Software Engineering

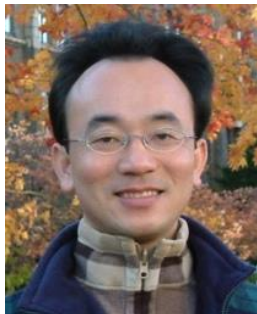## Security and Privacy; Reliable Programming; Testing: Test-driven development, and Code reviews; DevOps and Code Management: DevOps automation

Min-Yuh Day, Ph.D,
Professor

Institute of Information Management, **National Taipei University**

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

https://meet.google.com/
ish-gzmy-pmo

2025-05-21

Syllabus

Week   Date   Subject/Topics

1 2025/02/19 Introduction to Software Engineering

2 2025/02/26 Software Products and Project Management:
        Software product management and prototyping with
        Generative AI

3 2025/03/05 Agile Software Engineering:
        Agile methods, Scrum, and Extreme Programming

4 2025/03/12 Case Study on Software Engineering I

5 2025/03/19 Features, Scenarios, and Stories

6 2025/03/26 Software Architecture:
        Architectural design, System decomposition, and
        Distribution architecture

Syllabus

Week   Date   Subject/Topics

7 2025/04/02 Make-up holiday for NTPU Sports Day (No Classes)

8 2025/04/09 Midterm Project Report

9 2025/04/16 Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service

10 2025/04/23 Cloud Computing and Cloud Software Architecture

11 2025/04/30 Case Study on Software Engineering II

12 2025/05/07 Microservices Architecture, RESTful services, Service deployment

Syllabus

Week   Date   Subject/Topics

13 2025/05/14 Industry Practices of Software Engineering
[Invited Talk: Agentic AI for Agile Software Development and
Sustainable Digital Transformation,
Invited Speaker: Shihyu (Alex) Chu, Industry Consultant / Section Manager,
Software Industry Research Center, Market Intelligence & Consulting Institute (MIC)]

14 2025/05/21 Security and Privacy; Reliable Programming;
Testing: Functional testing, Test automation,
Test-driven development, and Code reviews;
DevOps and Code Management:
Code management and DevOps automation

15 2025/05/28 Final Project Report I
16 2025/06/04 Final Project Report II

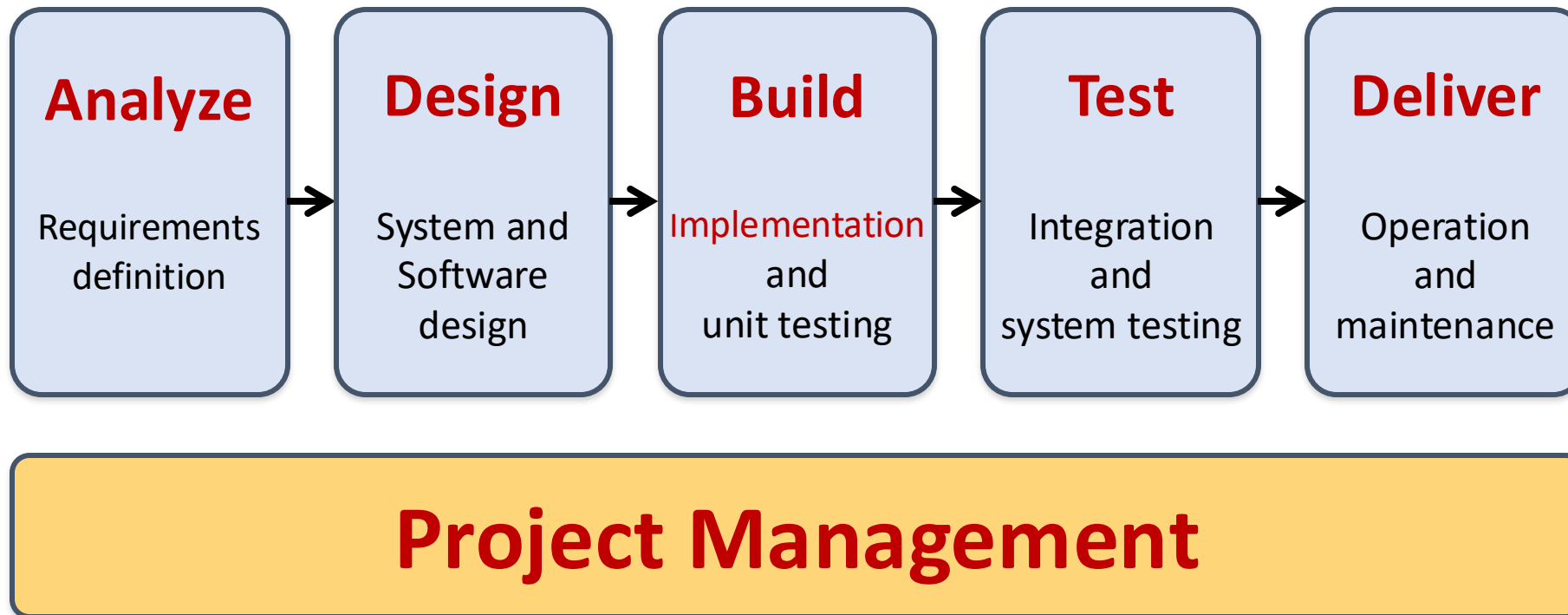# Security and Privacy;

Reliable Programming;

Testing:
Test-driven development, and Code reviews;

DevOps and Code Management:
DevOps automation

Software Engineering
and
Project Management

Analyze	Design	Build	Test	Deliver
Requirements definition	System and Software design	Implementation and unit testing	Integration and system testing	Operation and maintenance

Project Management

Information Management (MIS)
Information Systems

Source: Kenneth C. Laudon & Jane P. Laudon (2014), Management Information Systems: Managing the Digital Firm, Thirteenth Edition, Pearson.
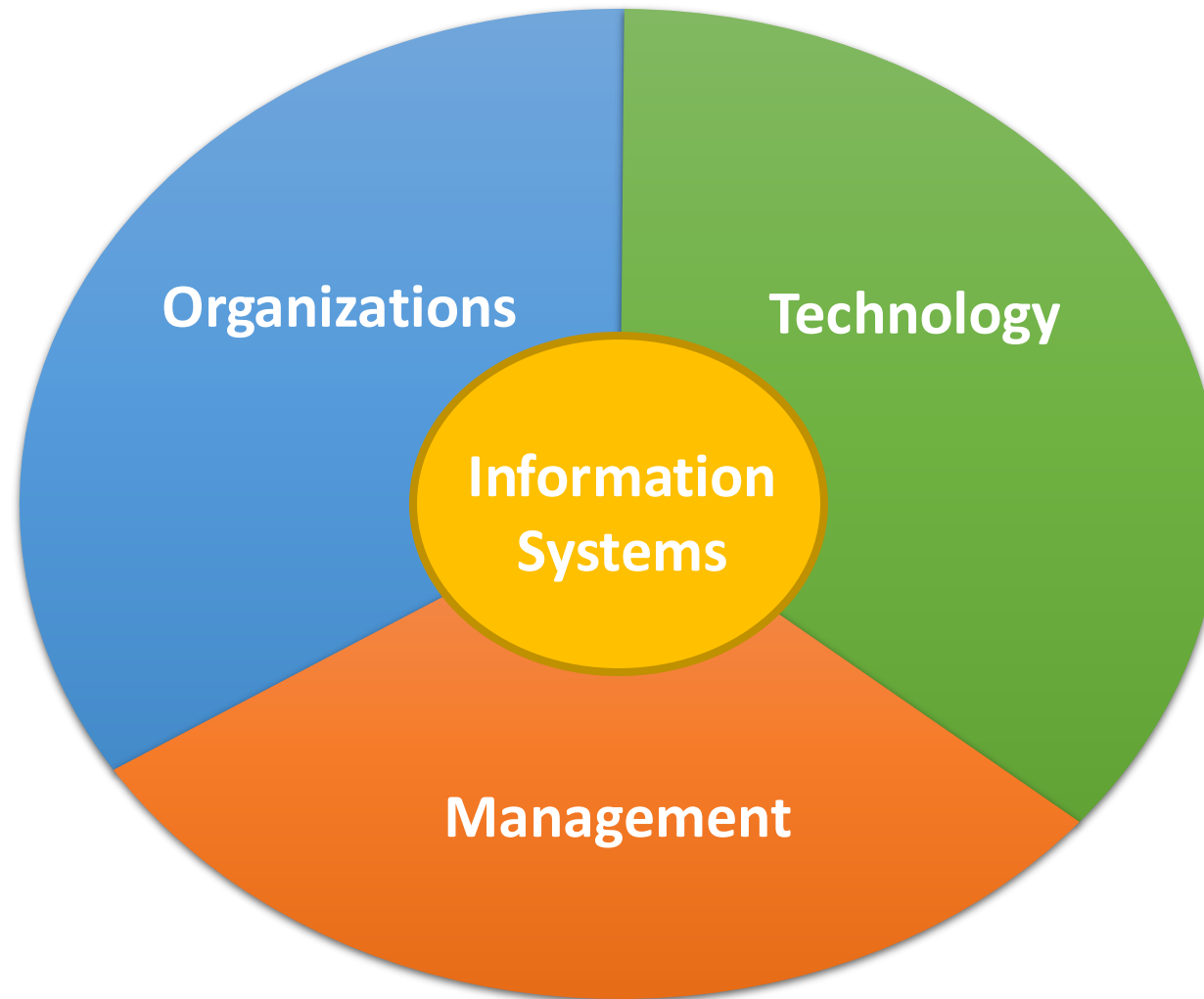
Fundamental MIS Concepts

Business Challenges

Management

Organization

Technology

Information System

Business Solutions
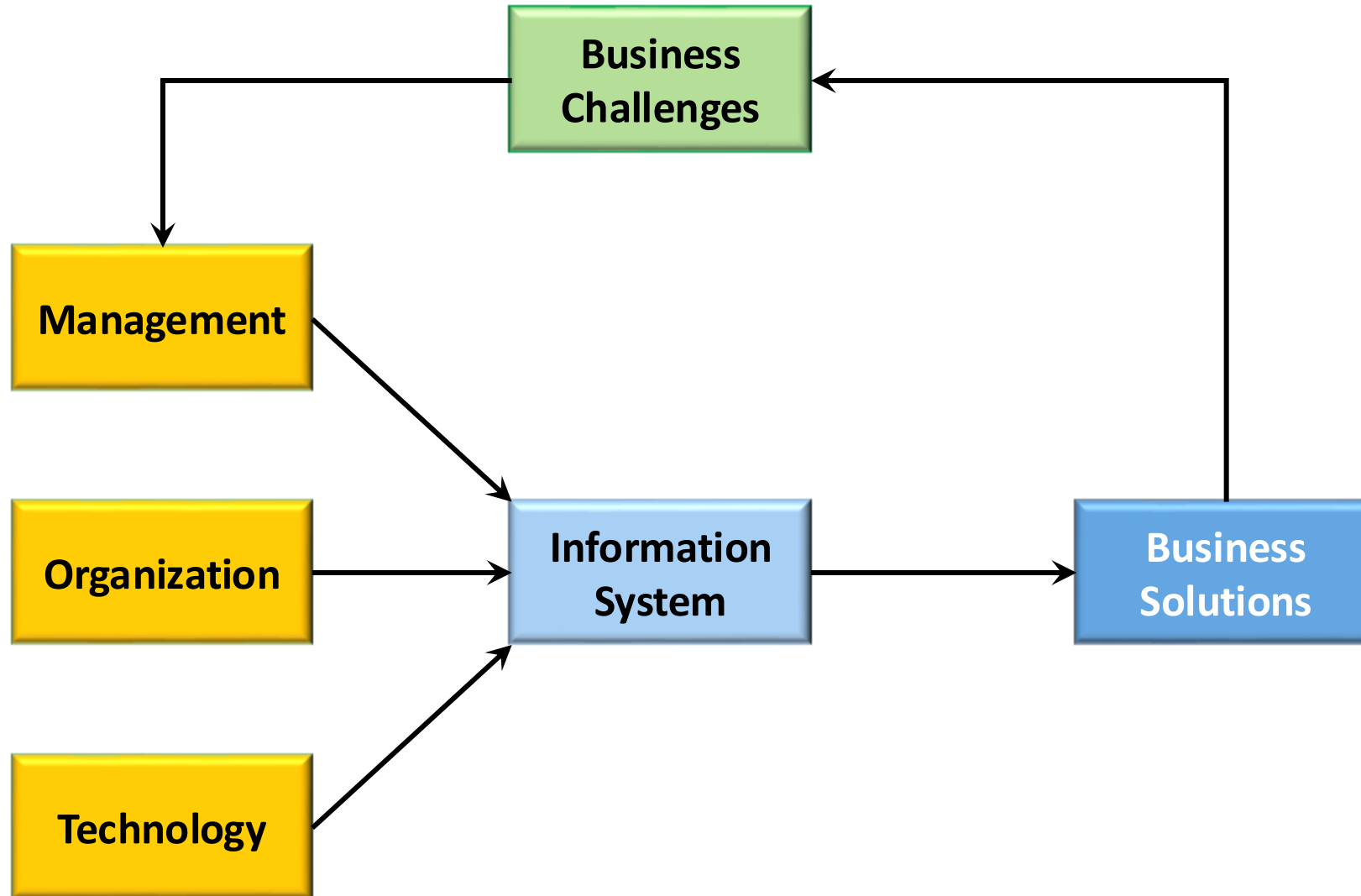
Project-based software engineering
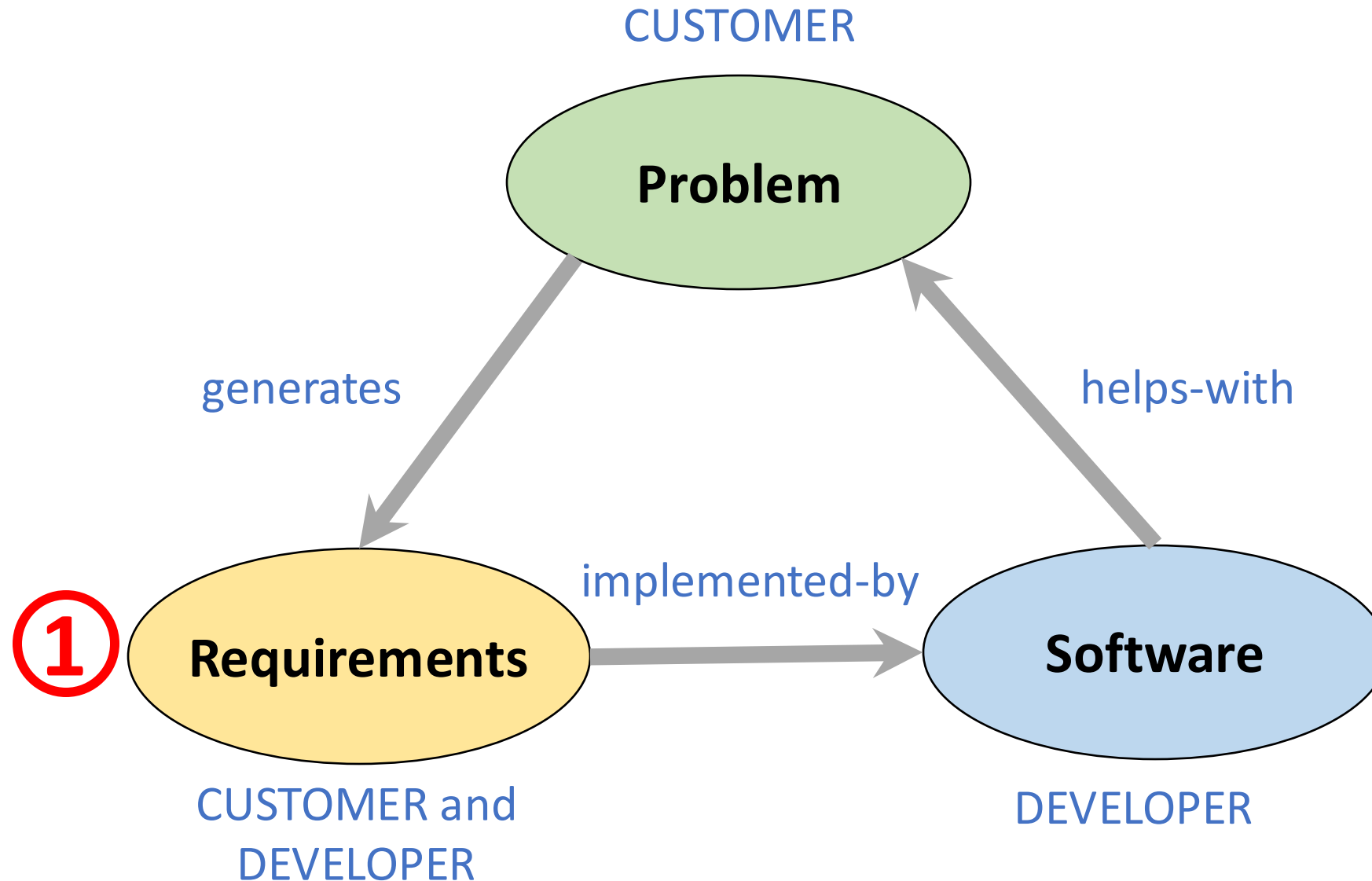
Product software engineering

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

10
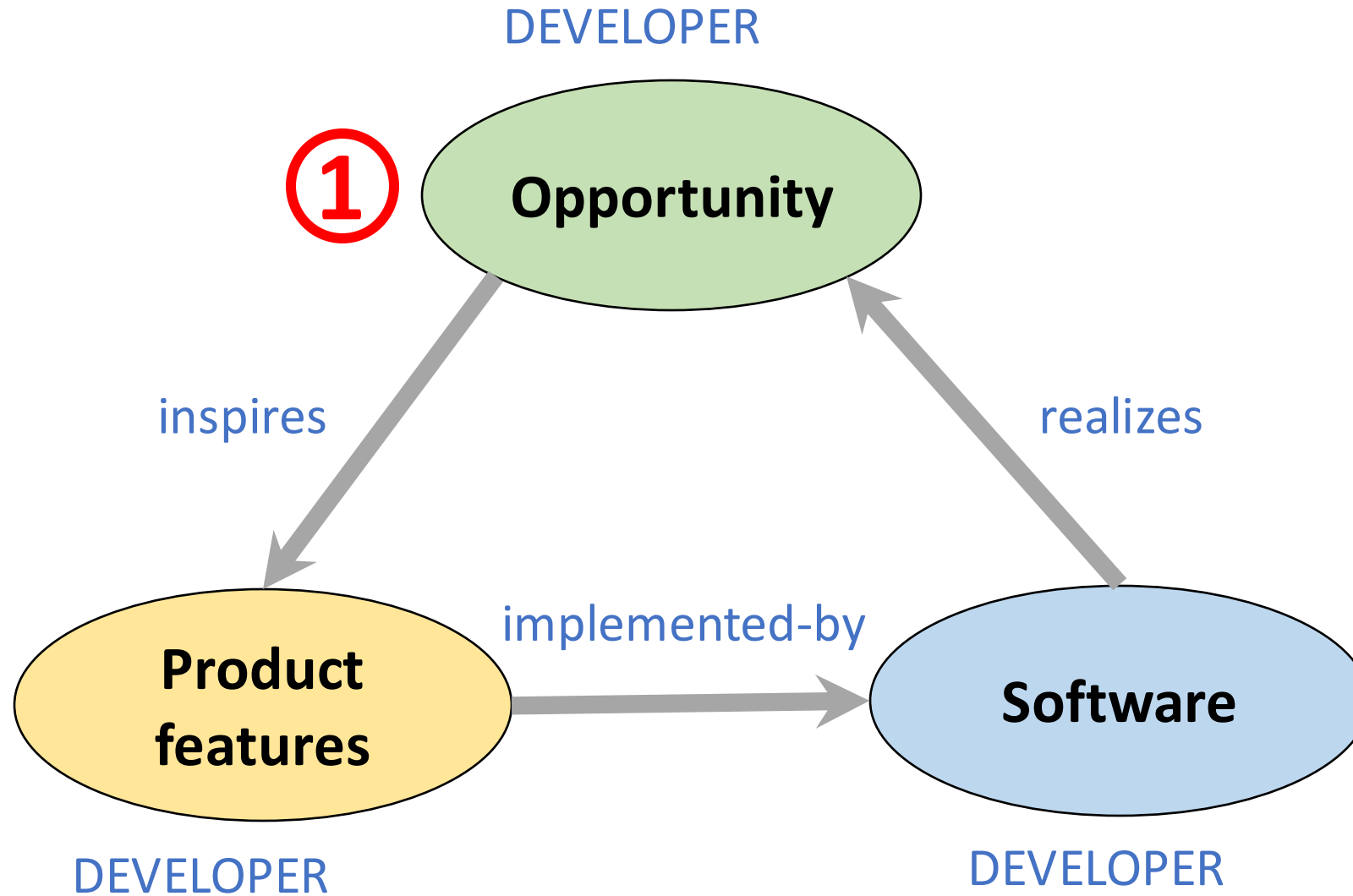
Software execution models

Stand-alone execution

User's computer

User interface
Product functionality
User data

Product updates

Vendor's servers

Hybrid execution

User's computer

User interface
Partial functionality
User data

Additional functionality
User data backups
Product updates

Vendor's servers

Software as a service

User's computer

User interface
(browser or app)

Product functionality
User data

Vendor's servers
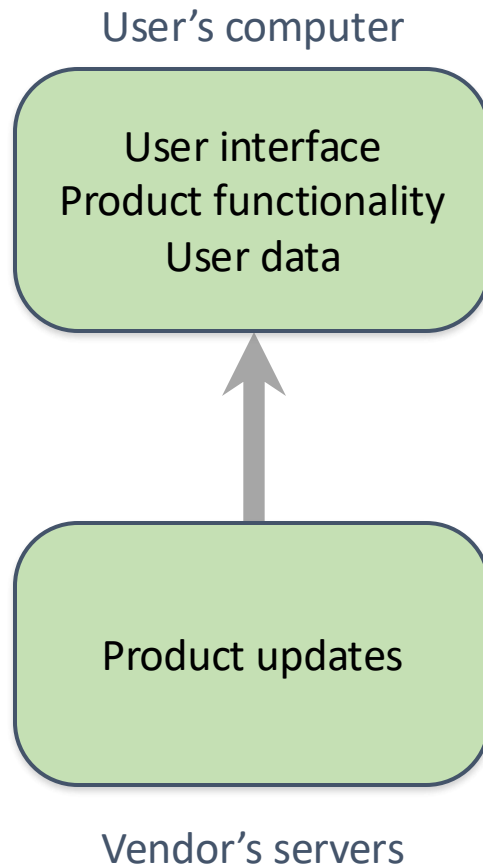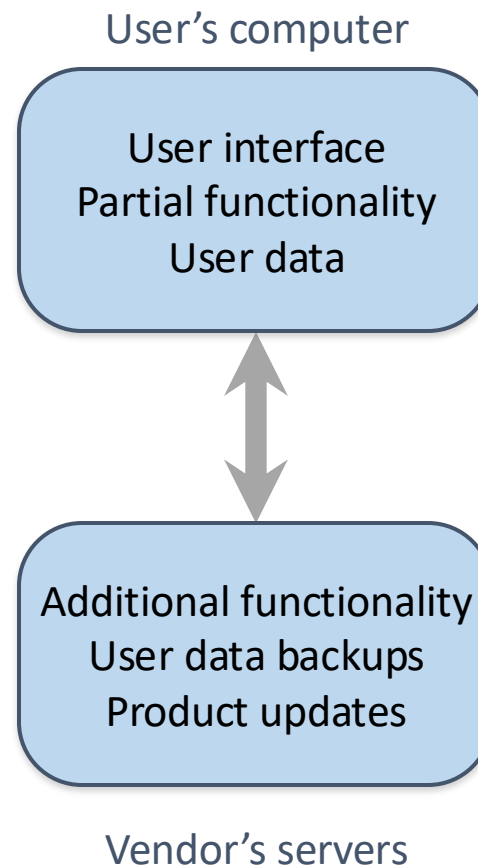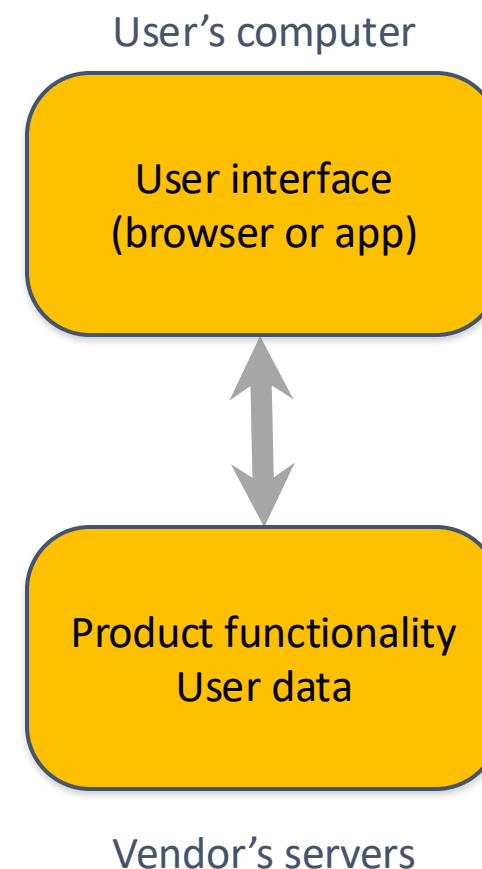
Product management concerns

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

12
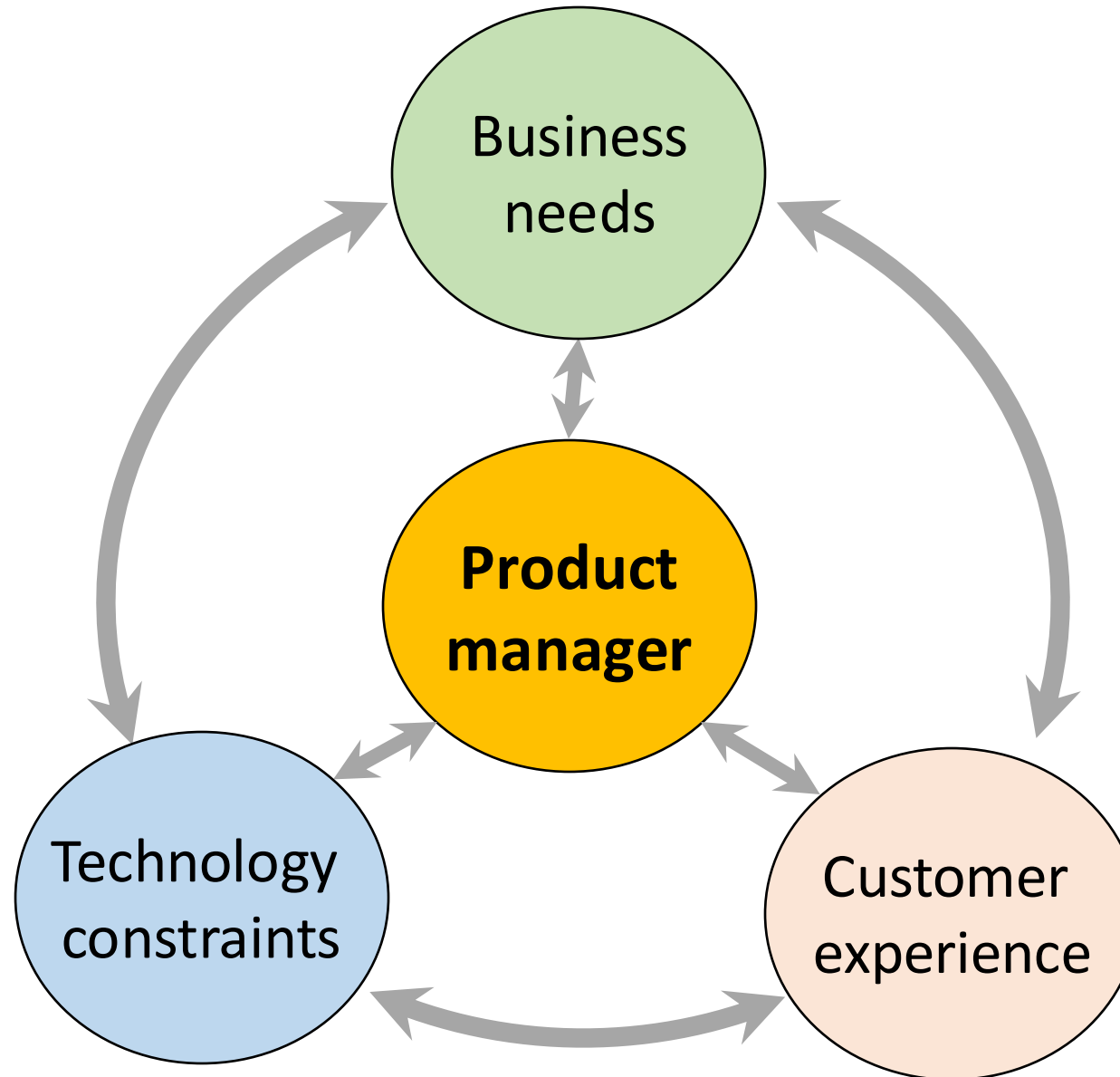
Technical interactions of product managers
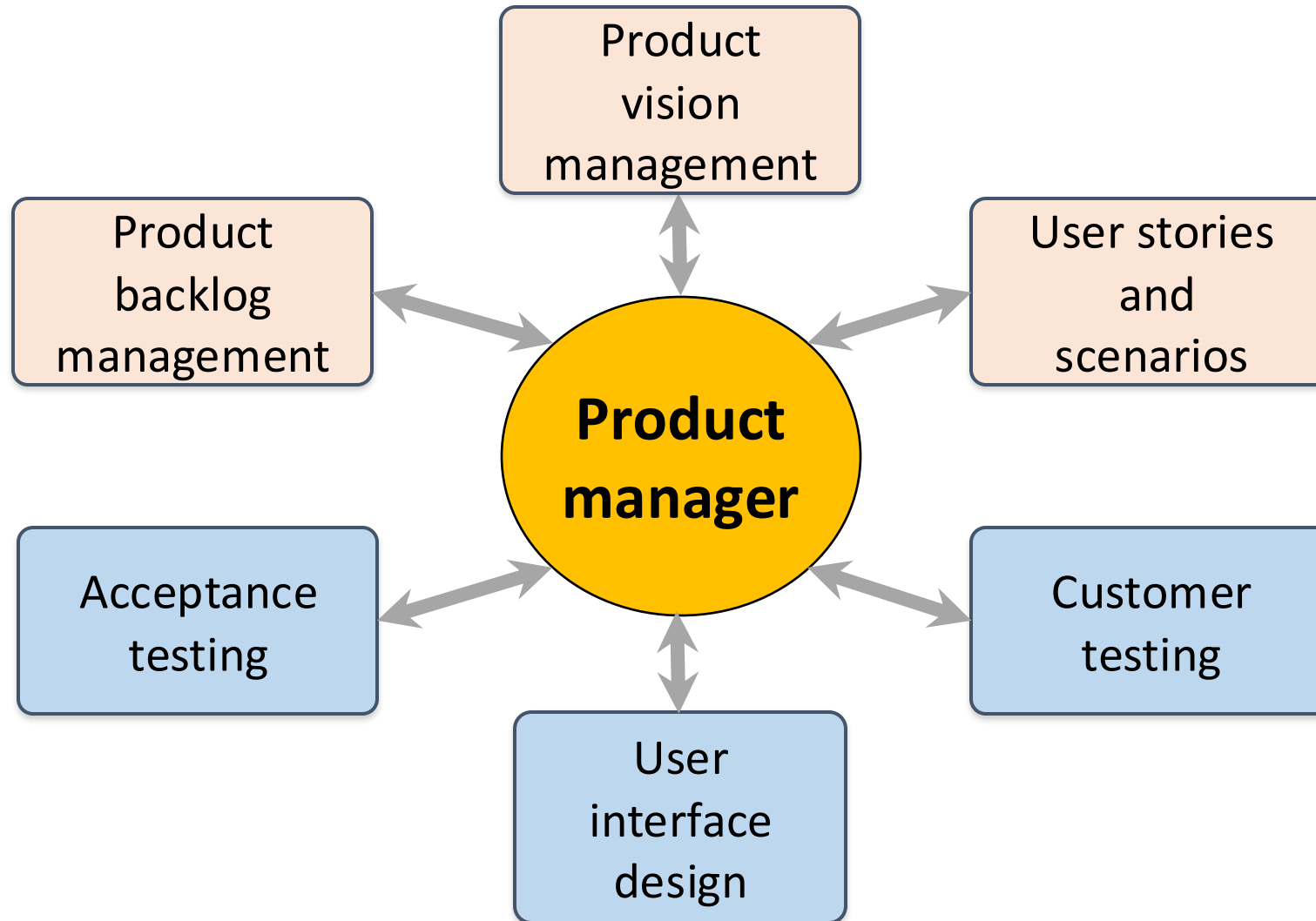
Software Development Life Cycle (SDLC)
The waterfall model

```
Requirements
definition
        │
        ▼
    System and
    Software design
            │
            ▼
        Implementation
        and unit testing
                │
                ▼
            Integration and
            system testing
                    │
                    ▼
                Operation and
                maintenance
```
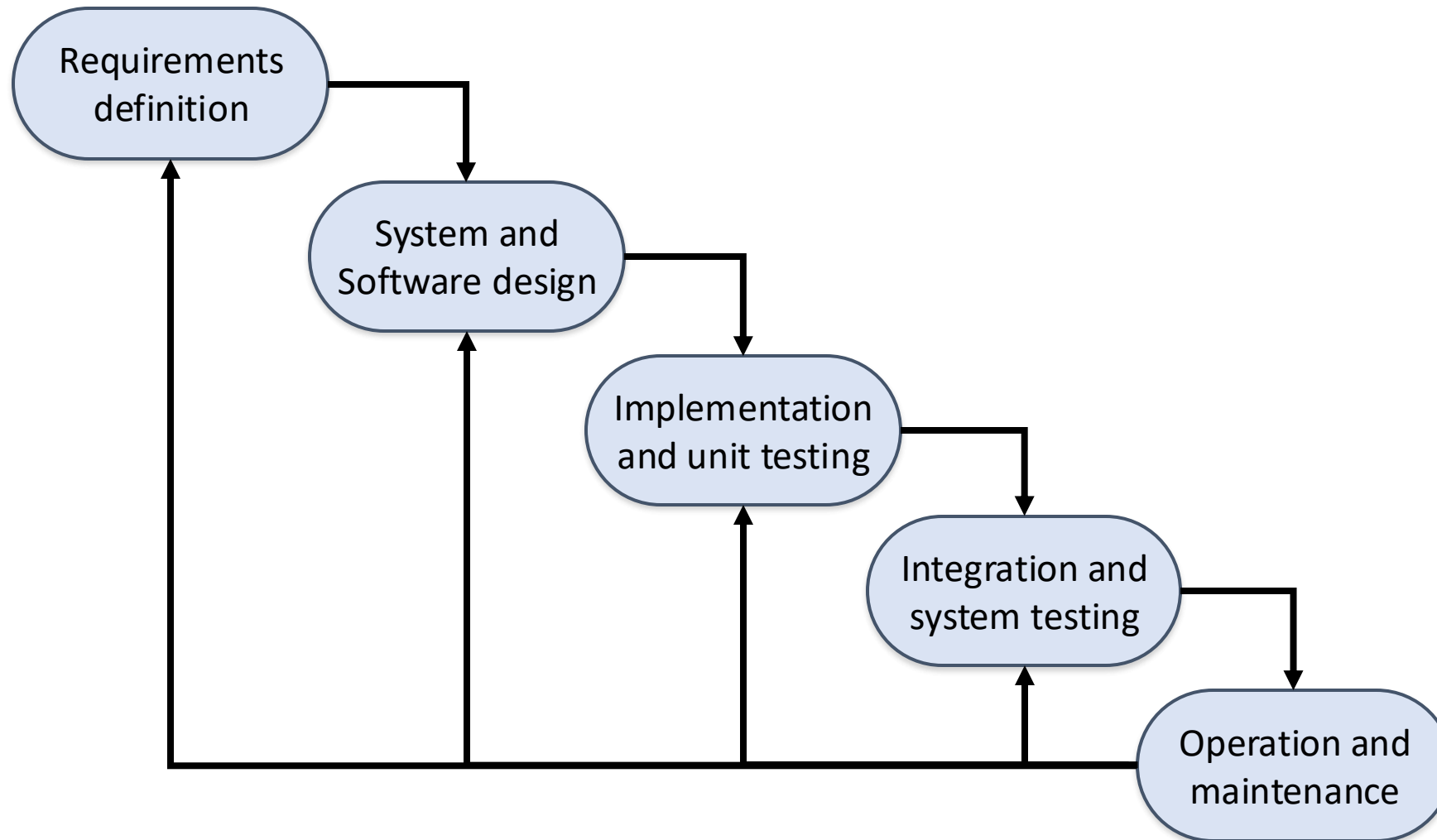
Plan-based and Agile development
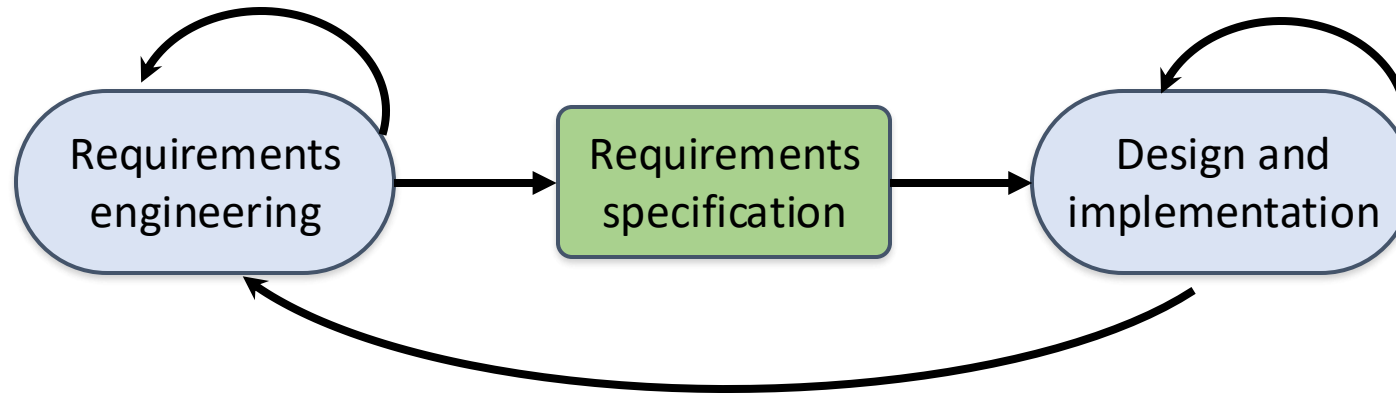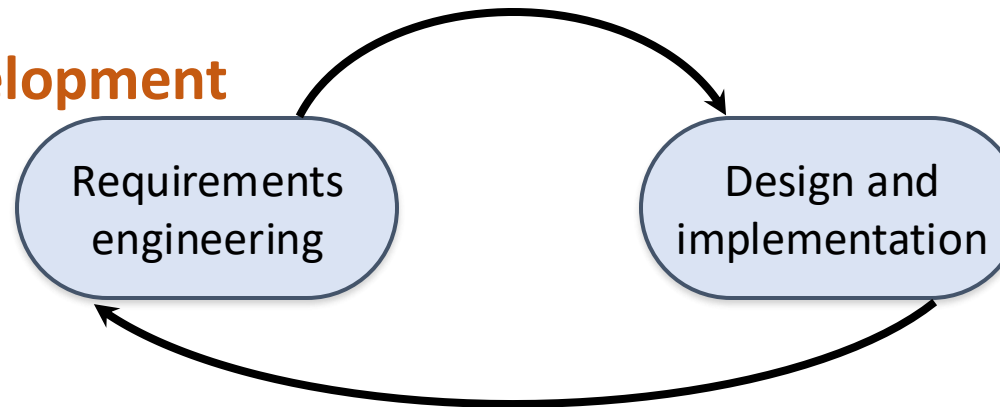
The Continuum of Life Cycles

Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Predictive Life Cycle

Analyze → **Design** → **Build** → **Test** → **Deliver**
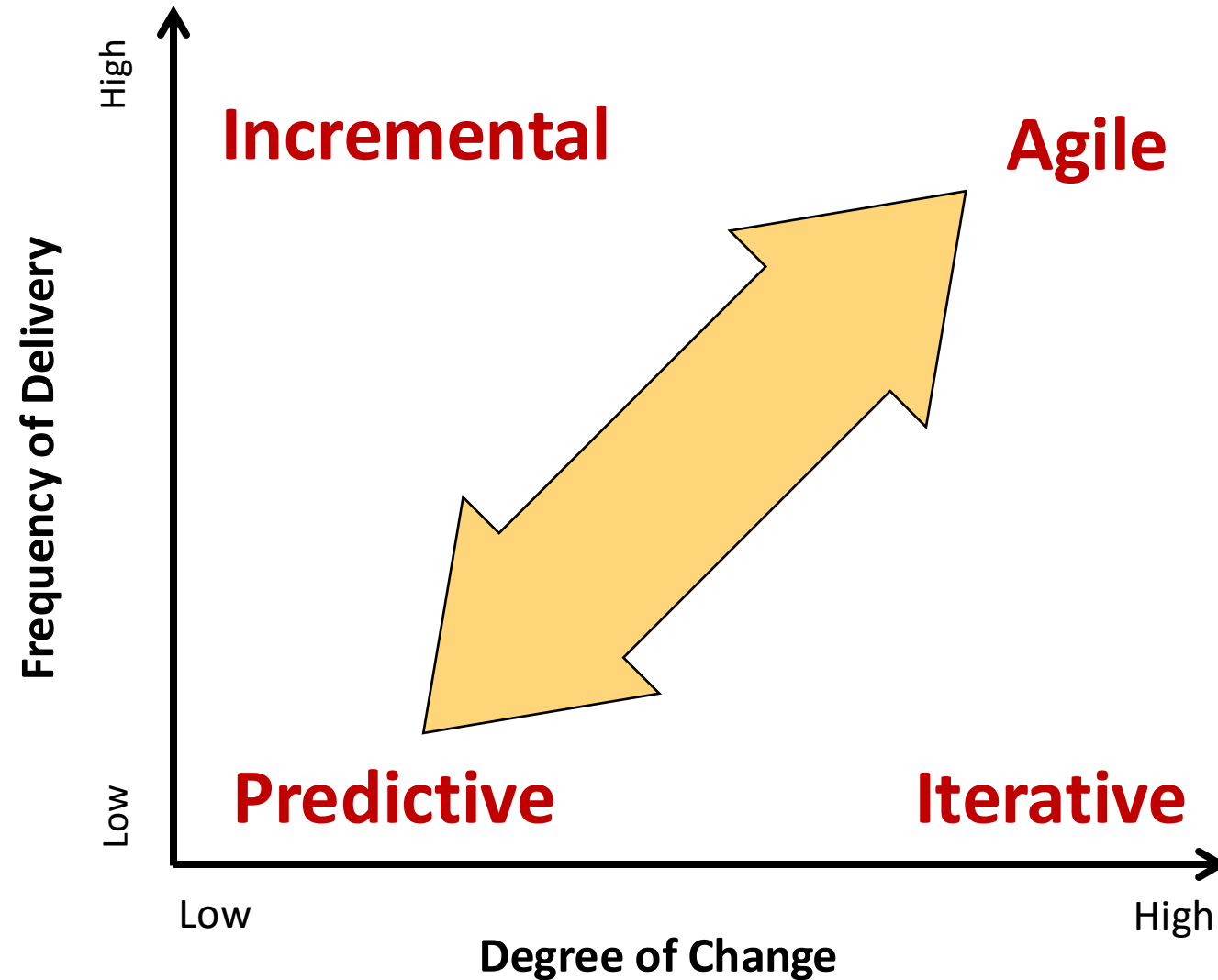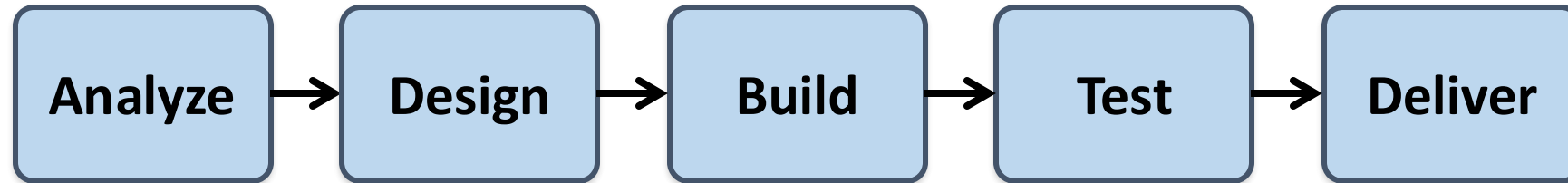
Iterative Life Cycle
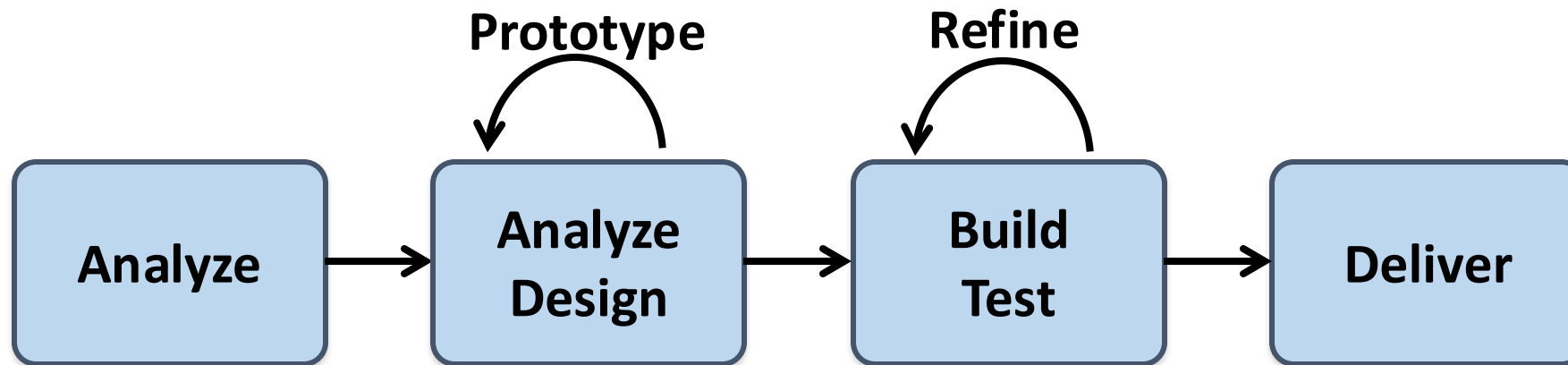
A Life Cycle of Varying-Sized Increments
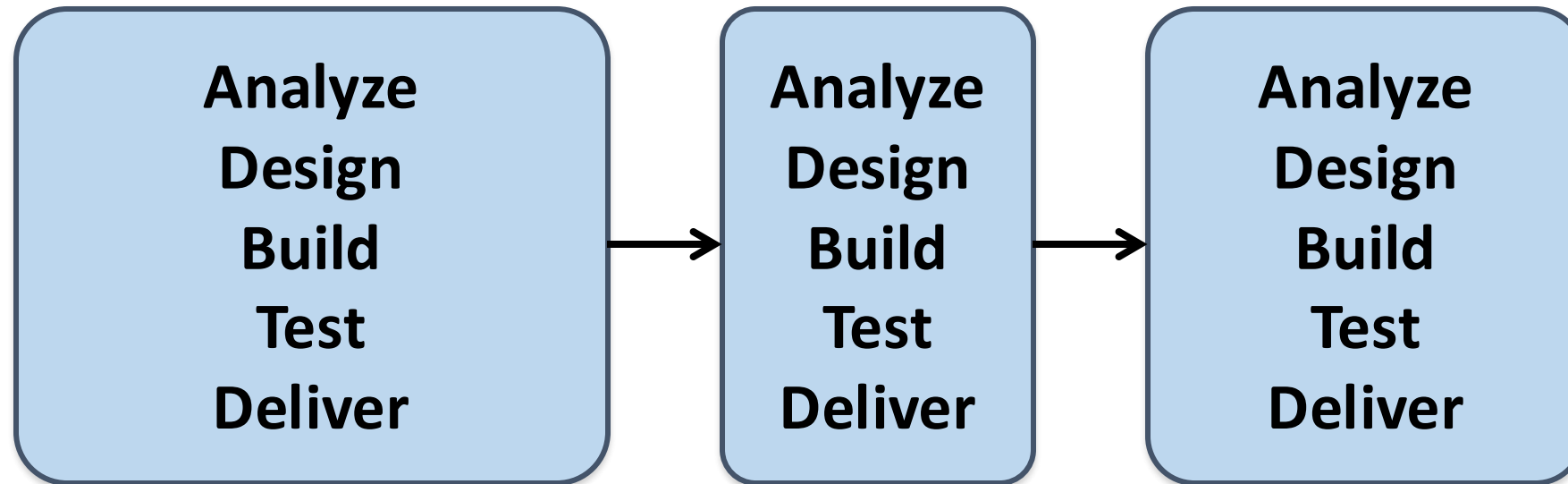
Iteration-Based and Flow-Based Agile Life Cycles

Iteration-Based Agile

Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Repeat as needed …	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test

Flow-Based Agile

Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit	Repeat as needed …	Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit

Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

From personas to features

1 **Personas** — A way of representing users

inspire

2 **Scenarios** — Natural language descriptions of a user interacting with a software product

are-developed-into

3 **Stories** — Natural language descriptions of something that is needed or wanted by users

define

inspire

4 **Features** — Fragments of product functionality
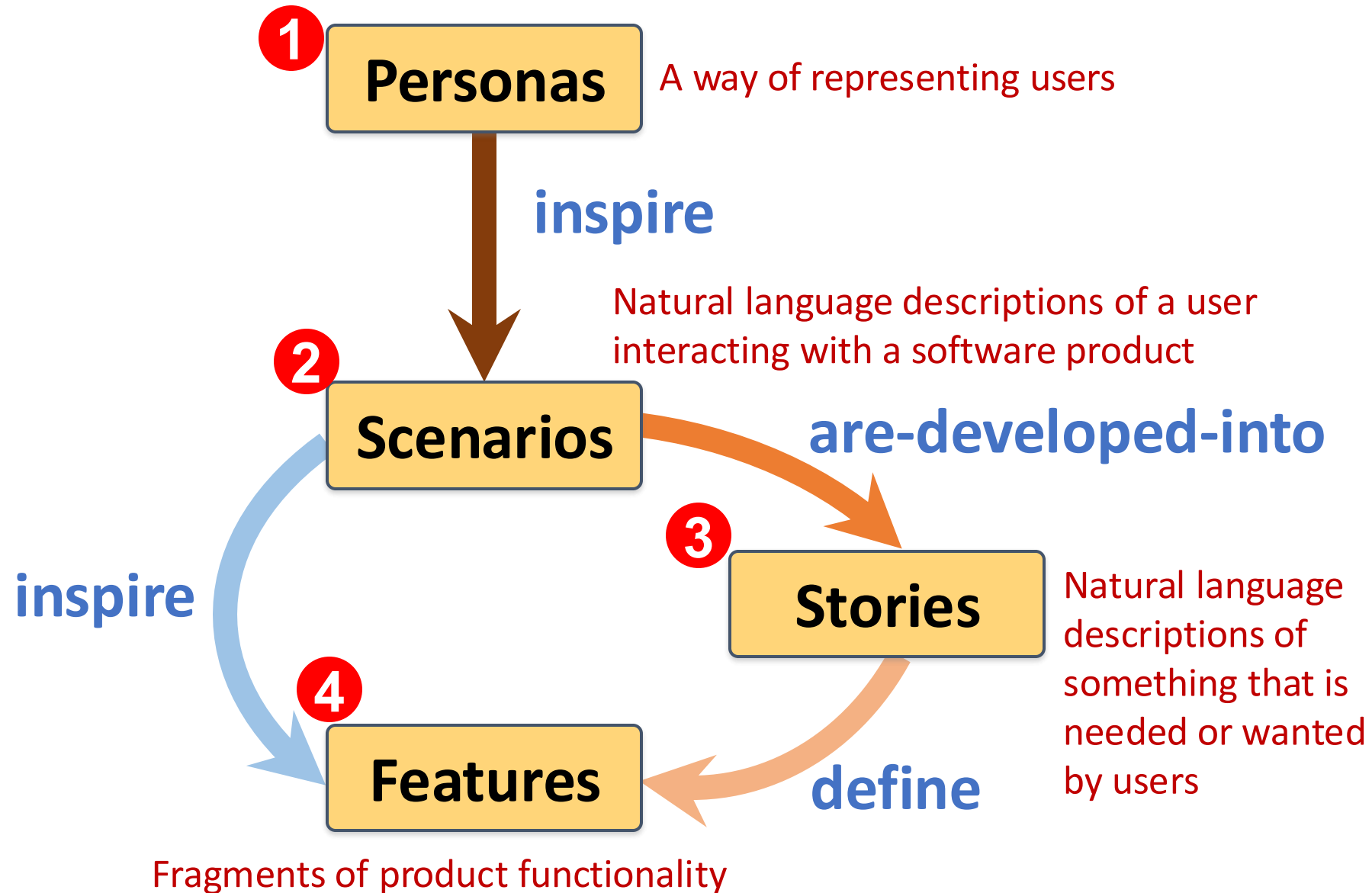
Multi-tier client-server architecture
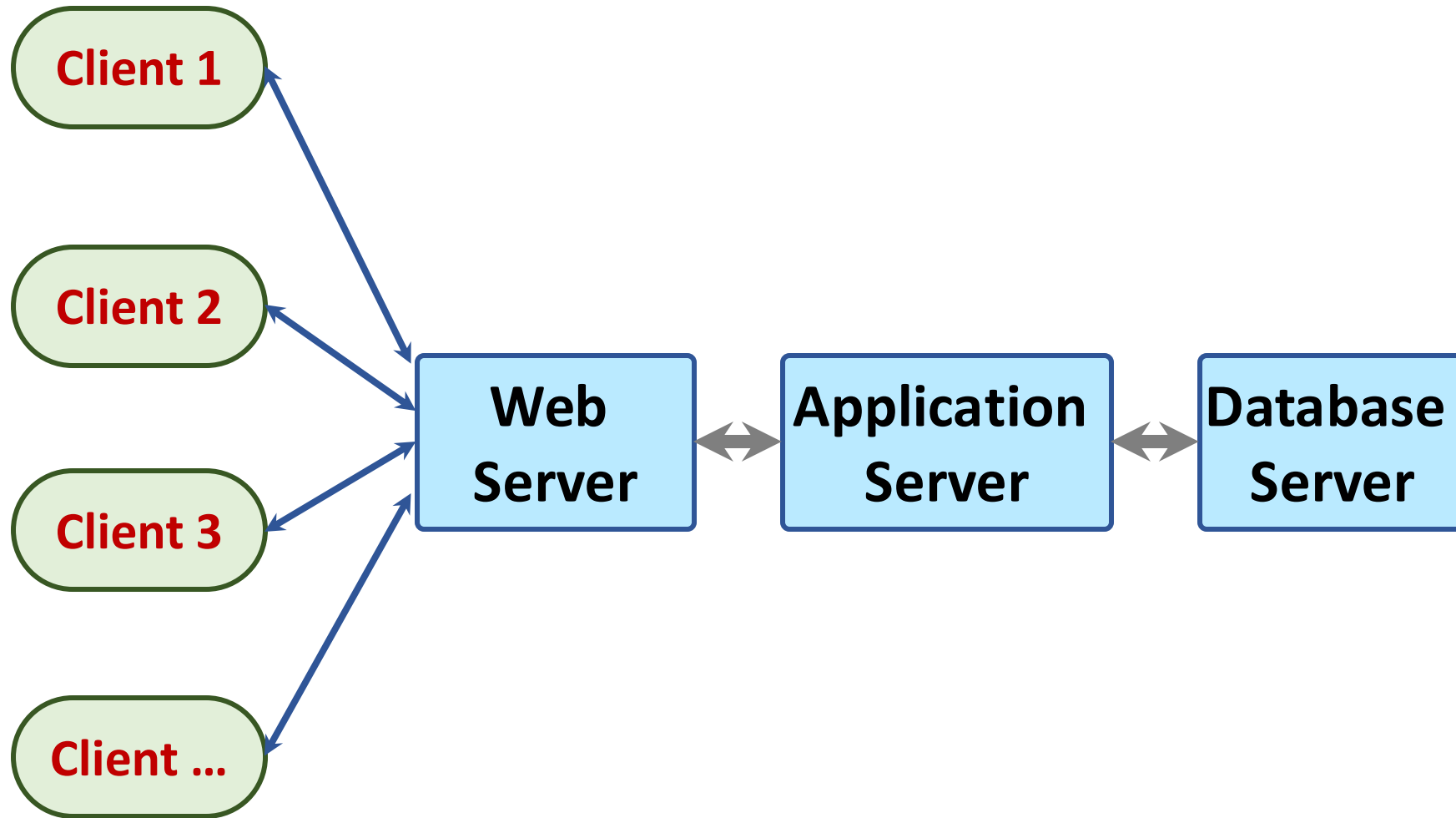
Service-oriented Architecture

Services

VM

Container

Virtual web server

Virtual mail server

User 1 Container 1

User 2 Container 2

Server software	Server software

Guest OS	Guest OS

Application software	Application software

Server software	Server software

Hypervisor

Container manager

Host OS

Host OS

Server Hardware

Server Hardware

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

24

Everything as a service

Photo editing

Cloud management Monitoring

Storage Network

Software as a service (SaaS)
Platform as a service (PaaS)
Infrastructure as a service (IaaS)
Cloud data center

Logistics management

Database Software development

Computing Virtualization

Software as a service

Software customers

Software provider

Cloud provider

Software services

Cloud Infrastructure

Microservices architecture – key design questions

What are the microservices that make up the system?

How should microservices communicate with each other?

How should data be distributed and shared?

Microservices architecture design

How should the microservices in the system be coordinated?

How should service failure be detected, reported and managed?

Types of security threat

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

Availability threats

SOFTWARE PRODUCT

An attacker attempts to damage the system or its data

Integrity threats

PROGRAM

DATA

Distributed denial of service (DDoS) attack

Virus

Ransomware

Data theft

Confidentiality threats

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

Software product quality attributes

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

29

A refactoring process

Start

1 Identify code 'smell'

2 Identify refactoring strategy

3 Make small improvement until strategy completed

4 Run automated code tests

Functional testing

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

31

Test-driven development (TDD)

Start → **1** Identify new functionality

2 Identify partial implementation of functionality

3 Write code stub that will fail test

4 Run all automated test

5 Implement code that should cause failing test to pass

6 Run all automated test

7 Refactor code if required

Functionality complete

Functionality incomplete

Test failure

All tests pass

DevOps

Development

Deployment

Support

Multi-skilled DevOps team

Code management and DevOps

DevOps automation

Continuous integration | Continuous deployment | Continuous delivery | Infrastructure as code

Code management system

Branching and merging

Recover version information

Code repository

Save and retrieve versions

Transfer code to/from developer's filestore

DevOps measurement

Data collection | Data analysis | Report generation

GitHub Copilot CODING AGENT

OpenAI Codex

A cloud-based software engineering agent
that can work on many tasks in parallel, powered by codex-1

SWE-Bench Verified

OpenAI Internal SWE tasks

SWE-bench Verified Leaderboard

AI coding agents ranking by software engineering problems resolved rate

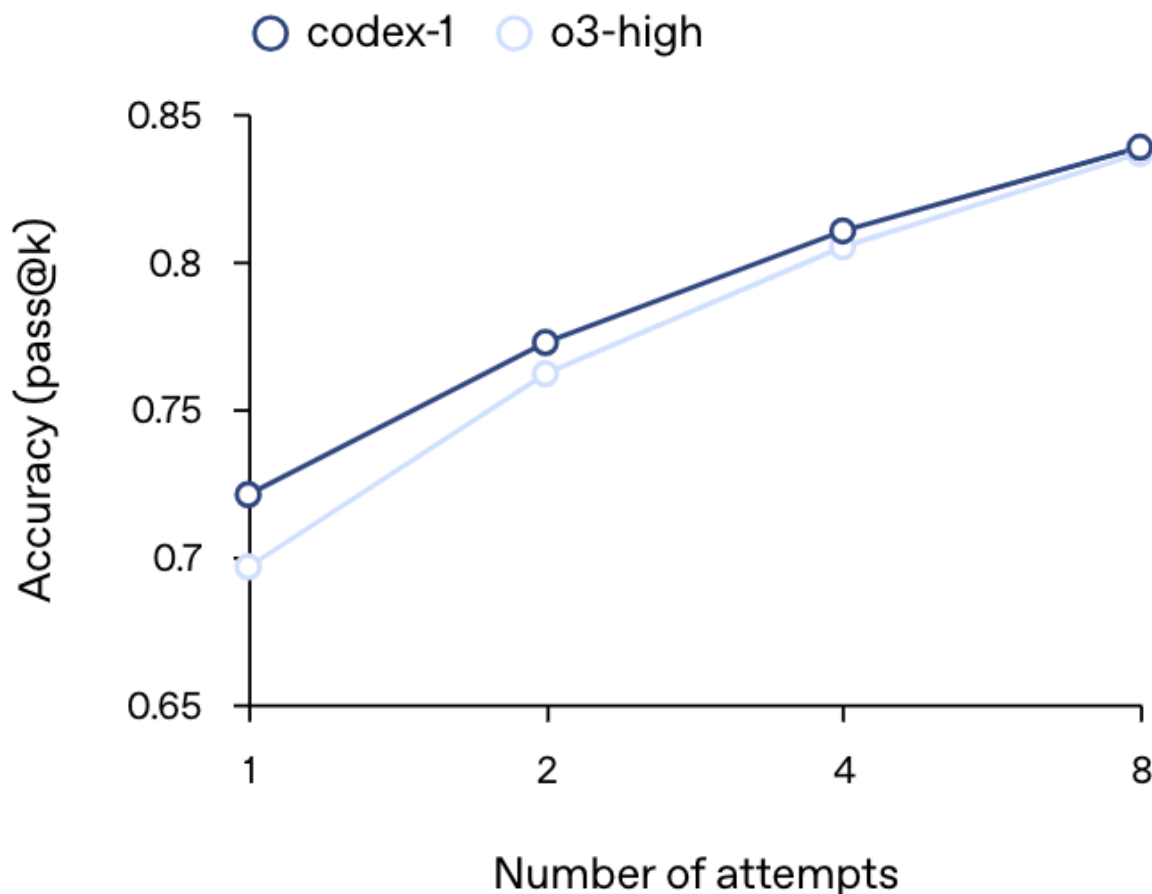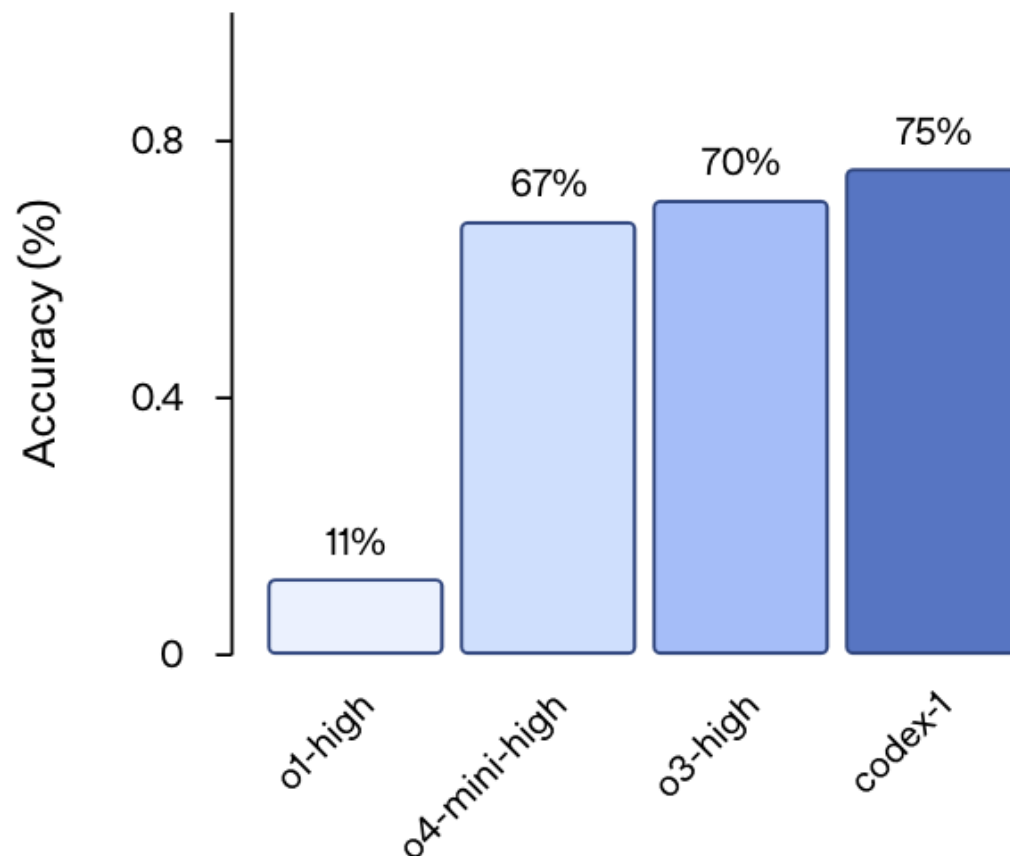| Model	% Resolved	Org	Date
🆕✅ OpenHands	65.80		2025-04-15
Augment Agent v0	65.40		2025-03-16
🆕 Amazon Q Developer Agent (v20250405-dev)	65.40	aws	2025-04-05
W&B Programmer O1 crosscheck5	64.60		2025-01-17
🆕 PatchPilot-v1.1	64.60		2025-05-03
AgentScope	63.40		2025-02-06
Tools + Claude 3.7 Sonnet (2025-02-24)	63.20	A\	2025-02-24
Blackbox AI Agent	62.80	–	2025-01-10
EPAM AI/Run Developer Agent v20250219 + Anthopic Claude 3.5 Sonnet	62.80		2025-02-28
✅ SWE-agent + Claude 3.7 Sonnet w/ Review Heavy	62.40		2025-02-25

Source: https://www.swebench.com/

WebDev Arena Leaderboard

AI coding competition in web development challenges

Rank (UB) ⓘ		Model	Arena Score	95% CI ⓘ	Votes	Organization	License
1	G	Gemini-2.5-Pro-Preview-05-06	1414.64	+13.57 / −15.14	3,464	Google	Proprietary
2	A\	Claude 3.7 Sonnet (20250219)	1357.05	+9.03 / −6.72	7,481	Anthropic	Proprietary
3	G	Gemini-2.5-Flash-Preview-05-20	1310.42	+19.10 / −21.23	981	Google	Proprietary
4		GPT-4.1-2025-04-14	1257.20	+9.49 / −8.26	4,880	OpenAI	Proprietary
5	A\	Claude 3.5 Sonnet (20241022)	1237.74	+4.15 / −4.66	26,338	Anthropic	Proprietary
6		DeepSeek-V3-0324	1206.67	+20.99 / −20.94	1,097	DeepSeek	MIT
6		DeepSeek-R1	1198.68	+10.46 / −8.71	3,760	DeepSeek	MIT
6		o3-2025-04-16	1190.47	+10.43 / −9.42	3,617	OpenAI	Proprietary
6		GPT-4.1-mini-2025-04-14	1185.05	+10.34 / −10.54	2,995	OpenAI	Proprietary
6		Qwen3-235B-A22B	1177.78	+13.42 / −16.64	2,024	Alibaba	Apache 2.0

Source: https://web.lmarena.ai/leaderboard

AutoDev: Automated AI-Driven Development

Conversation

- 🎯 Test this method
- ✒ write <file> <content>
- 📋 Run test
- ✕ Error <failure log>
- 🔍 retrieval signature
- 📄 Content result
- ✒ write <file> <fix>
- 📋 Run test
- ✅ Success

Eval Environment

- Actions </>
- Repository

User 👤

AI Agent

OpenAI GPT-4
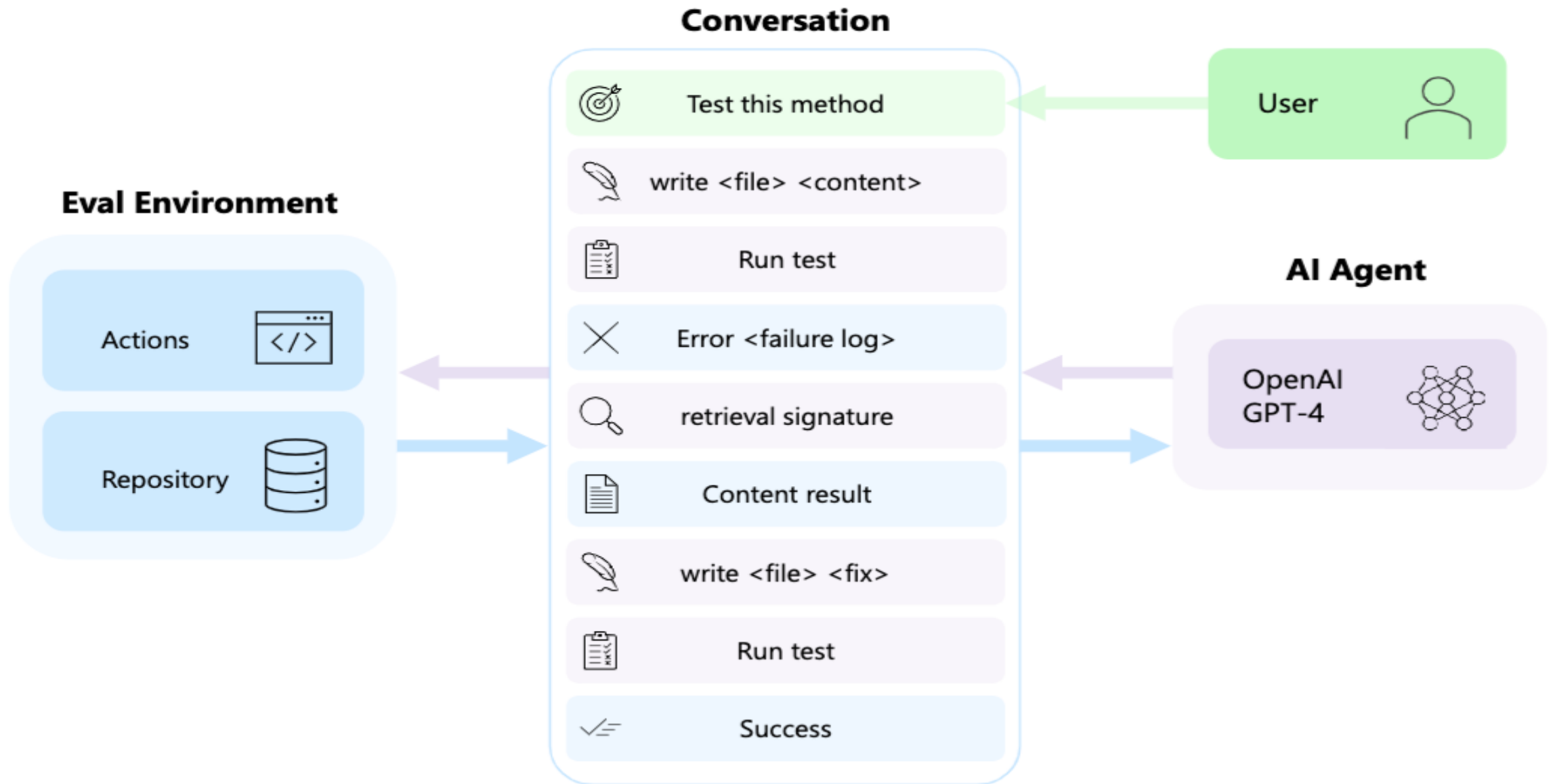
AutoDev: Automated AI-Driven Development
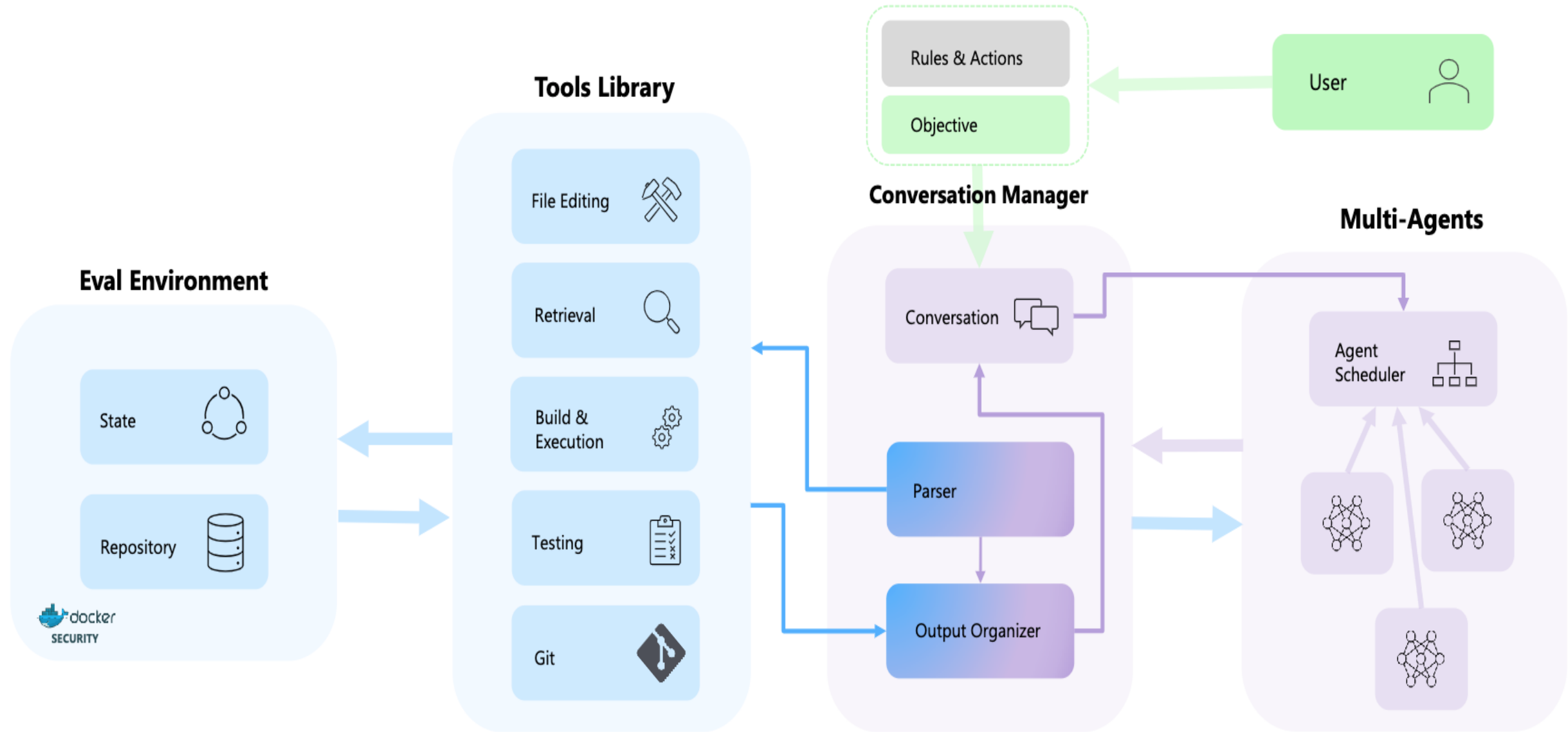
A Pilot Study on AI-Assisted Code Generation with Large Language Models for Software Engineering

Initials: P: Python, J: Java, JS: JavaScript, R: Ruby

Ref	Model	Parameter	Backbone	Year	Pretrained Dataset	Size of Dataset	Programming Language
[1]	Codex	12B	GPT-3	2021	Python from GitHub	100G	P, C++, J, JS, Go
[2]	PolyCoder	2.7B	GPT-2	2022	Multi-Lingual Corpus	249G	C, C#, C++, Go, J, JS, PHP, P, R, Rust, Scala, TypeScript
[4]	GPT-3.5	-	GPT-3	2022	-	-	P, C, C++, J, JS, Go, Julia. Perl, R, Ruby, Smalltalk
[7]	PaLM	540B	PaLM	2022	-	-	P
[9]	MarianCG	-	MarianMT	2022	CoNaLa and DJANGO	-	P, J
[12]	CodeMaster	-	CodeT5	2022	-	-	P, J
[15]	PyCodeGPT	110M	GPT-2	2022	Python and Java from CodeSearchNet	1.1M	P
[17]	PanGu-Coder2 15B	317M/2.6B	PanGu-alpha	2023	Python from GitHub	147G	P
[20]	Reflexion	-	GPT-4	2023	-	-	P
[22]	phi-1-base	1.3B	-	2023	Subset of The Stack and StackOverflow	7B	P
[25]	WizardCoder	15B	StarCoder	2023	-	1T	P
[26]	OctoGeeX	6B	-	2023	Python from GitHub	-	P, JS, J, Go, C++, Rust
[27]	Code Llama	34B	Llama-2	2023	Proprietary dataset	-	Python
[28]	CodeGeeX	13B	GPT	2023	Python from GitHub	158B	23 languages
[29]	CodeGen	16.1B	-	2023	-	-	
[30]	SantaCoder	1.1B	GPT-2	2023	subset of The Stack	268G	P, J, JS
[31]	StarCoder	15B	StarCoder-base	2023	The Stack	35B	80 languages
[32]	CodeT5+	16B	T5	2023	CodeSearchNet and C/C# datasets	52B	R, JS, GO, P, J, PHP, C, C#

41

AI-Assisted Code Generation Tools:
GitHub Copilot, Amazon CodeWhisperer, and ChatGPT

- **RQ1 What is the quality of the code generated by the code generation tools?**
  - **RQ1.1 How valid are the code generation tools' code suggestions?**
  - **RQ1.2 How correct are code generation tools' code suggestions?**
  - **RQ1.3 How secure are code generation tools' code suggestions?**
  - **RQ1.4 How reliable are code generation tools' code suggestions?**
  - **RQ1.5 How maintainable are code generation tools' code suggestions?**
- **RQ2 What is the impact of using the docstrings on the generated code quality?**
- **RQ3 What is the impact of using meaningful function names on the generated code quality?**
- **RQ4 How did the code generation tools evolve over time?**

AI-Assisted Code Generation Tools: GitHub Copilot, Amazon CodeWhisperer, and ChatGPT

Features	ChatGPT	Amazon CodeWhisperer	GitHub Copilot
IDE Support	No IDE Support	JetBrains, Visual Studio Code, AWS Cloud9, or the AWS Lambda console	IntelliJ IDEA, Android Studio, AppCode, CLion, Code With Me Guest, DataGrip, DataSpell, GoLand, JetBrains Client, MPS, PhpStorm, PyCharm, Rider, RubyMine, WebStorm
First Release Time	Nov-30-2022	June-23-2022	Oct-29-2021
Developer	OpenAI	AWS	OpenAI-Microsoft
Providing References to Suggestions	NO	YES	NO
Explanation of Suggestions	YES	NO	NO
Providing Multiple Suggestions	NO (Theoretically user can manually ask for another suggestion.)	YES (Up to 5)	YES (Up to 10)
Training Data Source	GitHub Repositories, OpenAI Codex Dataset, other code repositories such as GitLab, Bitbucket, and SourceForge	"Vast amounts of publicly available code"	"...trained on all languages that appear in public repositories" (Fine-tuned)
Programming Languages work best with (according to the vendor)	N/A	C#, Java, JavaScript, Python, and TypeScript	C, C++, C#, Go, Java, JavaScript, PHP, Python, Ruby, Scala, and TypeScript
Multipurpose (other than programming)	YES	NO	NO
Subscription	ChatGPT Free ChatGPT Plus ($20 per month)	Free Preview	Copilot for Students (Free) Copilot for Individuals ($10 per month) Copilot for Business ($19 per user, per month)
Can be Used Offline?	NO	NO	NO
Can it Access Local Files?	NO	YES	YES

Source: Yetiştiren, Burak, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT." arXiv preprint arXiv:2304.10778 (2023).

AI-Assisted Code Generation Tools: Experiment Setup
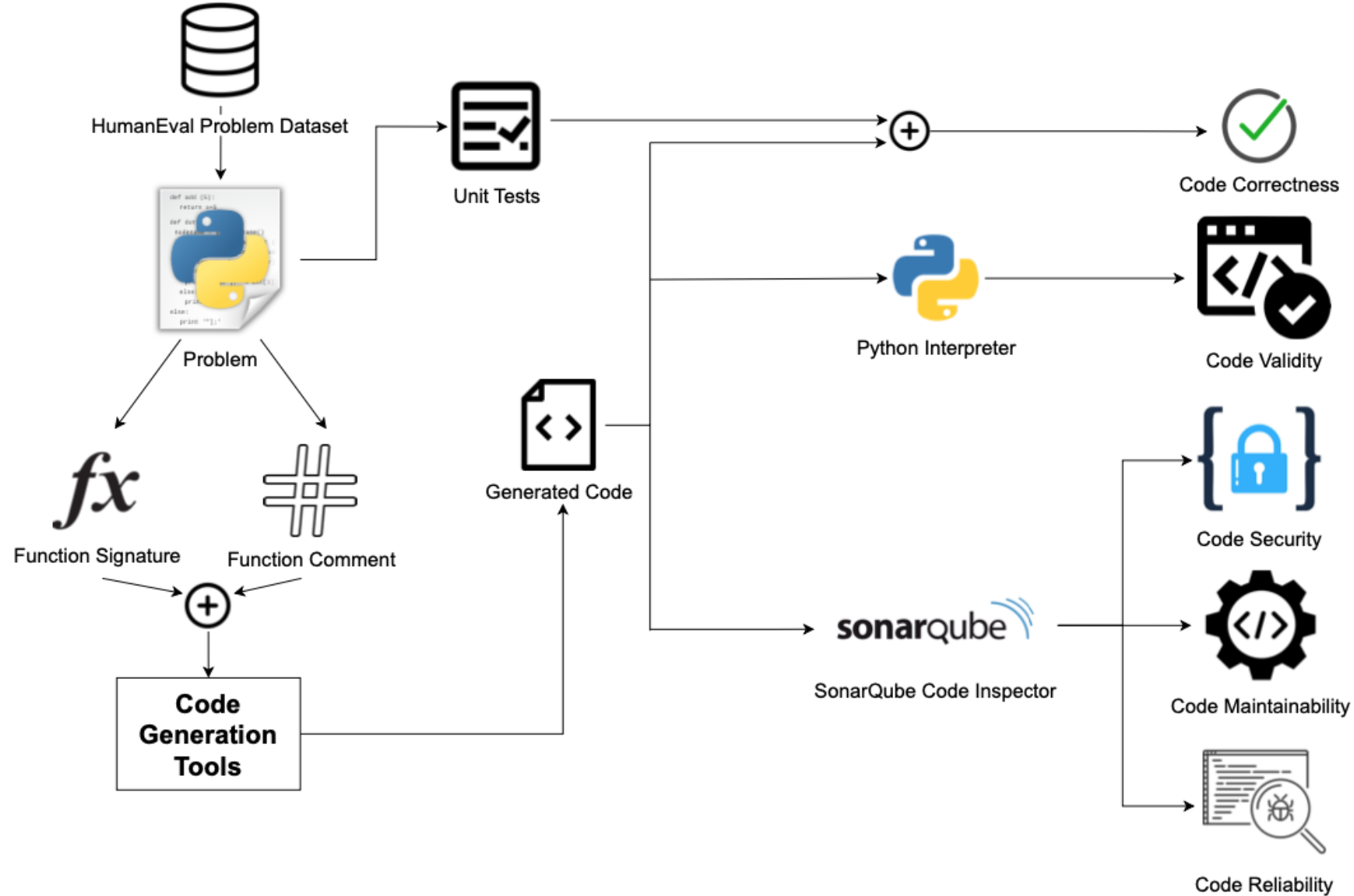
AI-Assisted Code Generation Tools: Experiment Workflow

Source: Yetiştiren, Burak, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT." arXiv preprint arXiv:2304.10778 (2023).

AI-Assisted Code Generation Tools: Code Correctness

Code Correctness among the Code Generation Tools

Source: Yetiştiren, Burak, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT." arXiv preprint arXiv:2304.10778 (2023).

AI-Assisted Code Generation Tools: Code Correctness

ChatGPT

- 12.2%
- 22.6%
- 65.2%

Legend:
- Proportion of Correct Generations (green)
- Proportion of Partially Correct Generations (purple)
- Proportion of Incorrect Generations (yellow)

GitHub Copilot

- 30.5%
- 46.3%
- 23.2%

Legend:
- Proportion of Correct Generations (green)
- Proportion of Partially Correct Generations (purple)
- Proportion of Incorrect Generations (yellow)

Source: Yetiştiren, Burak, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT." arXiv preprint arXiv:2304.10778 (2023).

Code Generation Models: BERT, T5, GPT-3, CodeBERT CodeGPT, CodeX

Pretrained Model	Backbone	Parameter	Pretrained Dataset	Size of Dataset	Trained Language
CuBERT	BERT	-	Python from Github	6.6M	Python
CodeBERT	BERT	125M	CodeSearchNet	3.5G	Ruby/JavaScript/Java/ Python/GO/PHP/English
GraphCodeB ERT	BERT	125M	CodeSearchNet	3.5G	Ruby/JavaScript/Java/ Python/GO/PHP/English
CodeGPT	GPT-2	124M	Python and Java from CodeSearchNet	Python 1.1M Java 1.6M	Java/Python
CoTexT	T5	-	CodeSearchNet AND Java and Python from BigQuery	-	Java/Python
CodeT5	T5	60M/223M/ 770M	CodeSearchNet and C/C# datasets	8.35G	Ruby/JavaScript/GO/ Python/Java/PHP/C/C#
PLBART	BART	140M	Java and Python from BigQuery AND SO posts	655G	Java/Python/English
CodeGen	-	350M/2.7B/ 6. 1B/16.1B	THEPILE/BIGQUERY/ BIGPYTHON	THEPILE 825G	C/C++/Go/Java/ JavaScript/Python/English
InCoder	-	1. 3B/6.7B	content from StackOverflow	159G	PTYHON and 28 other languages
CodeX	G P T-3	300M/2.5B/ 12B	Python from GitHub	159G	Python
AlphaCode	-	300M/1B/ 3B/9B/41B	a snapshot of github	715.1G	C++/C#/Java/JavaScript/ Lua/PHP/Python/Ruby/ Go/Rust/Scala/TypeScript
PanGu-Coder	PanGu-alpha	317M/2.6B	Python from GitHub	147G	Python
CodeGeeX	-	13B	open-sourced code datasets, The Pile and CodeParrot	-	C++/Python/C/Java/ JavaScript/Go/HTML/PHP/ Shell/CSS/Others
aiXcoder L	G P T-2	1.3B	Java from GitHub	-	Java
aiXcoder XL	-	13B	Open-sourced code from GitHub	-	Java

Security and Privacy

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

SOFTWARE PRODUCT

PROGRAM

DATA

Virus

Ransomware

Data theft

Confidentiality threats

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

System infrastructure stack

Operational Environment

Application

Frameworks and application libraries

Browsers and messaging

System libraries

Database

Operating system

Software Infrastructure

Network
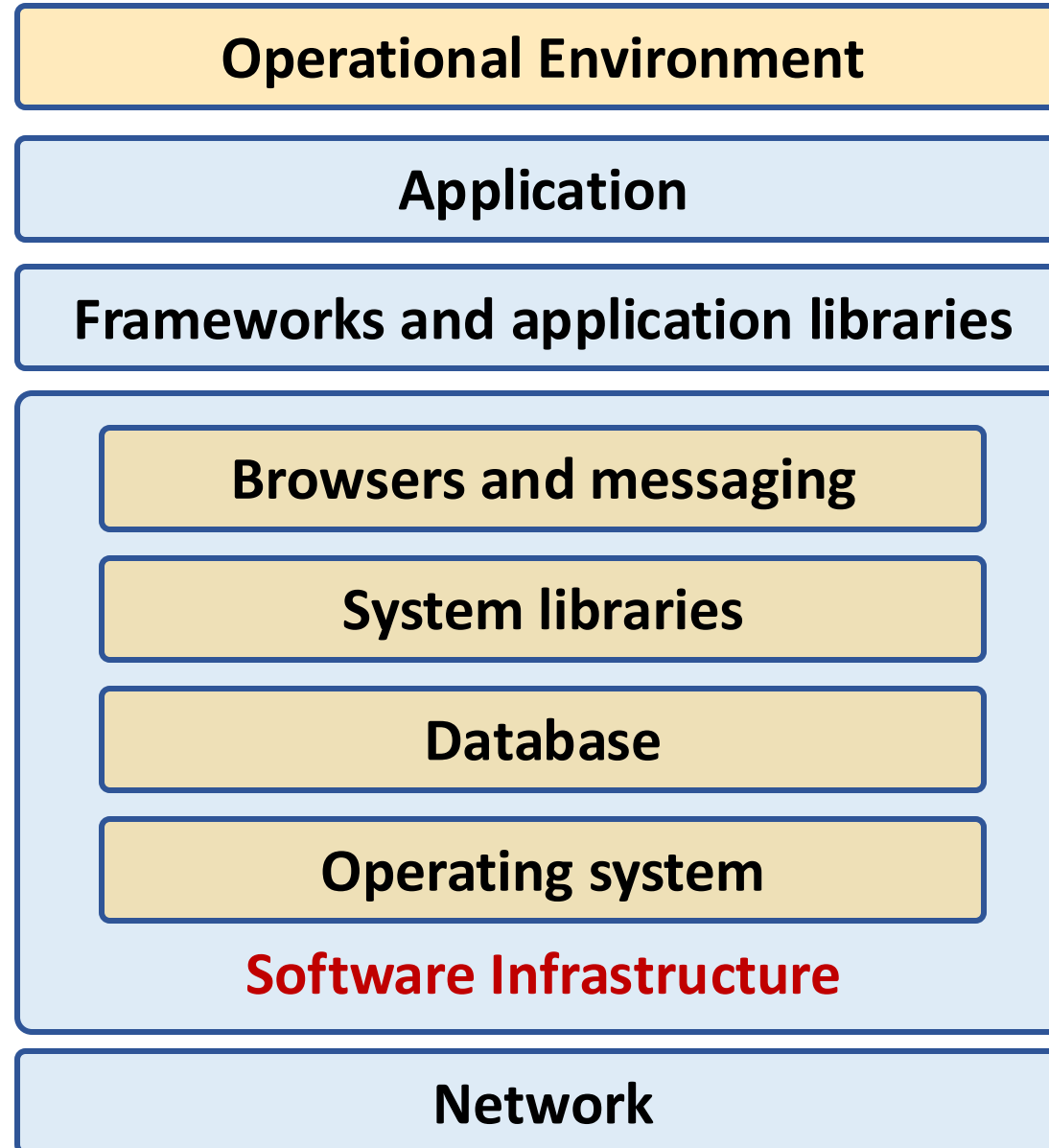
Security management

- ## 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

Attacker

Browser

Website

1. Introduce malicious code

Product website

Malicious code added to valid data

3. Malware script sends session cookie to attacker

Valid request for data from website

Browser

Victim

2. Data delivered and malware script installed in victim's browser
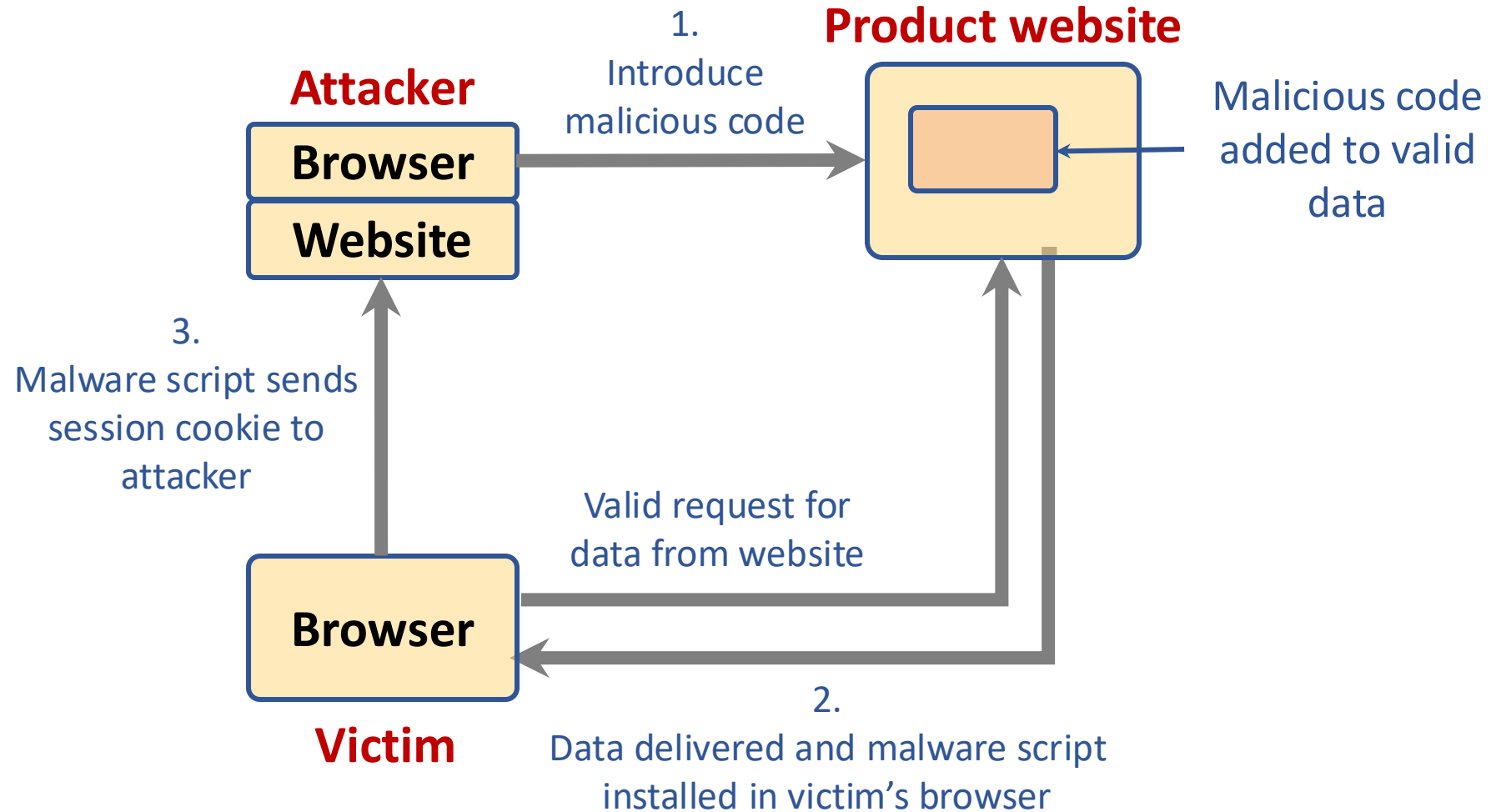
Session hijacking attacks

- **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

Authentication approach

Example

Authenticating user

Knowledge	→	Password
Possession	→	Mobile device
Attribute	→	Fingerprint
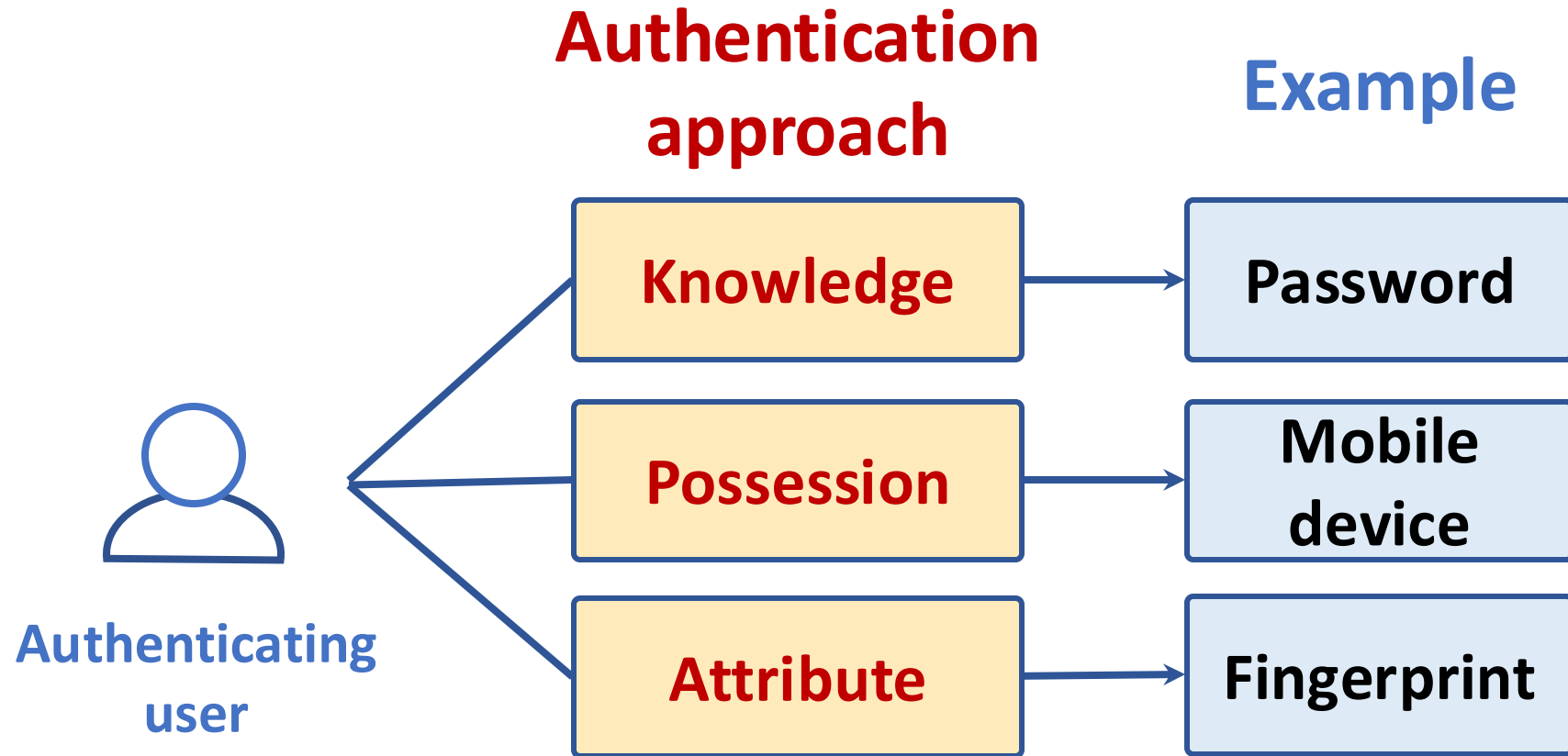
Weaknesses of password-based authentication

- **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
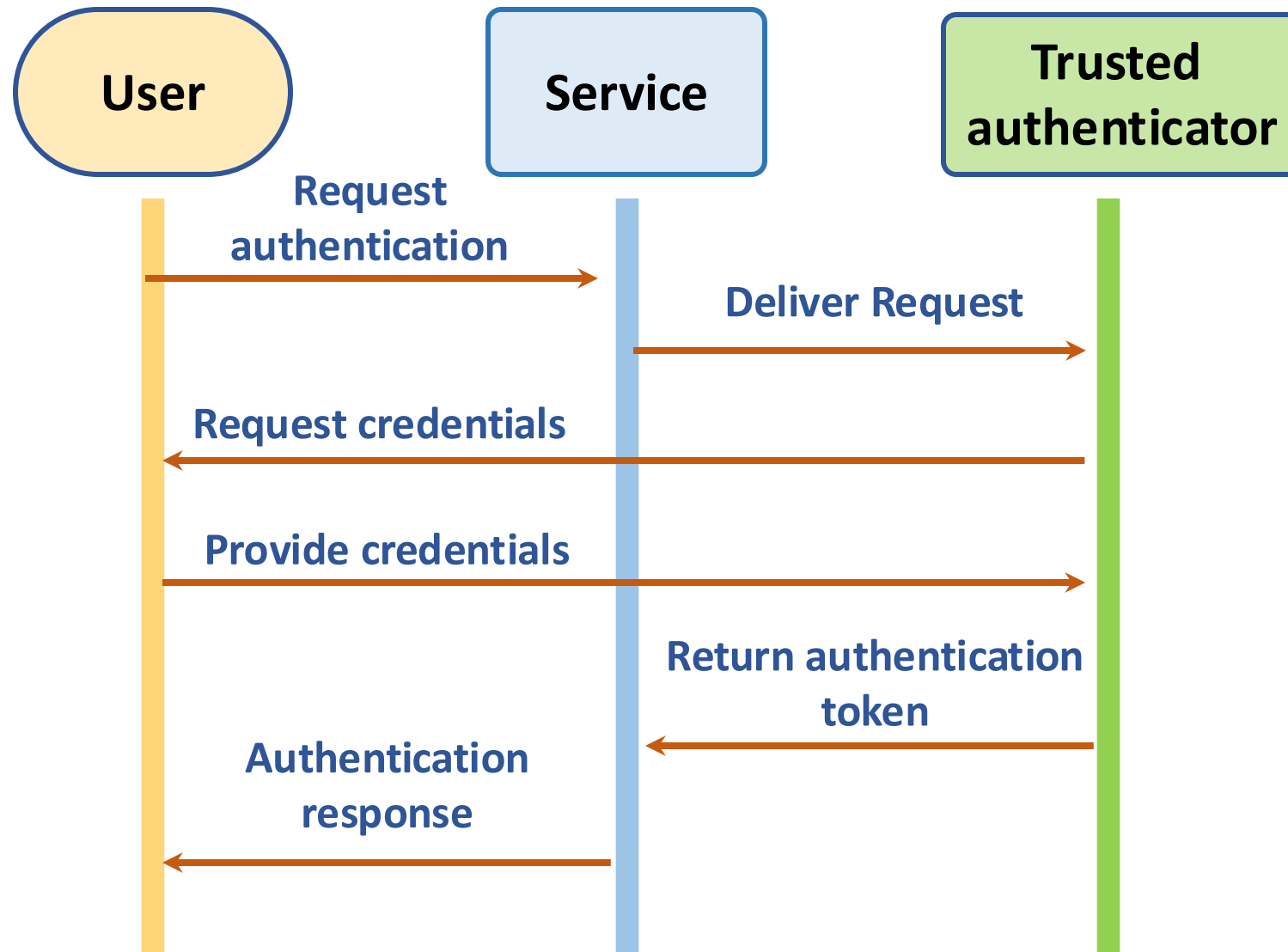
Authorization

- **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

Alice

Bob

Encryption key
a7Dr6yYf9F...

Encryption key
a7Dr6yYf9F...

Encrypt

Decrypt

Secret message → **Encrypted text** → **Secret message**

Asymmetric encryption

Alice

Bob

Bob's **public** key
dr5ts3TR9dt x4ztmRsYY…

Bob's **private** key
hTr34BbfsDy 9r3g5HHt76…

Encrypt

Decrypt

Secret message → **Encrypted text** → **Secret message**
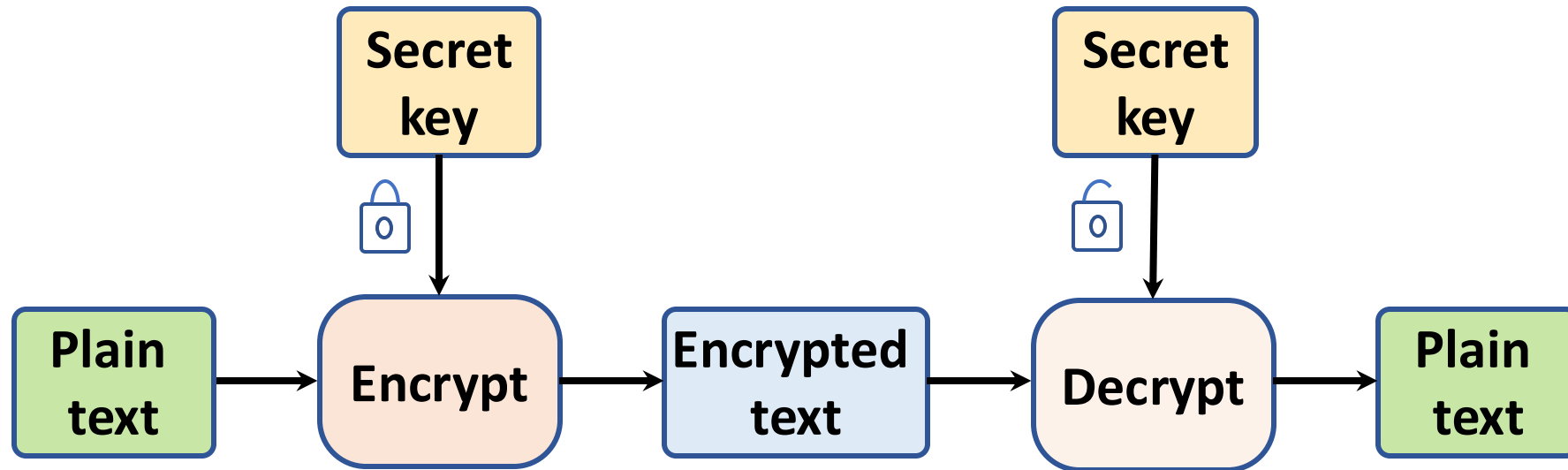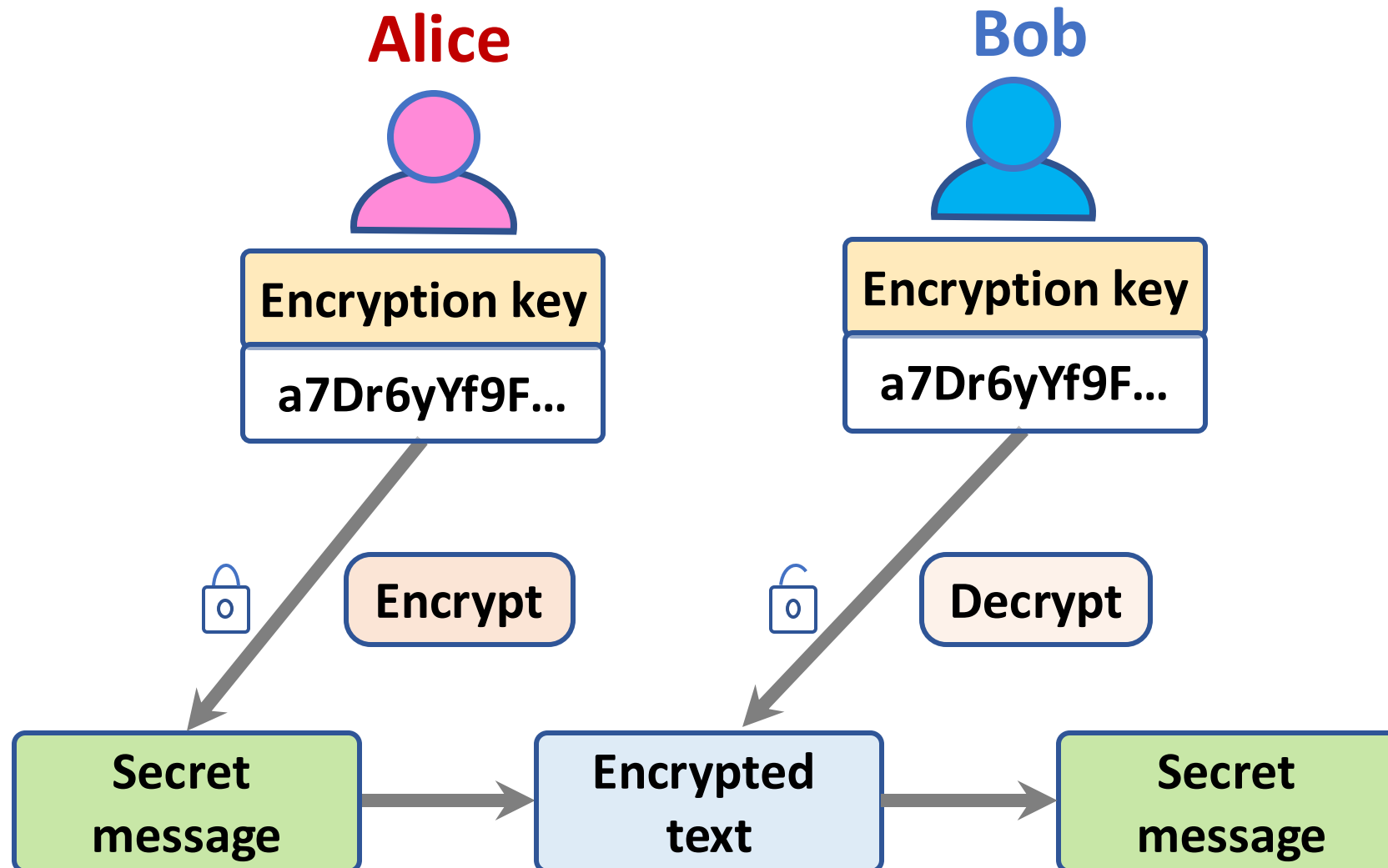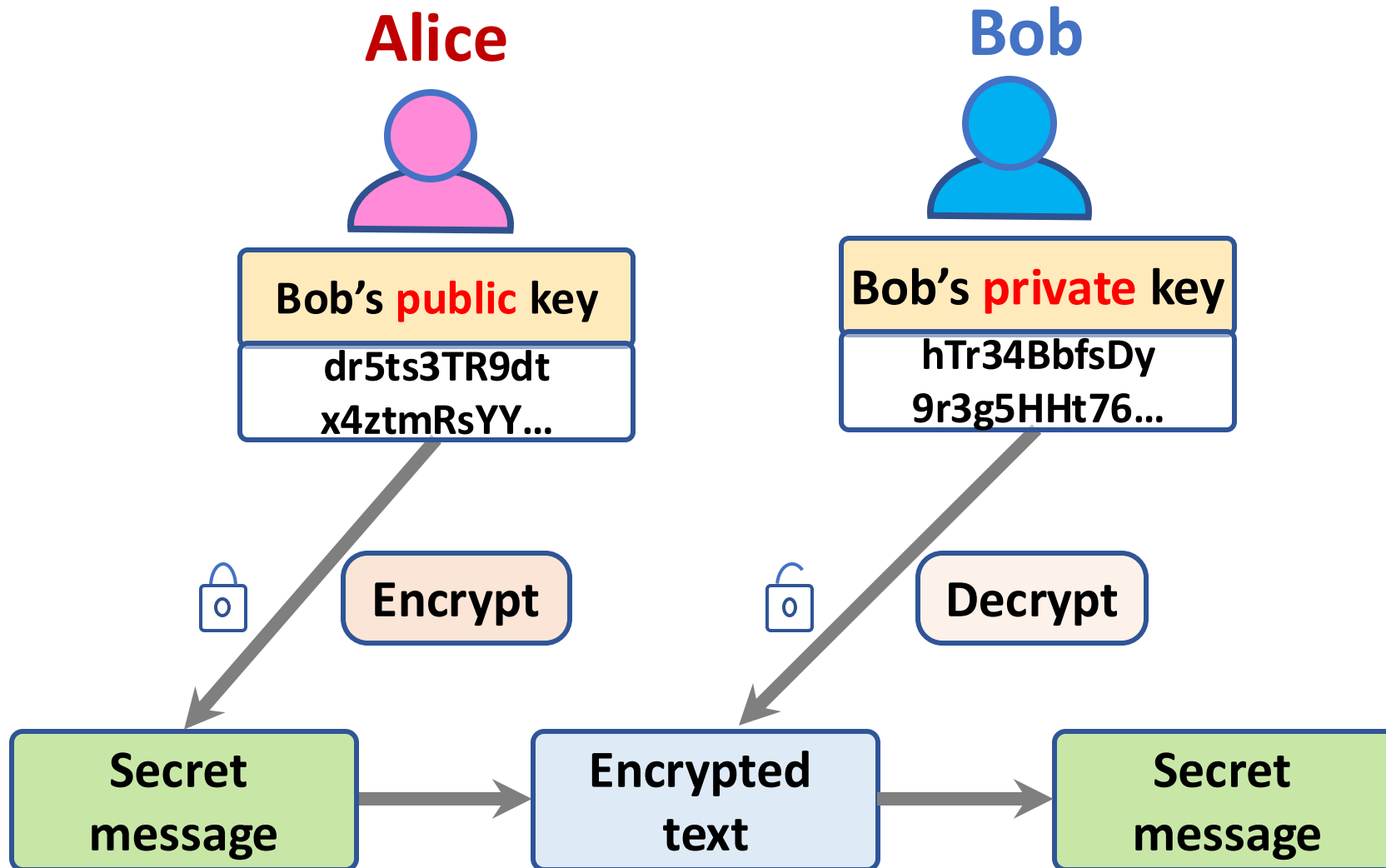
Encryption for authentication

Bob

Alice

Bob's **private** key
hTr34BbfsDy 9r3g5HHt76...

Bob's **public** key
dr5ts3TR9dt x4ztmRsYY...

Encrypt

Decrypt

| I am really Bob | → | Encrypted text | → | I am really Bob |
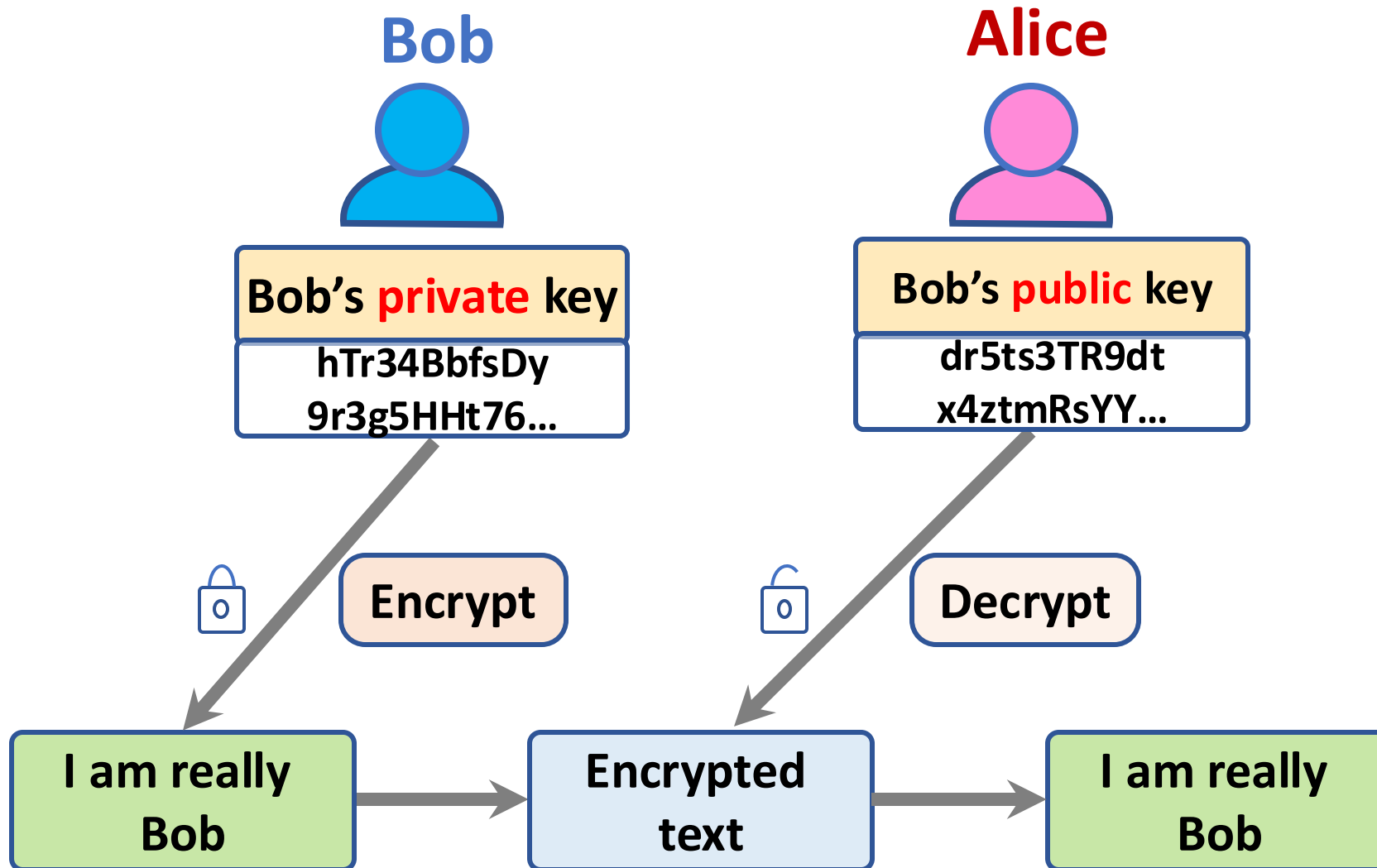
TLS and digital certificates

- **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

Application

Calls

API

Key management system (KMS)

Keys

Key store

Unencrypted data

Encryption engine

Stored encrypted data
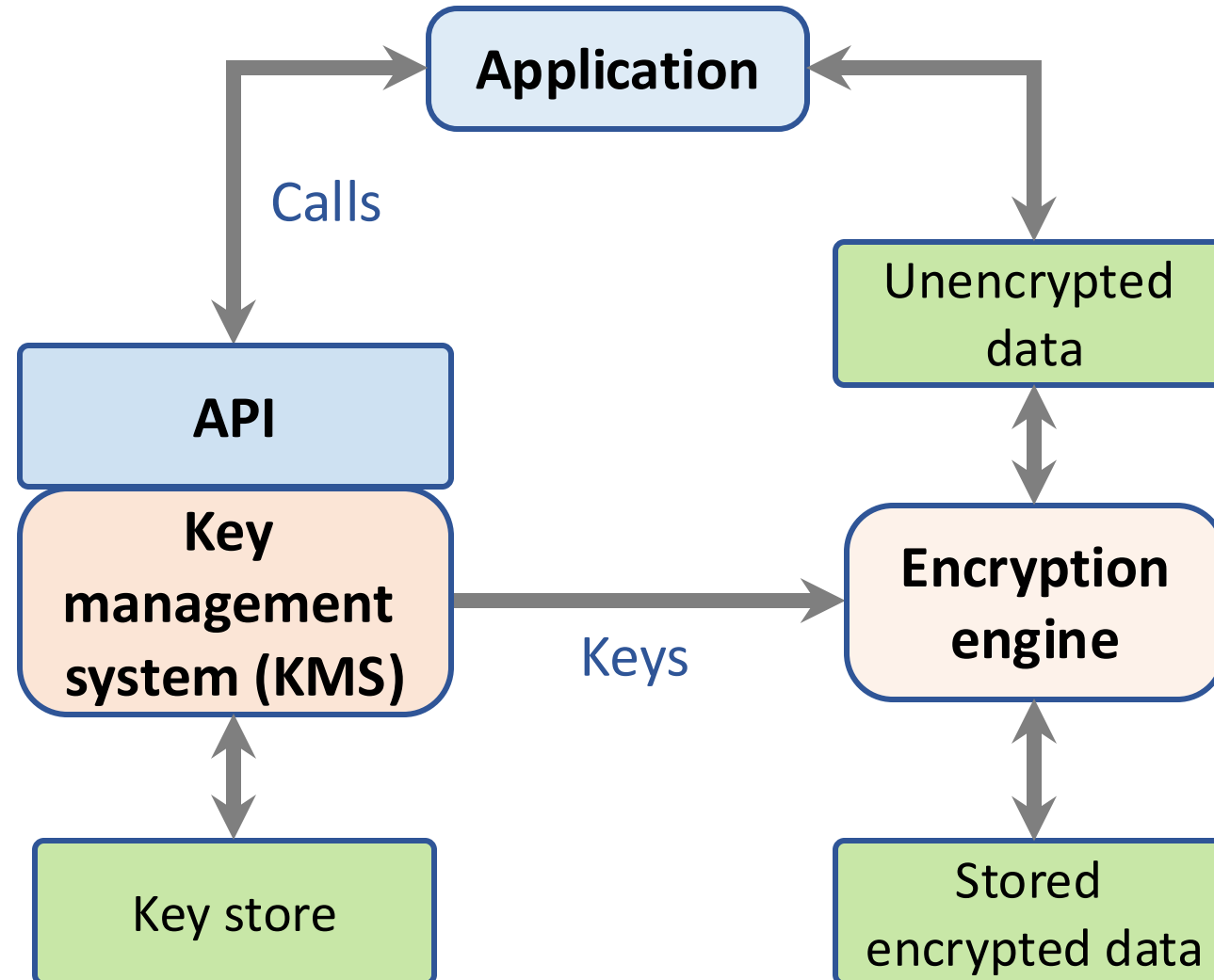
Long-term key storage

- **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 law

Responsibilities of the data controller

Data storage
Data use
Security
Subject access

Rights of the data subject

Data access
Error correction
Data deletion
Consent
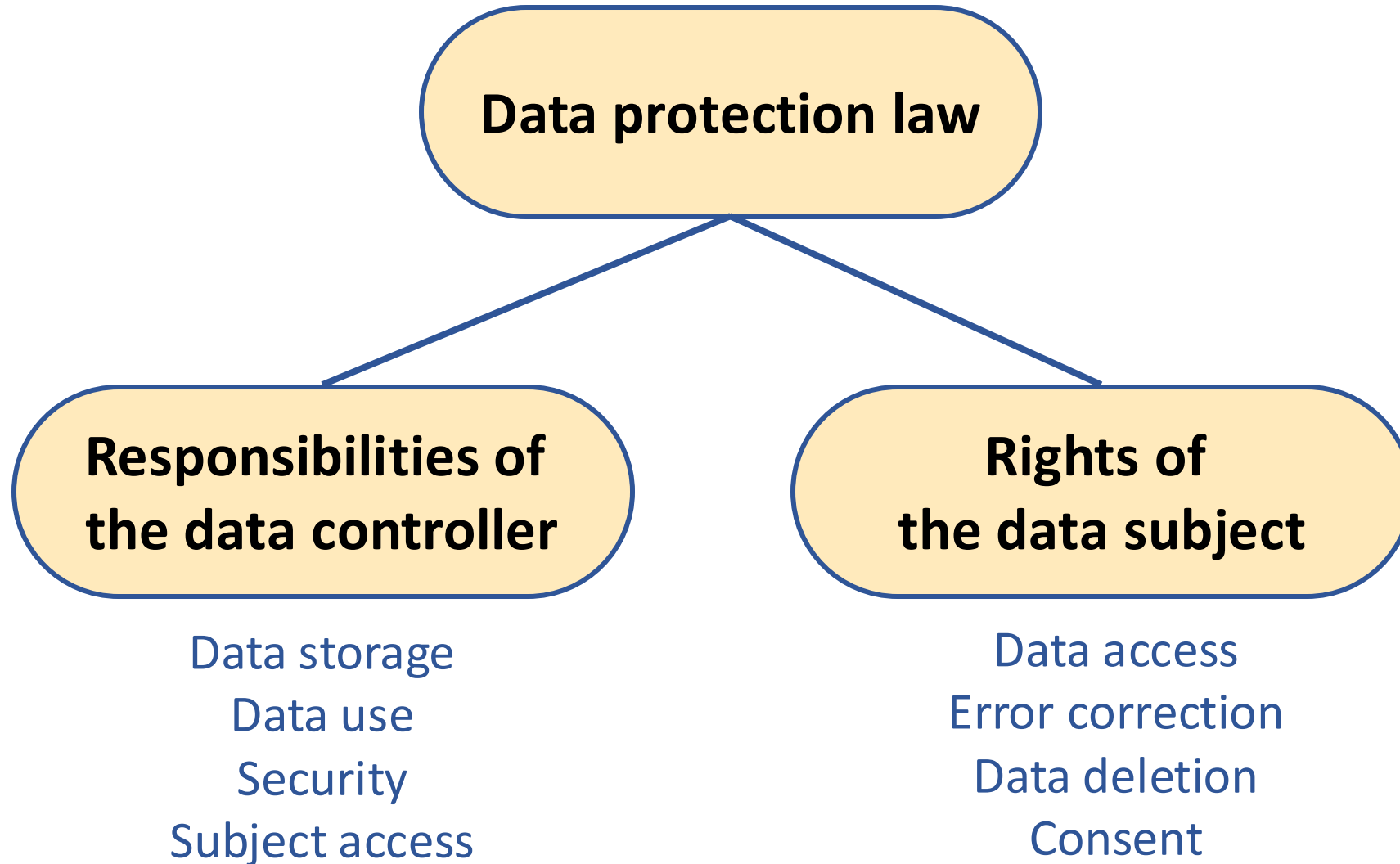
Data protection principles

- **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.**

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

94

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**.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

96

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

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

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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

Problem

Technology

Programming language, libraries, database, IDE, etc.
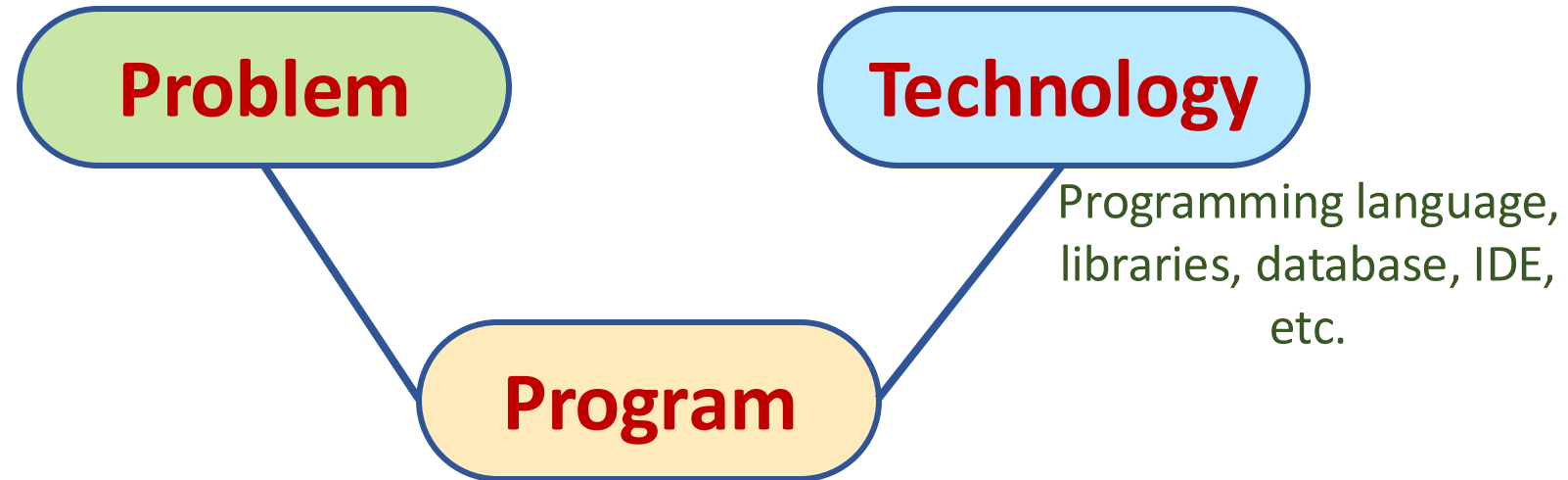
Program

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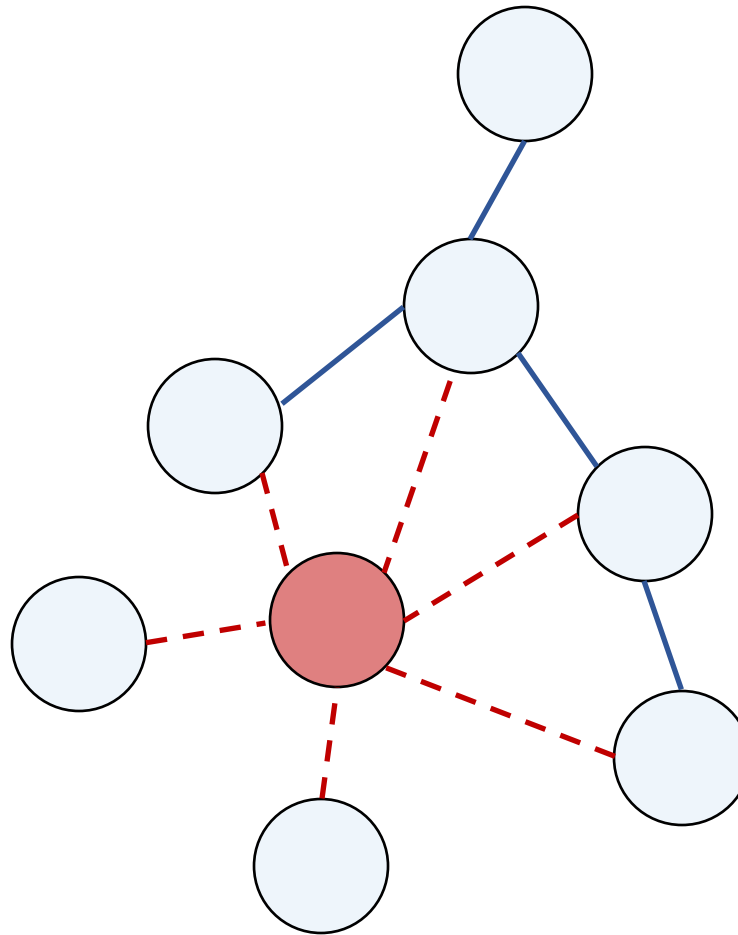
The shaded node interacts, in some ways, with
the linked nodes shown by the dotted line

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.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

106

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

DeviceInventory
laptops
tablets
phones
device_assignment
addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment

(a)

DeviceInventory
laptops
tablets
phones
device_assignment
addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment
printInventory
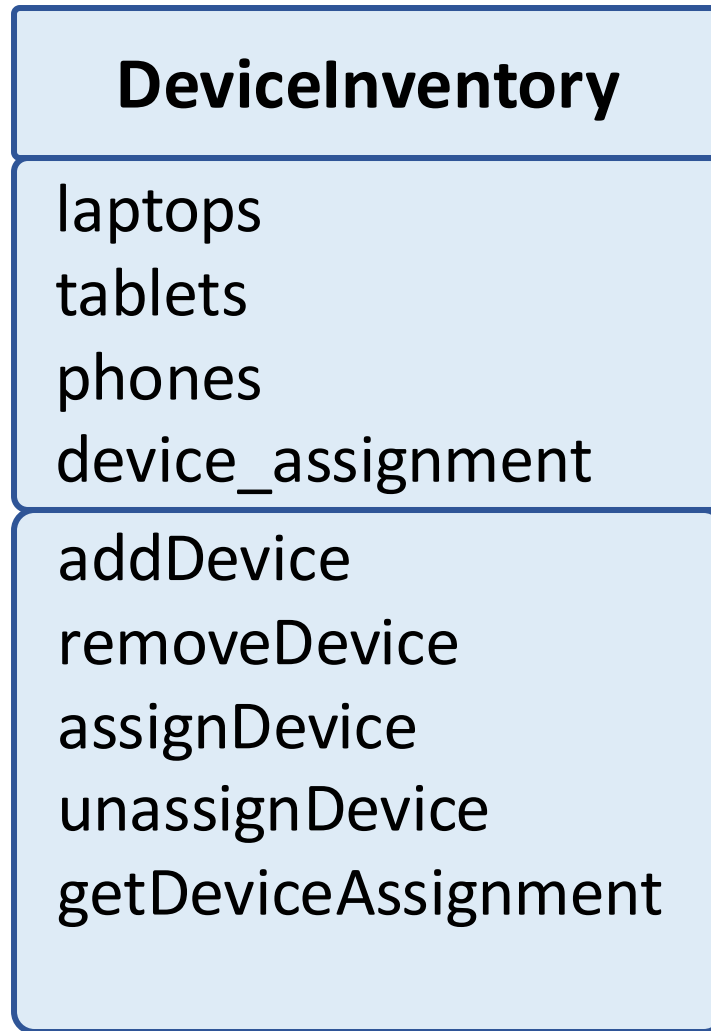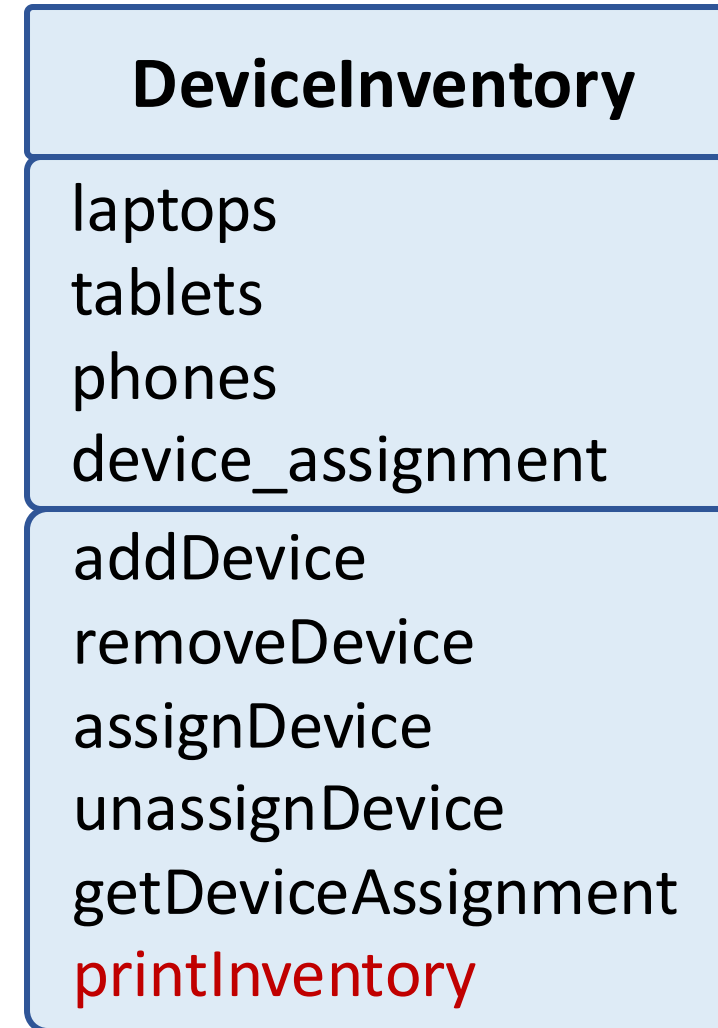
(b)

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

DeviceInventory

laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment

InventoryReport

report_data
report_format

updateData
updateFormat
print
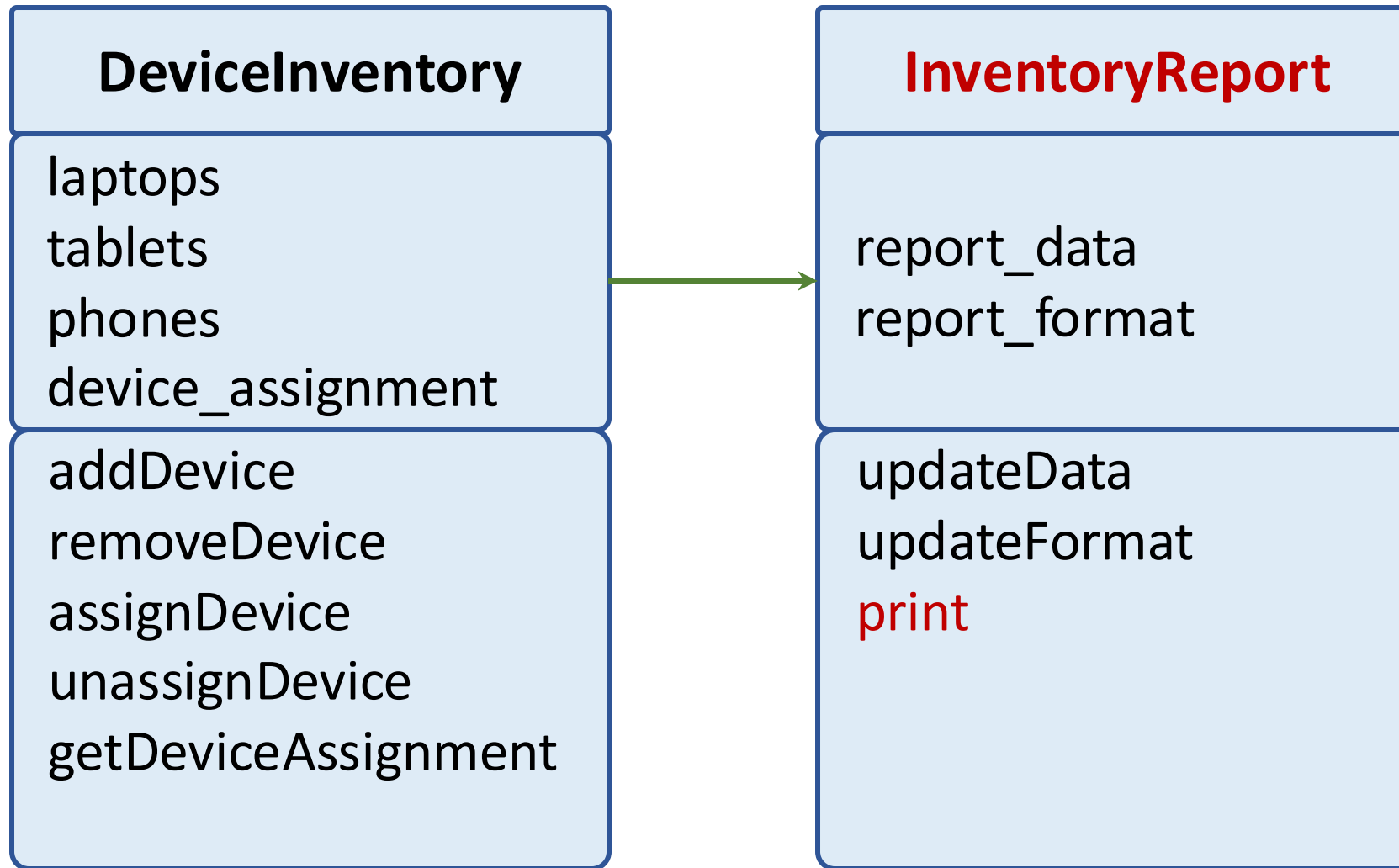
Avoid deeply nested conditional statements

- **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

```python
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

```python
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
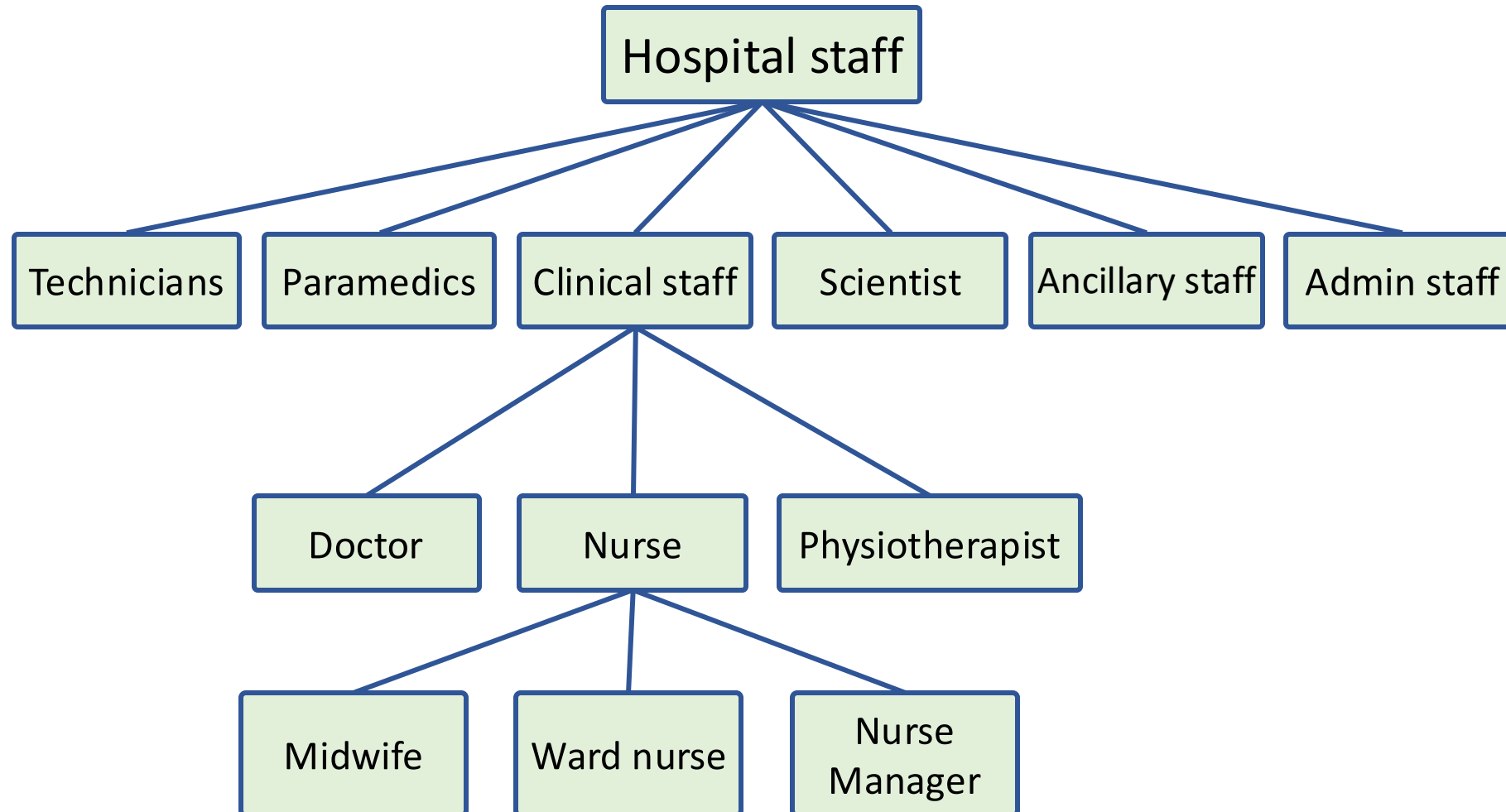
Design pattern definition

- **Definition**

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

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

121

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

Start

① Identify code 'smell'

② Identify refactoring strategy

③ Make small improvement until strategy completed

④ Run automated code tests
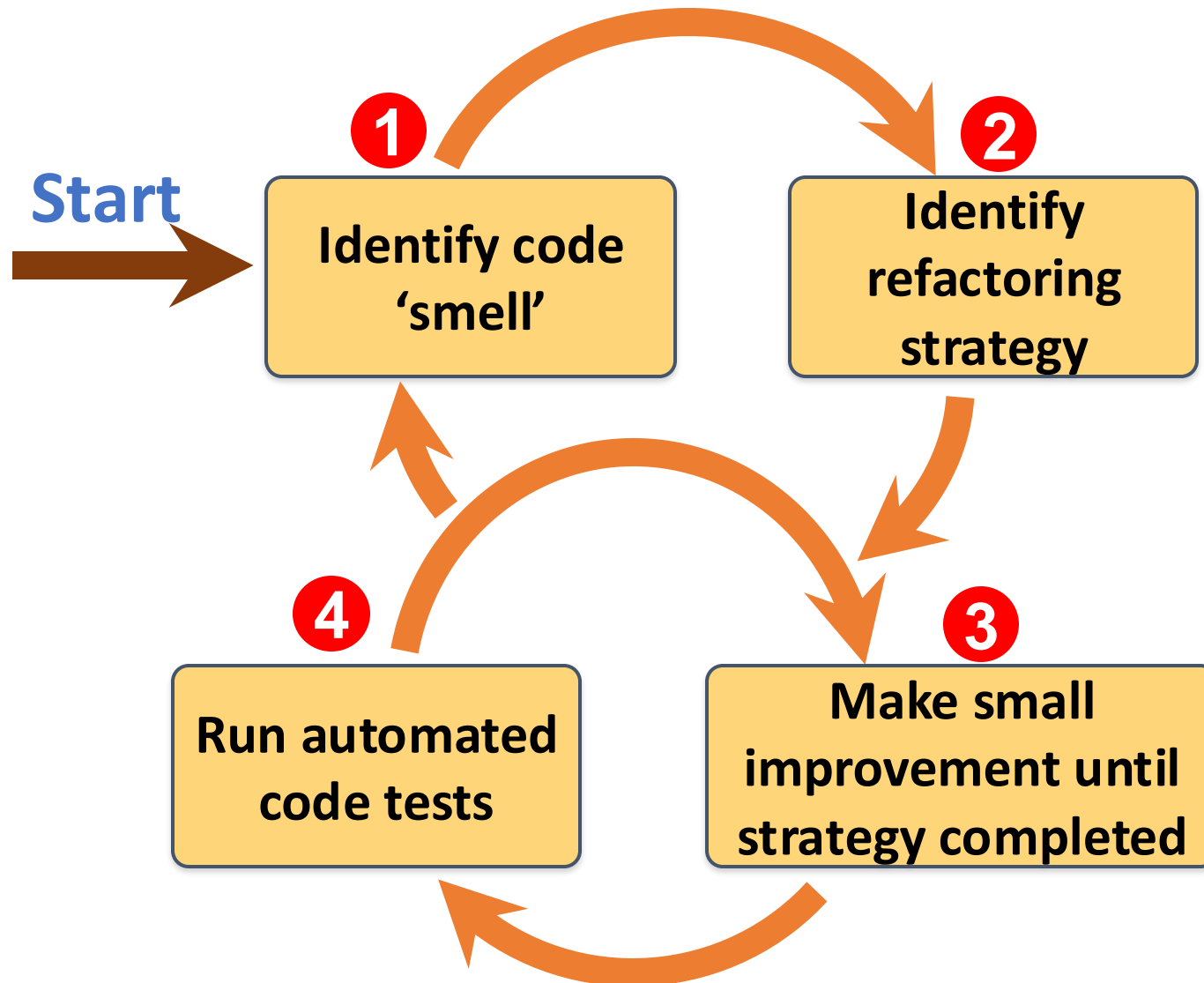
Code smells

- **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

Normal processing

Exception raised

Normal processing

Exit

Exception-handling block

Exception-handling code

Exception re-raised or abnormal exit
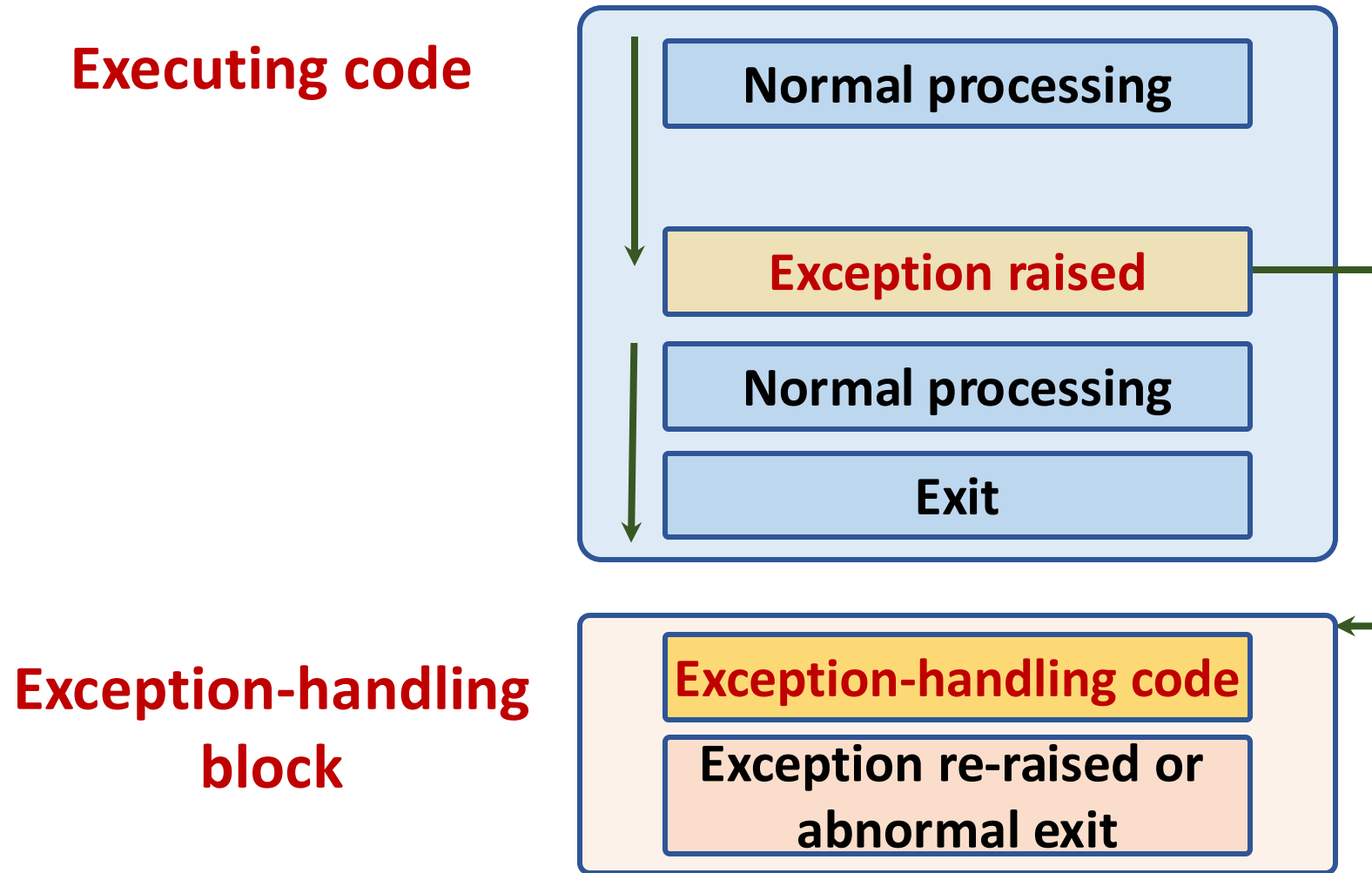
Python
try: except: finally:

```python
try:
  f = open("file1.txt")
  f.write("Hello World")
except:
  print("writing file error!")
finally:
  f.close()
```

Auto-save and activity logging
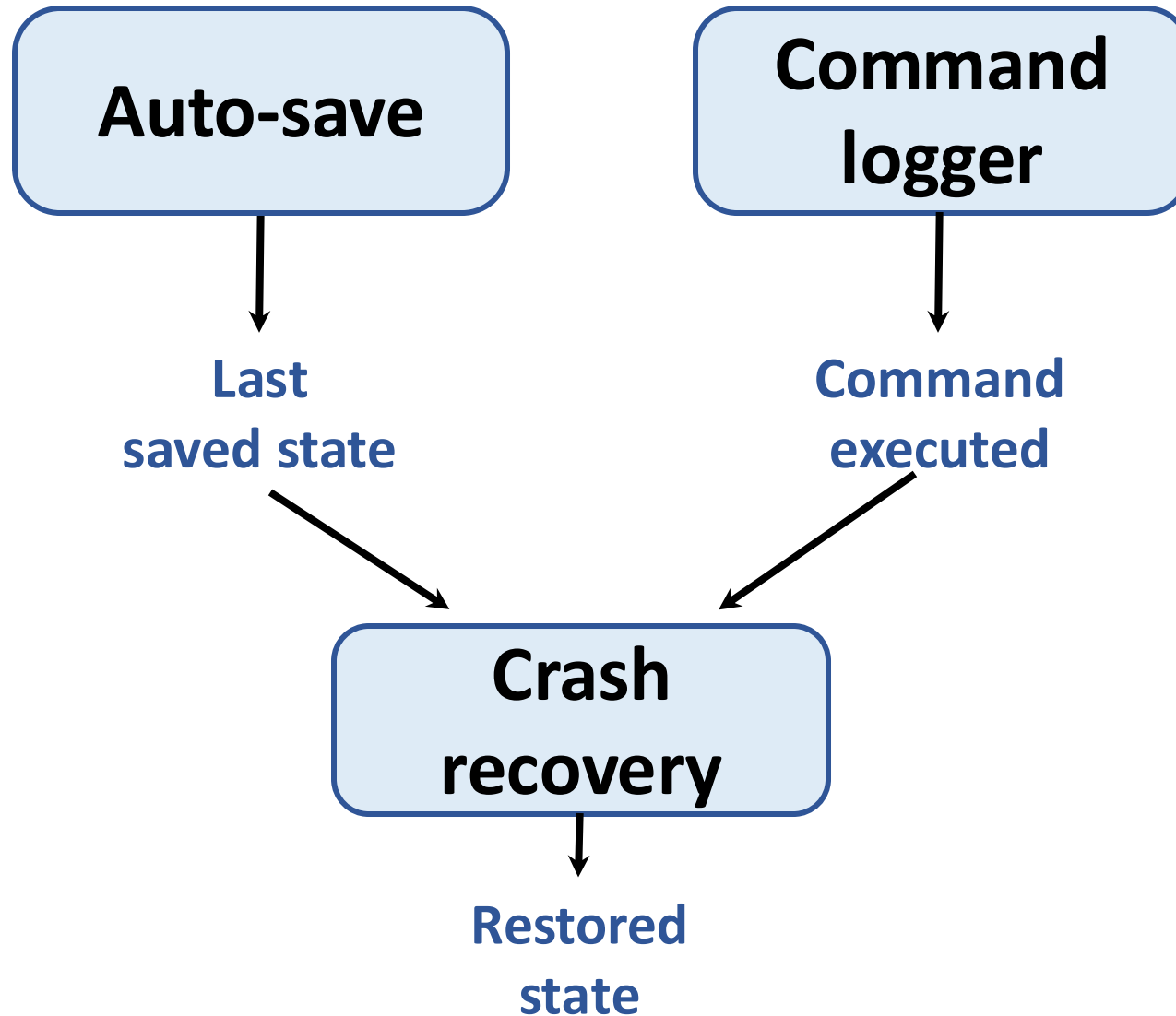
Summary

- **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.

Testing:
Functional testing,
Test automation,
Test-driven development,
and Code reviews

Outline

- **Software testing**

- **Functional testing**

- **Test automation**

- **Test-driven development**

- **Code reviews**

Software testing

- **Software testing is a process in which you execute your program using data that simulates user inputs.**

- **You observe its behaviour to see whether or not your program is doing what it is supposed to do.**

  - **Tests pass if the behaviour is what you expect.**
    Tests fail if the behaviour differs from that expected.

  - **If your program does what you expect, this shows that for the inputs used, the program behaves correctly.**

- **If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.**

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

145

Program bugs

- **If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.**

- **There are two causes of program bugs:**

  - **Programming errors**

    - You have accidentally included faults in your program code.
      For example: 'off-by-1' error

  - **Understanding errors**

    - You have misunderstood or have been unaware of some of the details of what the program is supposed to do.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

146

Types of testing

Functional testing	Test the functionality of the overall system.
User testing	Test that the software product is useful to and usable by end-users.
Performance and load testing	Test that the software works quickly and can handle the expected load placed on the system by its users.
Security testing	Test that the software maintains its integrity and can protect user information from theft and damage.

Functional testing

- **Functional testing** involves developing a **large set of program tests** so that, ideally, **all of a program's code is executed at least once**.

- **The number of tests needed obviously depends on the size and the functionality of the application.**

- **For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.**

Functional testing

- **Functional testing is a staged activity in which you initially test individual units of code.
You integrate code units with other units to create larger units then do more testing.**

- **The process continues until you have created a complete system ready for release.**

Functional testing

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.
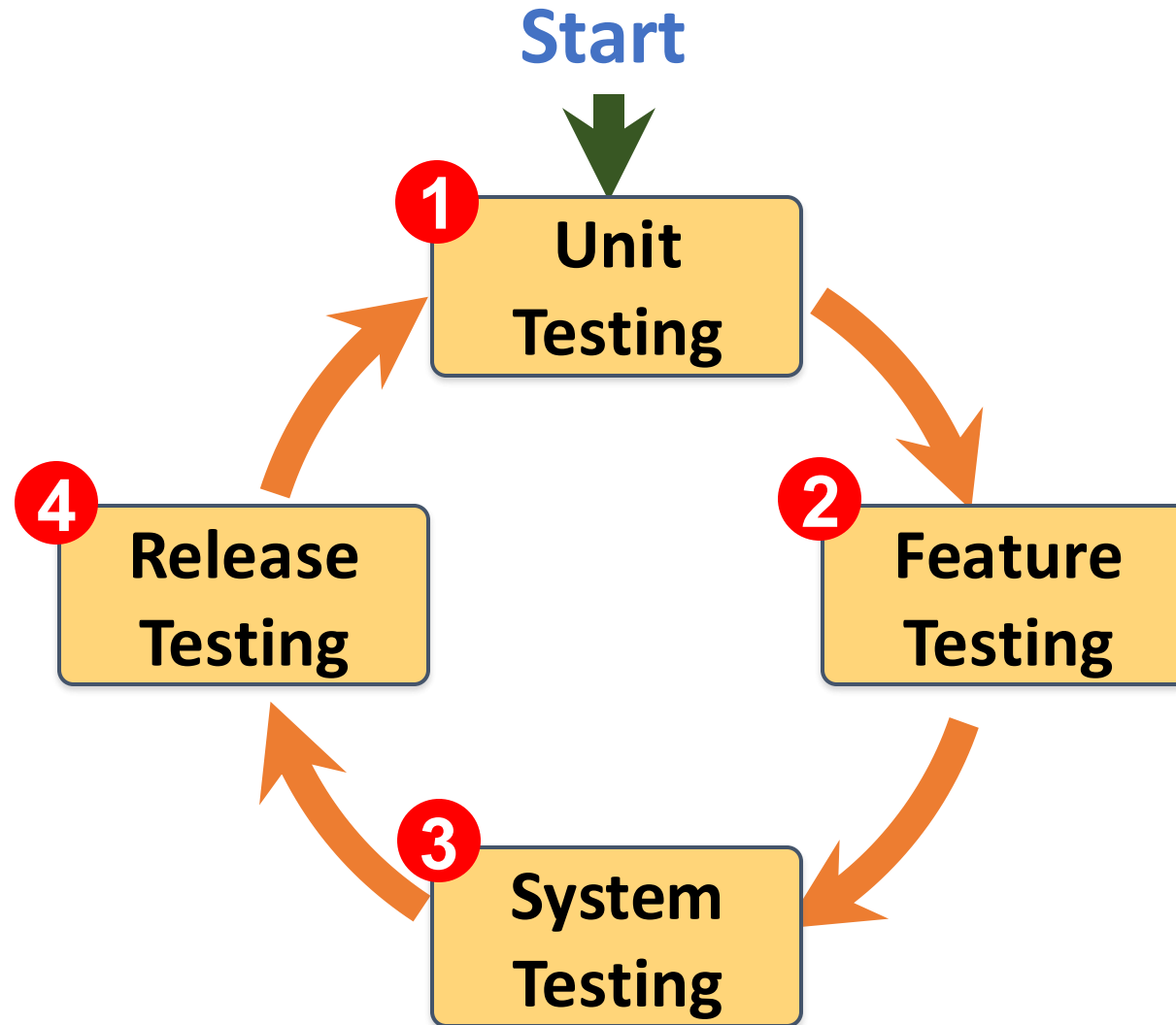
A name checking function

```python
def namecheck(s):

    # Checks that a name only includes alphabetic characters, - or
    # a single quote. Names must be between 2 and 40 characters long
    # quoted strings and -- are disallowed


    namex = r"^[a-zA-Z][a-zA-Z-']{1,39}$"
    if re.match(namex, s):
        if re.search("'.*'", s) or re.search("--", s):
            return False
        else:
            return True
    else:
        return False
```

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

151

Equivalence partitions for the name checking function

- **Correct names 1
  The inputs only includes alphabetic characters and are between 2 and 40 characters long.**

- **Correct names 2
  The inputs only includes alphabetic characters, hyphens or apostrophes and are between 2 and 40 characters long.**

- **Incorrect names 1
  The inputs are between 2 and 40 characters long but include disallowed characters.**

- **Incorrect names 2
  The inputs include allowed characters but are either a single character or are more than 40 characters long.**

Unit testing guidelines (1)

- **Test edge cases**
  If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.

- **Force errors**
  Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.

- **Fill buffers**
  Choose test inputs that cause all input buffers to overflow.

- **Repeat yourself**
  Repeat the same test input or series of inputs several times.

Unit testing guidelines (2)

- **Overflow and underflow**
  If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.

- **Don't forget null and zero**
  If your program uses pointers or strings, always test with null pointers and strings.

- **Keep count**
  When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.

- **One is different**
  If your program deals with sequences, always test with sequences that have a single value.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

154

Feature testing

- **Features** have to be tested to show that the **functionality** is implemented as **expected** and that the **functionality meets the real needs** of users.

  - For example, if your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.

  - You may want to check that it gives users the option to sign up for email information about your product.

Feature testing

- **Normally, a feature that does several things is implemented by multiple, interacting, program units.**

- **These units may be implemented by different developers and all of these developers should be involved in the feature testing process.**

Types of feature test

- ## Interaction tests

  - These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.

  - These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.

  - The integration may also reveal bugs in program units, which were not exposed by unit testing.

- ## Usefulness tests

  - These test that the feature implements what users are likely to want.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

157

User stories for the sign-in with Google feature

- **User registration**
  As a user, I want to be able to login without creating a new account so that I don't have to remember another login id and password.

- **Information sharing**
  As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don't want to share this information.

- **Email choice**
  As a user, I want to be able to choose the types of email that I'll get from you when I register for an account.

Feature tests for sign-in with Google

- **Initial login screen**
  Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the 'Sign-in with Google' link. Test that the login is completed if the user is already logged in to Google.

- **Incorrect credentials**
  Test that the error message and retry screen is displayed if the user inputs incorrect Google credentials.

Feature tests for sign-in with Google

- **Shared information**
  Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is cancelled if the cancel option is chosen.

- **Email opt-in**
  Test that the user is offered a menu of options for email information and can choose multiple items to opt-in to emails. Test that the user is not registered for any emails if no options are selected.

System and release testing

- **System testing** involves testing the system as a whole, rather than the individual system features.

System testing

- **System testing** should focus on **four things**:

  - **Testing to discover if there are unexpected and unwanted interactions between the features in a system.**

  - **Testing to discover if the system features work together effectively to support what users really want to do with the system.**

  - **Testing the system to make sure it operates in the expected way in the different environments where it will be used.**

  - **Testing the responsiveness, throughput, security and other quality attributes of the system.**

Scenario-based testing

- **The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.**

- **Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system.**

- **An end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.**

Choosing a holiday destination
End-to-end pathways

1. User inputs departure airport and chooses to see only direct flights. User quits.

2. User inputs departure airport and chooses to see all flights. User quits.

3. User chooses destination country and chooses to see all flights. User quits.

4. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User quits.

5. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User selects a displayed flight and clicks through to airline website. User returns to holiday planner after booking flight.

Release testing

- **Release testing** is a **type of system testing** where a system that's intended for **release to customers** is tested.

- **Preparing a system for release involves packaging that system for deployment** (e.g. in a container if it is a cloud service) and **installing software and libraries** that are used by your product.

- You must **define configuration parameters** such as the name of a root directory, the database size limit per user and so on.

Release testing and System testing

- **The fundamental differences between release testing and system testing are:**

  - **Release testing** tests the system in its **real operational environment** rather than in a **test environment**. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.

  - **The aim of release testing is to decide if the system is good enough to release**, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.

Test automation

- **Automated testing** is based on the idea that **tests should be executable.**

- An **executable test** includes the **input data** to the unit that is being tested, the **expected result** and a **check** that the unit returns the expected result.

- You run the test and the **test passes** if the **unit returns the expected result.**

- Normally, you should develop **hundreds or thousands of executable tests** for a software product.

Automated testing

Files of executable tests

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

168
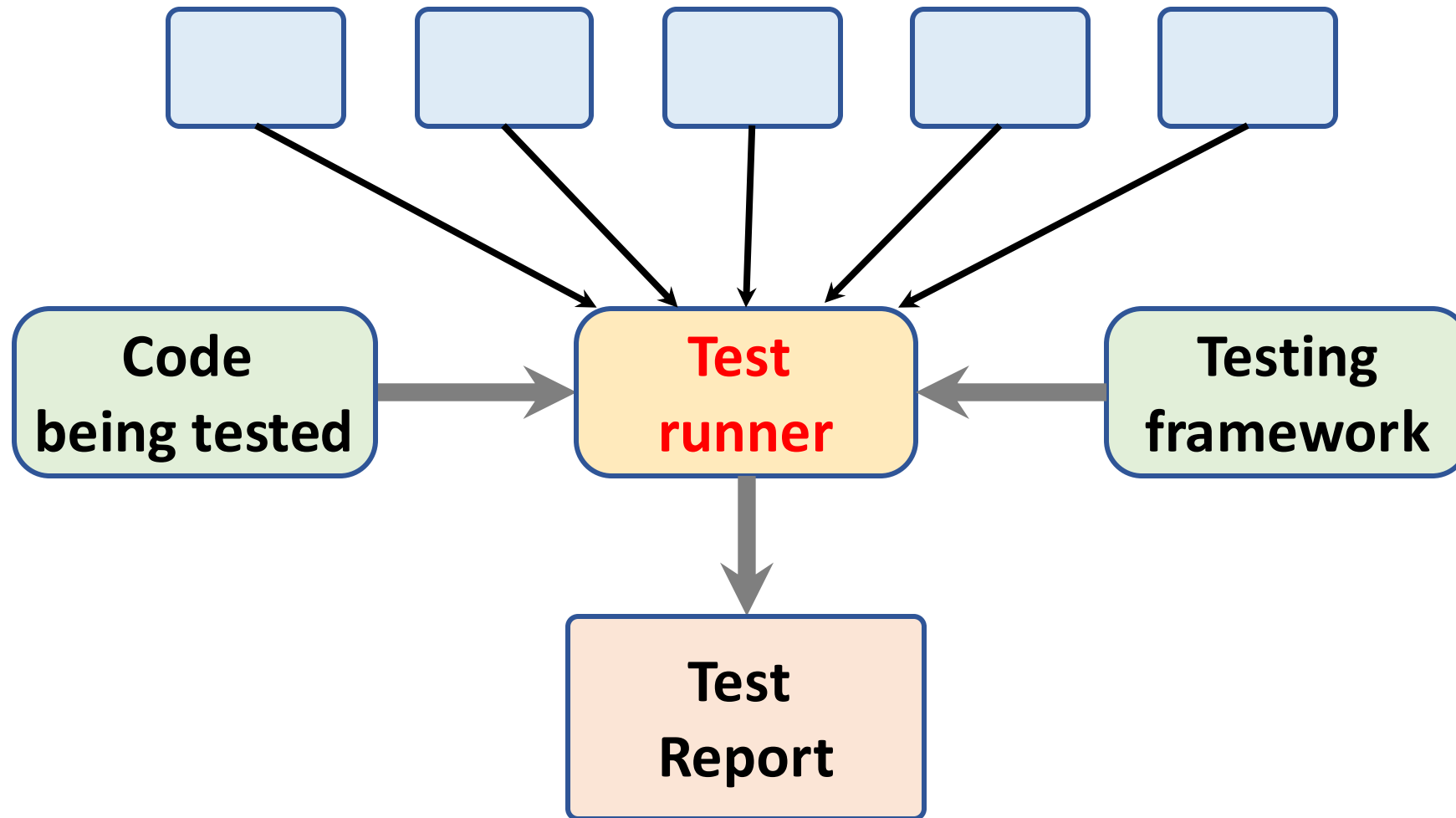
Test methods for an interest calculator

```python
# TestInterestCalculator inherits attributes and methods from the class
# TestCase in the testing framework unittest

class TestInterestCalculator(unittest.TestCase):
    # Define a set of unit tests where each test tests one thing only
    # Tests should start with test_ and the name should explain what is being tested
    def test_zeroprincipal(self):
        #Arrange - set up the test parameters
        p = 0; r = 3; n = 31
        result_should_be = 0
        #Action - Call the method to be tested
        interest = interest_calculator (p, r, n)
        #Assert - test what should be true
        self.assertEqual(result_should_be, interest)


    def test_yearly_interest(self):
        #Arrange - set up the test parameters
        p = 17000; r = 3; n = 365
        #Action - Call the method to be tested
        result_should_be = 270.36
        interest = interest_calculator(p, r, n)
        #Assert - test what should be true
        self.assertEqual(result_should_be, interest)
```

Automated tests

- **It is good practice to
  structure automated tests into three parts:**

  1. **Arrange**

     - You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

  2. **Action**

     - You call the unit that is being tested with the test parameters.

  3. **Assert**

     - You make an assertion about what should hold if the unit being tested has executed successfully.
       AssertEquals: checks if its parameters are equal.

Executable tests for the namecheck function (1)

```python
import unittest
from RE_checker import namecheck

class TestNameCheck (unittest.TestCase):

    def test_alphaname (self):
        self.assertTrue (namecheck ('Sommerville'))

    def test_doublequote (self):
        self.assertFalse (namecheck ("Thisis'maliciouscode'"))

    def test_namestartswithhyphen (self):
        self.assertFalse (namecheck ('-Sommerville'))

    def test_namestartswithquote (self):
        self.assertFalse (namecheck ("'Reilly"))

    def test_nametoolong (self):
        self.assertFalse (namecheck ('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))

    def test_nametooshort (self):
        self.assertFalse (namecheck ('S'))
```

Executable tests for the namecheck function (2)

```python
def test_namewithdigit (self):
    self.assertFalse (namecheck('C-3PO'))

def test_namewithdoublehyphen (self):
    self.assertFalse (namecheck ('--badcode'))

def test_namewithhyphen (self):
    self.assertTrue (namecheck ('Washington-Wilson'))

def test_namewithinvalidchar (self):
    self.assertFalse (namecheck('Sommer_ville'))

def test_namewithquote (self):
    self.assertTrue (namecheck ("O'Reilly"))

def test_namewithspaces (self):
    self.assertFalse (namecheck ('Washington Wilson'))

def test_shortname (self):
    self.assertTrue ('Sx')

def test_thiswillfail (self)
    self.assertTrue (namecheck ("O Reilly"))
```

Code to run unit tests from files

```python
import unittest

loader = unittest.TestLoader()

#Find the test files in the current directory

tests = loader.discover('.')

#Specify the level of information provided by the test runner

testRunner = unittest.runner.TextTestRunner(verbosity=2)
testRunner.run(tests)
```

The test pyramid

Increased automation
Reduced costs

System tests

Feature tests

Unit tests
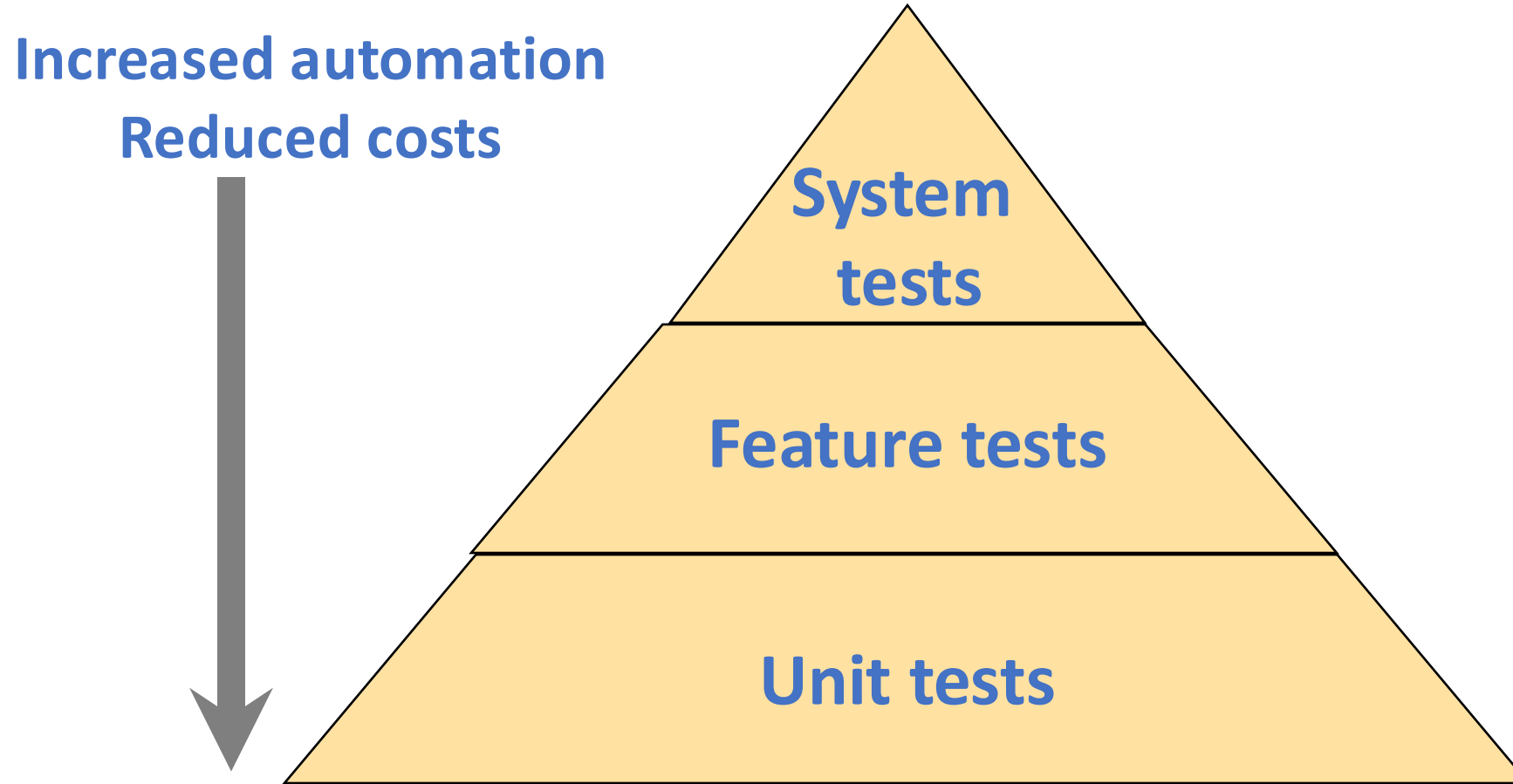
Feature editing through an API
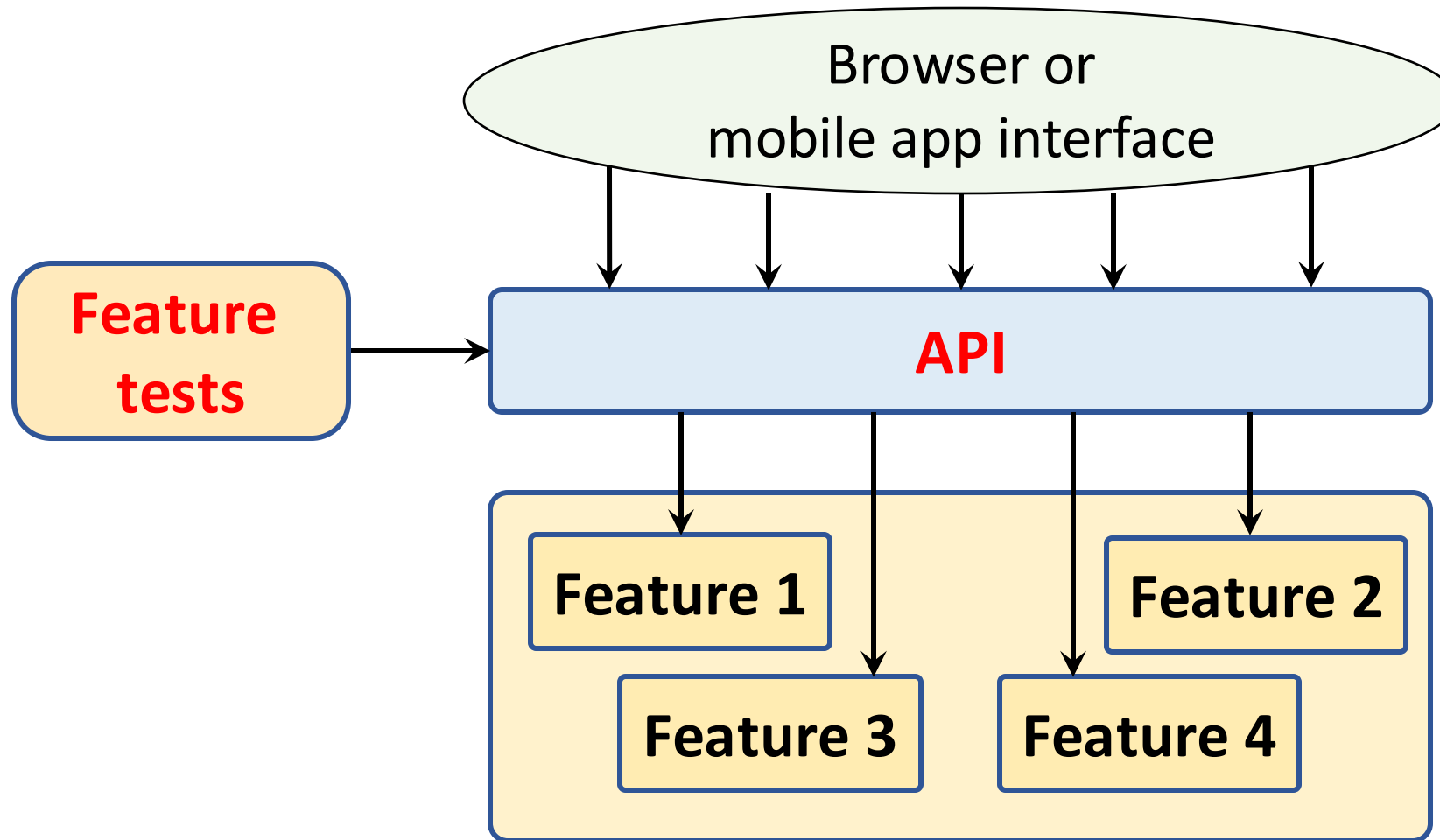
Interaction recording and playback

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

176
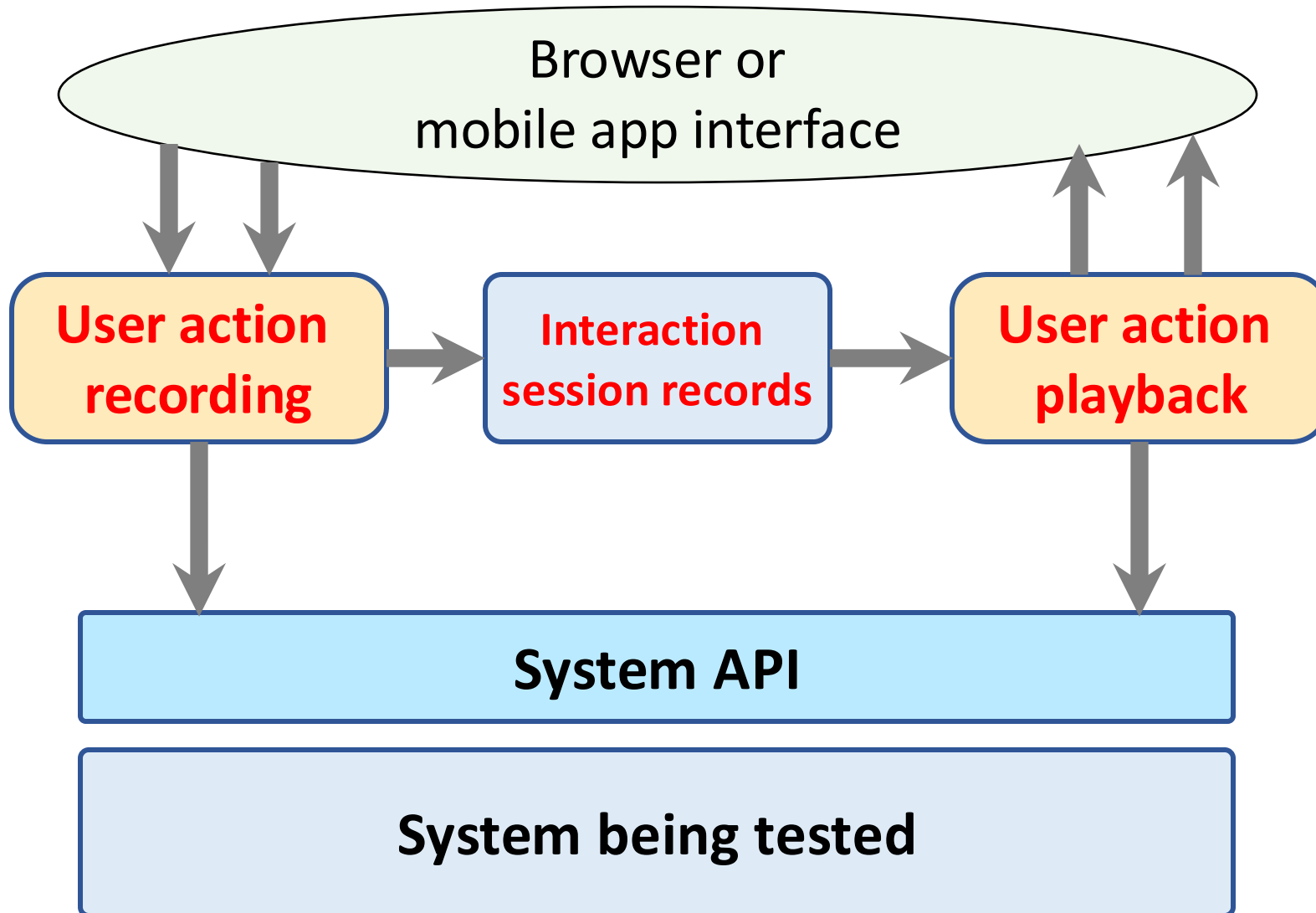
Test-driven development (TDD)

- **Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.**

- **It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.**

- **Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.**

- **Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.**

Test-driven development (TDD)

Start → ① **Identify new functionality**

② **Identify partial implementation of functionality**

③ **Write code stub that will fail test**

④ **Run all automated test**

⑤ **Implement code that should cause failing test to pass**

⑥ **Run all automated test**

⑦ **Refactor code if required**

Functionality complete

Functionality incomplete

Test failure

All tests pass
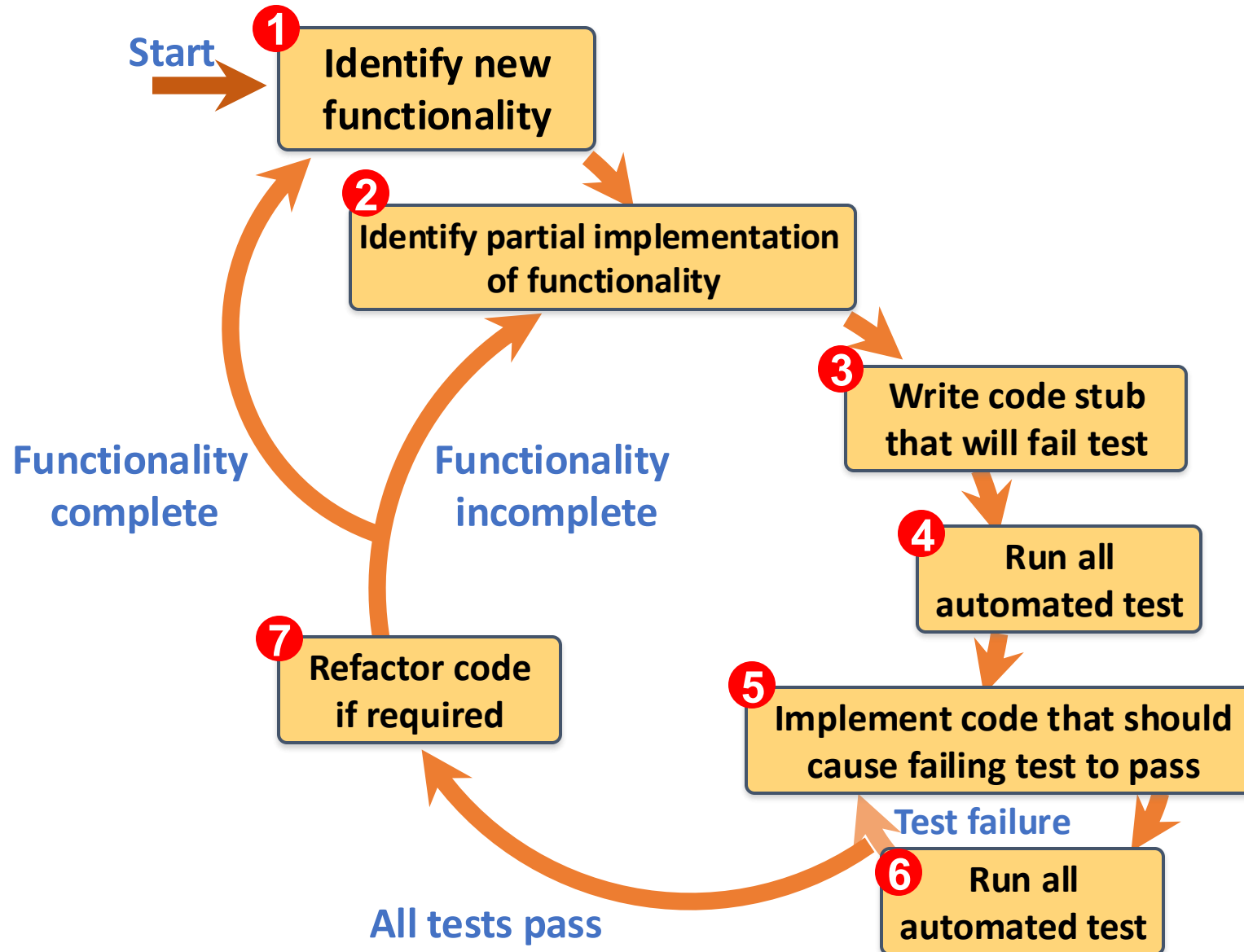
Stages of test-driven development

1. **Identify partial implementation**
   Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.

2. **Write mini-unit tests**
   Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.

3. **Write a code stub that will fail test**
   Write incomplete code that will be called to implement the mini-unit. You know this will fail.

4. **Run all existing automated tests**
   All previous tests should pass. The test for the incomplete code should fail.

Stages of test-driven development

5. **Implement code that should cause the failing test to pass**
    Write code to implement the mini-unit, which should cause it to operate correctly

6. **Rerun all automated tests**
    If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.

7. **Refactor code if necessary**
    If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Benefits of test-driven development

- **It is a systematic approach to testing in which tests are clearly linked to sections of the program code.**

  - **This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code.**

- **The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.**

- **Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.**

- **TDD leads to simpler code as programmers only write code that's necessary to pass tests. They don't over-engineer their code with complex features that aren't needed.**

Reasons for not using TDD

- **TDD discourages radical program change**

- **I focused on the tests rather than the problem I was trying to solve**

- **I spent too much time thinking about implementation details rather than the programming problem**

- **It is hard to write 'bad data' tests**

Security testing

- **Security testing** aims to **find vulnerabilities** that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.

- The tests should demonstrate that the system can **resist attacks** on its **availability, attacks** that try to **inject malware** and **attacks** that try to **corrupt or steal users' data and identity**.

- **Comprehensive security testing** requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

Risk-based security testing

- A **risk-based approach to security testing** involves identifying common risks and developing tests to demonstrate that the system protects itself from these risks.

- **You may also use automated tools that scan your system to check for known vulnerabilities**, such as unused HTTP ports being left open.

- **Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable.**

- **It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files.**
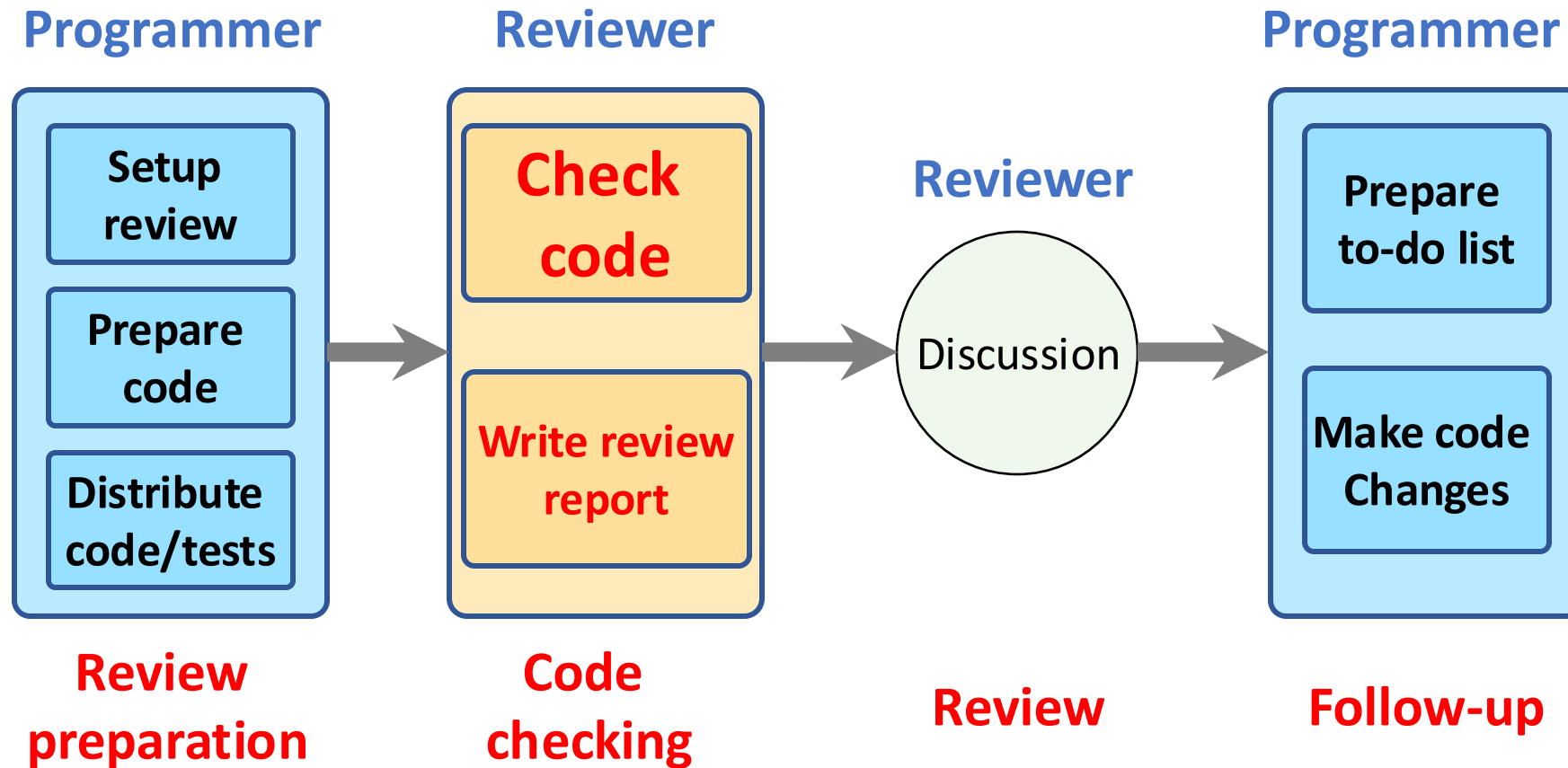
Risk analysis

- **Once you have identified security risks, you then analyze them to assess how they might arise.**

  - **The user has set weak passwords that can be guessed by an attacker.**

  - **The system's password file has been stolen and passwords discovered by attacker.**

- **Develop tests to check some of these possibilities.**

  - **For example, you might run a test to check that the code that allows users to set their passwords always checks the strength of passwords.**

Code reviews

- **Code reviews** involve one or more people **examining** the **code** to **check** for **errors** and **anomalies** and **discussing issues** with the developer.

- If problems are identified, it is the developer's responsibility to **change the code to fix** the problems.

- **Code reviews complement testing**. They are effective in finding bugs that arise through misunderstandings and bugs that may only arise when unusual sequences of code are executed.

- Many software companies insist that all code has to go through a process of **code review** before it is integrated into the product codebase.

Code reviews

Programmer

- Setup review
- Prepare code
- Distribute code/tests

Review preparation

Reviewer

- **Check code**
- **Write review report**

Code checking

Reviewer

Discussion

Review

Programmer

- Prepare to-do list
- Make code Changes

Follow-up

Summary

- **The aim of program testing is to find bugs and to show that a program does what its developers expect it to do.**

- **Four types of testing that are relevant to software products are functional testing, user testing, load and performance testing and security testing.**

- **Unit testing involves testing program units such as functions or class methods that have a single responsibility.**

- **Feature testing focuses on testing individual system features.**

Summary

- **System testing** tests the system as a whole to check for unwanted interactions between features and between the system and its environment.

- **Identifying equivalence partitions**, in which all inputs have the same characteristics, and choosing test inputs at the boundaries of these partitions, is an effective way of finding bugs in a program.

- **User stories** may be used as a basis for deriving feature tests.

Summary

- **Test automation** is based on the idea that tests should be executable. You develop a set of executable tests and run these each time you make a change to a system.

- **The structure of an automated unit test** should be **arrange-action-assert**. You set up the test parameters, call the function or method being tested, and make an assertion of what should be true after the action has been completed.

Summary

- **Test-driven development** is an approach to development where executable tests are written before the code. Code is then developed to pass the tests.

- **A disadvantage of test-driven development is that programmers focus on the detail of passing tests rather than considering the broader structure of their code and algorithms used.**

Summary

- **Security testing** may be risk driven where a list of security risks is used to identify tests that may identify system vulnerabilities.

- **Code reviews** are an effective supplement to testing. They involve people checking the code to comment on the code quality and to look for bugs.

DevOps and Code Management: Code management and DevOps automation

Outline

- **Source code management**

- **DevOps automation**
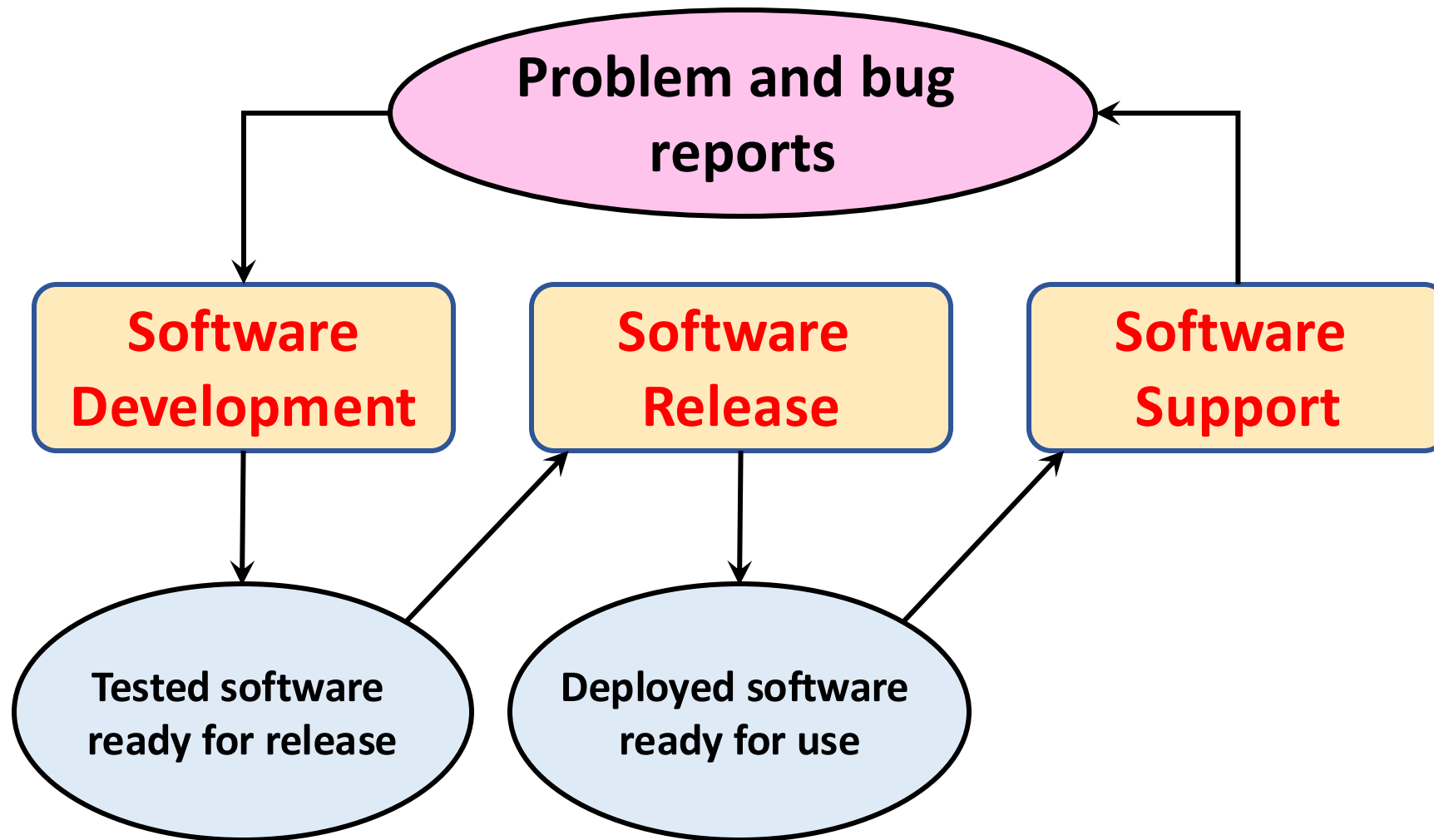
- **DevOps measurement**

DevOps

Source: The DevOps Institute (2022), https://thedevopsinstitute.com/?page_id=22
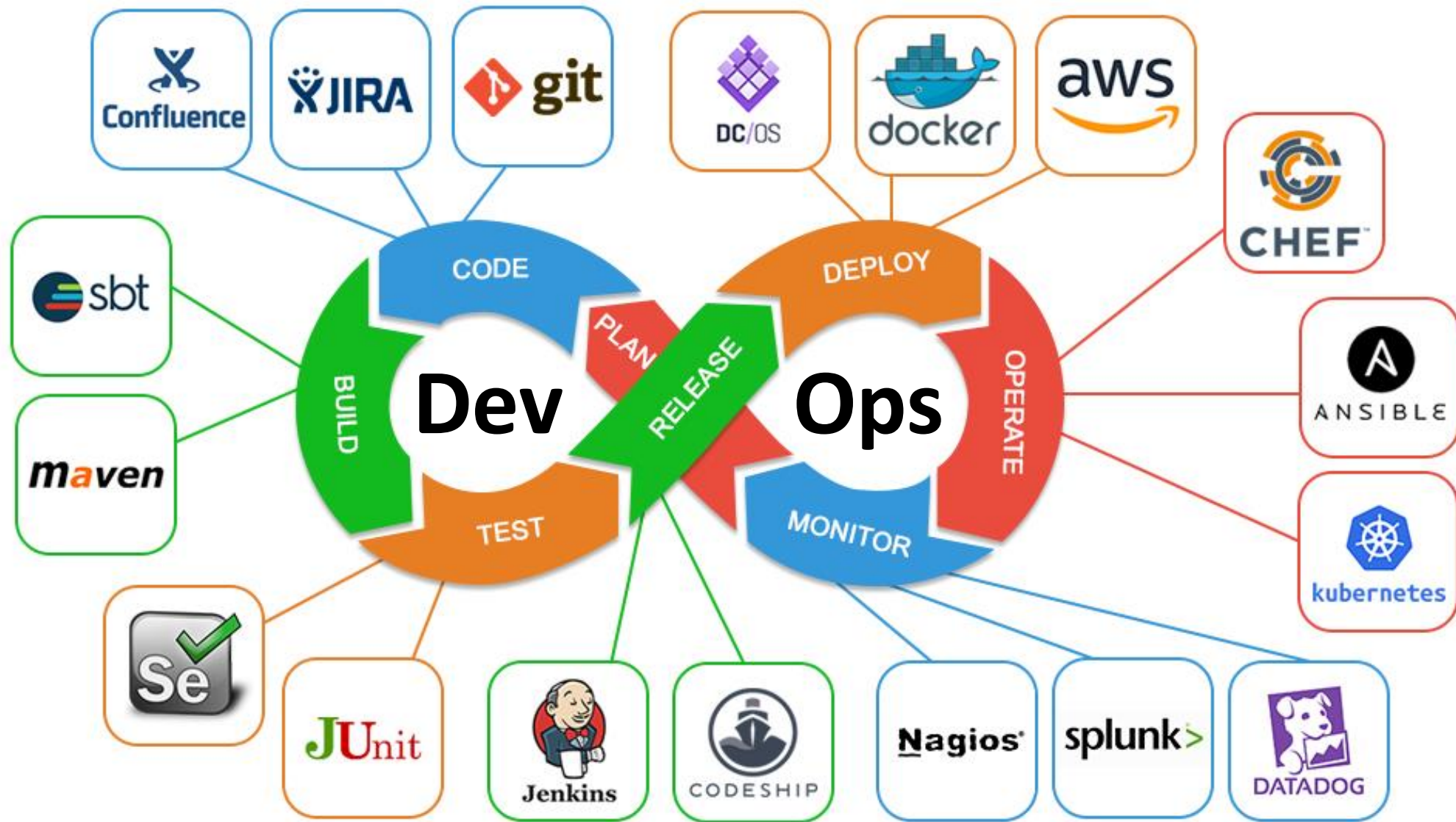
Software support

- **Traditionally, separate teams were responsible software development, software release and software support.**

- **The development team passed over a 'final' version of the software to a release team.**

  - **Built a release version, tested this and prepared release documentation before releasing the software to customers.**

- **A third team was responsible for providing customer support.**

  - **The original development team were sometimes also responsible for implementing software changes.**

  - **Alternatively, the software may have been maintained by a separate 'maintenance team'.**

Software Development, release and support

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

197

DevOps

- **There are inevitable delays and overheads in the traditional support model.**

- **To speed up the release and support processes, an alternative approach called DevOps (Development + Operations) has been developed.**

DevOps

- **Three factors** led to the development and widespread adoption of **DevOps**:

  - **Agile software engineering** reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.

  - **Amazon re-engineered** their **software** around **services** and introduced an approach in which a service was developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.

  - It became possible to **release software as a service, running on a public or private cloud**. Software products did not have to be released to users on physical media or downloads.

DevOps
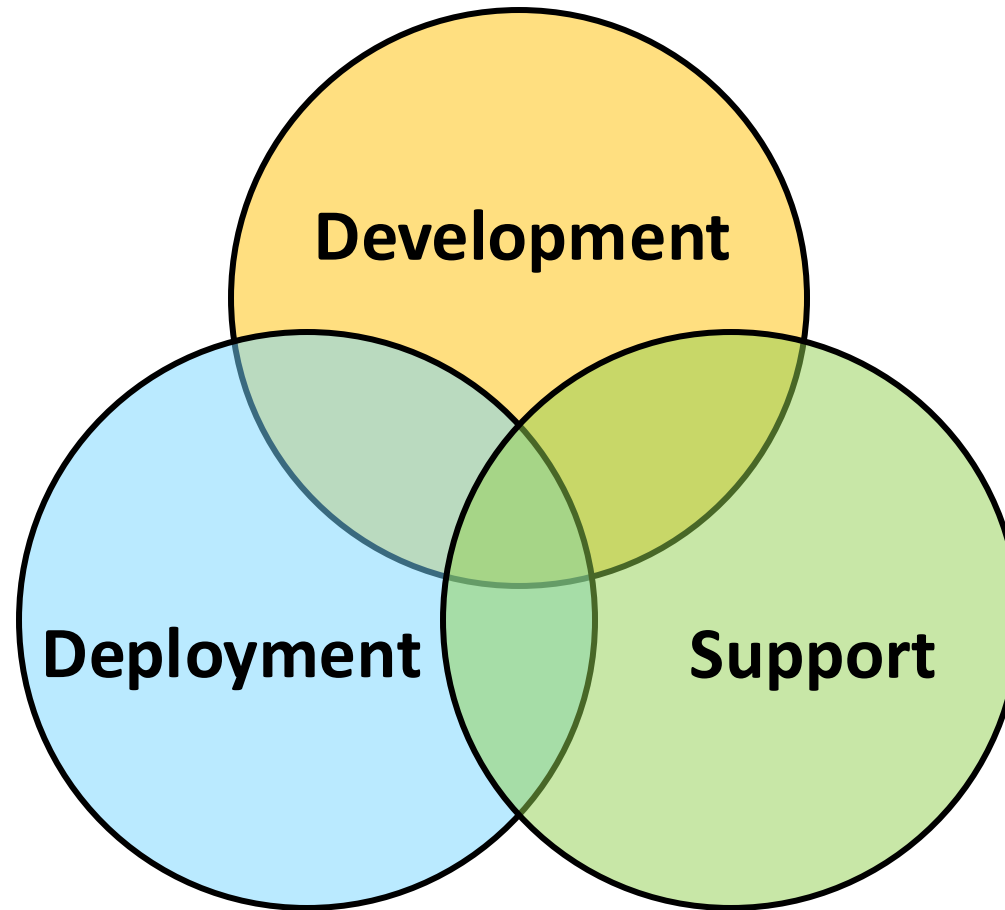
Multi-skilled DevOps team

DevOps principles

- **Everyone is responsible for everything**
  All team members have joint responsibility for developing, delivering and supporting the software.

- **Everything that can be automated should be automated**
  All activities involved in testing, deployment and support should be automated if it is possible to do so. There should be minimal manual involvement in deploying software.

- **Measure first, change later**
  DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Benefits of DevOps

Faster deployment	Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
Reduced risk	The increment of functionality in each release is small so there is less chance of feature interactions and other changes causing system failures and outages.
Faster repair	DevOps teams work together to get the software up and running again as soon as possible.
More productive teams	DevOps teams are happier and more productive than the teams involved in the separate activities.

Code management

- **Code management is a set of software-supported practices that is used to manage an evolving codebase.**

- **During the development of a software product, the development team will probably create tens of thousands of lines of code and automated tests.**

- **These will be organized into hundreds of files. Dozens of libraries may be used, and several, different programs may be involved in creating and running the code.**
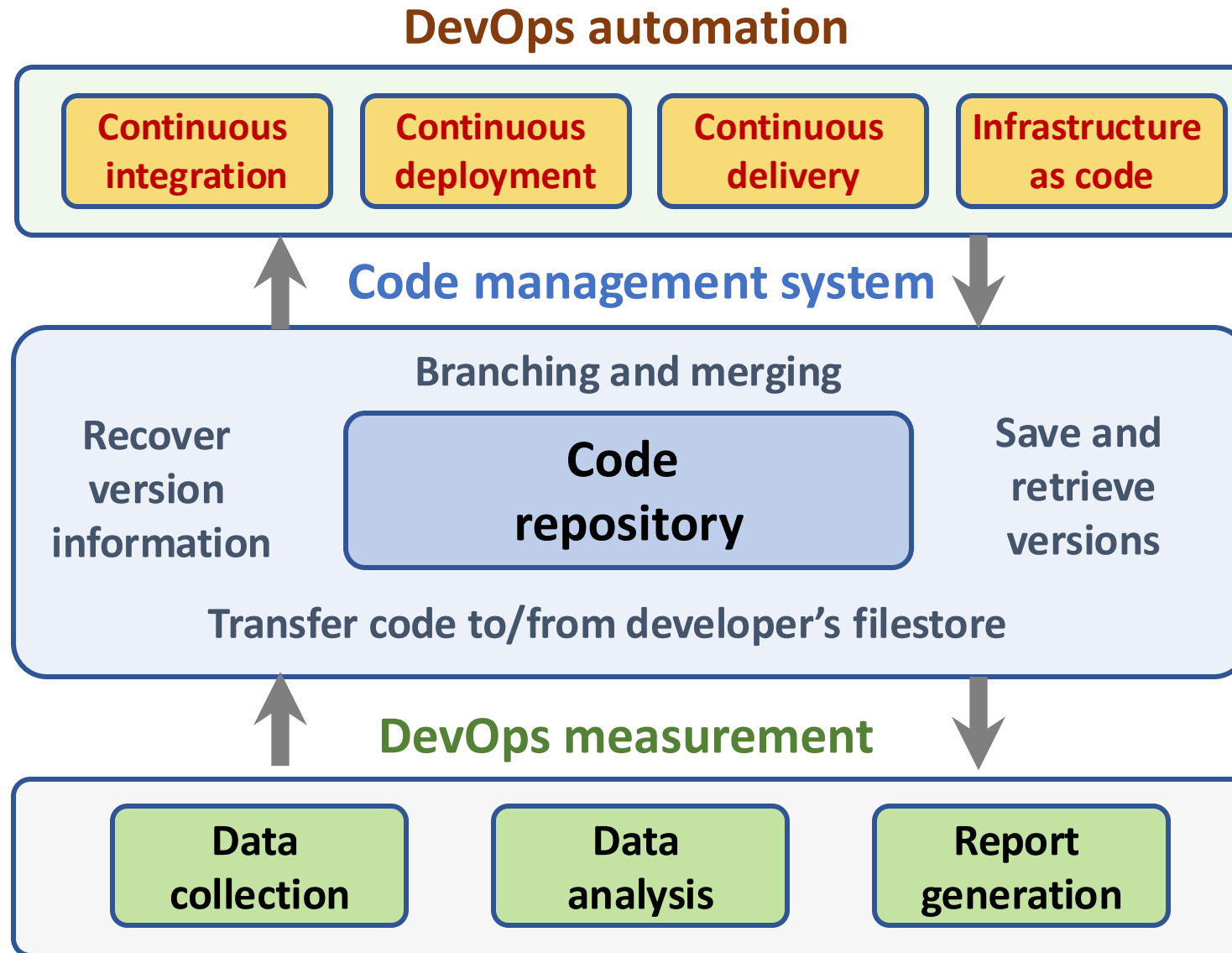
Code management

- **You need code management to ensure that changes made by different developers do not interfere with each other, and to create different product versions.**

- **Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.**

Code management and DevOps

- **Source code management**, combined with **automated system building**, is essential for **professional software engineering**.

- In companies that use **DevOps**, a **modern code management system** is a fundamental requirement for '**automating everything**'.

- Not only does it **store** the **project code** that is ultimately **deployed**, it also stores all other information that is used in **DevOps processes**.

- **DevOps automation and measurement tools** all interact with the code management system

Code management and DevOps

DevOps automation

| Continuous integration | Continuous deployment | Continuous delivery | Infrastructure as code |

Code management system

Branching and merging

Recover version information

Code repository

Save and retrieve versions

Transfer code to/from developer's filestore

DevOps measurement

| Data collection | Data analysis | Report generation |

Code management fundamentals

- **Code management systems** provide a set of features that support four general areas:

- **Code transfer**
  - Developers take code into their personal file store to work on it then return it to the shared code management system.

- **Version storage and retrieval**
  - Files may be stored in several different versions and specific versions of these files can be retrieved.

- **Merging and branching**
  - Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.

- **Version information**
  - Information about the different versions maintained in the system may be stored and retrieved

Code repository

- **All source code management systems have the general form with a shared repository and a set of features to manage the files in that repository:**

  - **All source code files and file versions are stored in the repository, as are other artefacts such as configuration files, build scripts, shared libraries and versions of tools used.**

  - **The repository includes a database of information about the stored files such as version information, information about who has changed the files, what changes were made at what times, and so on.**

Code repository

- **Files can be transferred to and from the repository and information about the different versions of files and their relationships may be updated.**

  - Specific versions of files and information about these versions can always be retrieved from the repository.

Features of
code management systems

Version and release identification

Change history recording

Independent development

Project support

Storage management

Git

- In 2005, **Linus Torvalds**, the developer of Linux, revolutionized **source code management** by developing a **distributed version control system (DVCS)** called **Git** to manage the code of the Linux kernel.

- This was geared to supporting **large-scale open source development**. It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.

- Instead of only keeping the copies of the files that users are working on, **Git maintains a clone of the repository** on every user's computer

Benefits of
# distributed code management

Resilience

Speed

Flexibility

Branching and merging

- **Branching and merging** are fundamental ideas that are supported by all code management systems.

- A **branch** is an **independent, stand-alone version** that is created when a developer wishes to change a file.

- The changes made by developers in their own branches may be **merged** to create a new shared branch.

- The repository ensures that **branch files** that have been changed cannot overwrite repository files without a **merge operation**.

Branching and merging

Feature experiment branch

Alice

Bob

Merge

Master branch

Bug fix branch
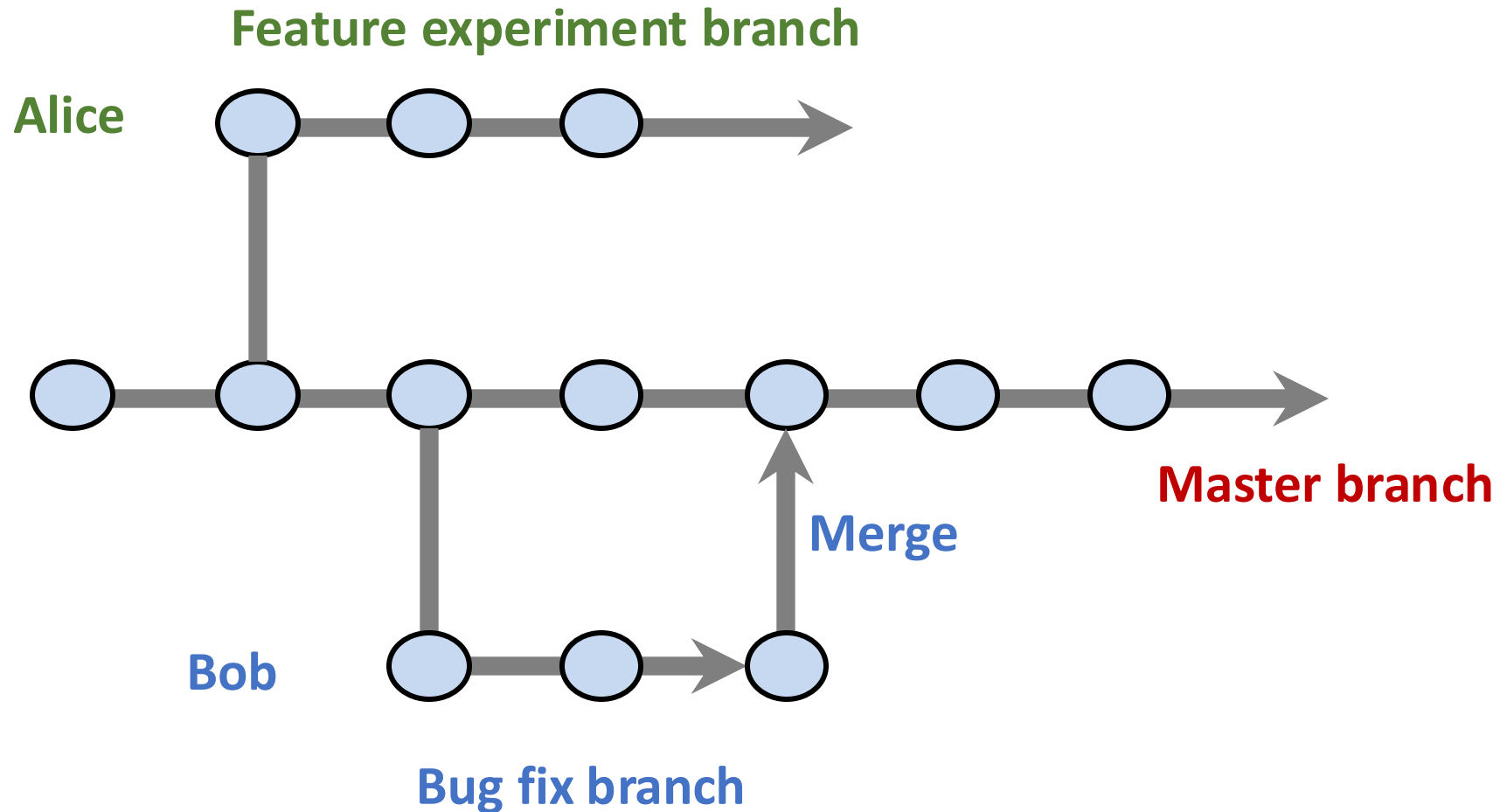
DevOps automation

- **By using DevOps with automated support, you can dramatically reduce the time and costs for integration, deployment and delivery.**

- **Everything that can be, should be automated is a fundamental principle of DevOps.**

- **As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.**

- **Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.**

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

216

Aspects of DevOps automation

Continuous integration	Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
Continuous delivery	A simulation of the product's operating environment is created and the executable software version is tested.
Continuous deployment	A new release of the system is made available to users every time a change is made to the master branch of the software.
Infrastructure as code	Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform.

Characteristics of infrastructure as code

Visibility

Reproducibility

Reliability

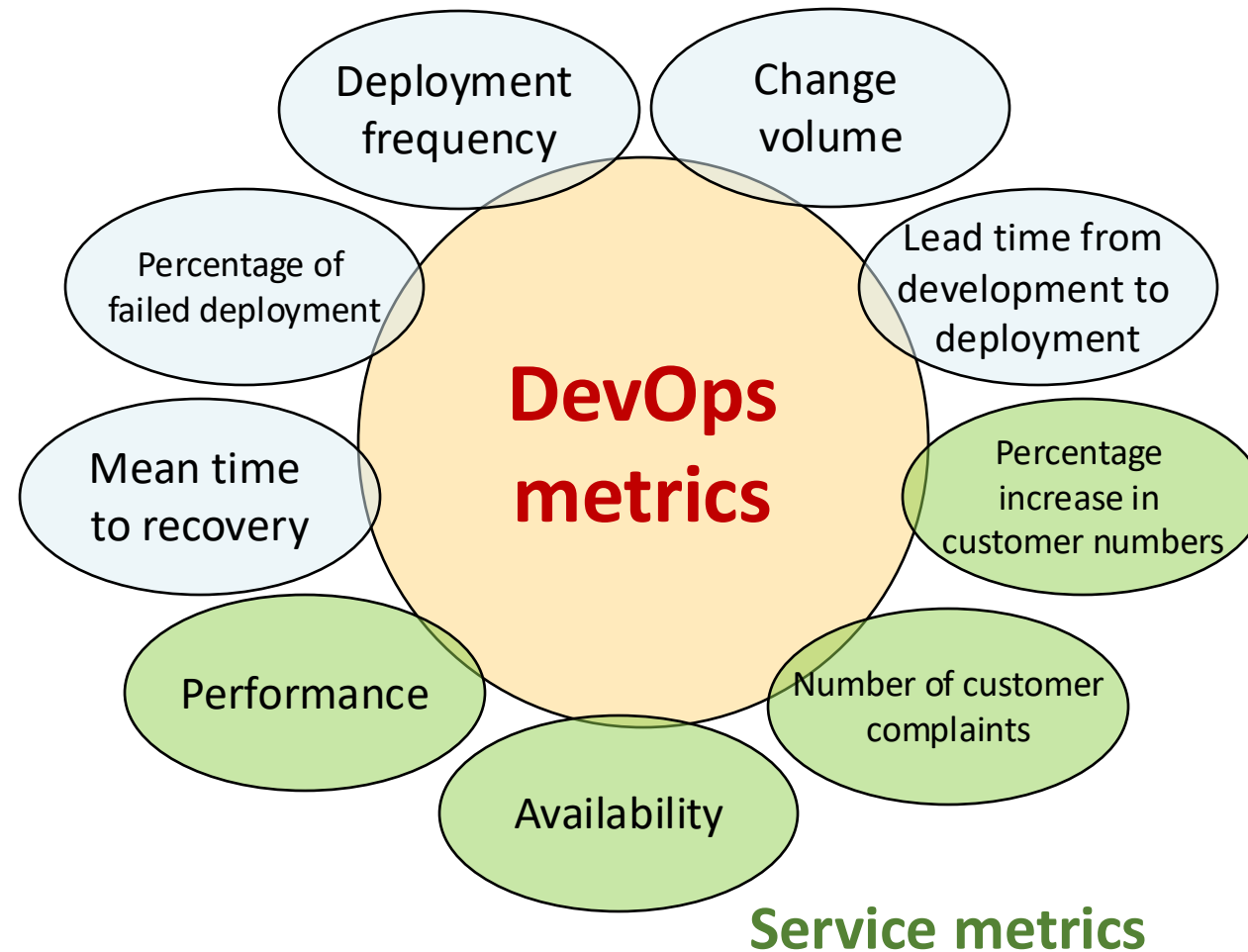**Recovery**

DevOps measurement

- **After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software.**

- **There are four types of software development measurement:**

  - **Process measurement**

  - **Service measurement**

  - **Usage measurement**

  - **Business success measurement**

Automating measurement

- **As far as possible, the DevOps principle of automating everything should be applied to software measurement.**

- **You should instrument your software to collect data about itself and you should use a monitoring system to collect data about your software's performance and availability.**

- **Some process measurements can also be automated.**

  - **However, there are problems in process measurement because people are involved. They work in different ways, may record information differently and are affected by outside influences that affect the way they work.**

Metrics used in the DevOps scorecard

Process metrics

Service metrics

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Summary

- **DevOps** is the **integration of software development and the management** of that software once it has been deployed for use. The same team is responsible for development, deployment and software support.

- **The benefits of DevOps** are **faster deployment, reduced risk, faster repair of buggy code and more productive teams**.

- **Source code management** is essential to avoid changes made by different developers interfering with each other.

Summary

- All **code management systems** are based around a **shared code repository** with a set of features that support code transfer, version storage and retrieval, branching and merging and maintaining version information.

- **Git** is a **distributed code management system** that is the most widely used system for software product development. Each developer works with their own copy of the repository which may be merged with the shared project repository.

Summary

- **DevOps** is the **integration of software development and the management** of that software once it has been deployed for use. The same team is responsible for development, deployment and software support.

- **The benefits of DevOps** are **faster deployment, reduced risk, faster repair of buggy code and more productive teams**.

- **Source code management** is essential to avoid changes made by different developers interfering with each other.

Summary

- **Continuous integration** means that as soon as a change is committed to a project repository, it is integrated with existing code and a new version of the system is created for testing.

- **Automated system building tools** reduce the time needed to compile and integrate the system by only recompiling those components and their dependents that have changed.

- **Continuous deployment** means that as soon as a change is made, the deployed version of the system is automatically updated. This is only possible when the software product is delivered as a cloud-based service.

Summary

- **Infrastructure as code** means that the infrastructure (network, installed software, etc.) on which software executes is defined as a machine-readable model. Automated tools, such as Chef and Puppet, can provision servers based on the infrastructure model.

- **Measurement** is **a fundamental principle of DevOps**. You may make both process and product measurements. **Important process metrics** are **deployment frequency, percentage of failed deployments, and mean time to recovery from failure**.

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.