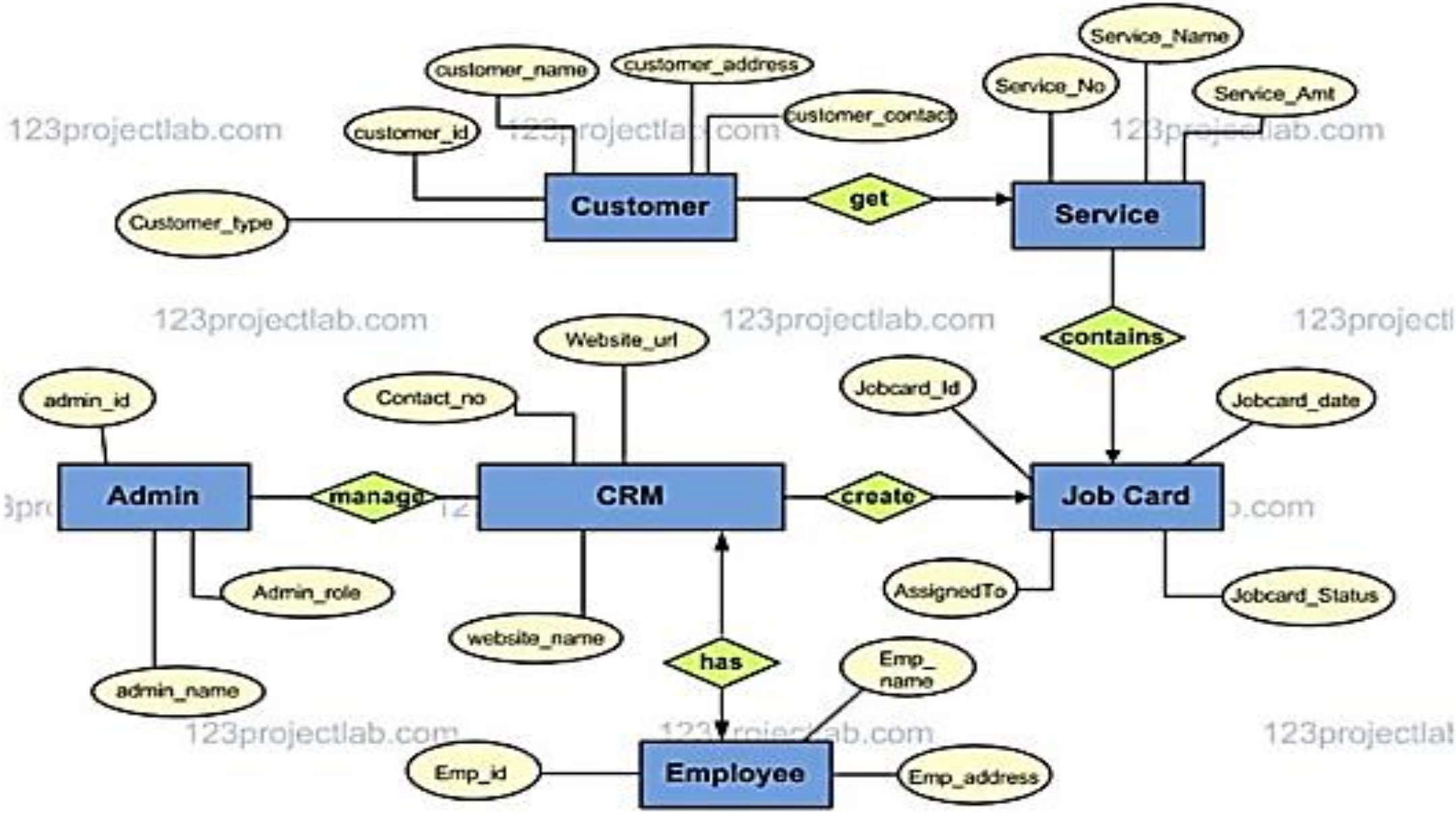
Data modeling levels



EntityRelationship (ER) Diagrams and Relational Database Design

ER Diagram for a Customer Relationship Management (CRM) System:

This ER diagram shows the entities and relationships between entities in a CRM system. The entities include customers, contacts, accounts, and opportunities. The relationships include one-to-many relationships between customers and contacts, one-to-many relationships between customers and accounts, and many-to-many relationships between accounts and opportunities.

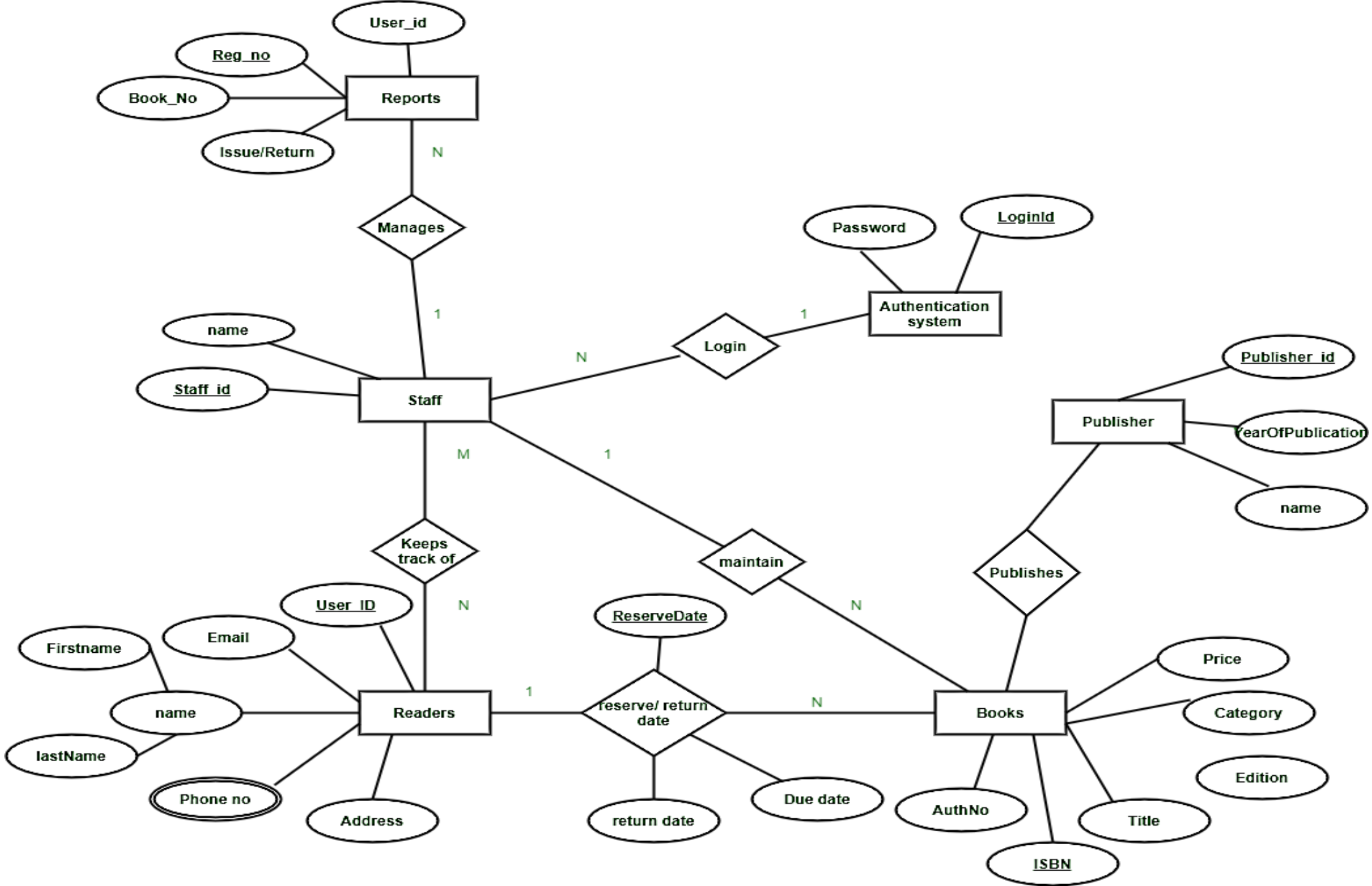


ER-Diagram for Customer Relationship Management System

123projectlab.com

Relational Database Design for a Library System:

This relational database design shows the tables and columns in a library system. The tables include books, authors, subjects, and borrowers. The columns in the books table include the book's title, author, subject, and publication date. The columns in the authors table include the author's name, birth date, and death date. The columns in the subjects table include the subject's name and description. The columns in the borrowers table include the borrower's name, address, and phone number.


User_id
Reg_no
Book_No
Issue/Return
Reports
N
Manages
1
Password
LoginId
Authentication system
1
Login
N
name
Staff_id
Staff
M
1
Keeps track of
maintain
N
N
Publisher_id
Publisher
YearOfPublication
name
Publishes
User_ID
Email
Firstname
name
lastName
Phone no
Address
Readers
1
ReserveDate
reserve/ return date
N
return date
Due date
Books
Price
Category
Edition
AuthNo
Title
ISBN

Normalization Techniques for Database Optimization

- First Normal Form (1NF): A table is in 1NF if all of its columns contain atomic values. This means that each column should contain a single value, and no column should contain repeating groups of values.

- Second Normal Form (2NF): A table is in 2NF if it is in 1NF and all of its non-key columns are dependent on the entire primary key. This means that no non-key column should be dependent on only part of the primary key.

- Third Normal Form (3NF): A table is in 3NF if it is in 2NF and no non-prime attribute is transitively dependent on the primary key. This means that no non-key column should be dependent on another non-key column.

Some normalization techniques for database optimization

- Boyce-Codd Normal Form (BCNF): A table is in BCNF if it is in 3NF and no non-prime attribute is functionally dependent on any other non-prime attribute. This is a stricter form of 3NF that eliminates transitive dependencies.

- Fourth Normal Form (4NF): A table is in 4NF if it is in BCNF and there are no multivalued dependencies. This means that no non-key column should contain multiple values for the same row.

- Fifth Normal Form (5NF): A table is in 5NF if it is in 4NF and there are no join dependencies. This means that no non-key column should be dependent on another non-key column through a join operation.

Some normalization techniques for database optimization

Database Normalization Activities

Designing System Architecture and Components

Designing system architecture involves creating a high-level plan that outlines how various components and modules of a system will interact and work together to achieve the desired functionality, scalability, reliability, and maintainability. It involves defining the structure of the system, the relationships between its components, and the data flow within the system.

Components in System Architecture:

Components are the building blocks of a system, each responsible for specific functionality or services. These components can be software modules, hardware devices, databases, APIs, services, or even third-party integrations. The interaction and cooperation between these components form the basis of the system's architecture.

Example: In a web application, the main components might include the front-end user interface, a backend server, a database, and various APIs or microservices.

Decoupling:

A well-designed architecture often promotes decoupling, which means reducing dependencies between components. This allows changes in one component without affecting others, enhancing the system's flexibility and maintainability.

Example: In a microservices architecture, each microservice is decoupled from others, allowing independent scaling and deployments.

Scalability:

A scalable system can handle increased workload and user demand efficiently. Designing for scalability involves distributing the load across multiple components or instances.

Example: In a web application, the main components might include the front-end user interface, a backend server, a database, and various APIs or microservices.

Reliability and Redundancy:

A reliable system minimizes downtime and data loss. Redundancy involves having backup components or systems to take over if a primary component fails.

Example: Employing a redundant database setup with master-slave replication to ensure data availability even if the primary database server goes down.

Security:

Security is a crucial aspect of system architecture, involving measures to protect data, prevent unauthorized access, and safeguard against potential vulnerabilities.

Example: Implementing encryption for sensitive data and employing firewalls to protect against unauthorized access.

Data Storage and Retrieval:

Designing the data storage and retrieval mechanism involves selecting appropriate databases, caching strategies, and data retrieval patterns.

Example: Using a NoSQL database for high write-throughput requirements or a relational database for complex data relationships.

Communication and Integration:

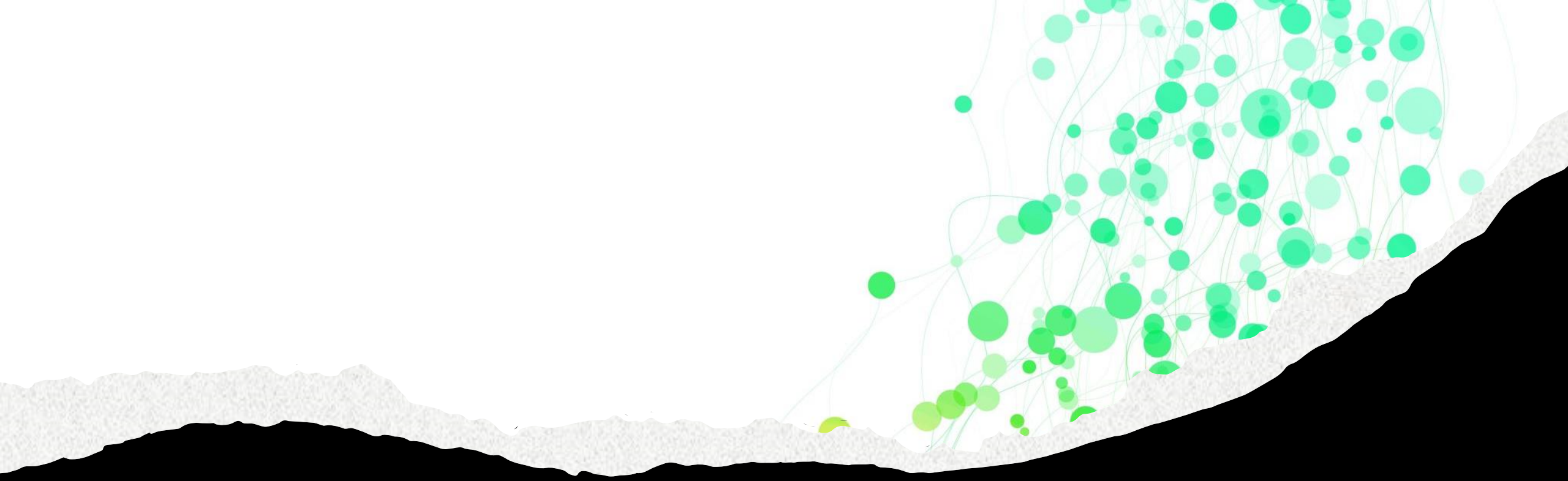
A system often involves communication between various components or services, which requires designing effective integration methods and protocols.

Example: Using RESTful APIs for communication between different services in a distributed system.

Monitoring and Logging:

An effective architecture includes provisions for monitoring the system's health and logging relevant data for debugging and analysis.

Example: Implementing logging mechanisms to record important events and using monitoring tools to track system performance.



Design Patterns and Best Practices

Design patterns and best practices are essential tools for creating maintainable, scalable, and efficient software solutions. Here are some general recommendations for using design patterns and best practices in your projects:

✓ **Follow SOLID Principles:**

SOLID is an acronym that stands for five design principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. Adhering to these principles helps create modular, flexible, and easy-to-maintain code.

✓ **Use Design Patterns Wisely:**

Familiarize yourself with various design patterns such as Factory, Singleton, Observer, Strategy, and others. Choose the appropriate pattern for your specific problem and avoid overusing patterns just for the sake of using them.

✓ **Encapsulate Wherever Possible:**

Encapsulation is fundamental to object-oriented programming. Hide implementation details within classes and provide a well-defined interface for interaction.

✓ **Write Readable Code:**

Write code that is easy for others to understand. Use meaningful variable and function names, proper indentation, and clear comments when necessary.

✓ **Keep Classes and Methods Small:**

Follow the Single Responsibility Principle (SRP) and keep classes and methods focused on a single task. This improves code readability and makes debugging and testing easier.

✓ **Avoid Code Duplication:**

DRY (Don't Repeat Yourself) principle encourages the elimination of duplicated code. Create reusable functions, classes, or modules to reduce redundancy.

✓ **Use Version Control:**

Utilize version control systems like Git to track changes, collaborate with others, and easily revert to previous versions when needed.

✓ **Automated Testing:**

Implement automated testing (unit tests, integration tests, etc.) to ensure code correctness and detect regressions early in the development process.

✓ **Optimize Performance Prudently:**

Optimize critical code sections when necessary, but prioritize readability and maintainability over micro-optimizations. Use profiling tools to identify performance bottlenecks accurately.

✓ **Security Considerations:**

Be mindful of security vulnerabilities, such as SQL injection, XSS, and CSRF. Use parameterized queries and follow security best practices to prevent attacks.

✓ **Error Handling:**

Implement proper error handling to handle exceptions and errors gracefully. Provide helpful error messages and avoid exposing sensitive information.

✓ **Documentation:**

Document your code, APIs, and system architecture to make it easier for other developers (or your future self) to understand and use your codebase effectively.

✓ **Review Code:**

Encourage code reviews within your team to get feedback on design choices, identify potential issues, and share knowledge.

✓ **Keep Abreast of New Technologies:**

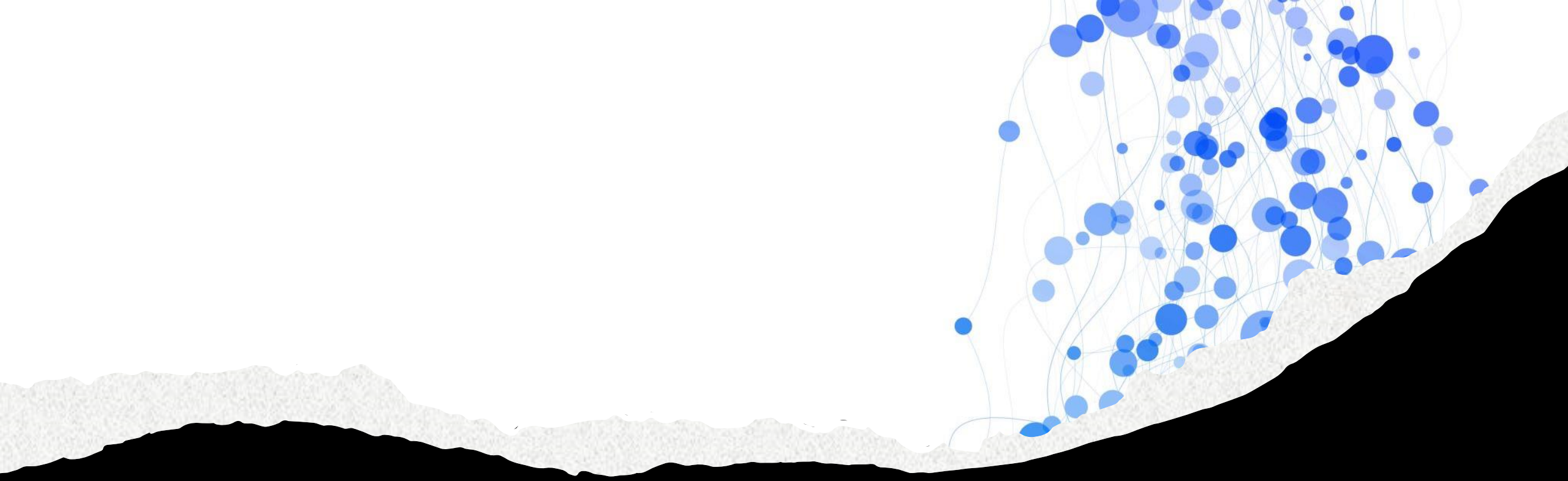
Stay updated with the latest trends, libraries, and tools in the tech industry to leverage new advancements in your projects.

✓ **Maintain a Consistent Coding Style:**

Adopt a consistent coding style across your project or team to enhance code readability and collaboration.

User Interface Design and Usability Considerations

Here are some general insights and considerations for UI design and usability:	**Know Your Users:**
	Keep it Simple and Intuitive:
	Consistency is Key:
	Clear and Concise Language:
	Prioritize Content:
	Responsive Design:
	Visual Hierarchy:
	Minimize User Effort:
	Feedback and Validation:
	Error Handling and Recovery:
	Test and Iterate:
	Accessibility:
	Performance Optimization:
	Visual Appeal:
	Contextual Help and Documentation:
	Learn from Competitors and Trends:



Strategies for System Implementation

System implementation is a critical phase where the designed system is put into action and made operational.

Phased Approach
Strong Project Management
Cross-Functional Collaboration
Training and Support
Data Migration and Integration
Pilot Testing
Change Management
Scalability and Performance
Security and Compliance
Backup and Disaster Recovery
User Acceptance Testing (UAT)
Continuous Monitoring and Evaluation
Go-Live Plan
Celebrate Success

Coding Practices and Development Tools

General Coding Practices

- Consistent Code Formatting:
- Meaningful Variable and Function Names:
- Commenting and Documentation:
- Modularity and Reusability:
- Avoid Magic Numbers and Strings:
- Error Handling:
- Version Control:
- Testing:
- Performance Optimization:
- Avoid Global Variables:

Development Tools

- **Integrated Development Environments (IDEs)**
- **Version Control Systems (VCS)**
- **Package Managers**
- **Build Tools**
- **Code Linters and Formatters**
- **Continuous Integration (CI) Servers**
- **Debugging Tools**
- **Performance Profilers**
- **Collaboration Tools**
- **Containerization Tools**

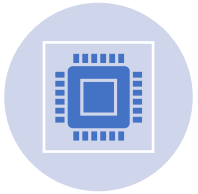
Software Testing Techniques and Methodologies

Types of Software Testing:

Unit Testing: Testing individual units or components in isolation to verify their correctness.

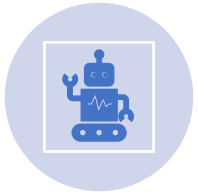
Integration Testing: Testing the interactions and data flow between integrated components.

Functional Testing: Verifying that the software functions correctly according to the specified requirements.



Performance Testing: Evaluating the software's responsiveness, scalability, and resource usage under various conditions.

Security Testing: Identifying vulnerabilities and potential security risks in the software.

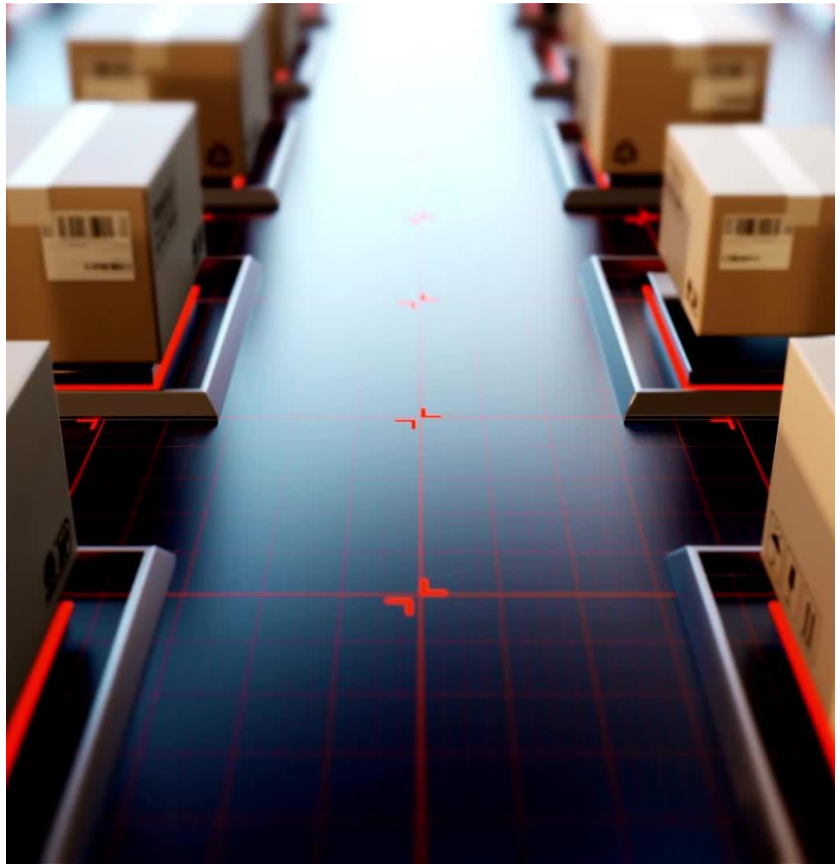


User Acceptance Testing (UAT): Testing the software with end-users to ensure it meets their needs and expectations.

Deployment Strategies and Considerations

Deployment Types

- Rolling Deployment: Gradually deploying new versions while maintaining some instances with the previous version until the update is complete.
- Blue-Green Deployment: Running two identical environments (blue and green), with one active for production while the other is for updates. After testing, the switch is made, making the updated version active.
- Canary Deployment: Releasing the new version to a small subset of users first to verify its stability and gradually increasing the user base.
- Feature Toggles: Enabling or disabling new features through configuration switches without deploying a new version.



Automated Deployment

Implement automated deployment pipelines to minimize manual intervention, reduce the risk of human errors, and speed up the release process.

Continuous Deployment vs. Continuous Delivery

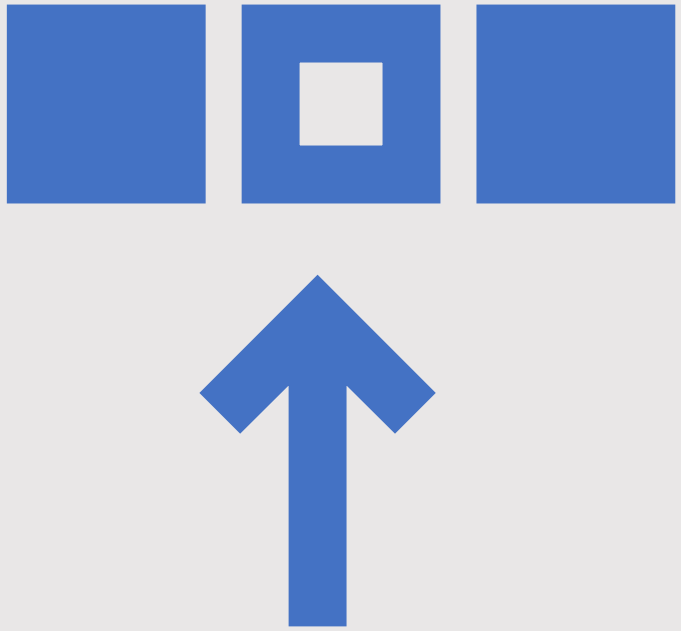
- Continuous deployment automatically releases every change that passes automated tests to the production environment.
- Continuous delivery ensures every change is deployable to production but allows for manual approval before release.

Rollback Plan

Always have a rollback plan in place to revert to the previous stable version in case the new deployment encounters critical issues.

Monitoring and Alerts

Implement monitoring and alerting mechanisms to promptly detect and respond to performance issues, errors, and other anomalies.

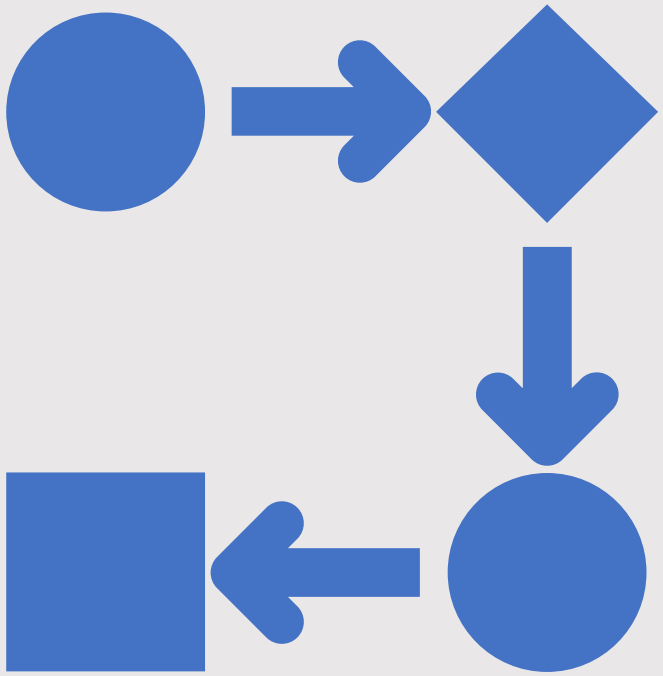


Versioning

Maintain version control for all deployed artifacts to facilitate rollbacks and traceability.

Security Considerations

Securely configure the production environment, use encryption where necessary, and follow security best practices to protect data and systems.



Dependency Management

Carefully manage dependencies to ensure consistent and predictable deployment environments.

Disaster Recovery and Redundancy

Implement redundancy and disaster recovery measures to ensure high availability and data integrity.

Downtime Mitigation

Minimize downtime during deployments by leveraging deployment strategies like rolling updates or blue-green deployments.

User Communication

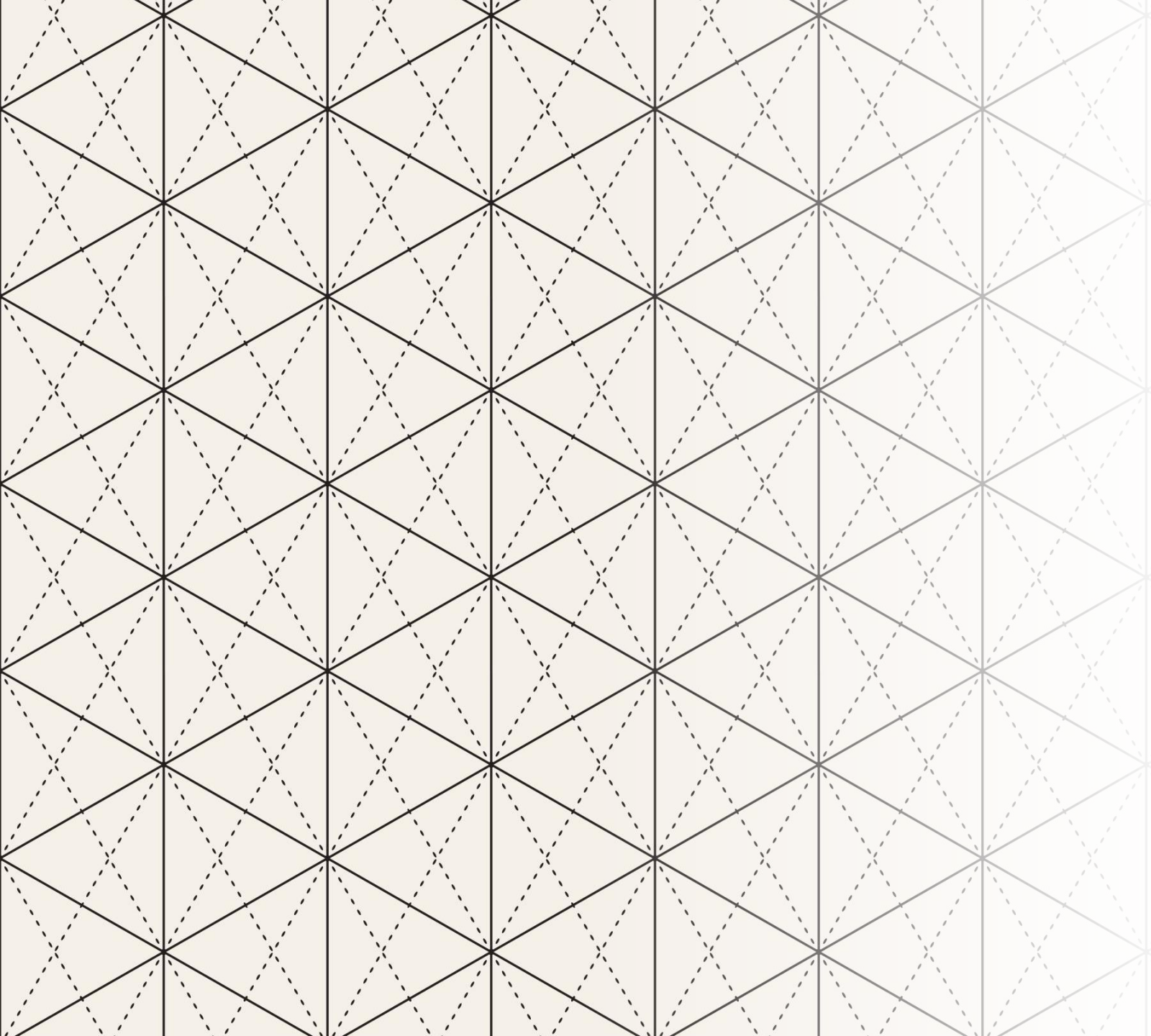
Communicate planned maintenance or updates to users to manage expectations and reduce surprises.

Compliance and Legal Considerations

Ensure compliance with legal and regulatory requirements when deploying software that involves sensitive data or operates in specific jurisdictions.

System Maintenance and Evaluation

Importance of system maintenance

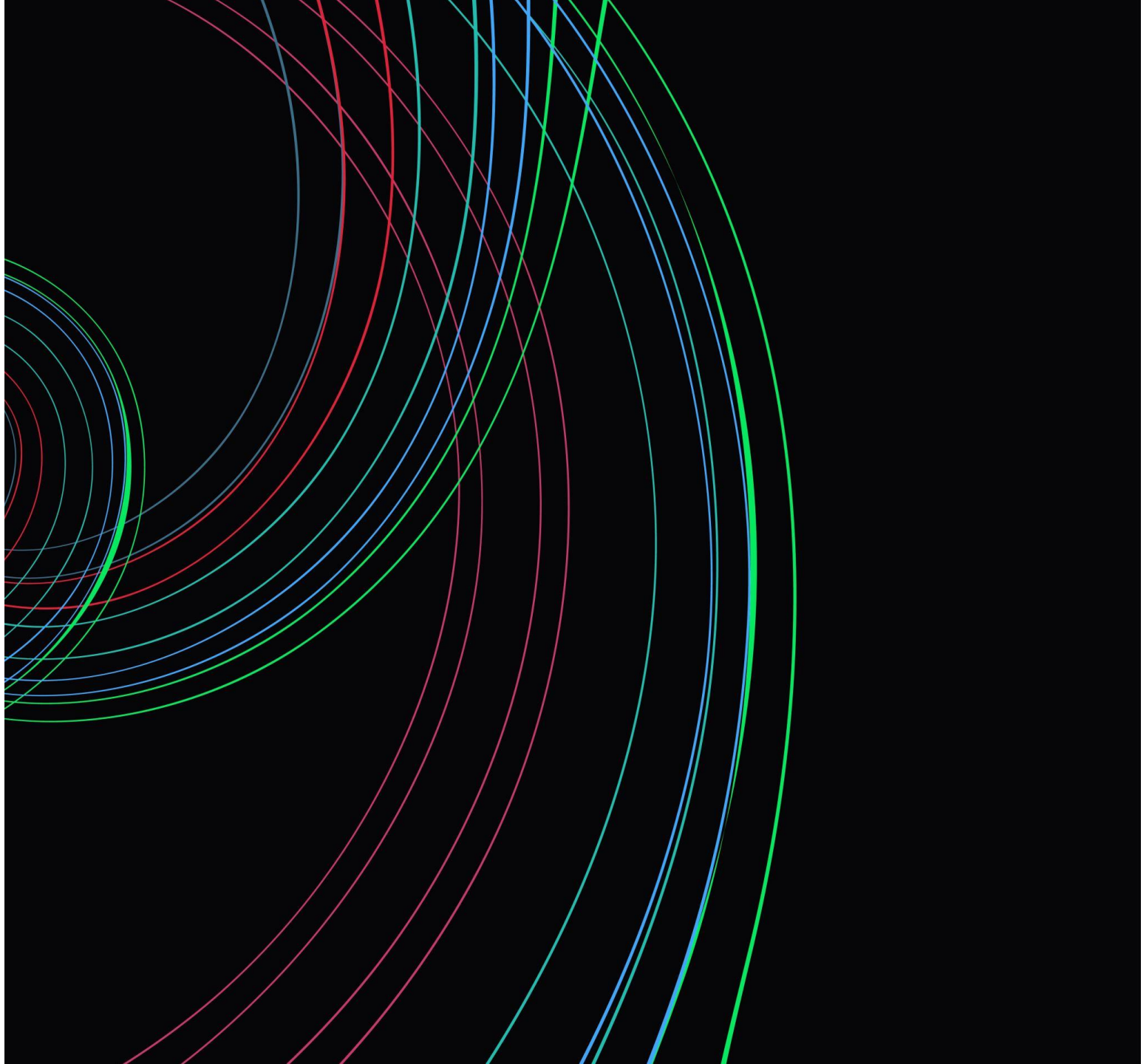


Handling system updates, bug fixes, and enhancements

User support and troubleshooting

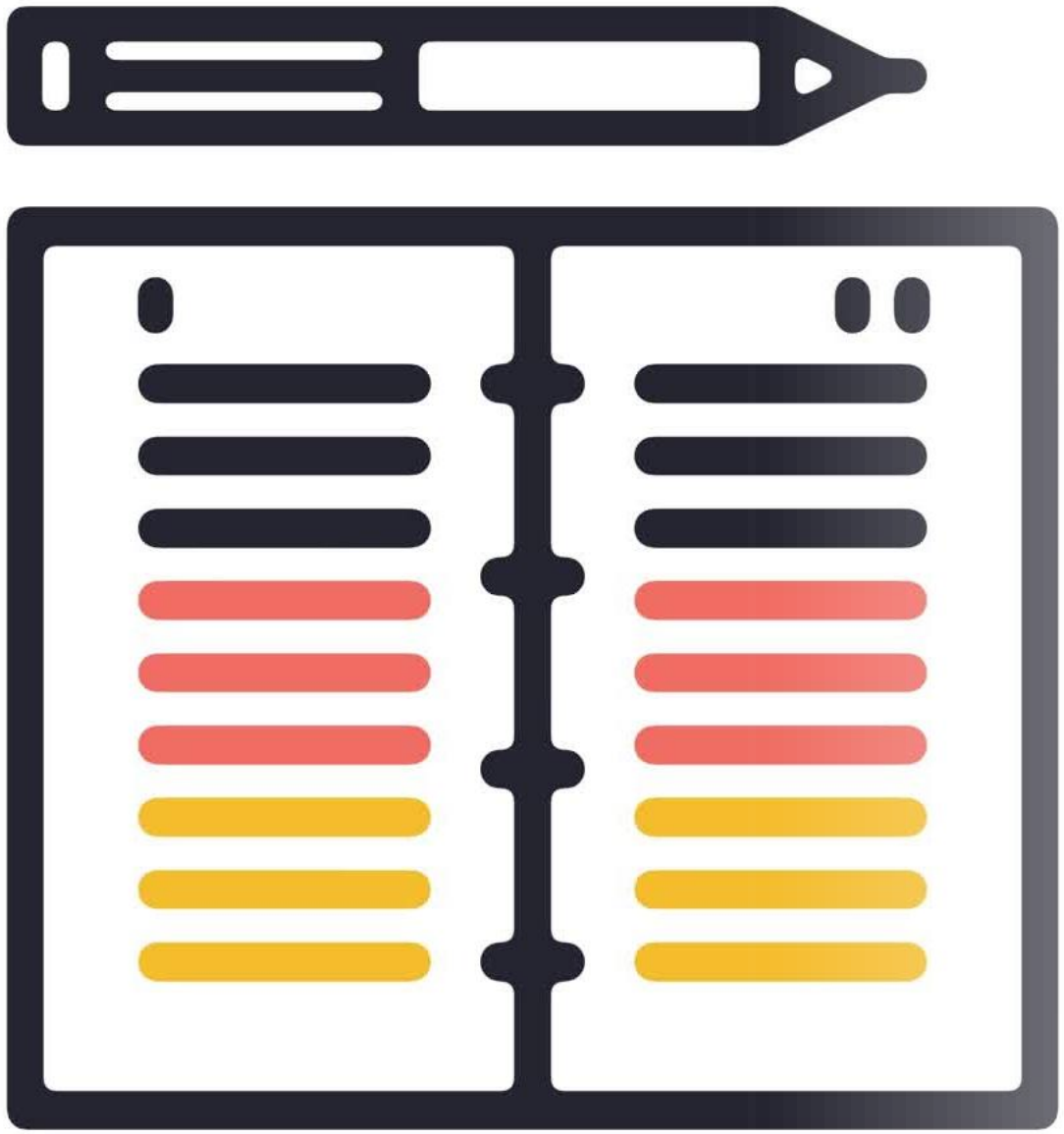
Evaluating system performance and effectiveness

Quality assurance techniques and metrics



User acceptance testing and feedback collection

Importance of system documentation



Creating user manuals, technical guides, and system documentation

Knowledge management practices for system maintenance

Thank You!