

# 軟體工程 (Software Engineering)

## 安全和隱私 (Security and Privacy) 可靠的程式設計 (Reliable Programming)

1101SE09

MBA, IM, NTPU (M6131) (Fall 2021)

Thu 11, 12, 13 (19:25-22:10) (209)



Min-Yuh Day

戴敏育

Associate Professor

副教授

Institute of Information Management, National Taipei University

國立臺北大學 資訊管理研究所

<https://web.ntpu.edu.tw/~myday>

2021/12/02



# 課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
1	2021/09/23	軟體工程概論 (Introduction to Software Engineering)
2	2021/09/30	軟體產品與專案管理：軟體產品管理，原型設計 (Software Products and Project Management: Software product management and prototyping)
3	2021/10/07	敏捷軟體工程：敏捷方法、Scrum、極限程式設計 (Agile Software Engineering: Agile methods, Scrum, and Extreme Programming)
4	2021/10/14	功能、場景和故事 (Features, Scenarios, and Stories)
5	2021/10/21	軟體工程個案研究 I (Case Study on Software Engineering I)
6	2021/10/28	軟體架構：架構設計、系統分解、分散式架構 (Software Architecture: Architectural design, System decomposition, and Distribution architecture)

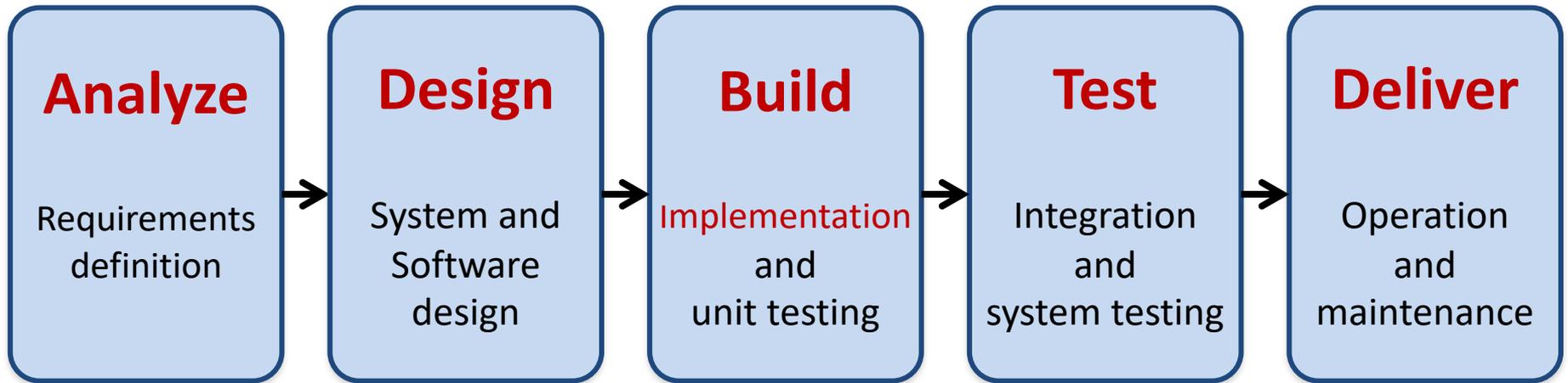
# 課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
7	2021/11/04	基於雲的軟體：虛擬化和容器、軟體即服務 (Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service)
8	2021/11/11	期中報告 (Midterm Project Report)
9	2021/11/18	雲端運算與雲軟體架構 (Cloud Computing and Cloud Software Architecture)
10	2021/11/25	微服務架構：RESTful服務、服務部署 (Microservices Architecture, RESTful services, Service deployment)
11	2021/12/02	安全和隱私 (Security and Privacy); 可靠的程式設計 (Reliable Programming)
12	2021/12/09	軟體工程個案研究 II (Case Study on Software Engineering II)

# 課程大綱 (Syllabus)

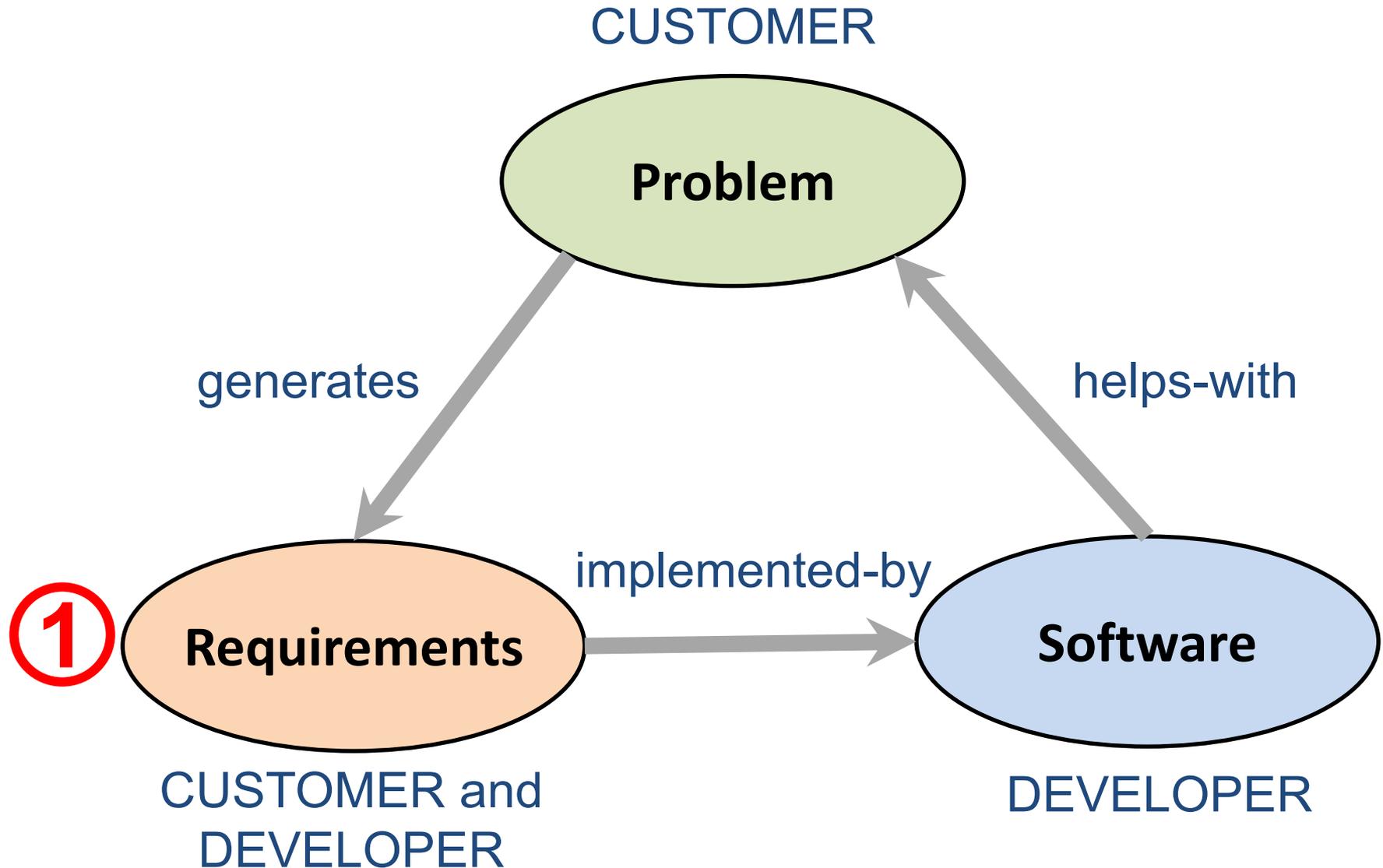
週次 (Week)	日期 (Date)	內容 (Subject/Topics)
13	2021/12/16	軟體工程產業實務 (Industry Practices of Software Engineering)
14	2021/12/23	測試：功能測試、測試自動化、 測試驅動的開發、程式碼審查 (Testing: Functional testing, Test automation, Test-driven development, and Code reviews); DevOps和程式碼管理：程式碼管理和DevOps自動化 (DevOps and Code Management: Code management and DevOps automation)
15	2021/12/30	期末報告 I (Final Project Report I)
16	2022/01/06	期末報告 II (Final Project Report II)
17	2022/01/13	學生自主學習 (Self-learning)
18	2022/01/20	學生自主學習 (Self-learning)

# Software Engineering and Project Management

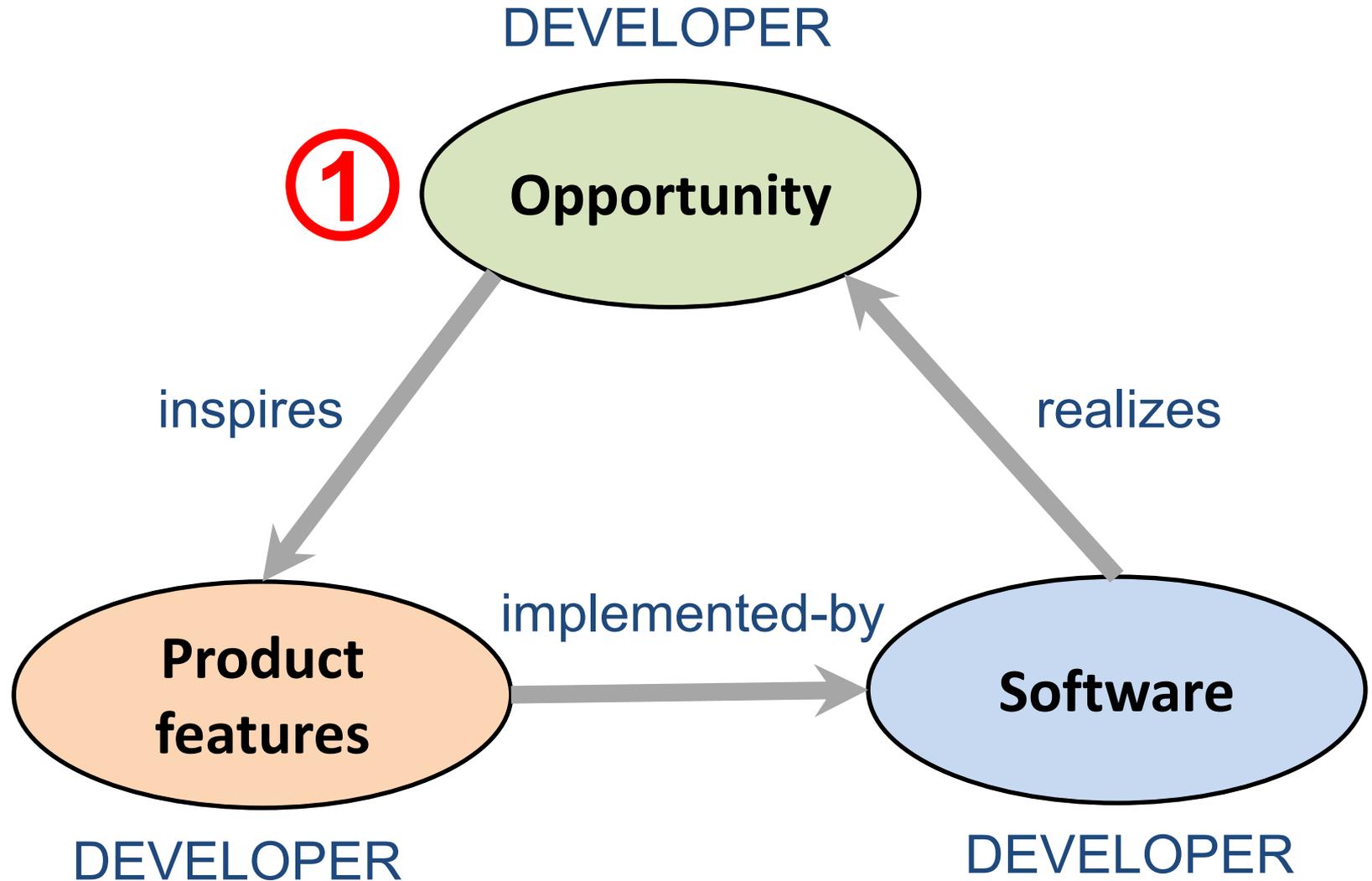


**Project Management**

# Project-based software engineering

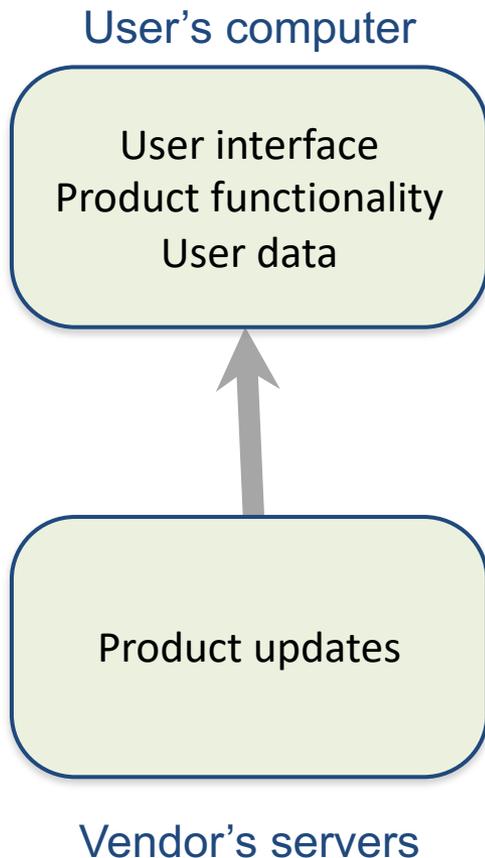


# Product software engineering

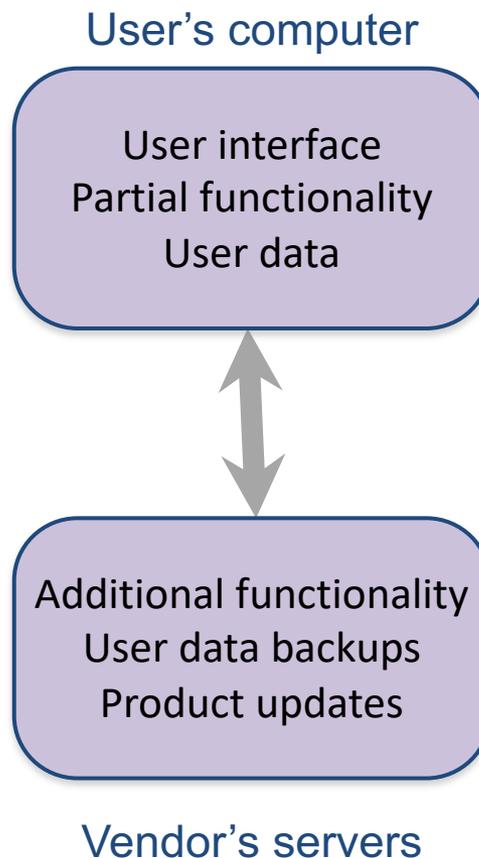


# Software execution models

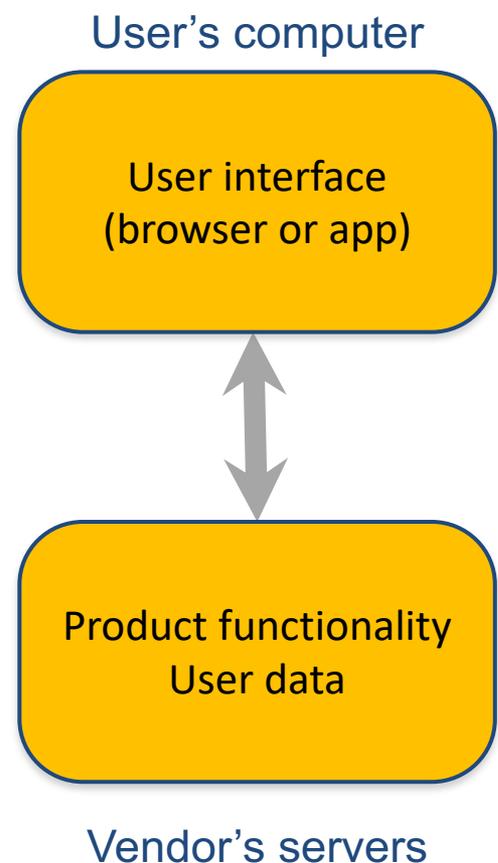
Stand-alone execution



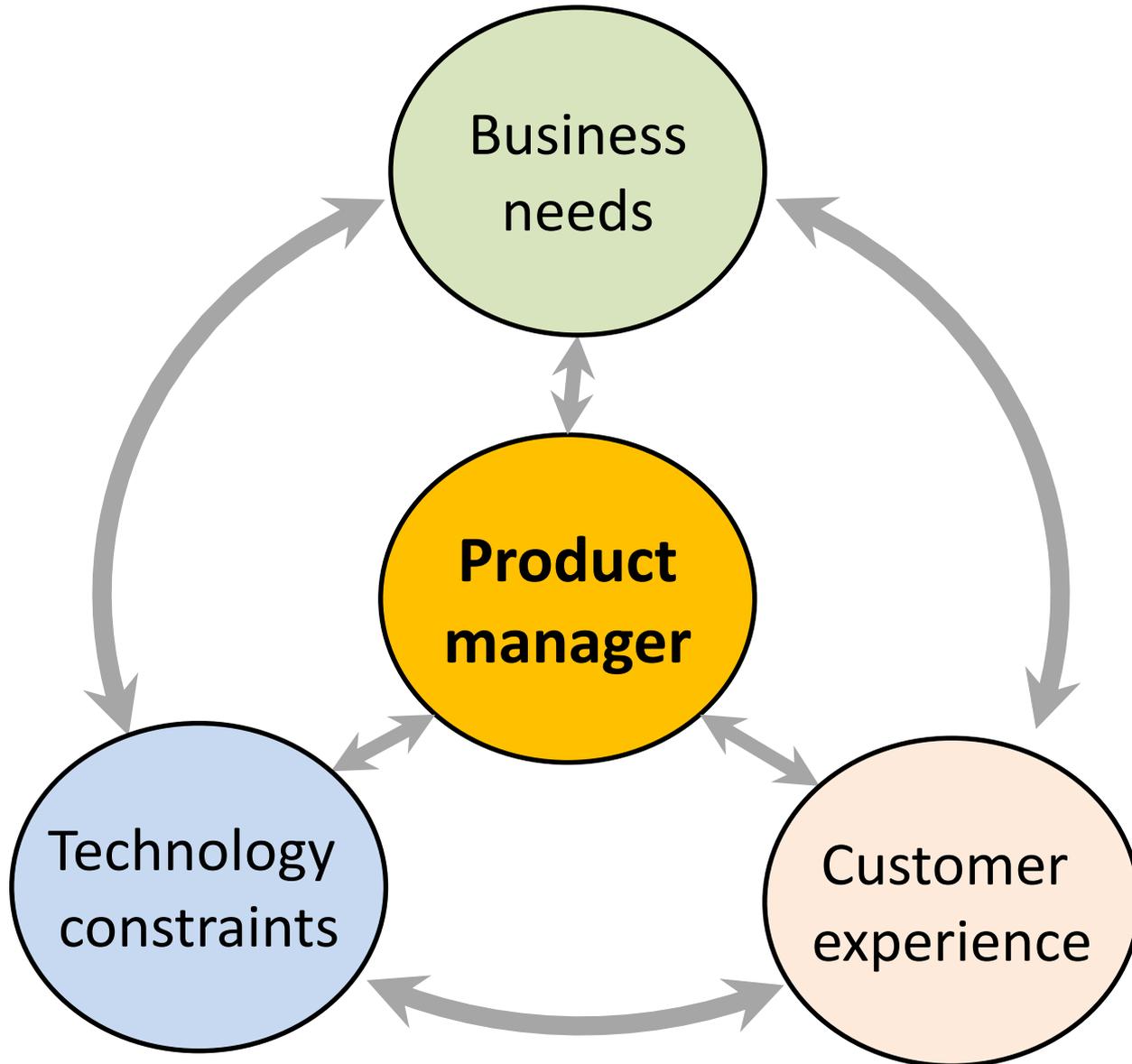
Hybrid execution



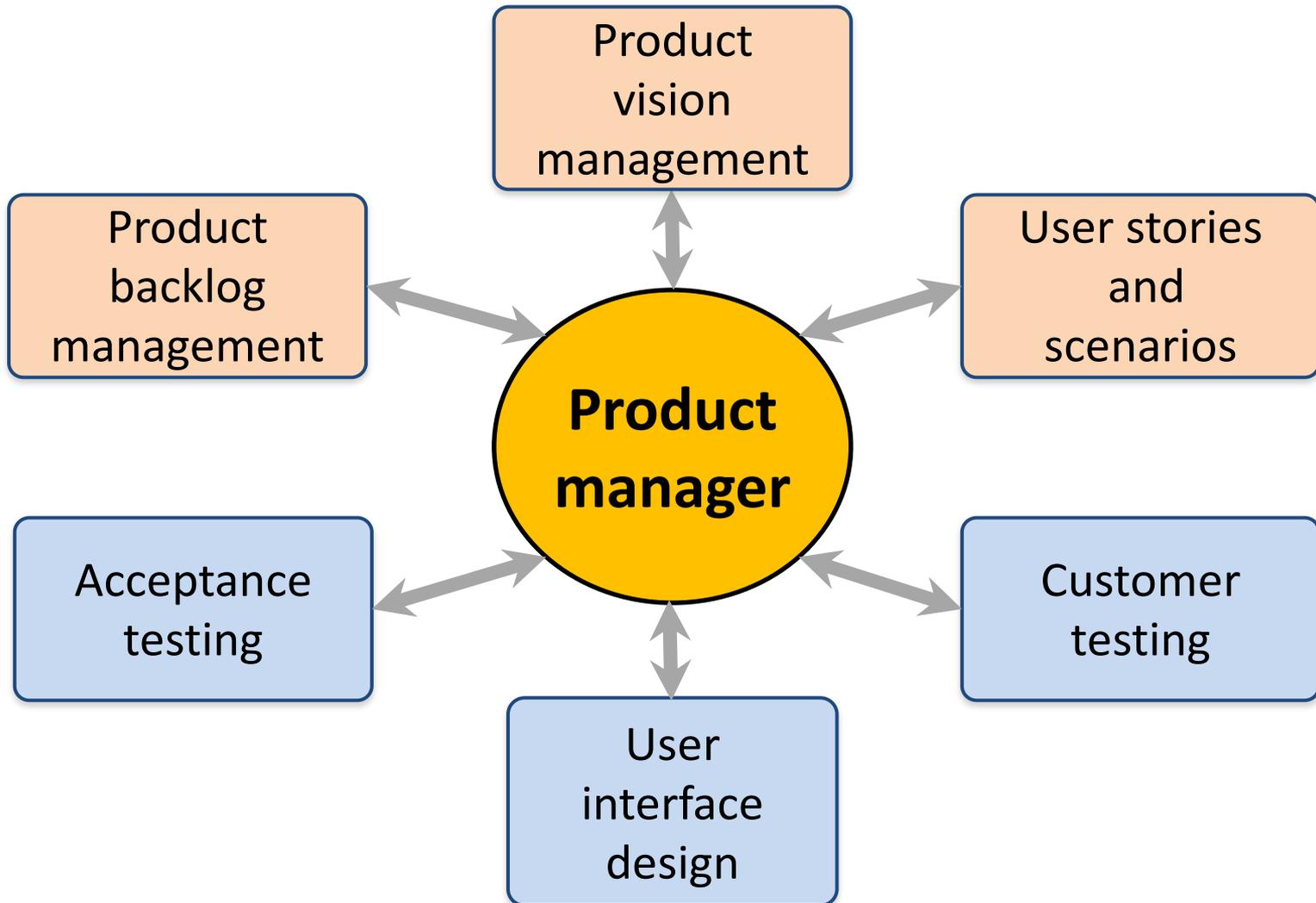
Software as a service



# Product management concerns

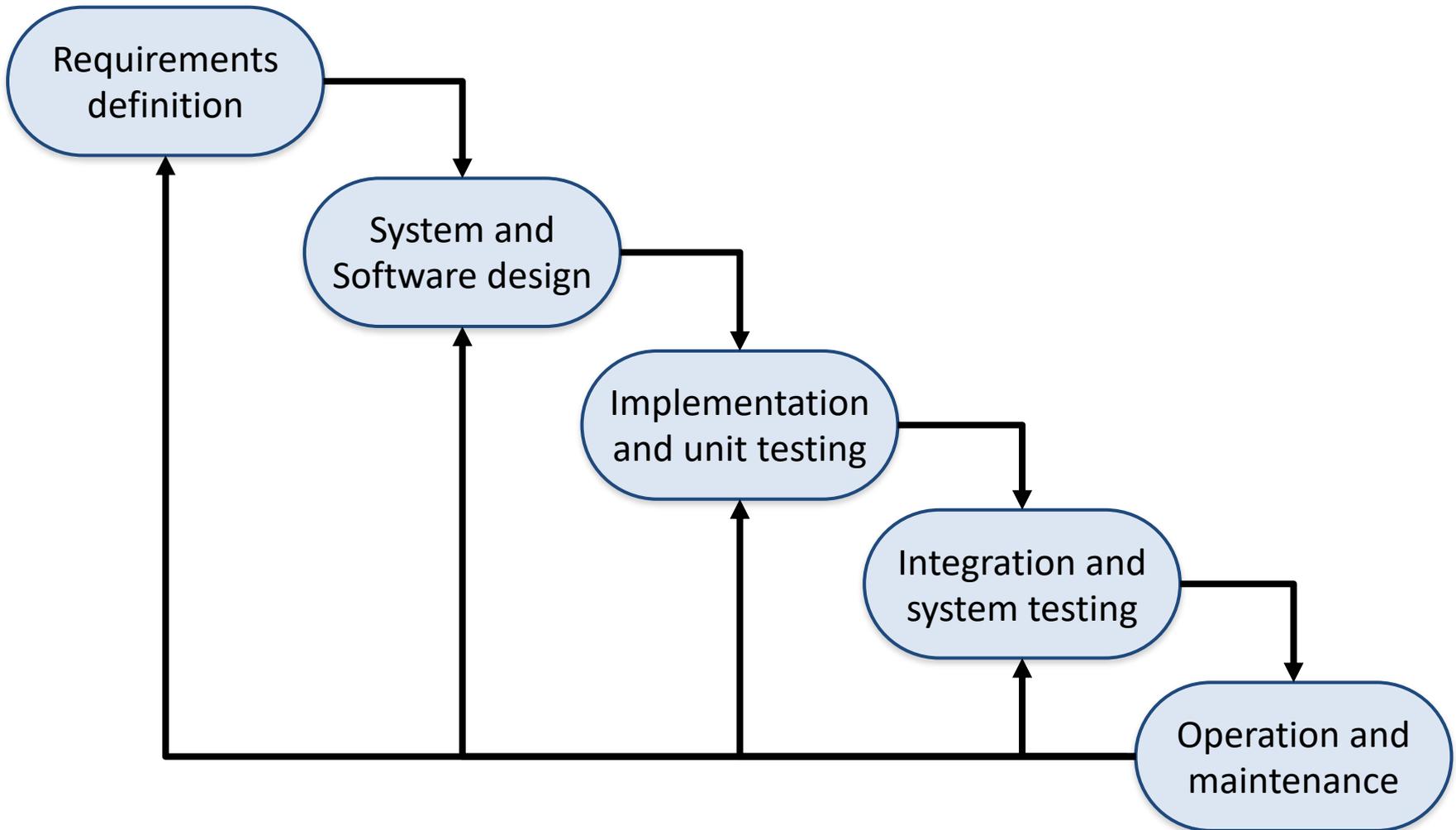


# Technical interactions of product managers



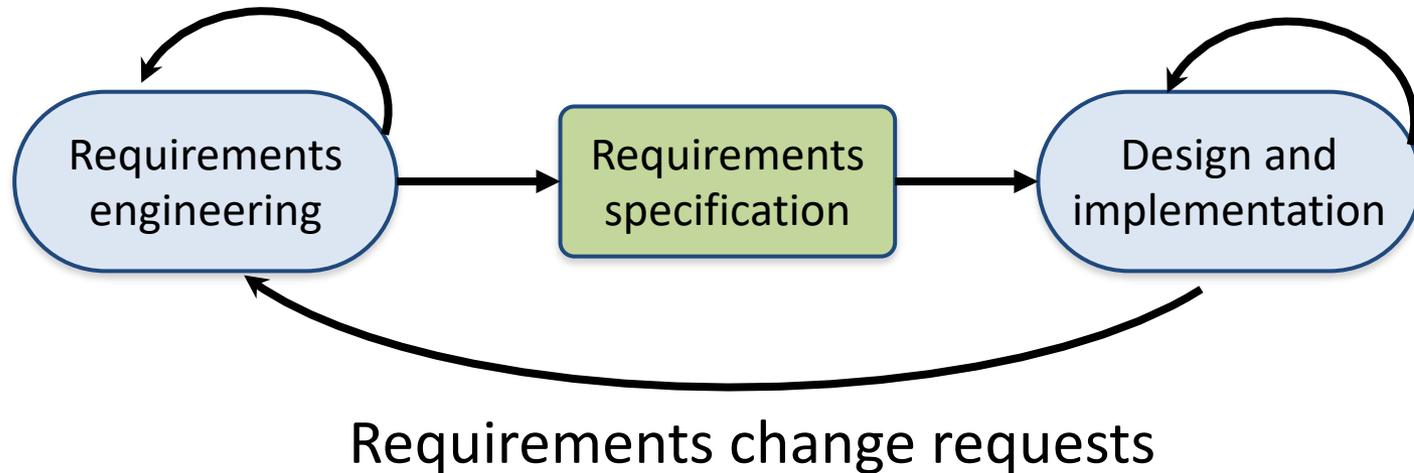
# Software Development Life Cycle (SDLC)

## The waterfall model

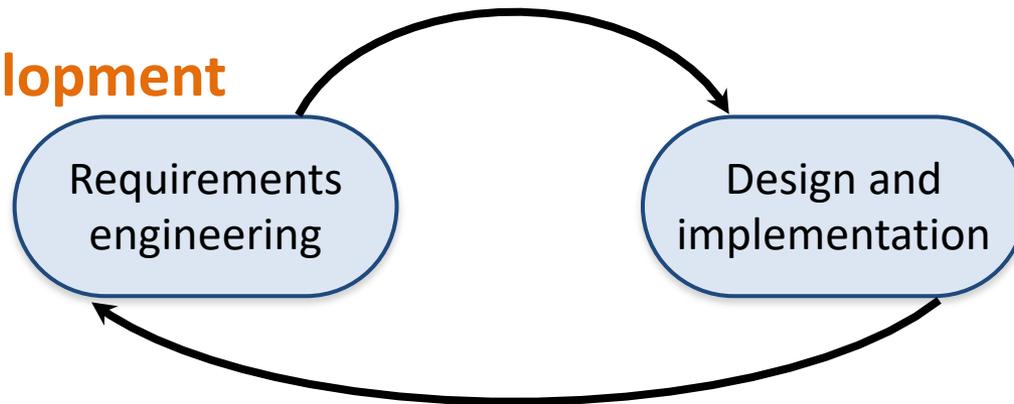


# Plan-based and Agile development

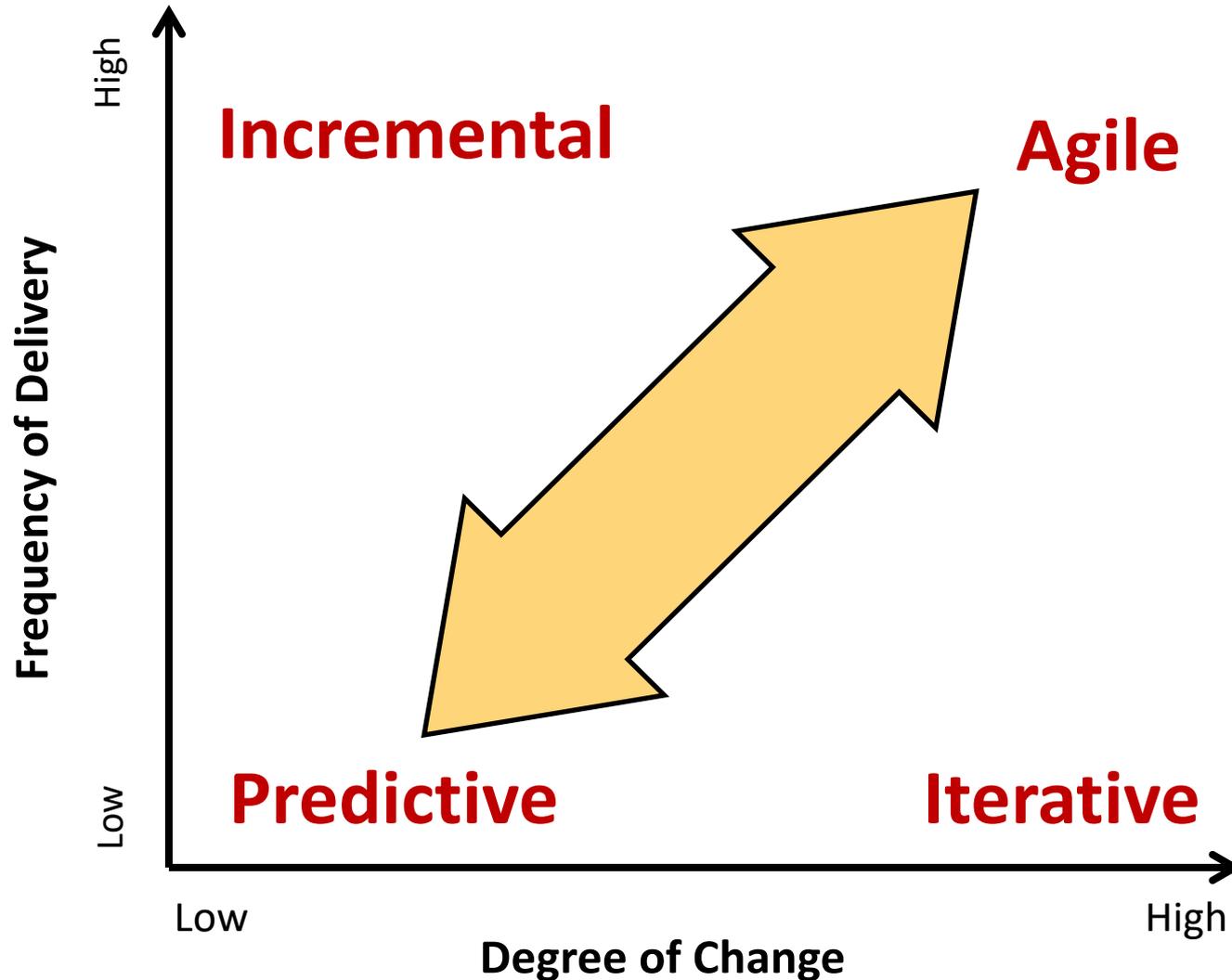
## Plan-based development



## Agile development



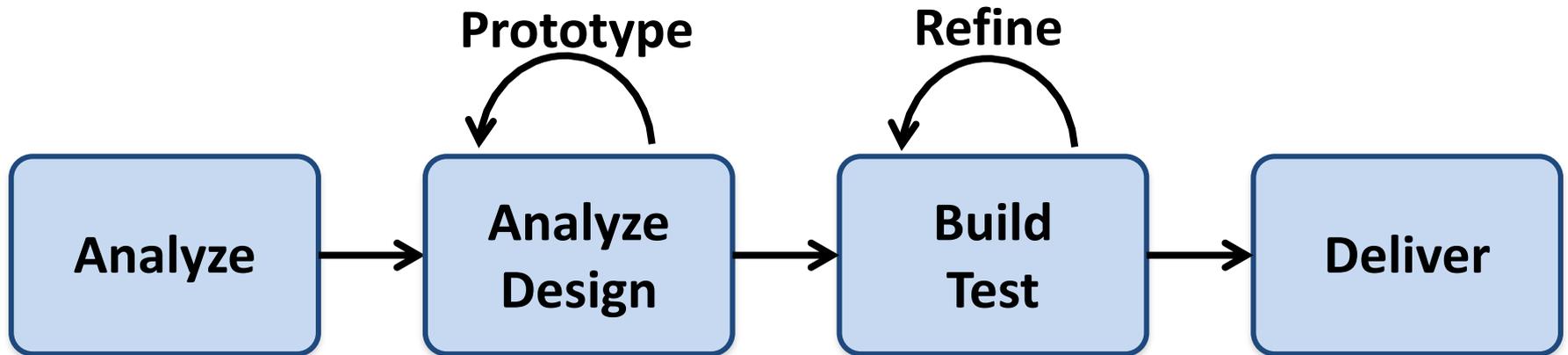
# The Continuum of Life Cycles



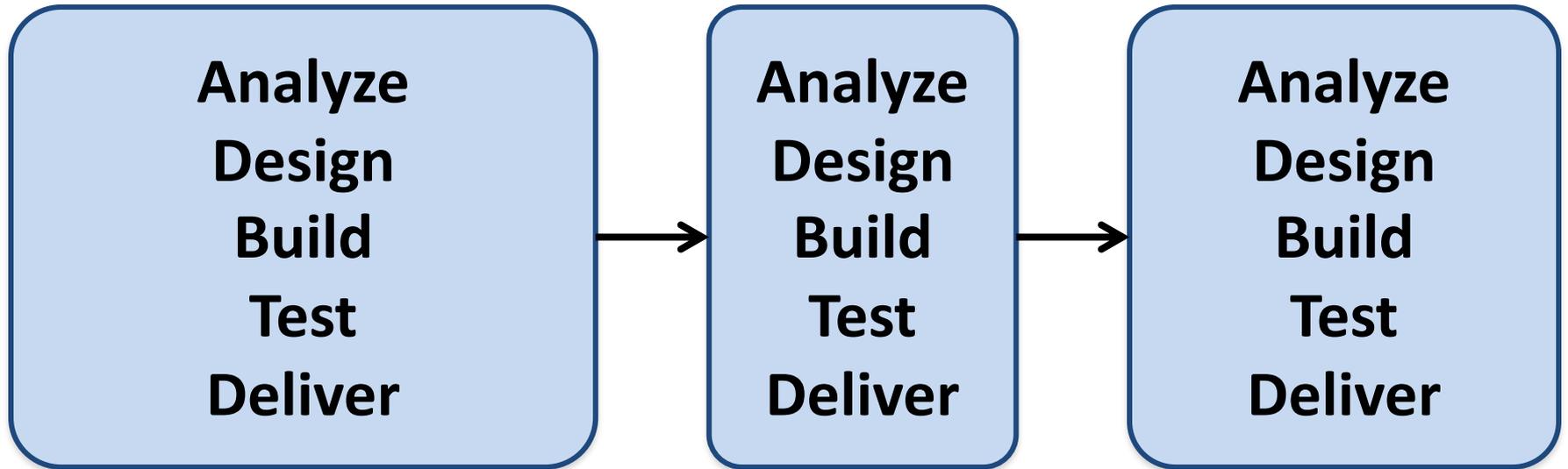
# Predictive Life Cycle



# Iterative Life Cycle

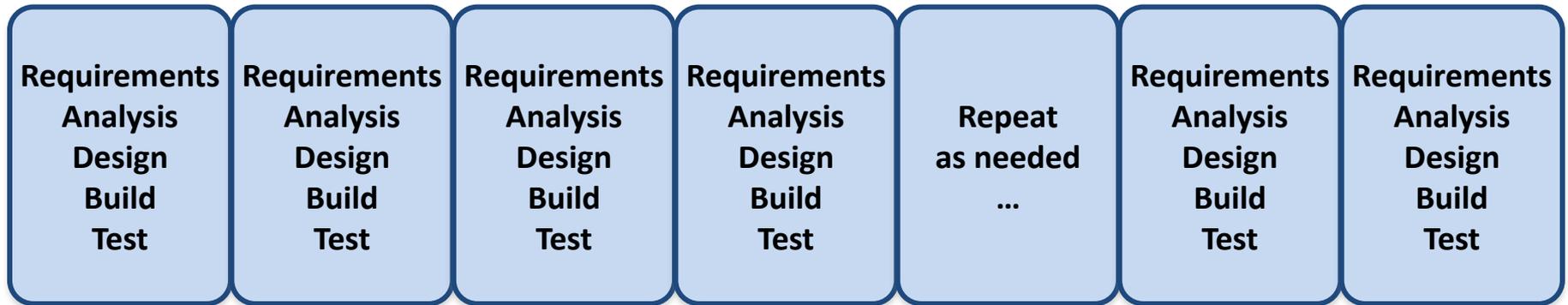


# A Life Cycle of Varying-Sized Increments

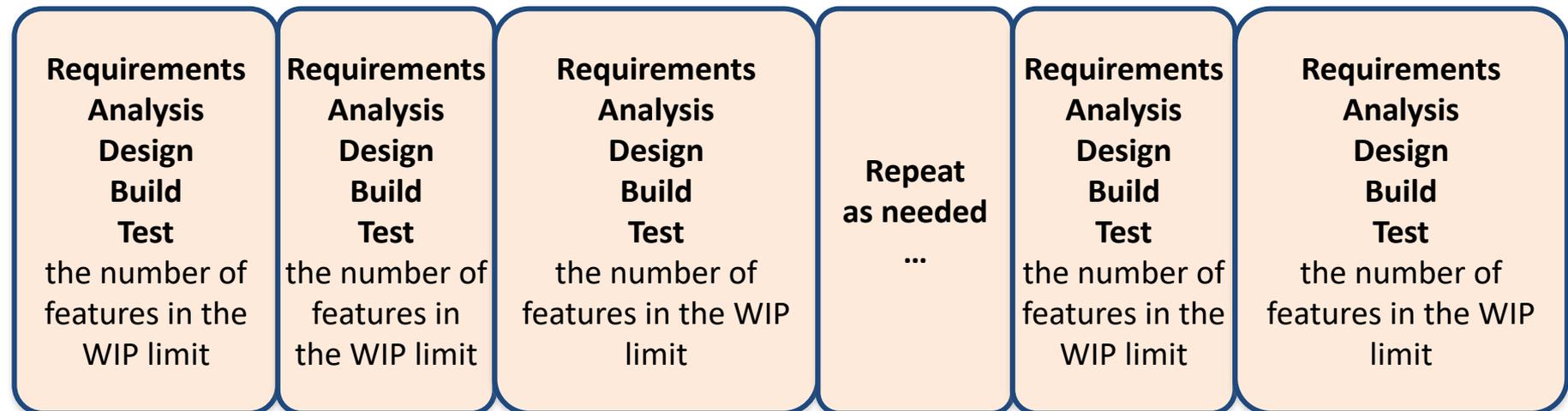


# Iteration-Based and Flow-Based Agile Life Cycles

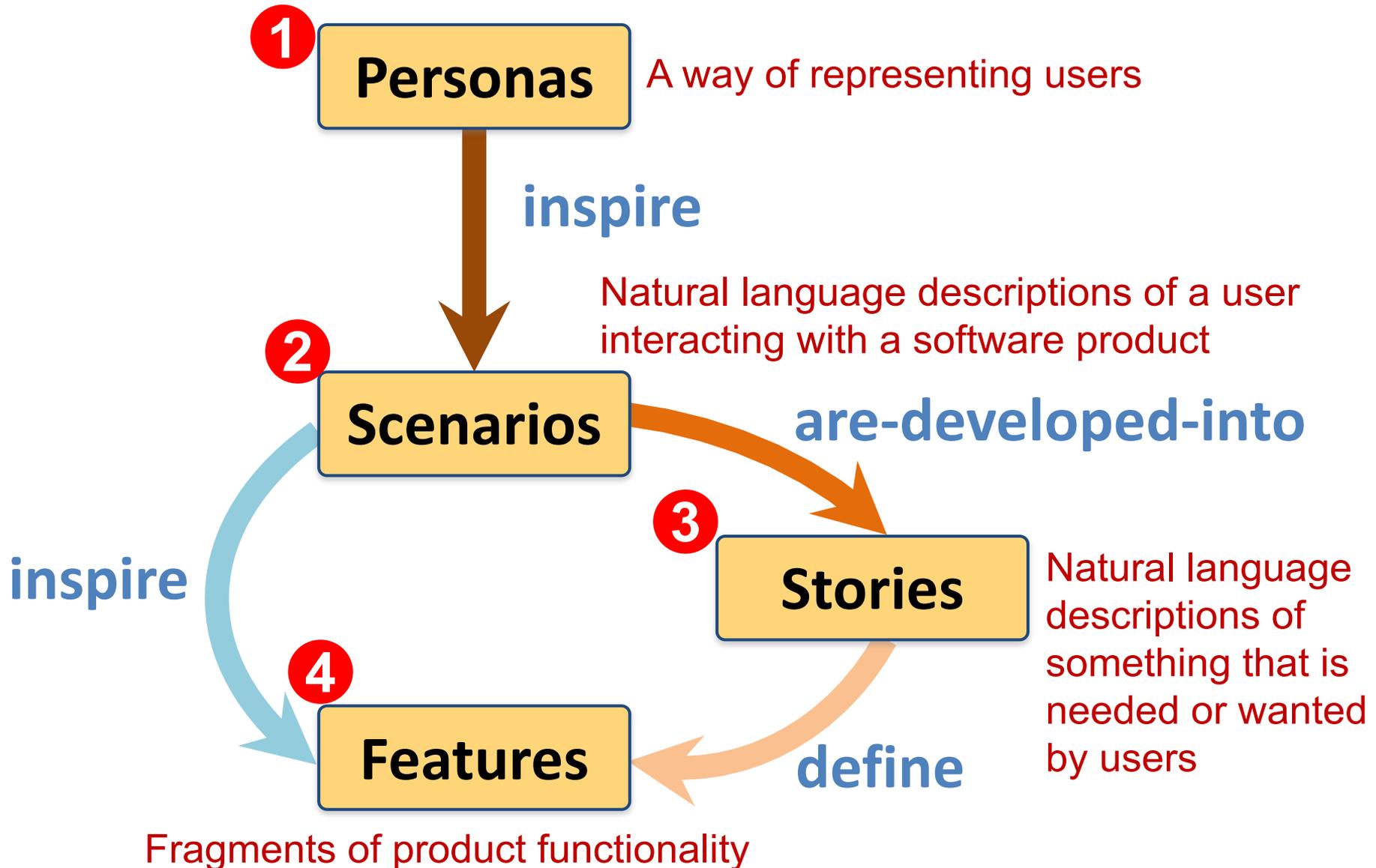
## Iteration-Based Agile



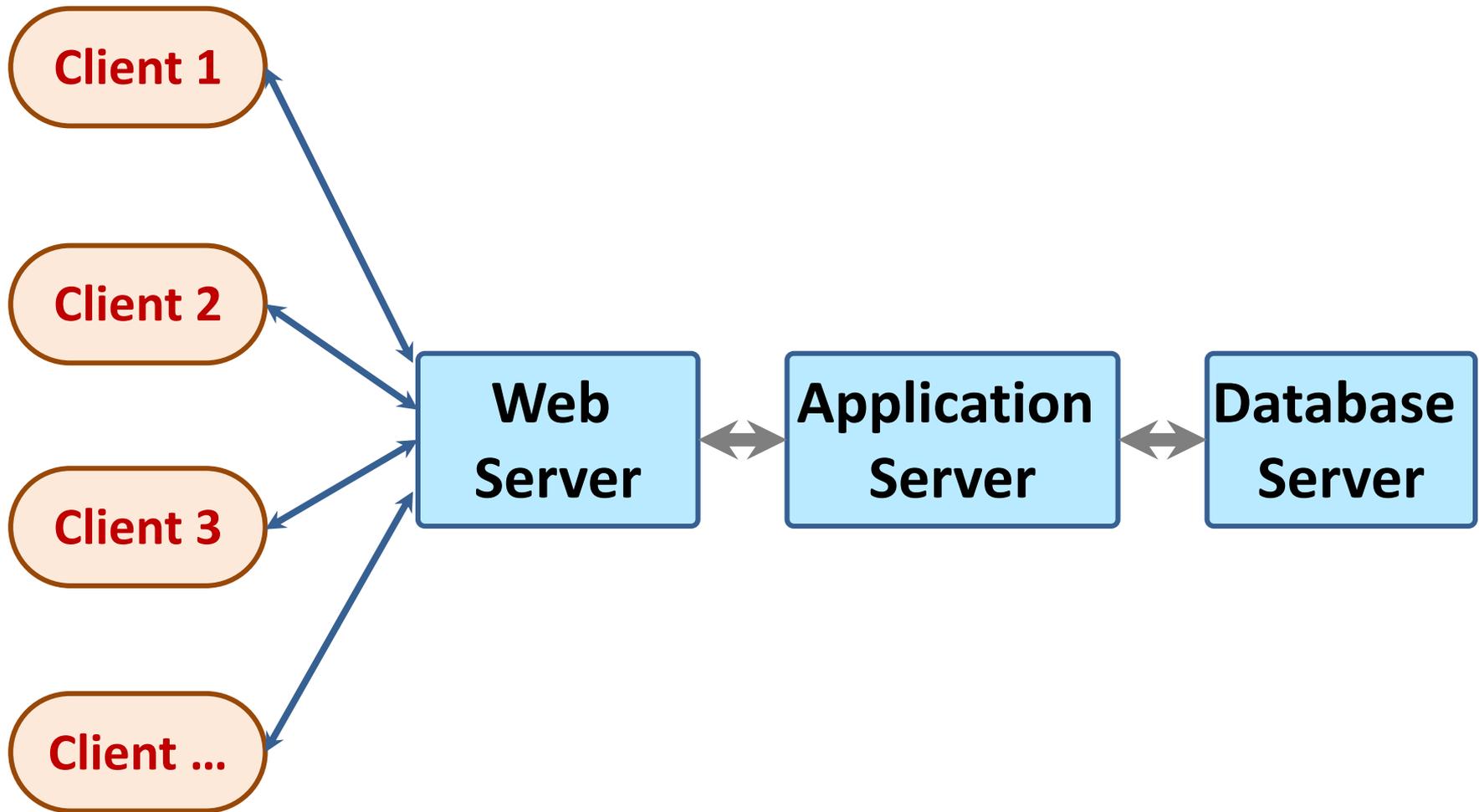
## Flow-Based Agile



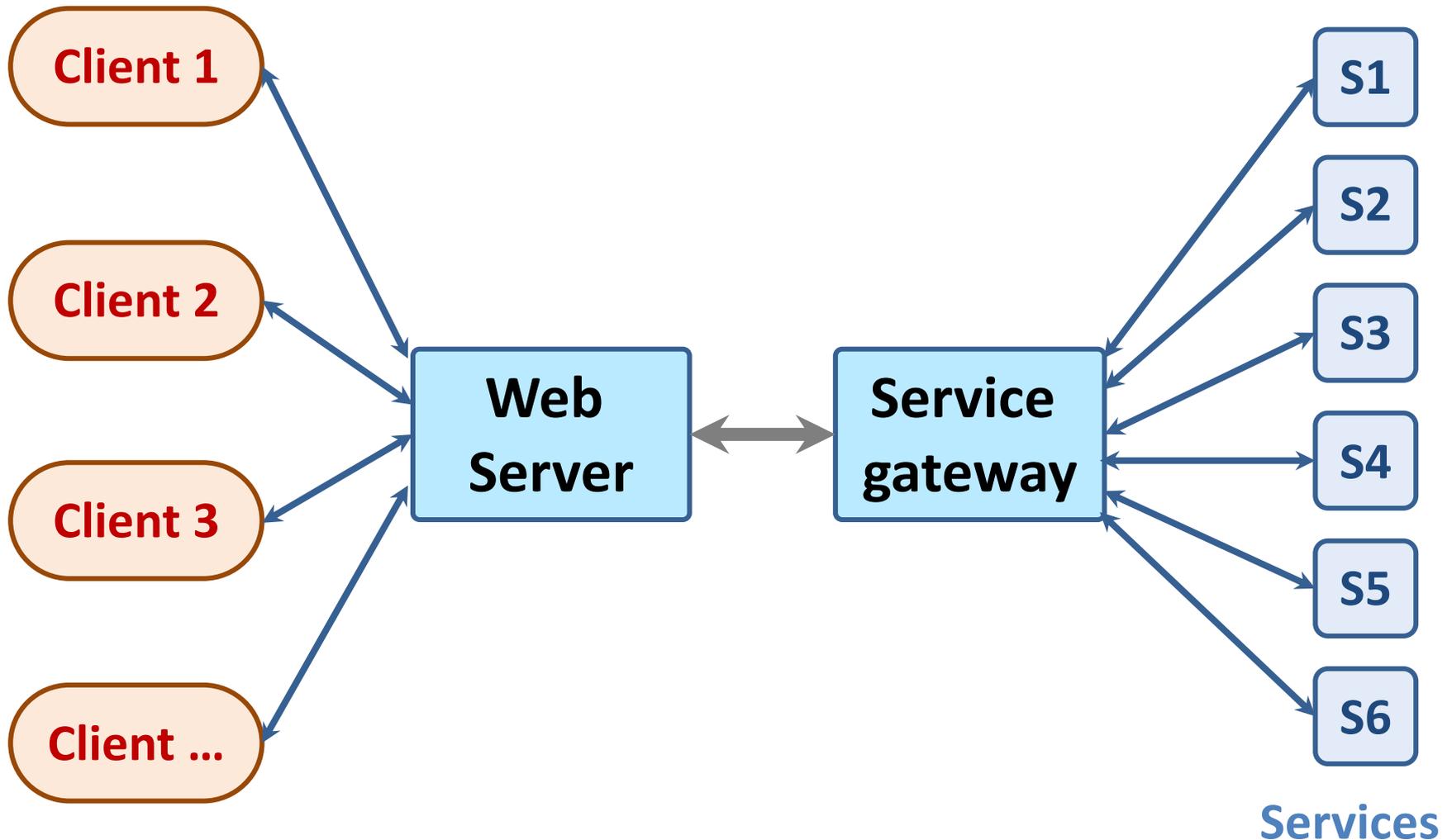
# From personas to features



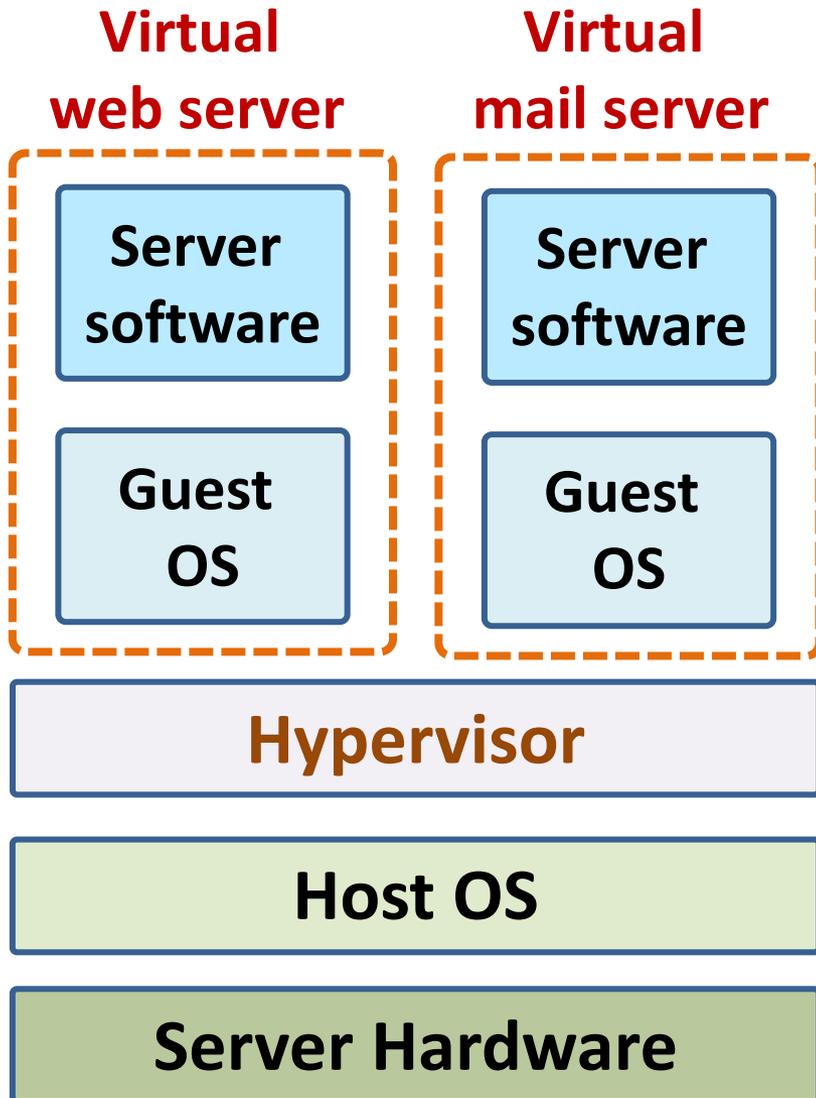
# Multi-tier client-server architecture



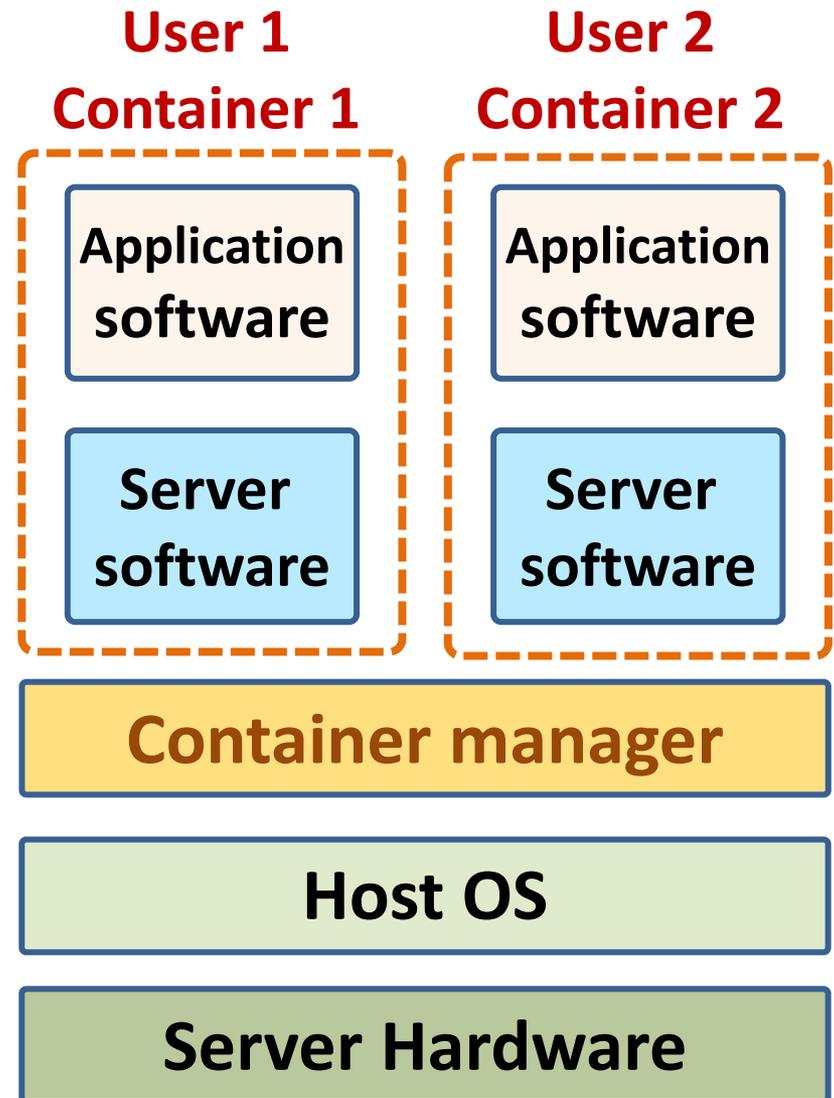
# Service-oriented Architecture



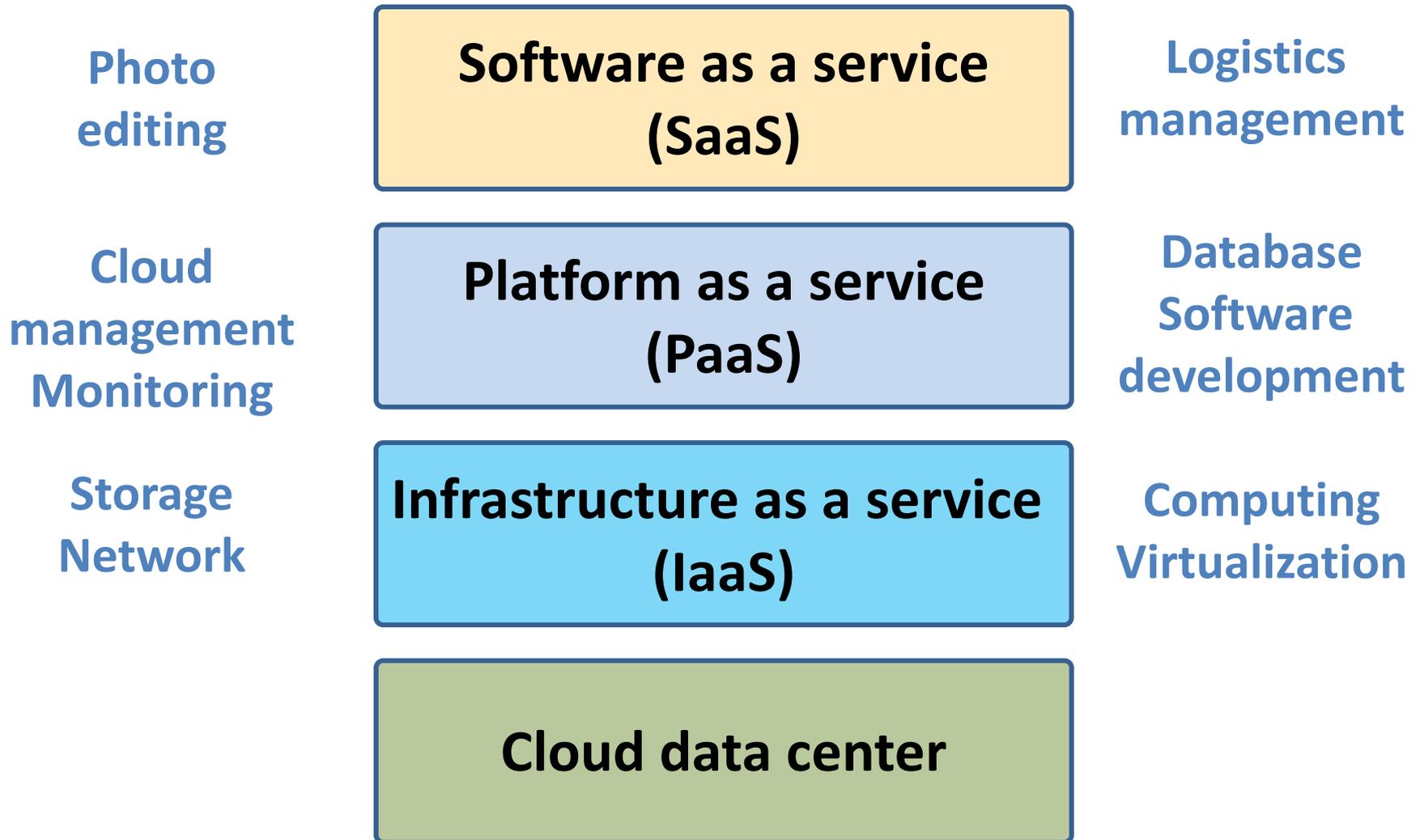
# VM



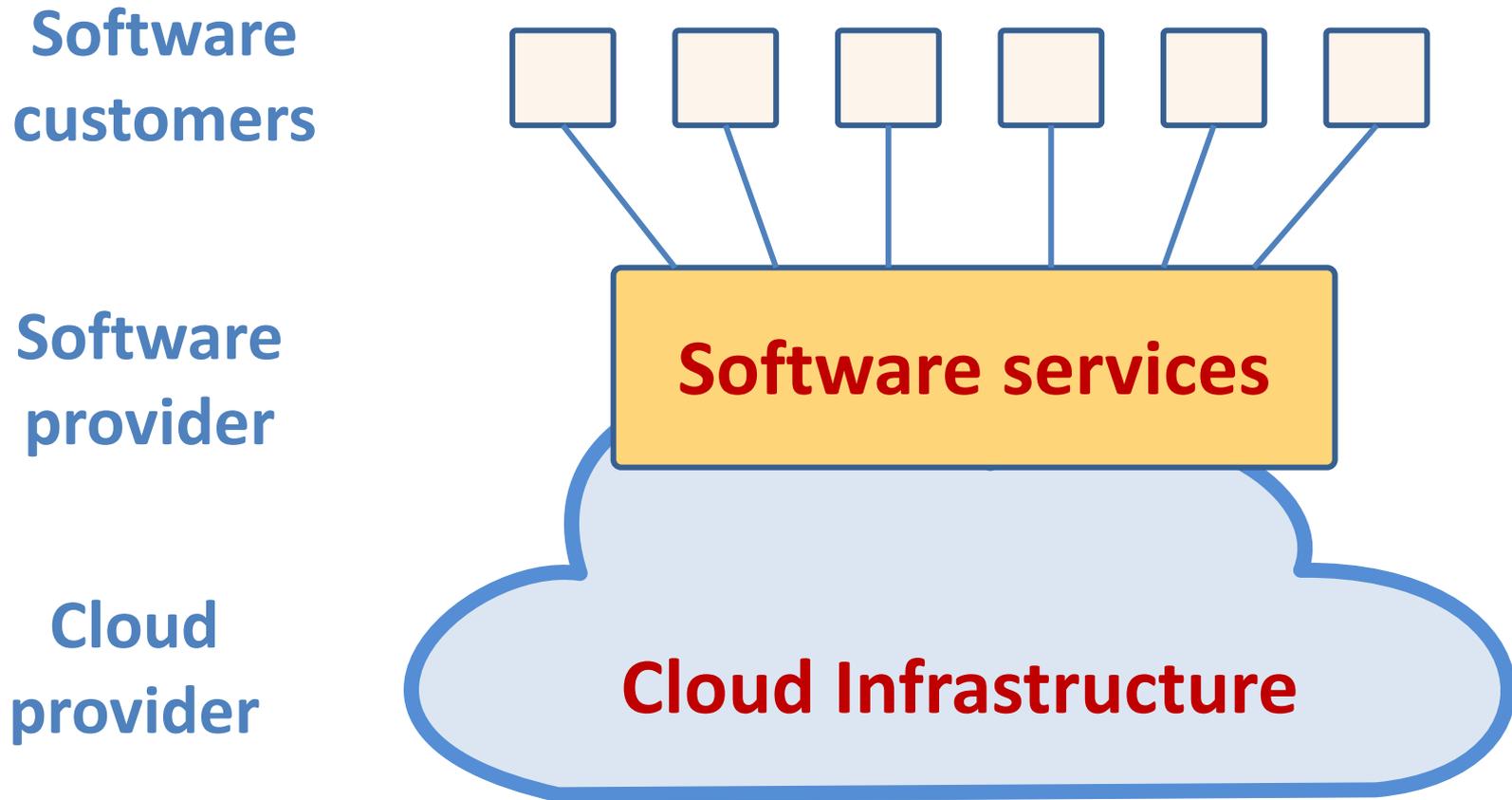
# Container



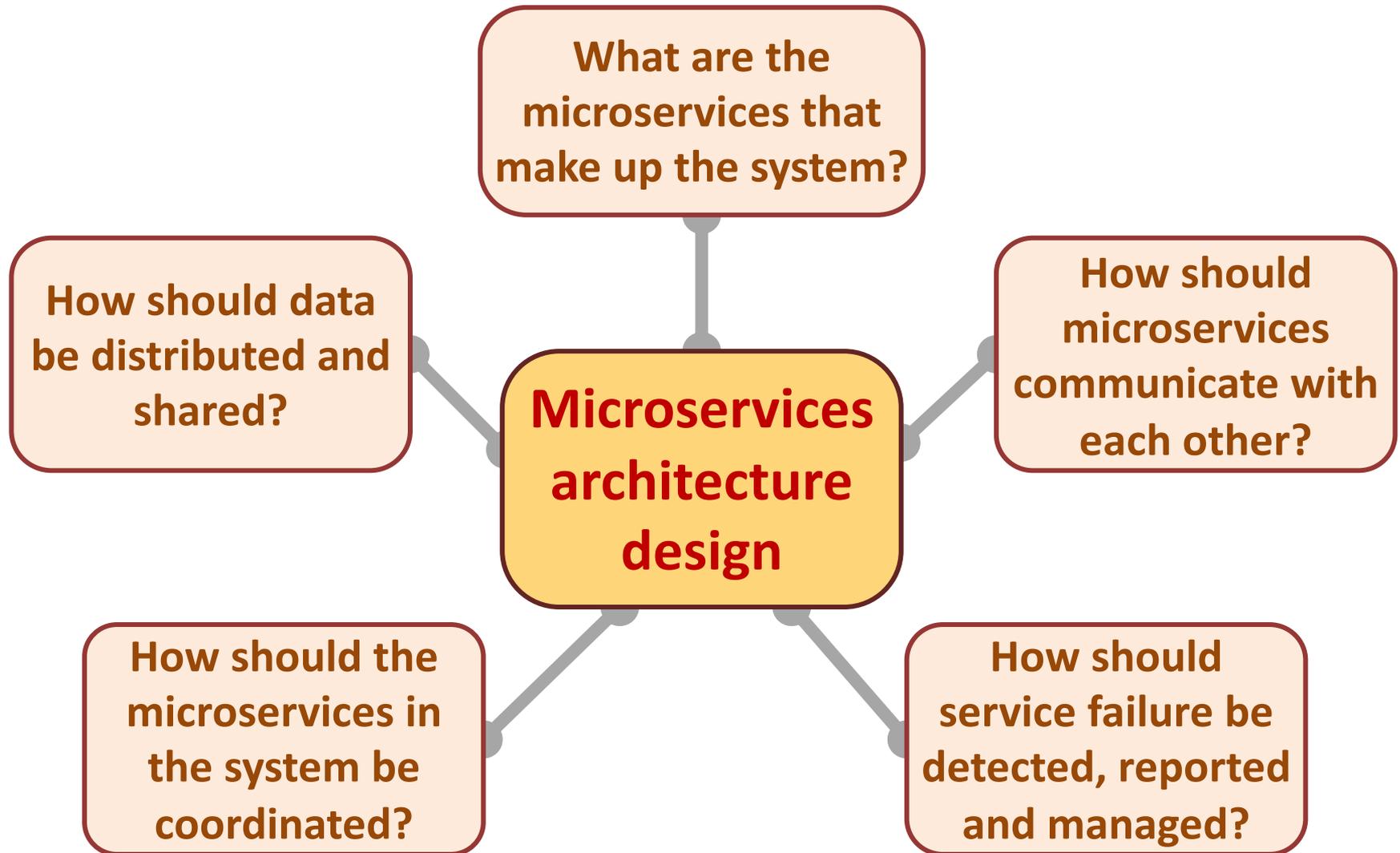
# Everything as a service



# Software as a service



# Microservices architecture – key design questions



# Types of security threat

An attacker attempts to deny access to the system for legitimate users

**Availability threats**

Distributed denial of service (DDoS) attack

An attacker attempts to damage the system or its data

**Integrity threats**

Virus

Ransomware

**SOFTWARE PRODUCT**

**PROGRAM**

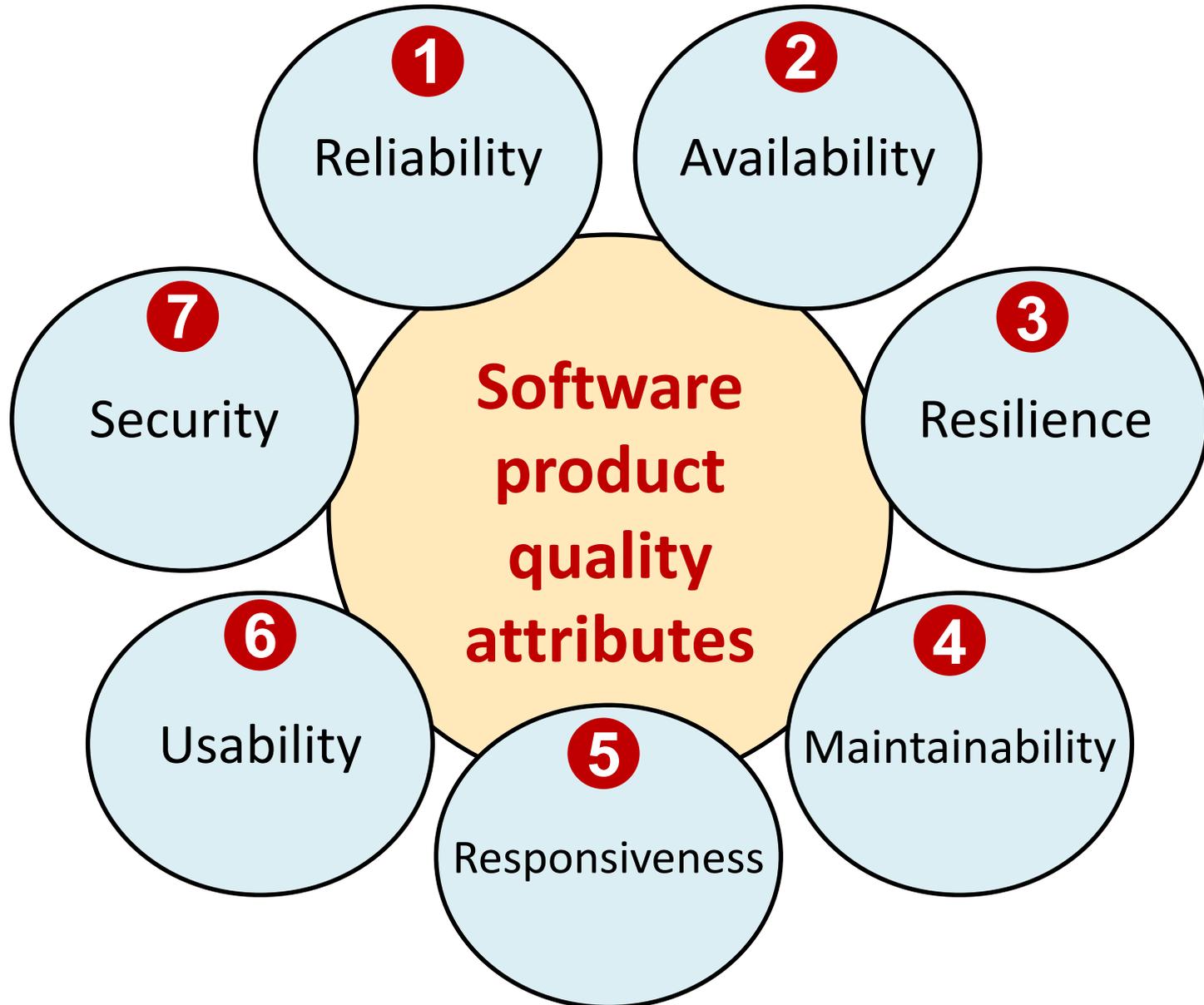
**DATA**

Data theft

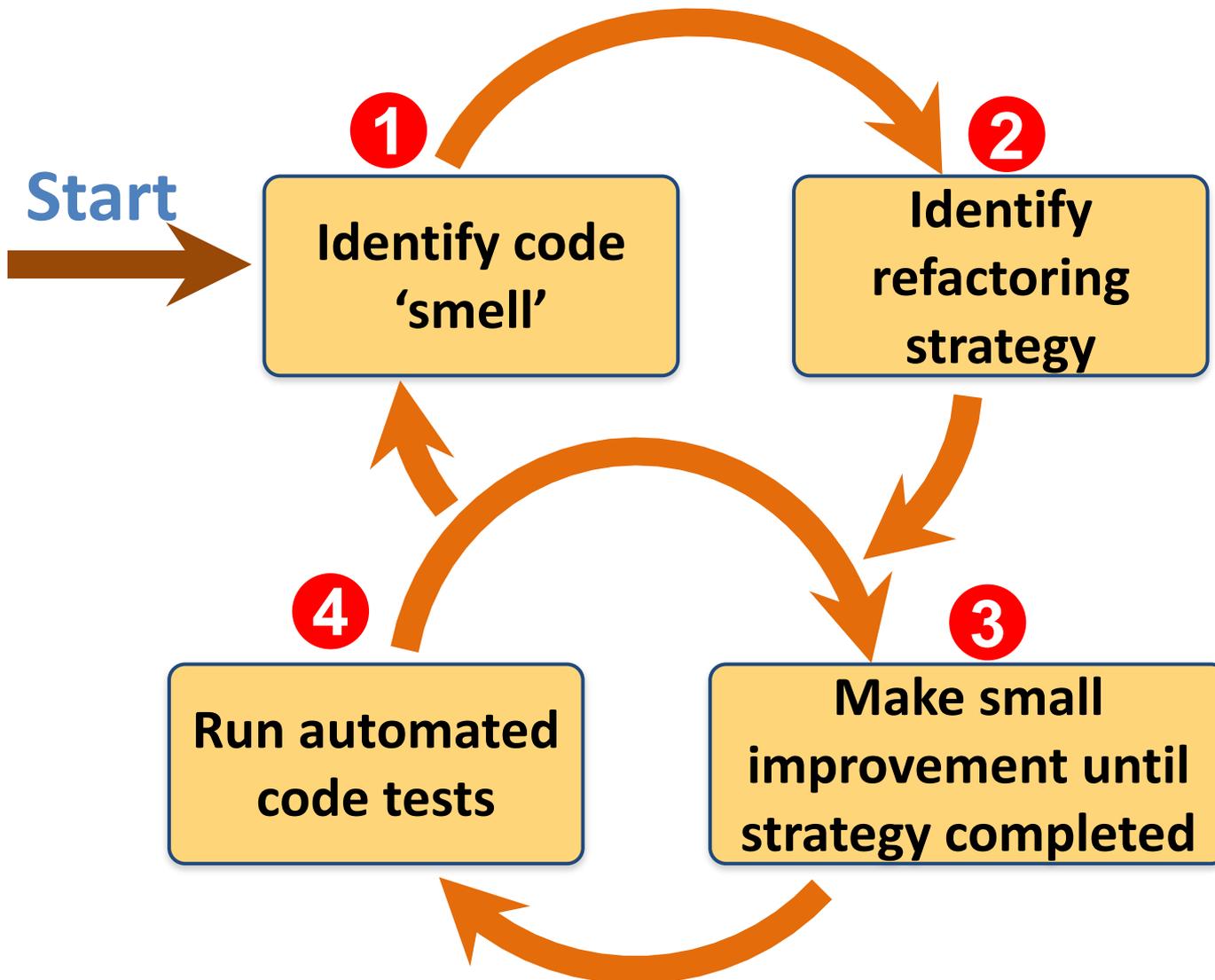
**Confidentiality threats**

An attacker tries to gain access to private information held by the system

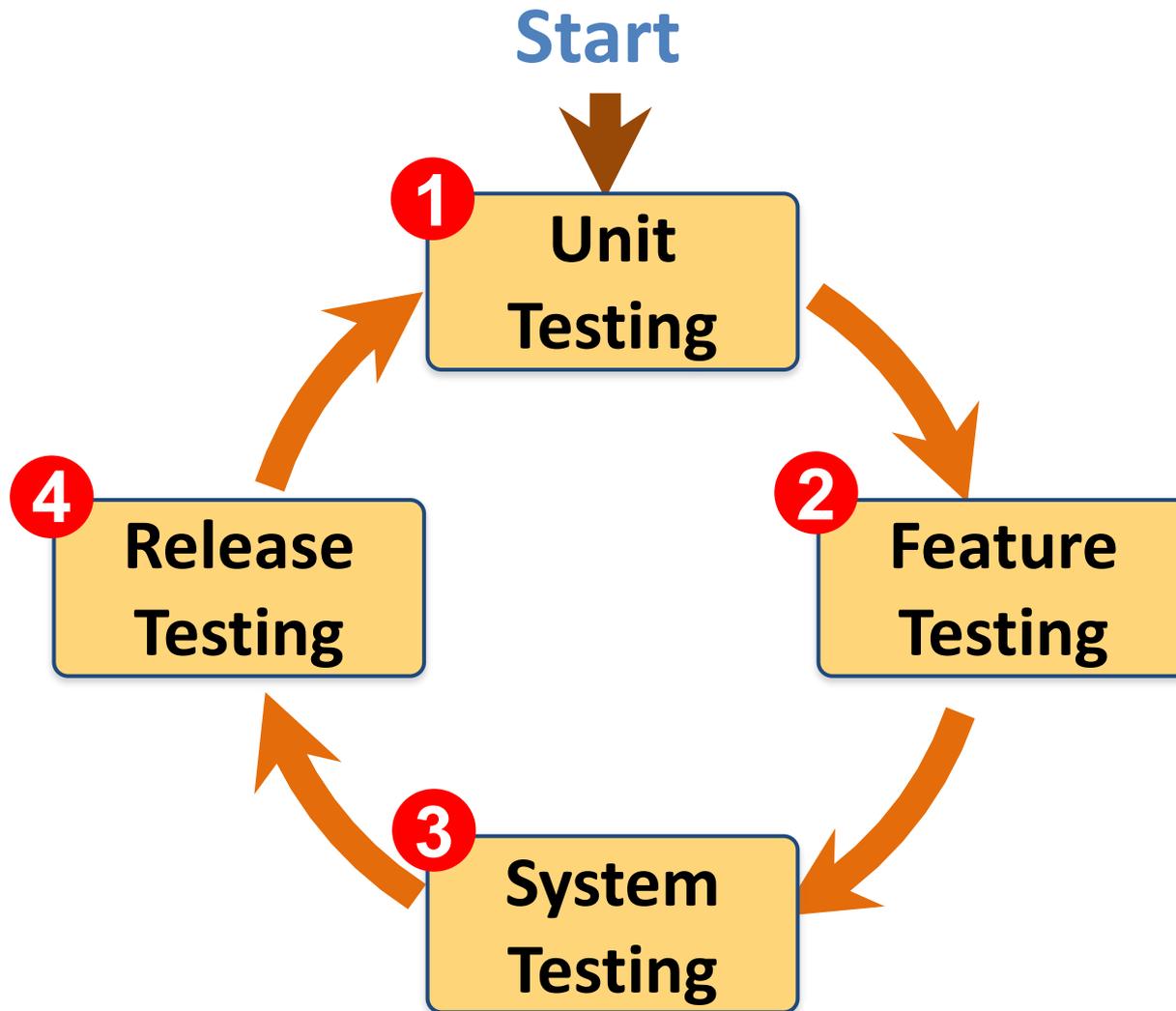
# Software product quality attributes



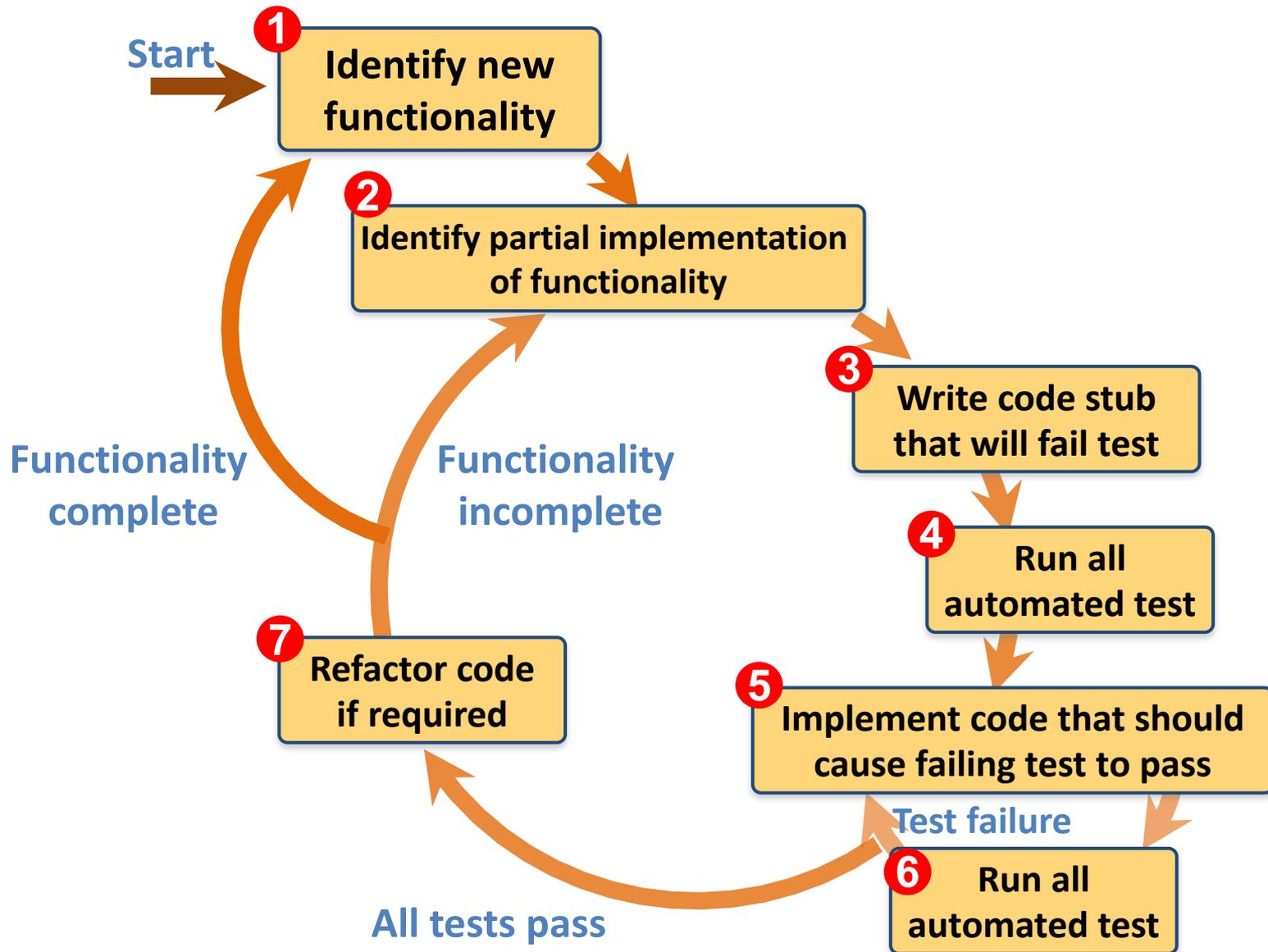
# A refactoring process



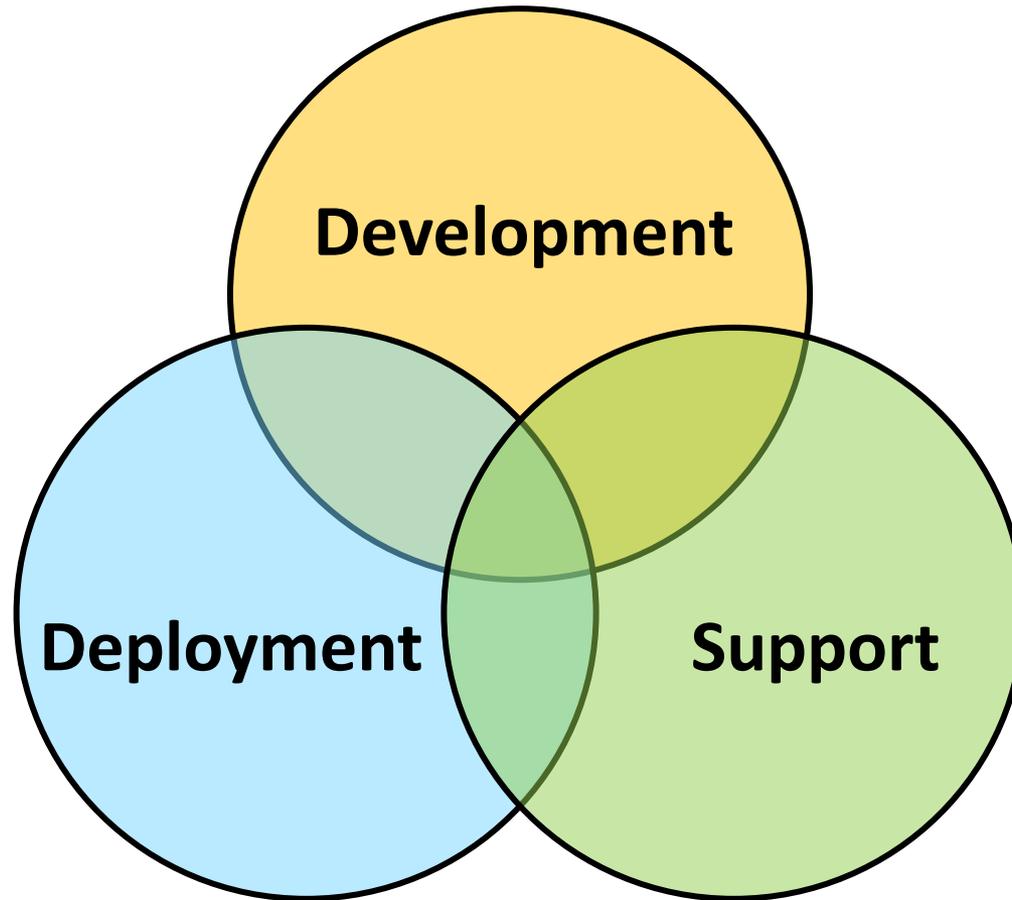
# Functional testing



# Test-driven development (TDD)

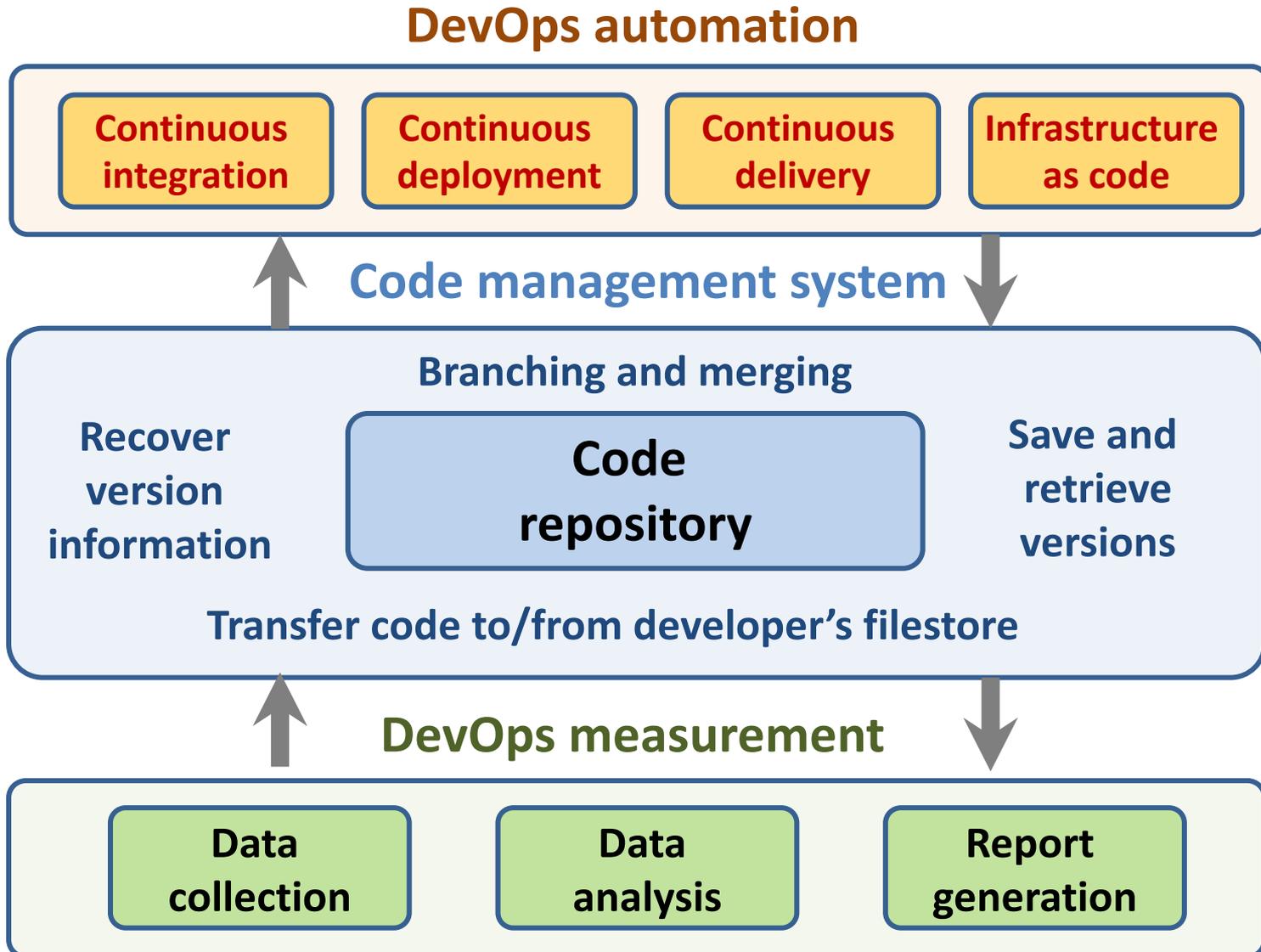


# DevOps



## Multi-skilled DevOps team

# Code management and DevOps



# Security and Privacy

# Outline

- **Security**
- **Privacy**

# Software security

- **Software security** should always be a **high priority** for product developers and their users.
- If you don't **prioritize security**, you and your customers will inevitably suffer losses from **malicious attacks**.
- In the worst case, these attacks could can put product providers out of business.
  - If their product is unavailable or if customer data is compromised, customers are liable to cancel their subscriptions.
- Even if they can recover from the attacks, this will take time and effort that would have been better spent working on their software.

# Types of security threat

An attacker attempts to deny access to the system for legitimate users

**Availability threats**

Distributed denial of service (DDoS) attack

An attacker attempts to damage the system or its data

**Integrity threats**

Virus

Ransomware

**SOFTWARE PRODUCT**

**PROGRAM**

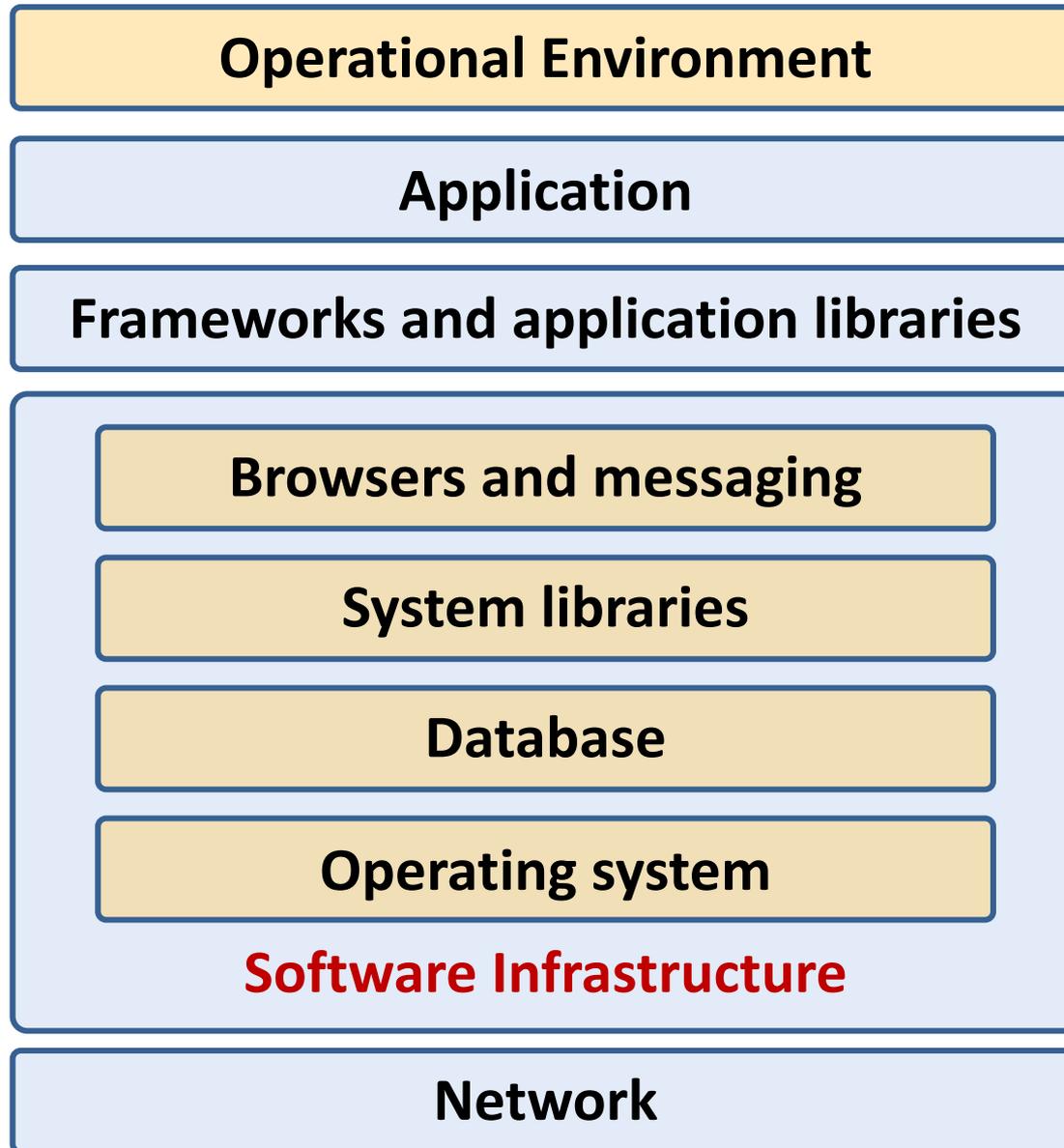
**DATA**

Data theft

**Confidentiality threats**

An attacker tries to gain access to private information held by the system

# System infrastructure stack



# Security management

- **Authentication and authorization**

You should have authentication and authorization standards and procedures that ensure that all users have strong authentication and that they have properly access permissions properly.

- **System infrastructure management**

Infrastructure software should be properly configured and security updates that patch vulnerabilities should be applied as soon as they become available.

- **Attack monitoring**

The system should be regularly checked for possible unauthorized access. If attacks are detected, it may be possible to put resistance strategies in place that minimize the effects of the attack.

- **Backup**

Backup policies should be implemented to ensure that you keep undamaged copies of program and data files. These can then be restored after an attack.

# Operational security

- **Operational security** focuses on **helping users to maintain security**. User attacks try to trick users into disclosing their credentials or accessing a website that includes malware such as a key-logging system.
- **Operational security procedures and practices**
  - **Auto-logout**, which addresses the common problem of users forgetting to logout from a computer used in a shared space.
  - **User command logging**, which makes it possible to discover actions taken by users that have deliberately or accidentally damaged some system resources.
  - **Multi-factor authentication**, which reduces the chances of an intruder gaining access to the system using stolen credentials.

# Injection attacks

- **Injection attacks** are a type of attack where a malicious user uses a valid input field to input malicious code or database commands.
- These **malicious instructions** are then executed, causing some damage to the system. Code can be injected that leaks system data to the attackers.
- Common types of injection attack include **buffer overflow attacks** and **SQL poisoning attacks**.

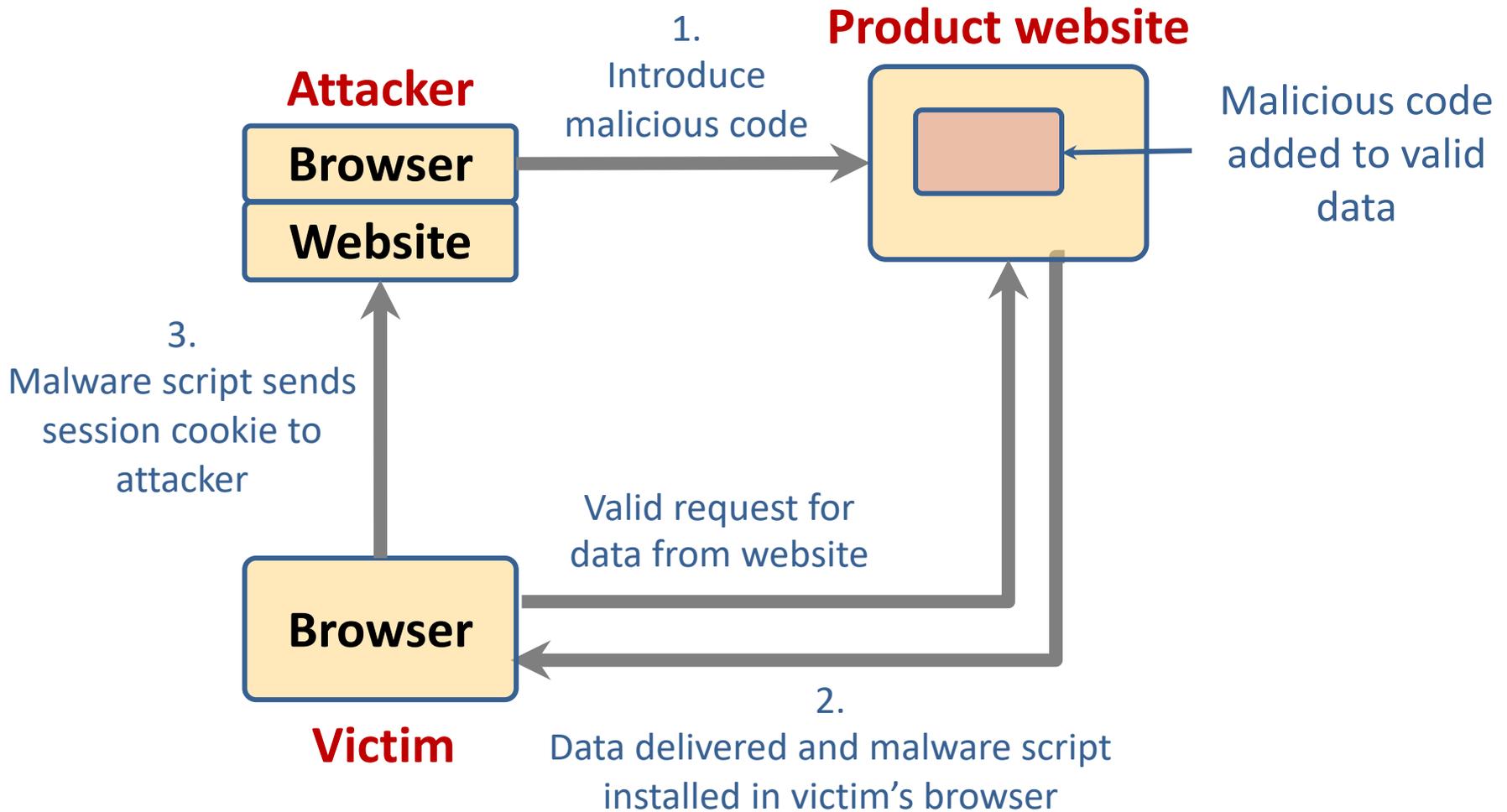
# SQL poisoning attacks

- **SQL poisoning attacks** are attacks on software products that use an SQL database.
- They take advantage of a situation where a **user input is used as part of an SQL command**.
- A malicious user uses a form input field to input a fragment of SQL that allows access to the database.
- The form field is added to the SQL query, which is executed and returns the information to the attacker.

# Cross-site scripting attacks

- **Cross-site scripting attacks** are another form of **injection attack**.
- An attacker adds **malicious Javascript code** to the web page that is returned from a server to a client and this script is executed when the page is displayed in the user's browser.
- The malicious script may steal customer information or direct them to another website.
- This may try to capture personal data or display advertisements.
- **Cookies** may be stolen, which makes a session **hijacking attack** possible.
- As with other types of injection attack, cross-site scripting attacks may be avoided by **input validation**.

# Cross-site scripting attack



# Session hijacking attacks

- When a **user authenticates** themselves with a web application, a session is created.
  - A session is a time period during which the user's **authentication is valid**. They don't have to re-authenticate for each interaction with the system.
  - The authentication process involves placing a session cookie on the user's device
- **Session hijacking** is a type of attack where an attacker gets hold of a session cookie and uses this to impersonate a legitimate user.

# Session hijacking attacks

- There are several ways that an attacker can find out the **session cookie value** including **cross-site scripting attacks** and **traffic monitoring**.
  - In a **cross-site scripting attack**, the installed malware sends **session cookies** to the attackers.
  - **Traffic monitoring** involves attackers capturing the traffic between the client and server. The session cookie can then be identified by analyzing the data exchanged.

# Actions to reduce the likelihood of hacking

- **Traffic encryption**

Always encrypt the network traffic between clients and your server. This means setting up sessions using https rather than http. If traffic is encrypted it is harder to monitor to find session cookies.

- **Multi-factor authentication**

Always use multi-factor authentication and require confirmation of new actions that may be damaging. For example, before a new payee request is accepted, you could ask the user to confirm their identity by inputting a code sent to their phone.

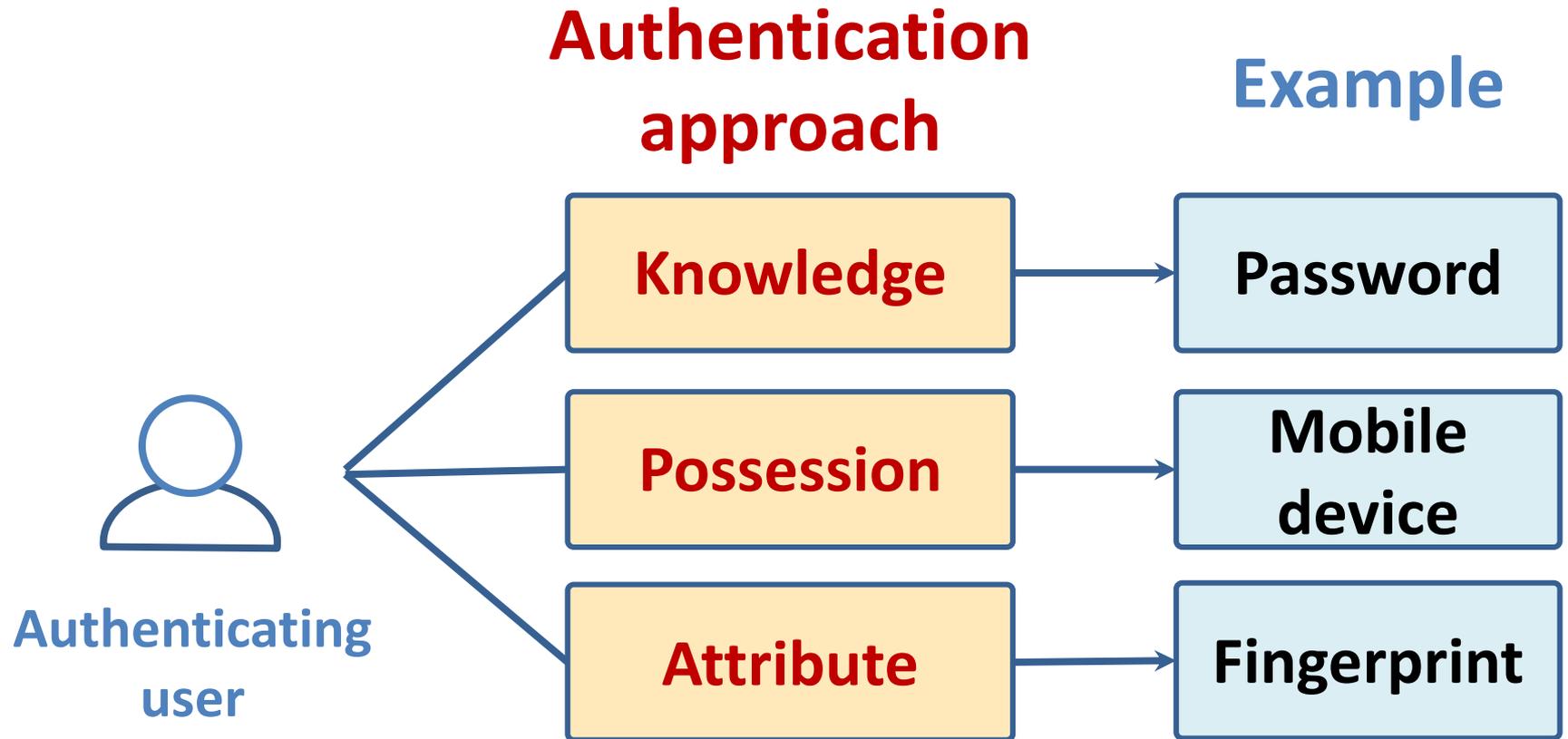
- **Short timeouts**

Use relatively short timeouts on sessions. If there has been no activity in a session for a few minutes, the session should be ended and future requests directed to an authentication page.

# Authentication

- **Authentication** is the process of **ensuring** that a **user** of your system is who they claim to be.
- You need **authentication** in all software products that **maintain user information**, so that only the providers of that information can access and change it.
- You also use **authentication** to learn about your users so that you can **personalize** their **experience** of using your product.

# Authentication approaches



# Weaknesses of password-based authentication

- **Insecure passwords**

Users choose passwords that are easy to remember.

- **Phishing attacks**

Users click on an email link that points to a fake site that tries to collect their login and password details.

- **Password reuse**

Users use the same password for several sites.

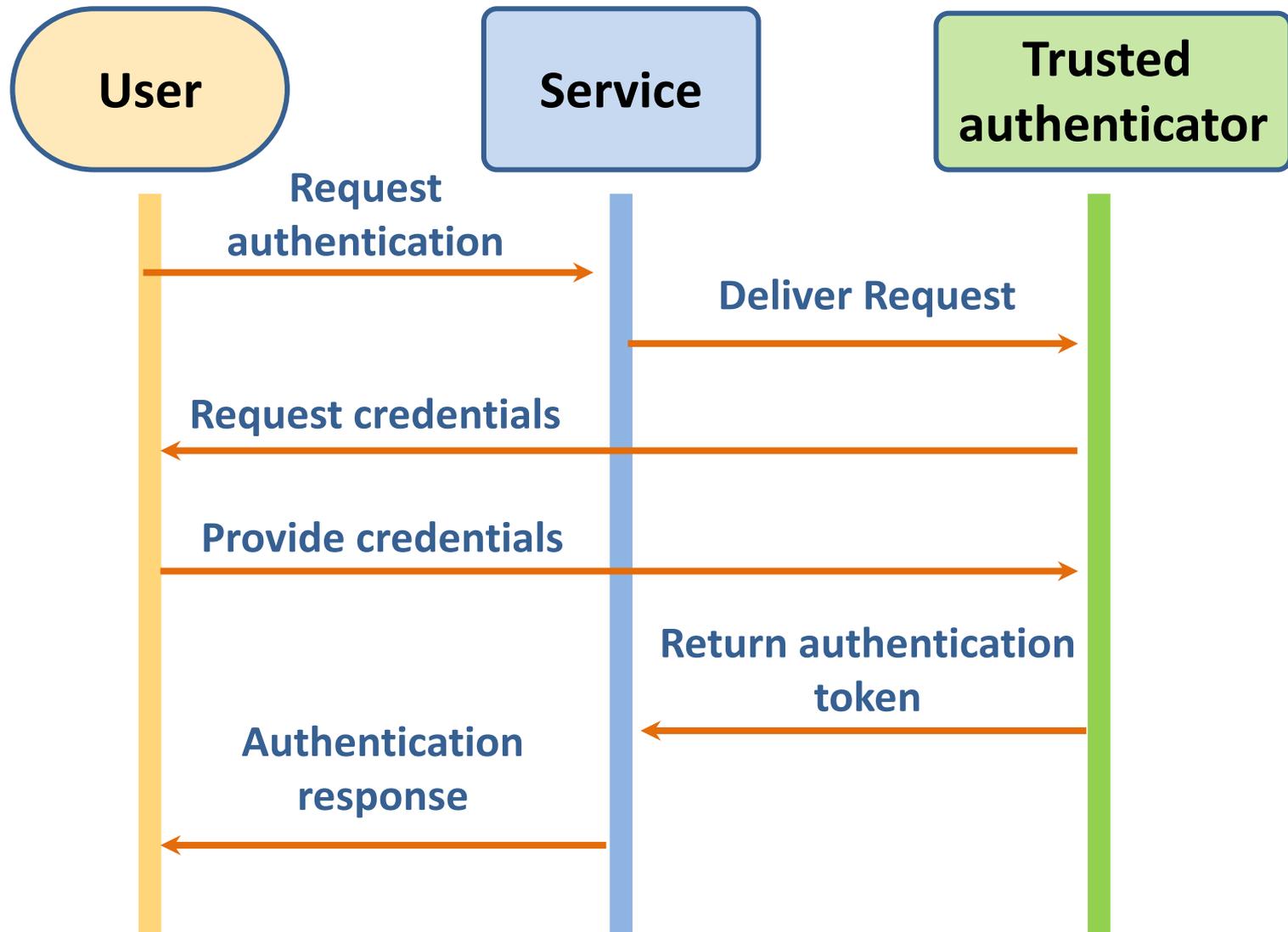
- **Forgotten passwords**

Users regularly forget their passwords so that you need to set up a password recovery mechanism to allow these to be reset.

# Federated identity

- **Federated identity** is an approach to authentication where you use an **external authentication service**.
- **‘Login with Google’** and **‘Login with Facebook’** are widely used examples of authentication using federated identity.
- The advantage of federated identity for a user is that they have a **single set of credentials** that are stored by a **trusted identity service**.
- Instead of logging into a service directly, a user provides their credentials to a known service who confirms their identity to the authenticating service.
- They don't have to keep track of different user ids and passwords.

# Federated identity



# Authorization

- **Authentication** involves a user proving their identity to a software system.
- **Authorization** is a complementary process in which that identity is used to control access to software system resources.
  - For example, if you use a shared folder on Dropbox, the folder's owner may authorize you to read the contents of that folder, but not to add new files or overwrite files in the folder.
- When a business wants to **define** the **type of access** that users get to resources, this is based on an **access control policy**.
  - This **policy** is a **set of rules** that define what information (data and programs) is controlled, who has access to that information and the type of access that is allowed

# Access control policies

- **Explicit access control policies** are important for both **legal** and **technical** reasons.
- **Data protection rules** limit the access the personal data and this must be reflected in the **defined access control policy**.
  - If this policy is incomplete or does not conform to the data protection rules, then there may be subsequent legal action in the event of a data breach.
- Technically, an **access control policy** can be a **starting point** for setting up the access control scheme for a system.
- For example, if the access control policy defines the access rights of students, then when new students are registered, they all get these rights by default.

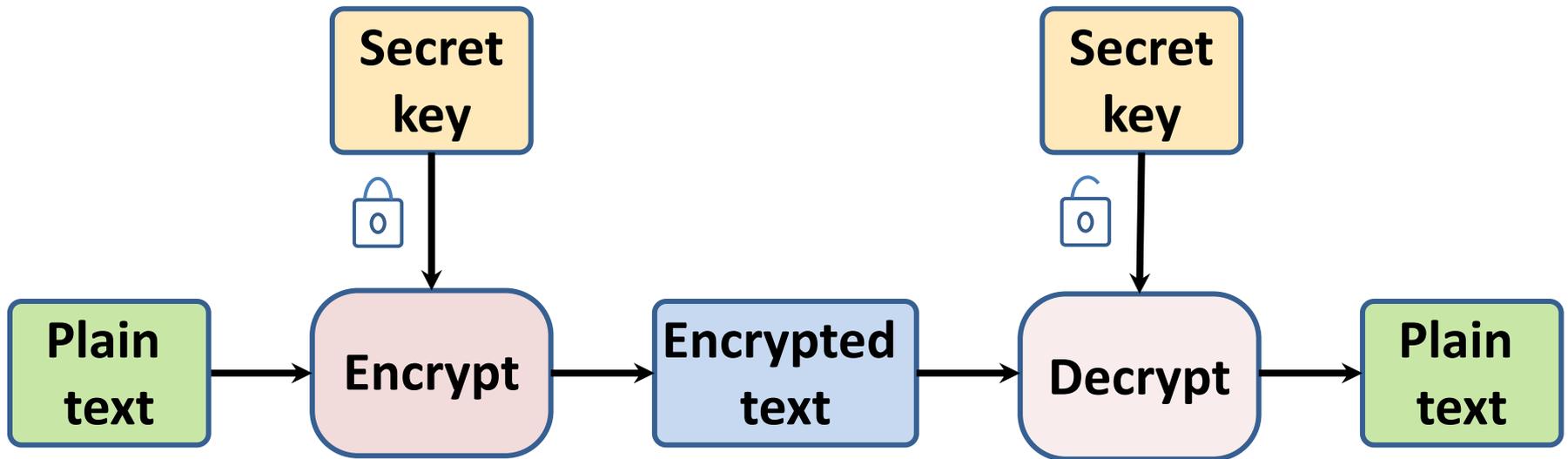
# Access Control Lists (ACL)

- **Access control lists (ACLs)** are used in most file and database systems to implement access control policies.
- **Access control lists** are **tables** that **link users with resources** and specify what those users are **permitted** to do.
- If access control lists are based on individual permissions, then these can become very large. However, you can dramatically cut their size by **allocating users to groups** and then **assigning permissions to the group**

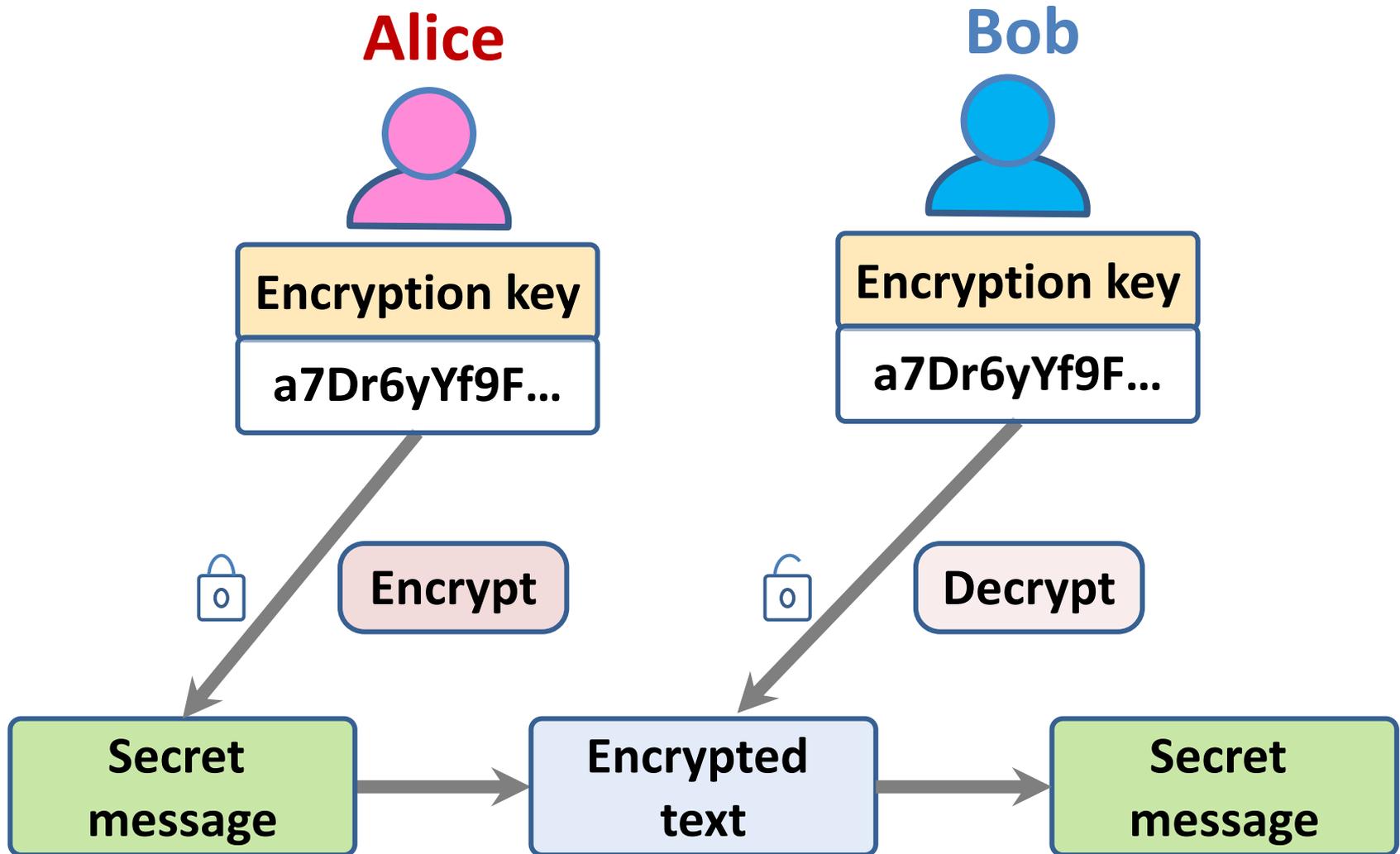
# Encryption

- **Encryption** is the process of making a document unreadable by applying an **algorithmic transformation** to it.
- A **secret key** is used by the **encryption algorithm** as the basis of this transformation. You can decode the encrypted text by applying the reverse transformation.
- Modern **encryption techniques** are such that you can **encrypt data** so that it is practically uncrackable using currently available technology.
- History has demonstrated that apparently **strong encryption may be crackable when new technology becomes available**.
- If commercial **quantum systems** become available, we will have to use a completely different approach to encryption on the Internet.

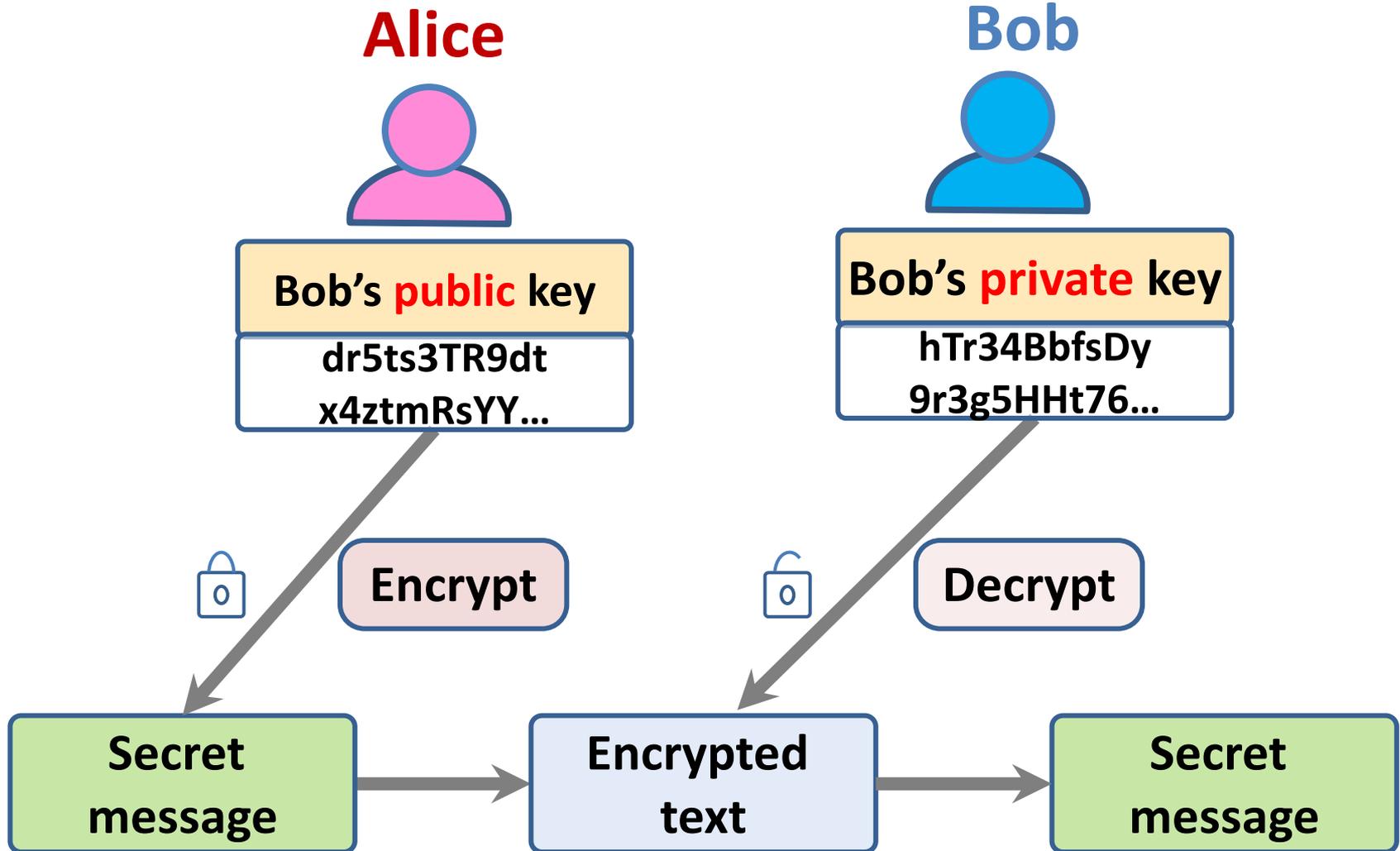
# Encryption and decryption



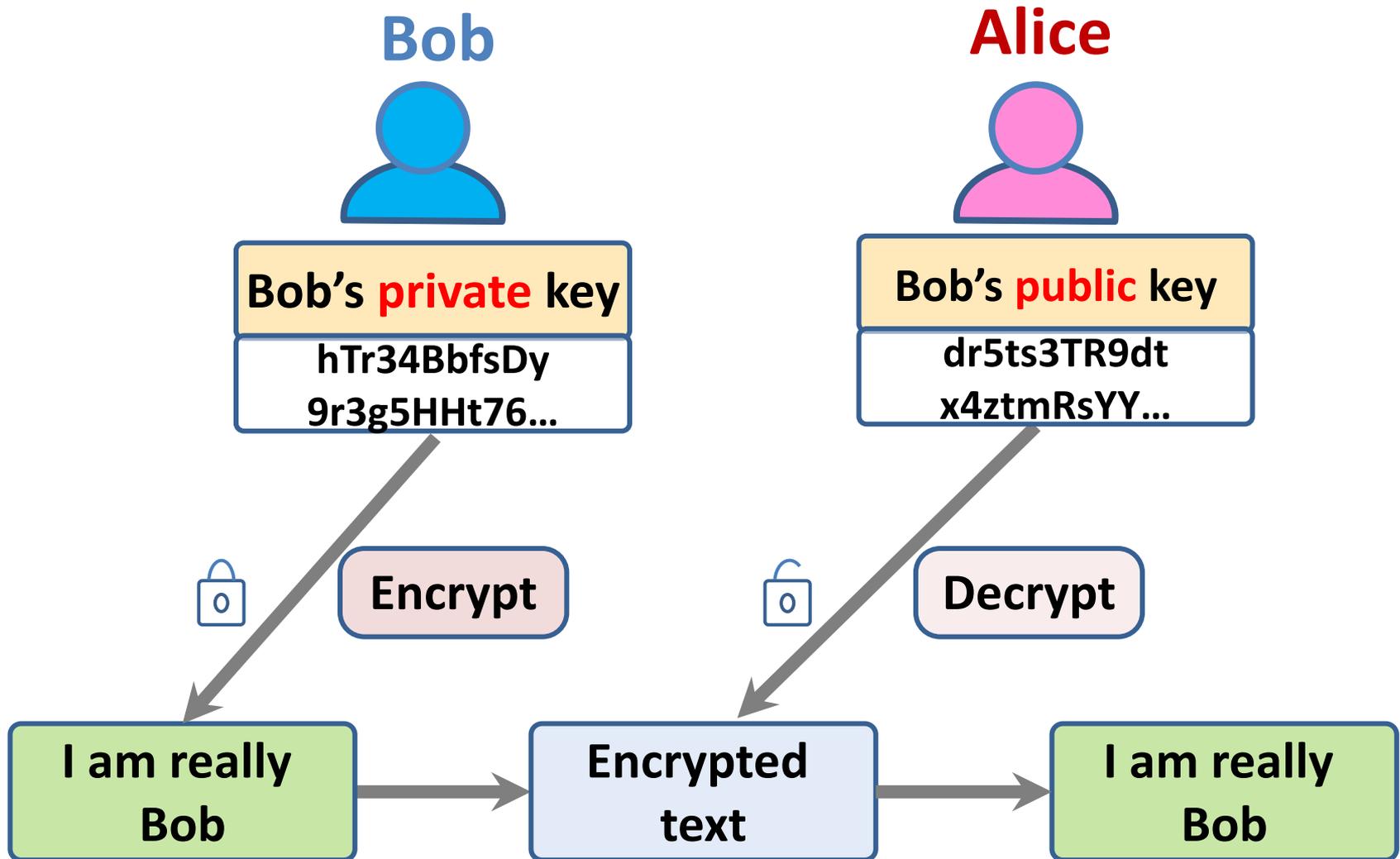
# Symmetric encryption



# Asymmetric encryption



# Encryption for authentication



# TLS and digital certificates

- The **https protocol** is a standard protocol for securely exchanging texts on the web.
- It is the standard http protocol plus an encryption layer called **TLS (Transport Layer Security)**.
- This encryption layer is used for 2 things:
  - to verify the identity of the web server;
  - to encrypt communications so that they cannot be read by an attacker who intercepts the messages between the client and the server

# TLS and digital certificates

- TLS encryption depends on a **digital certificate** that is sent from the web server to the client.
  - **Digital certificates** are issued by a **certificate authority (CA)**, which is a trusted identity verification service.
  - The **CA encrypts the information in the certificate** using their **private key** to create a unique signature. This signature is included in the certificate along with the **public key** of the CA. To check that the certificate is valid, you can decrypt the signature using the CA's public key.

# Key management

- **Key management** is the process of ensuring that encryption keys are **securely generated, stored and accessed** by **authorized users**.
- Businesses may have to manage tens of thousands of encryption keys so it is impractical to do key management manually and you need to use some kind of **automated key management system (KMS)**.

# Digital certificates

- **Subject information**

Information about the company or individual whose web site is being visited. Applicants apply for a digital certificate from a certificate authority who checks that the applicant is a valid organization.

- **Certificate authority information**

Information about the certificate authority (CA) who has issued the certificate.

- **Certificate information**

Information about the certificate itself, including a unique serial number and a validity period, defined by start and end dates.

# Digital certificates

- **Digital signature**

The combination of all of the above data uniquely identifies the digital certificate. The signature data is encrypted with the CA's private key to confirm that the data is correct. The algorithm used to generate the digital signature is also specified.

- **Public key information**

The public key of the CA is included along with the key size and the encryption algorithm used. The public key may be used to decrypt the digital signature.

# Data encryption

- As a product provider you inevitably store information about your users and, for cloud-based products, user data.
- **Encryption** can be used to reduce the damage that may occur from data theft. If information is encrypted, it is impossible, or very expensive, for thieves to access and use the unencrypted data.
  - **Data in transit**
  - **Data at rest**
  - **Data in use**

# Encryption levels

**Application**

**Database**

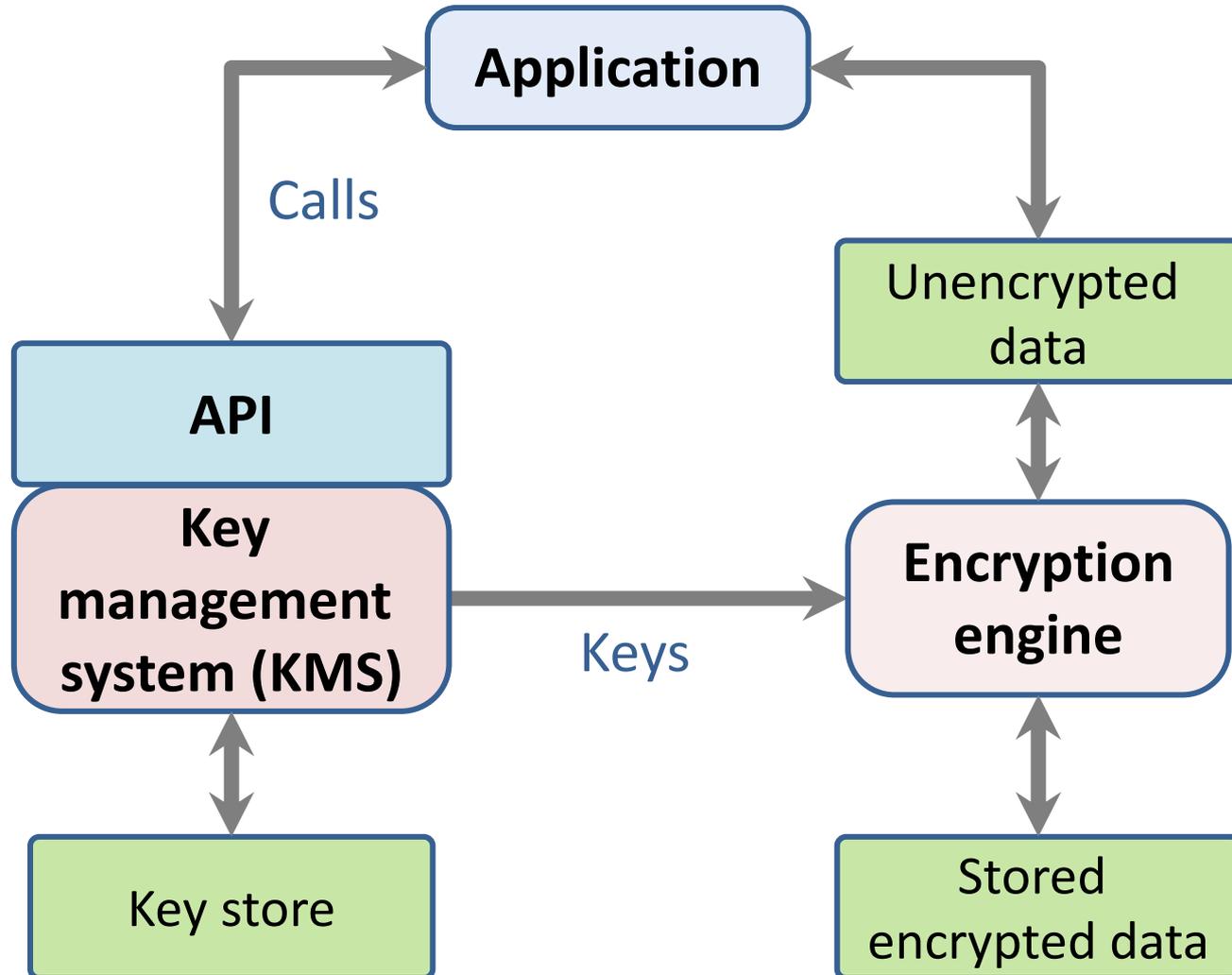
**Files**

**Media**

# Key management

- **Key management** is important because, if you get it wrong, unauthorized users may be able to access your keys and so decrypt supposedly private data. Even worse, if you lose encryption keys, then your encrypted data may be permanently inaccessible.
- A **key management system (KMS)** is a specialized database that is designed to securely store and manage encryption keys, digital certificates and other confidential information.

# Using a KMS for encryption management



# Long-term key storage

- Business may be required by **accounting** and other **regulations** to keep copies of all of their data for several years.
  - For example, in the UK, tax and company data has to be maintained for at least six years, with a longer retention period for some types of data. **Data protection regulations** may require that this data be stored securely, so the data should be encrypted.
- To reduce the risks of a security breach, **encryption keys should be changed regularly**. This means that archival data may be encrypted with a different key from the current data in your system.
- Therefore, **key management systems** must maintain multiple, timestamped versions of keys so that system backups and archives can be decrypted if required.

# Privacy

- **Privacy** is a **social concept** that relates to the collection, dissemination and appropriate use of personal information held by a third-party such as a company or a hospital.
- The importance of privacy has changed over time and individuals have their own views on what degree of privacy is important.
- **Culture and age** also affect peoples' views on what privacy means.
  - **Younger people** were early adopters of the first social networks and many of them seem to be less inhibited about **sharing personal information** on these platforms than older people.
  - In some countries, the level of **income** earned by an individual is seen as a private matter; in others, all **tax returns** are openly published.

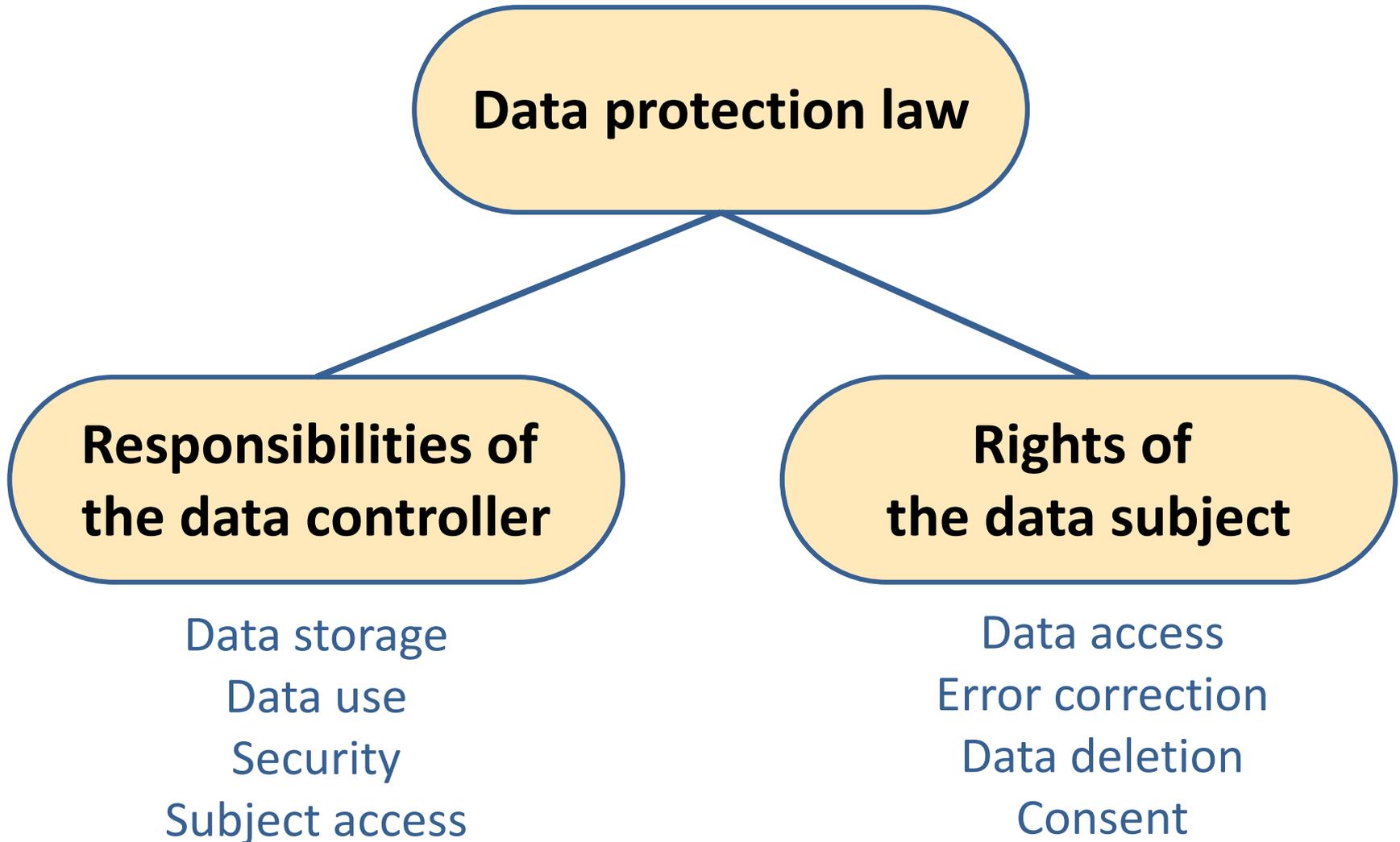
# Business reasons for privacy

- If you are offering a product directly to consumers and you fail to conform to **privacy regulations**, then you may be subject to **legal action** by product buyers or by a data regulator. If your conformance is weaker than the protection offered by data protection regulations in some countries, you won't be able to sell your product in these countries.
- If your product is a **business product**, business customers require privacy safeguards so that they are not put at **risk of privacy violations and legal action** by users.
- If personal information is **leaked** or **misused**, even if this is not seen as a **violation of privacy regulations**, this can lead to serious reputational damage. Customers may stop using your product because of this.

# Data protection laws

- In many countries, the right to **individual privacy** is protected by **data protection laws**.
- These laws limit the **collection, dissemination and use of personal data** to the purposes for which it was collected.
  - For example, a travel insurance company may collect health information so that they can assess their level of risk. This is legal and permissible.
  - However, it would not be legal for those companies to use this information to target online advertising of health products, unless their users had given specific permission for this.

# Data protection laws



# Data protection principles

- **Awareness and control**

Users of your product must be made aware of what data is collected when they are using your product, and must have control over the personal information that you collect from them.

- **Purpose**

You must tell users why data is being collected and you must not use that data for other purposes.

- **Consent**

You must always have the consent of a user before you disclose their data to other people.

- **Data lifetime**

You must not keep data for longer than you need to. If a user deletes their account, you must delete the personal data associated with that account.

# Data protection principles

- **Secure storage**

You must maintain data securely so that it cannot be tampered with or disclosed to unauthorized people.

- **Discovery and error correction**

You must allow users to find out what personal data that you store. You must provide a way for users to correct errors in their personal data.

- **Location**

You must not store data in countries where weaker data protection laws apply unless there is an explicit agreement that the stronger data protection rules will be upheld.

# Privacy policy

- You should to establish a **privacy policy** that defines how personal and sensitive information about users is **collected, stored and managed**.
- **Software products** use **data** in different ways, so your privacy policy has to define the personal data that you will **collect** and how you will **use** that data.
- Product users should be able to review your privacy policy and change their **preferences** regarding the information that you store.

# Privacy policy

- Your **privacy policy** is a **legal document** and it should be auditable to check that it is consistent with the **data protection laws** in countries where your software is sold.
- Privacy policies should not be expressed to users in a long '**terms and conditions**' document that, in practice, nobody reads.
- The **General Data Protection Regulation (GDPR)** now require software companies to include a **summary of their privacy policy**, written in **plain language** rather than legal jargon, on their **website**.

# Summary

- **Security** is a technical concept that relates to a software system's ability to **protect itself from malicious attacks** that may threaten its **availability**, the **integrity** of the system and/or its data, and the theft of **confidential** information.
- **Common types of attack on software products** include
  - **injection attacks,**
  - **cross-site scripting attacks,**
  - **session hijacking attacks,**
  - **denial of service attacks and**
  - **brute force attacks.**

# Summary

- **Authentication** may be based on something a user knows, something a user has, or some physical attribute of the user.
- **Federated authentication** involves devolving responsibility for authentication to a third-party such as Facebook or Google, or to a business's authentication service.
- **Authorization** involves controlling access to system resources based on the user's authenticated identity. Access control lists are the most commonly-used mechanism to implement authorization.

# Summary

- **Symmetric encryption** involves encrypting and decrypting information with the **same secret key**.
- **Asymmetric encryption** uses a **key pair** – a **private key** and a **public key**. Information encrypted using the public key can only be decrypted using the private key.
- A major issue in symmetric encryption is **key exchange**.
- The **Transport Layer Security (TLS) protocol**, which is used to secure web traffic, gets around this problem by using **asymmetric encryption** for **transferring information** used to generate a **shared key**.

# Summary

- If your product stores **sensitive user data**, you should **encrypt that data** when it is not in use.
- A **key management system (KMS)** stores encryption keys. Using a KMS is essential because a business may have to manage thousands or even millions of keys and may have to decrypt historic data that was encrypted using an obsolete encryption key.

# Summary

- **Privacy** is a **social concept** that relates to **how people feel** about the **release of their personal information to others**. Different countries and cultures have different ideas on what information should and should not be private.
- **Data protection laws** have been made in many countries to protect individual privacy. They require companies who manage user data to store it securely, to ensure that it is not used or sold without the permission of users, and to allow users to view and correct personal data held by the system.

# Reliable Programming

# Outline

- **Software quality**
- **Programming for reliability**
- **Design pattern**
- **Refactoring**

# Software quality

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

# Software product quality attributes



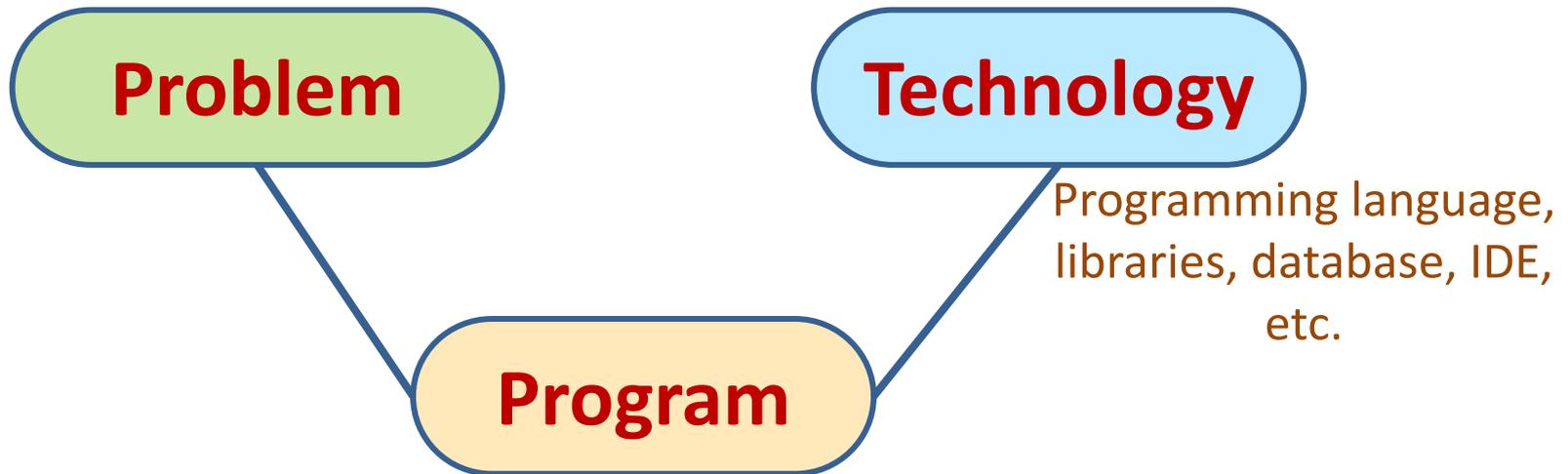
# Programming for reliability

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

# Underlying causes of program errors

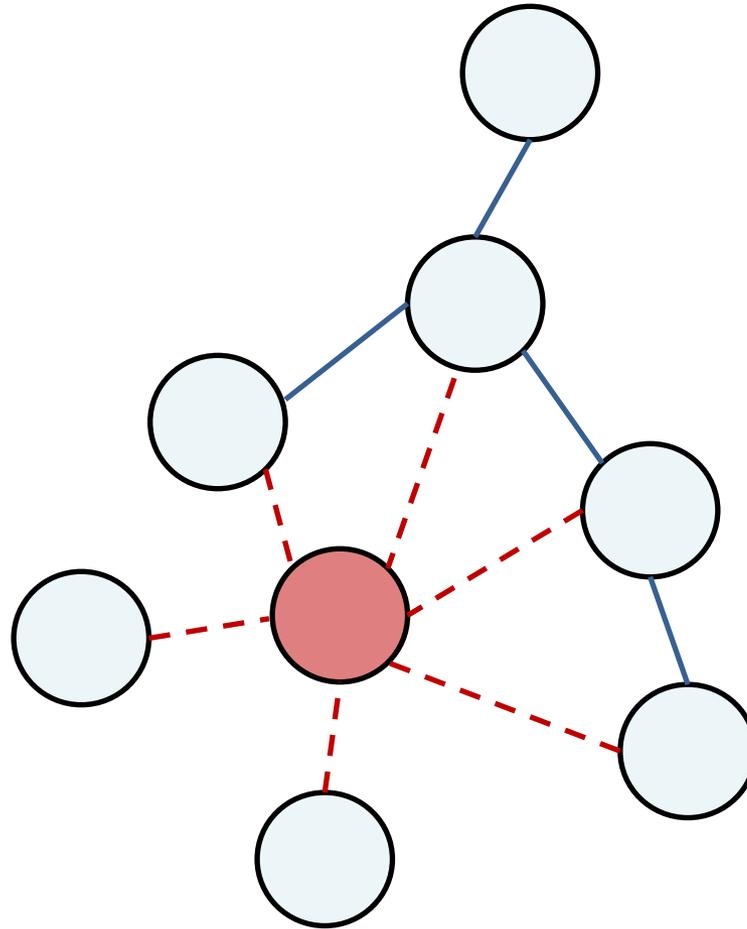
Programmers make mistakes because they don't properly understand the problem or the application domain

Programmers make mistakes because they use unsuitable technology or they don't properly understand the technologies used



Programmers make mistakes because they make simple slips or they do not completely understand how multiple program components work together the program's state.

# Software complexity



The shaded node interacts, in some ways, with the linked nodes shown by the dotted line

# Program complexity

- **Complexity** is related to the number of relationships between elements in a program and the type and nature of these relationships
- The number of relationships between entities is called the **coupling**. The higher the coupling, the more complex the system.
  - The shaded node has a relatively **high coupling** because it has relationships with six other nodes.

# Software complexity

- A **static relationship** is one that is stable and does not depend on program execution.
  - Whether or not one component is **part** of another component is a static relationship.
- **Dynamic relationships**, which change over time, are more complex than static relationships.
  - An example of a dynamic relationship is the **'calls'** relationship between functions.

# Types of complexity

## Reading complexity

This reflects how hard it is to read and understand the program.

## Structural complexity

This reflects the number and types of relationship between the structures (classes, objects, methods or functions) in your program.

## Data complexity

This reflects the representations of data used and relationships between the data elements in your program.

## Decision complexity

This reflects the complexity of the decisions in your program

# Complexity reduction guidelines

## Structural complexity

- Functions should do one thing and one thing only
- Functions should never have side-effects
- Every class should have a single responsibility
- Minimize the depth of inheritance hierarchies
- Avoid multiple inheritance
- Avoid threads (parallelism) unless absolutely necessary

# Complexity reduction guidelines

## Data complexity

- Define interfaces for all abstractions
- Define abstract data types
- Avoid using floating-point numbers
- Never use data aliases

# Complexity reduction guidelines

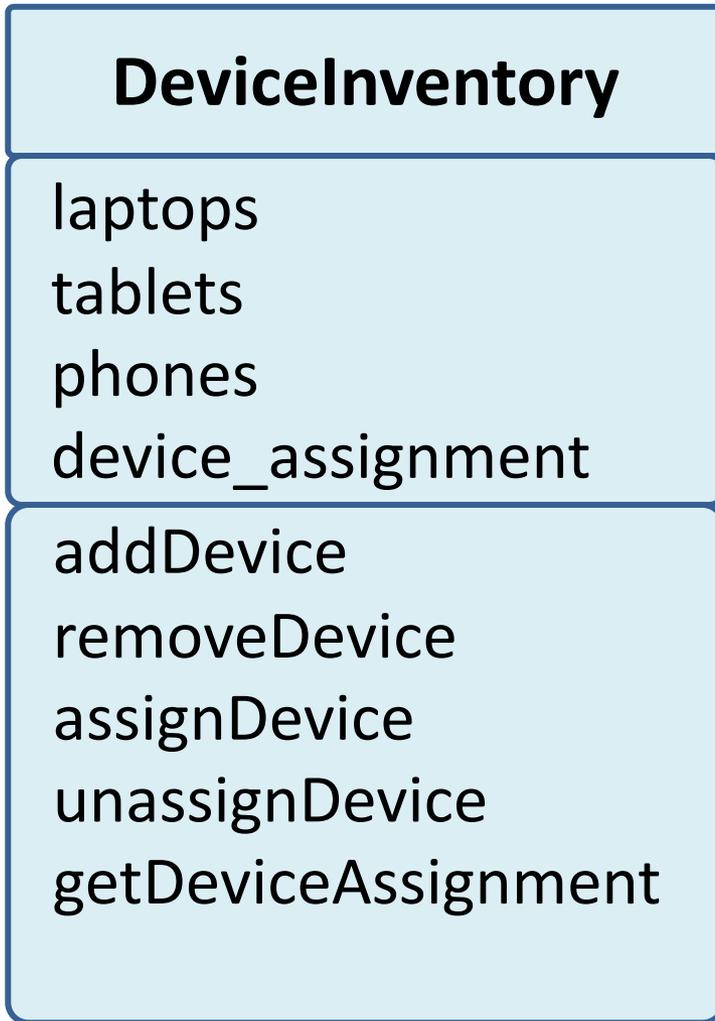
## Conditional complexity

- Avoid deeply nested conditional statements
- Avoid complex conditional expressions

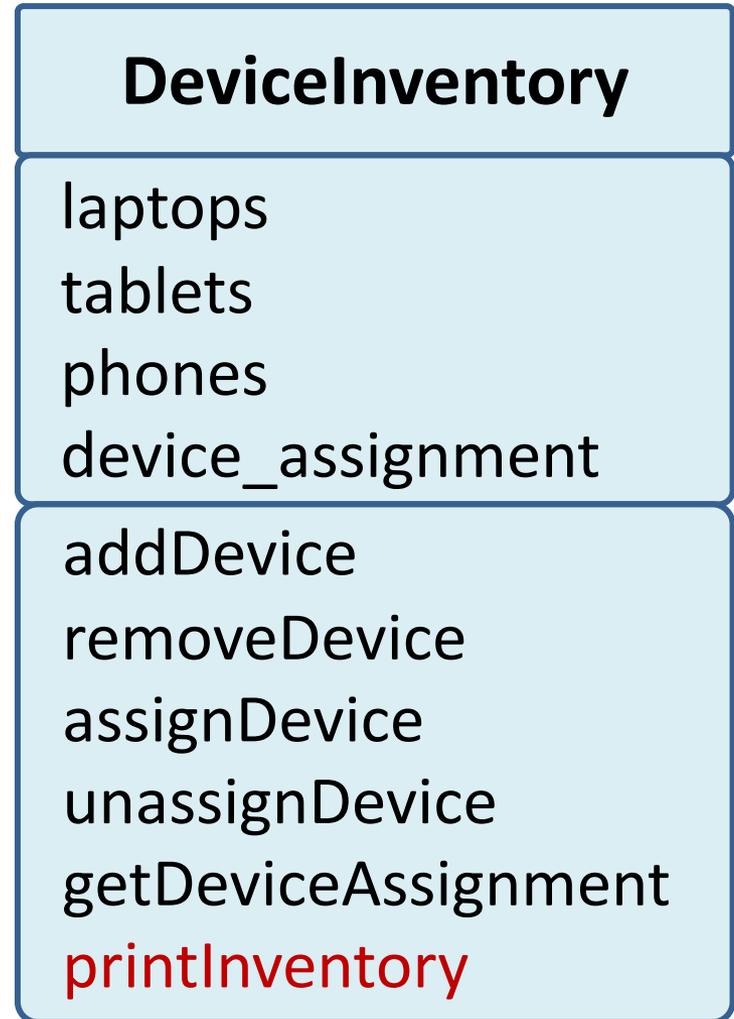
# Ensure that every class has a single responsibility

- You should design classes so that there is only **a single reason to change** a class.
  - If you adopt this approach, your classes will be smaller and more cohesive.
  - They will therefore be less complex and easier to understand and change.
- The **single responsibility principle**
  - Gather together the things that change for the same reasons.
  - Separate those things that change for different reasons

# The DeviceInventory class



(a)

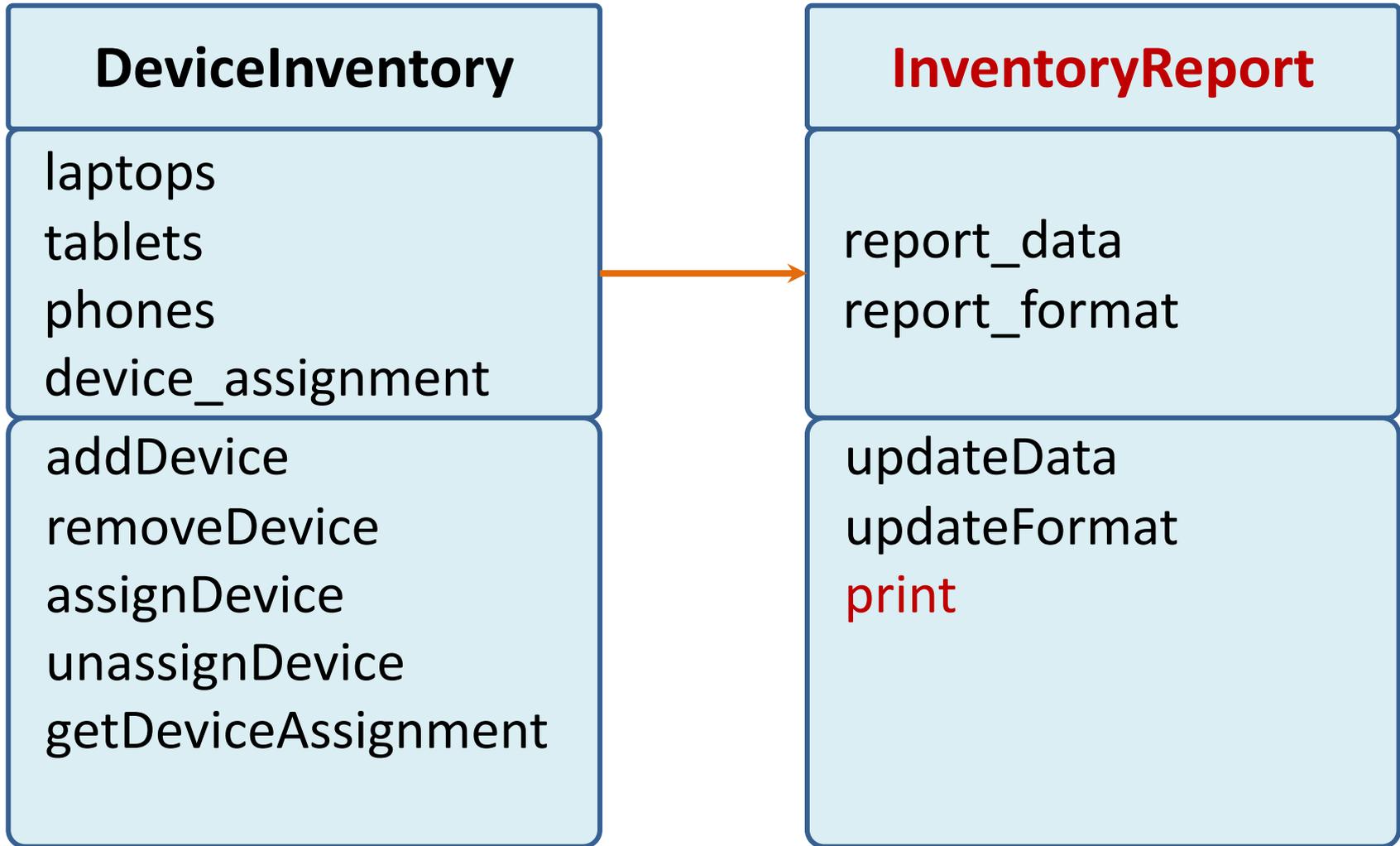


(b)

# Adding a printInventory method

- One way of making this change is to **add a printInventory method**
  - This change **breaks the single responsibility principle** as it then adds an additional ‘reason to change’ the class.
- Instead of adding a printInventory method to DeviceInventory, it is better to **add a new class** to represent the printed report.

# The DeviceInventory and InventoryReport classes



# Avoid deeply nested conditional statements

- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- For example, the function ‘agecheck’ is a short Python function that is used to calculate an age multiplier for insurance premiums.
  - The insurance company’s data suggests that the age and experience of drivers affects the chances of them having an accident, so premiums are adjusted to take this into account.
  - It is good practice to name constants rather than using absolute numbers, so Program names all constants that are used.

## Deeply nested if-then-else statements

```
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def agecheck(age, experience):
    # Assigns a premium multiplier depending on the age and experience of the driver
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER *
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

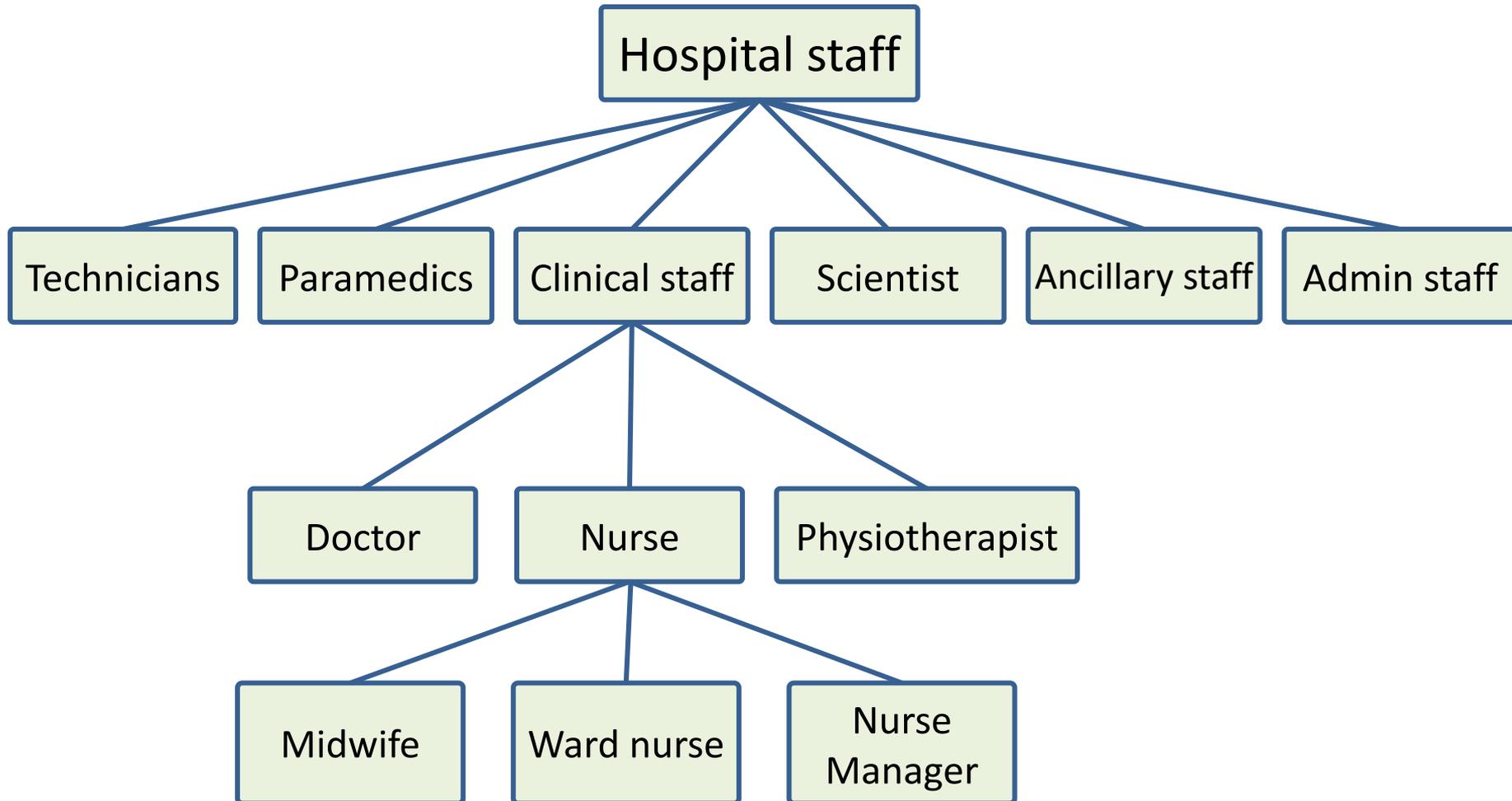
# Using guards to make a selection

```
def agecheck_with_guards(age, experience):  
  
    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <=  
YOUNG_DRIVER_EXPERIENCE:  
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER *  
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER  
    if age <= YOUNG_DRIVER_AGE_LIMIT:  
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER  
    if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE) and experience  
<= OLDER_DRIVER_EXPERIENCE:  
        return OLDER_DRIVER_PREMIUM_MULTIPLIER  
    if age > ELDERLY_DRIVER_AGE:  
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER  
    return NO_MULTIPLIER
```

# Avoid deep inheritance hierarchies

- **Inheritance** allows the attributes and methods of a **class**, such as RoadVehicle, can be inherited by **sub-classes**, such as Truck, Car and MotorBike.
- Inheritance appears to be an effective and efficient way of **reusing code** and of making changes that affect all subclasses.
- However, **inheritance increases the structural complexity** of code as it increases the coupling of subclasses.
- The problem with deep inheritance is that if you want to make changes to a class, you have to look at all of its superclasses to see where it is best to make the change.
- You also have to look at all of the related subclasses to check that the change does not have unwanted consequences. It's easy to make mistakes when you are doing this analysis and introduce faults into your program.

# Part of the inheritance hierarchy for hospital staff



# Design pattern definition

- Definition
  - **A general reusable solution to a commonly-occurring problem within a given context in software design.**

# Design pattern

- **Design patterns** are **object-oriented** and describe solutions in terms of **objects** and **classes**.
- They are not off-the-shelf solutions that can be directly expressed as code in an object-oriented language.
- They describe the **structure of a problem solution** but have to be adapted to suit your application and the programming language that you are using.

# Programming principles

- **Separation of concerns**
  - This means that each abstraction in the program (class, method, etc.) should address a separate concern and that all aspects of that concern should be covered there.
- **Separate the ‘what’ from the ‘how’**
  - If a program component provides a particular service, you should make available only the information that is required to use that service (the ‘what’). The implementation of the service (‘the how’) should be of no interest to service users.

# Common types of design patterns

- **Creational patterns**

- These are concerned with class and object creation. They define ways of instantiating and initializing objects and classes that are more abstract than the basic class and object creation mechanisms defined in a programming language.

- **Structural patterns**

- These are concerned with class and object composition. Structural design patterns are a description of how classes and objects may be combined to create larger structures.

- **Behavioural patterns**

- These are concerned with class and object communication. They show how objects interact by exchanging messages, the activities in a process and how these are distributed amongst the participating objects.

# Pattern description

- **Design patterns** are usually documented in the stylized way. This includes:
  - a meaningful name for the pattern and a brief description of what it does;
  - a description of the problem it solves;
  - a description of the solution and its implementation;
  - the consequences and trade-offs of using the pattern and other issues that you should consider.

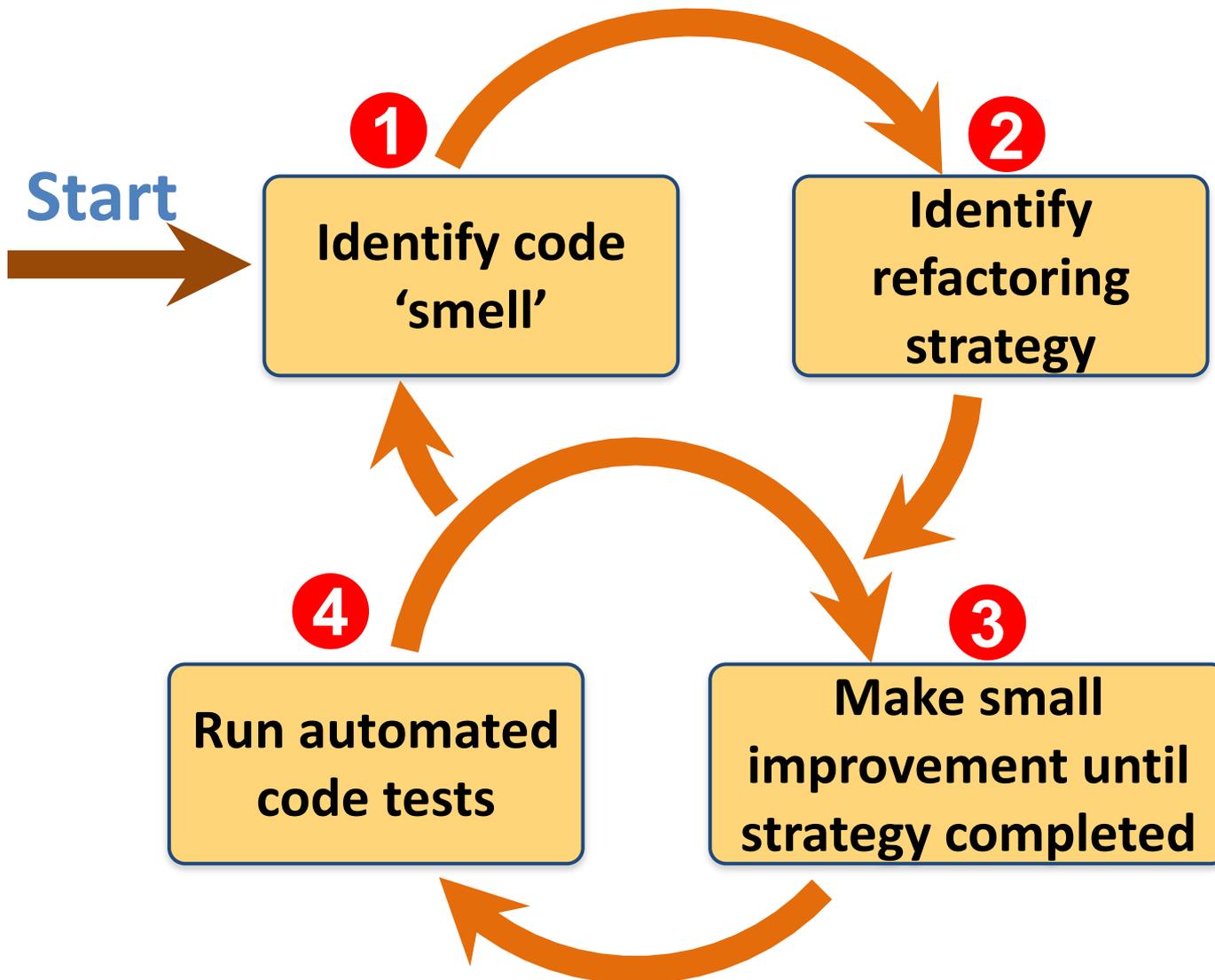
# Refactoring

- **Refactoring** means **changing a program to reduce its complexity** without changing the external behaviour of that program.
- **Refactoring** makes a program more **readable** (so reducing the ‘reading complexity’) and more **understandable**.
- It also makes it **easier to change**, which means that you reduce the chances of making mistakes when you introduce new features.

# Refactoring

- The reality of programming is that as you make **changes and additions to existing code**, you inevitably increase its **complexity**.
  - The code becomes harder to understand and change.
  - The abstractions and operations that you started with become more and more complex because you modify them in ways that you did not originally anticipate.

# A refactoring process



# Code smells

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

# Examples of code smells

- **Large classes**

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

- **Long methods/functions**

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

# Examples of code smells

- **Duplicated code**

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

- **Meaningless names**

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

# Examples of code smells

- **Unused code**

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

# Examples of refactoring for complexity reduction

- **Reading complexity**

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

- **Structural complexity**

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

# Examples of refactoring for complexity reduction

- **Data complexity**

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

- **Decision complexity**

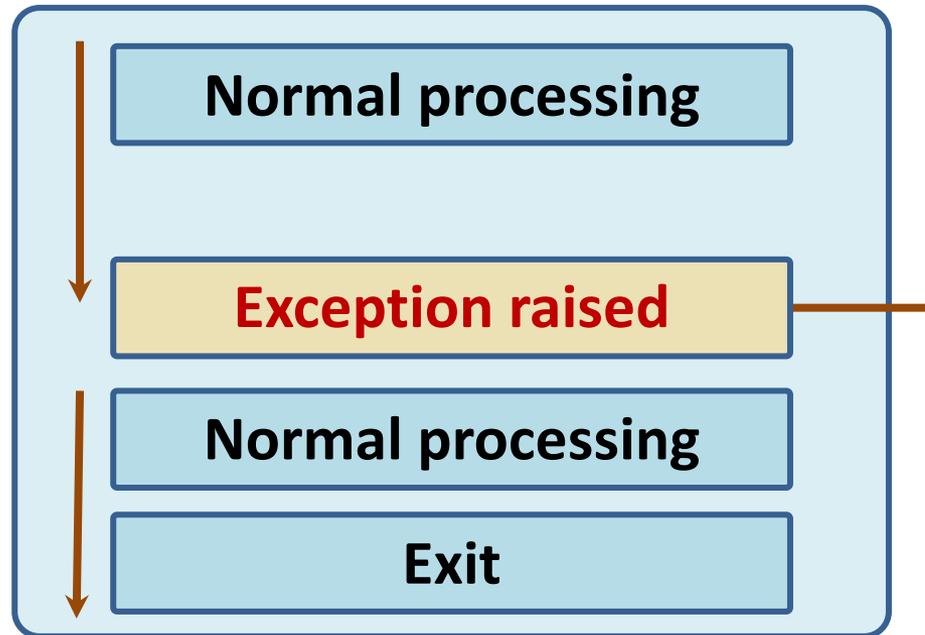
You can replace a series of deeply nested if-then-else statements with guard clauses.

# Exception handling

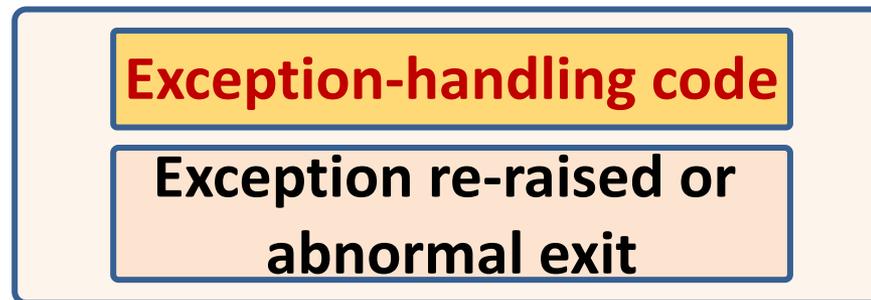
- Exceptions are events that disrupt the normal flow of processing in a program.
- When an exception occurs, control is automatically transferred to exception management code.
- Most modern programming languages include a mechanism for exception handling.
- In Python, you use `**try-except**` keywords to indicate exception handling code; in Java, the equivalent keywords are `**try-catch.**`

# Exception handling

**Executing code**



**Exception-handling  
block**



# Python

**try: except: finally:**

**try:**

```
f = open("file1.txt")  
f.write("Hello World")
```

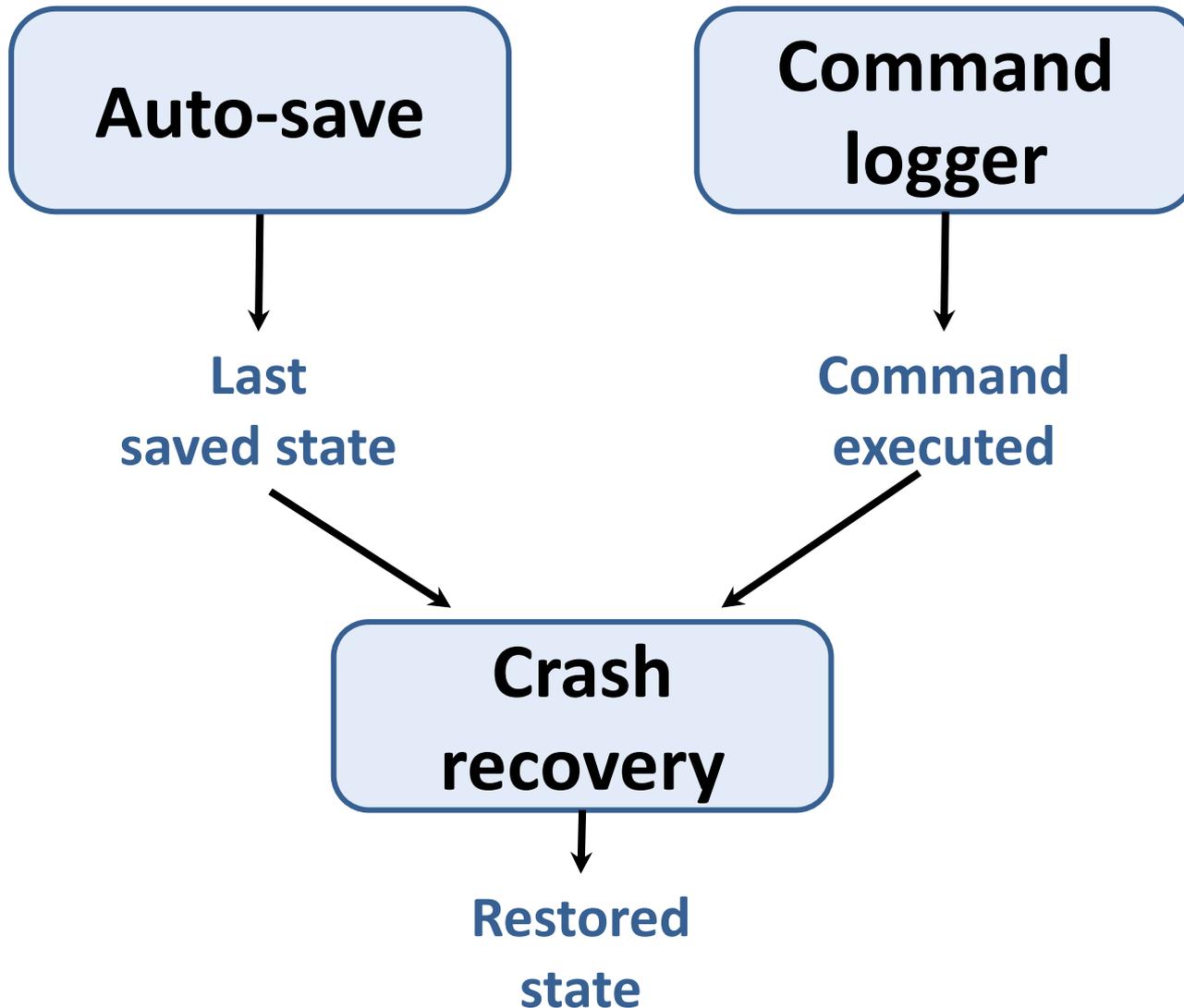
**except:**

```
print("writing file error!")
```

**finally:**

```
f.close()
```

# Auto-save and activity logging



# Summary

- The most important **quality** attributes for most software products are **reliability, security, availability, usability, responsiveness and maintainability**.
- To avoid introducing faults into your program, you should use **programming practices** that reduce the probability that you will make mistakes.
- You should always aim to **minimize complexity** in your programs. **Complexity** makes programs **harder to understand**. It increases the chances of programmer **errors** and makes the program more **difficult to change**.

# Summary

- **Design patterns** are tried and tested solutions to commonly occurring problems. Using patterns is an effective way of reducing program complexity.
- **Refactoring** is the process of reducing the complexity of an existing program without changing its functionality. It is good practice to refactor your program regularly to make it easier to read and understand.
- **Input validation** involves checking all user inputs to ensure that they are in the format that is expected by your program. Input validation helps avoid the introduction of malicious code into your system and traps user errors that can pollute your database.

# Summary

- **Regular expressions** are a way of defining patterns that can match a range of possible input strings. Regular expression matching is a compact and fast way of checking that an input string conforms to the rules you have defined.
- You should check that **numbers** have **sensible values** depending on the type of input expected. You should also check number sequences for feasibility.
- You should assume that your program may **fail** and to manage these failures so that they have minimal impact on the user.

# Summary

- **Exception management** is supported in most modern programming languages. Control is transferred to your own **exception handler** to **deal with the failure** when a program exception is detected.
- You should **log user updates** and **maintain user data snapshots** as your program executes. In the event of a failure, you can use these to recover the work that the user has done. You should also include ways of recognizing and recovering from external service failures.

# References

- Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.
- Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.
- Titus Winters, Tom Manshreck, and Hyrum Wright (2020), Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media.
- Project Management Institute (2021), A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Seventh Edition and The Standard for Project Management, PMI
- Project Management Institute (2017), A Guide to the Project Management Body of Knowledge (PMBOK Guide), Sixth Edition, Project Management Institute
- Project Management Institute (2017), Agile Practice Guide, Project Management Institute