Software Engineering

Security and Privacy; Reliable Programming; Testing: Test-driven development, and Code reviews

1142SE09
MBA, IM, NTPU (M5010) (Spring 2026)
Wed 2, 3, 4 (9:10-12:00) (B3F17)

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

Min-Yuh Day, Ph.D,
Professor and Director

Institute of Information Management, National Taipei University

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

2026-05-13

Syllabus

Week	Date	Subject/Topics
1	2026/02/25	Introduction to Software Engineering
2	2026/03/04	Software Products and Project Management: Software product management and prototyping with Generative AI and Agentic AI
3	2026/03/11	Agile Software Engineering: Agile methods, Scrum, and Extreme Programming
4	2026/03/18	Case Study on Software Engineering I
5	2026/03/25	Features, Scenarios, and Stories
6	2026/04/01	Software Architecture: Architectural design, System decomposition, and Distribution architecture

Syllabus

Week	Date	Subject/Topics
7	2026/04/08	Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service
8	2026/04/15	Midterm Project Report
9	2026/04/22	Cloud Computing and Cloud Software Architecture
10	2026/04/29	Microservices Architecture, RESTful services, Service deployment
11	2026/05/06	Case Study on Software Engineering II
12	2026/05/13	Security and Privacy; Reliable Programming; Testing: Functional testing, Test automation, Test-driven development, and Code reviews

Syllabus

Week Date Subject/Topics

13 2026/05/20 Industry Practices of Software Engineering

[Invited Talk: Agentic AI and Development Trends in Human–AI Collaborative Applications,
Invited Speaker: Shihyu (Alex) Chu, Industry Consultant / Section Manager,
Software Industry Research Center, Market Intelligence & Consulting Institute (MIC)]

14 2026/05/27 DevOps and Code Management:
Code management and DevOps automation

15 2026/06/03 Final Project Report I

16 2026/06/10 Final Project Report II

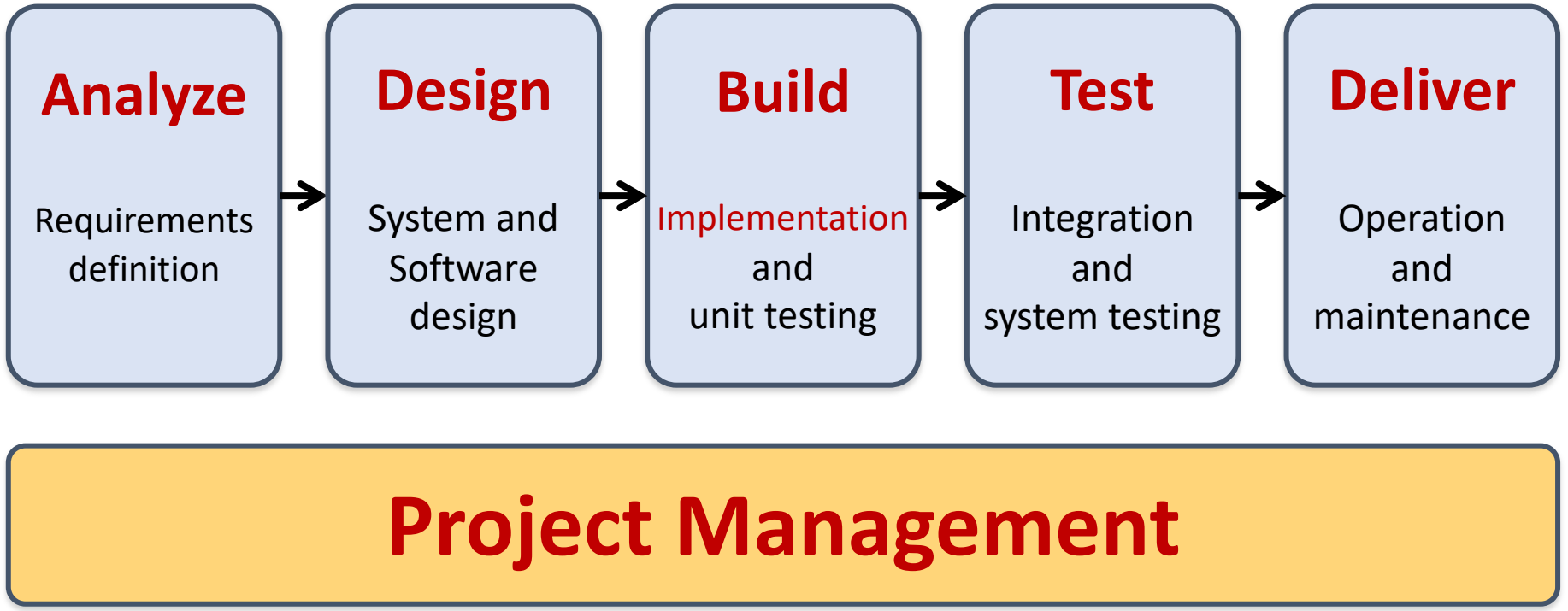
Security and Privacy;

Reliable Programming;

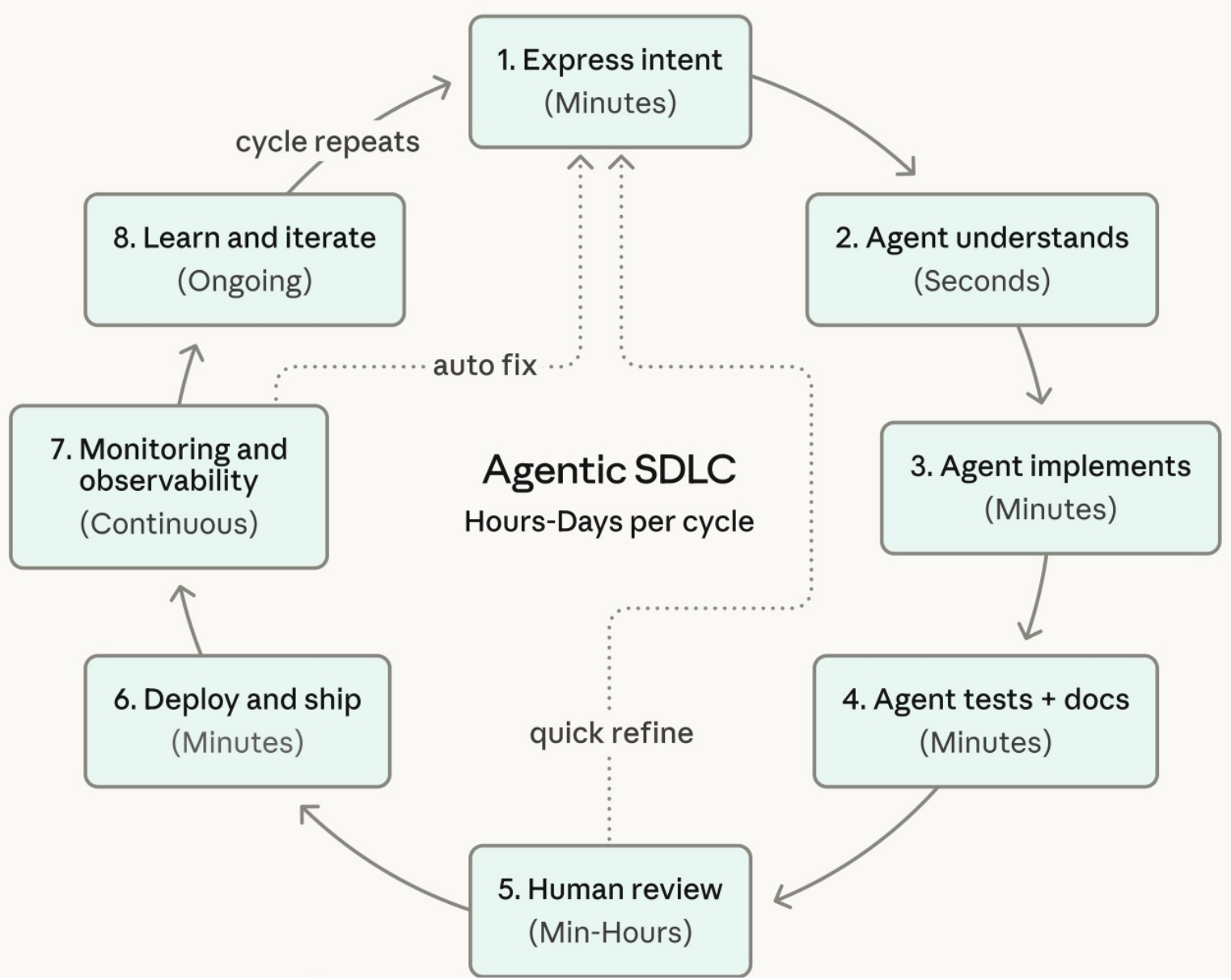
Testing:

Test-driven development, and Code reviews;

Software Engineering
and
Project Management



Agentic Coding Software Development Lifecycle



Source: Anthropic (2026), 2026 Agentic CodingTrends Report: How coding agents are reshaping software development, https://resources.anthropic.com/hubfs/2026%20Agentic%20Coding%20Trends%20Report.pdf

Information Management

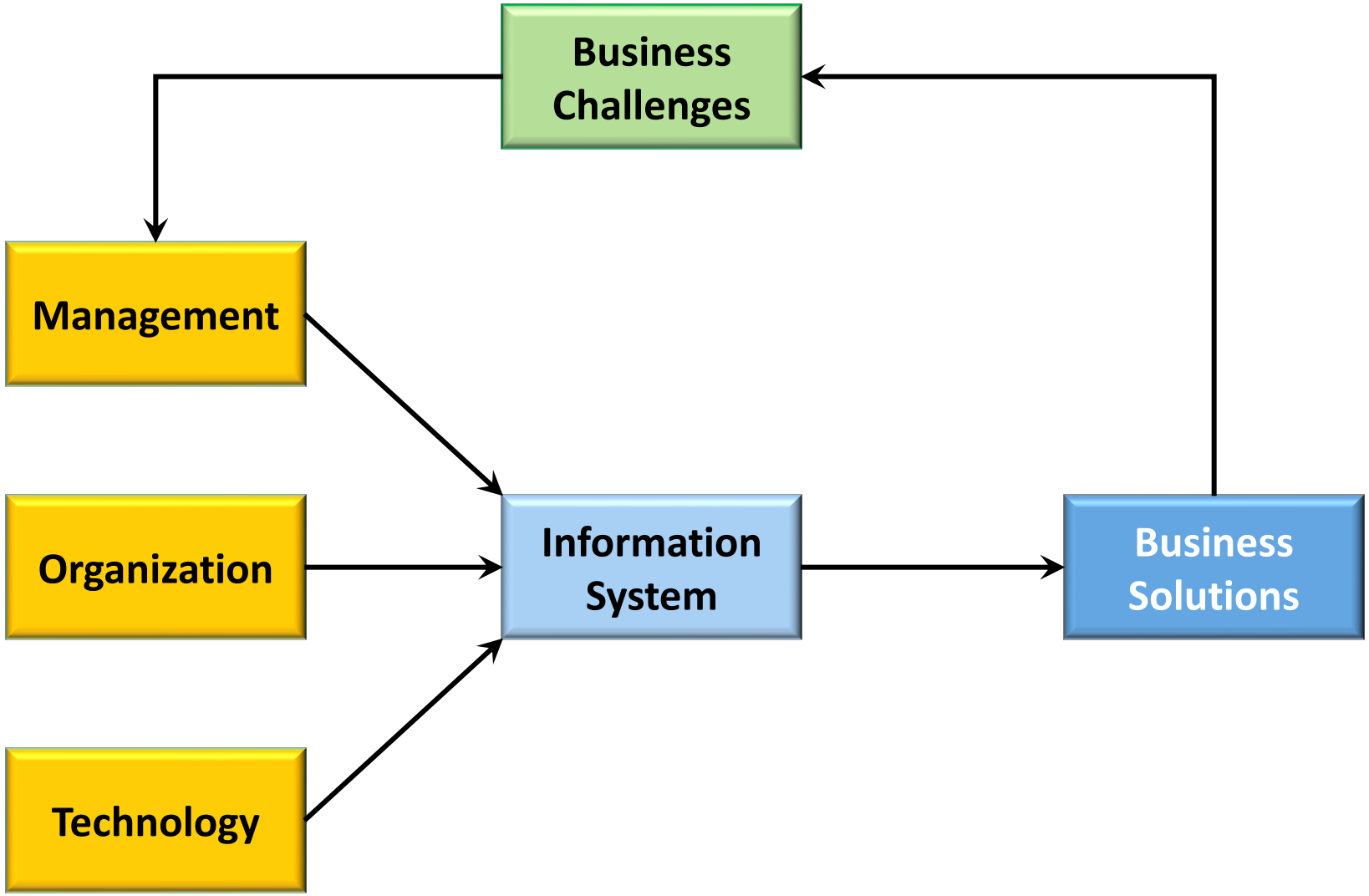
Management Information Systems (MIS)

Information Systems

Information Management (MIS)
Information Systems

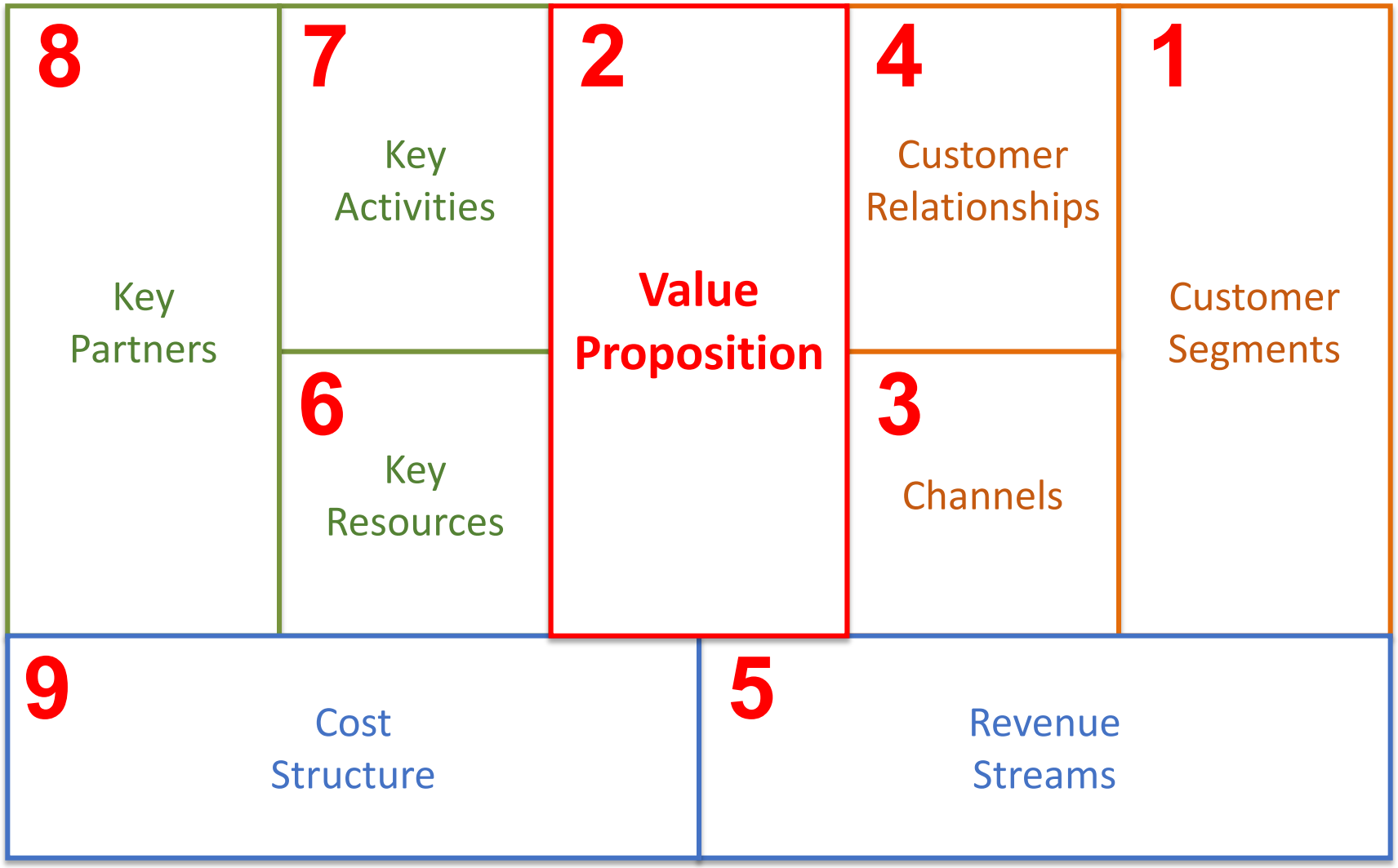
Source: Kenneth C. Laudon & Jane P. Laudon (2014), Management Information Systems: Managing the Digital Firm, Thirteenth Edition, Pearson.

Fundamental MIS Concepts



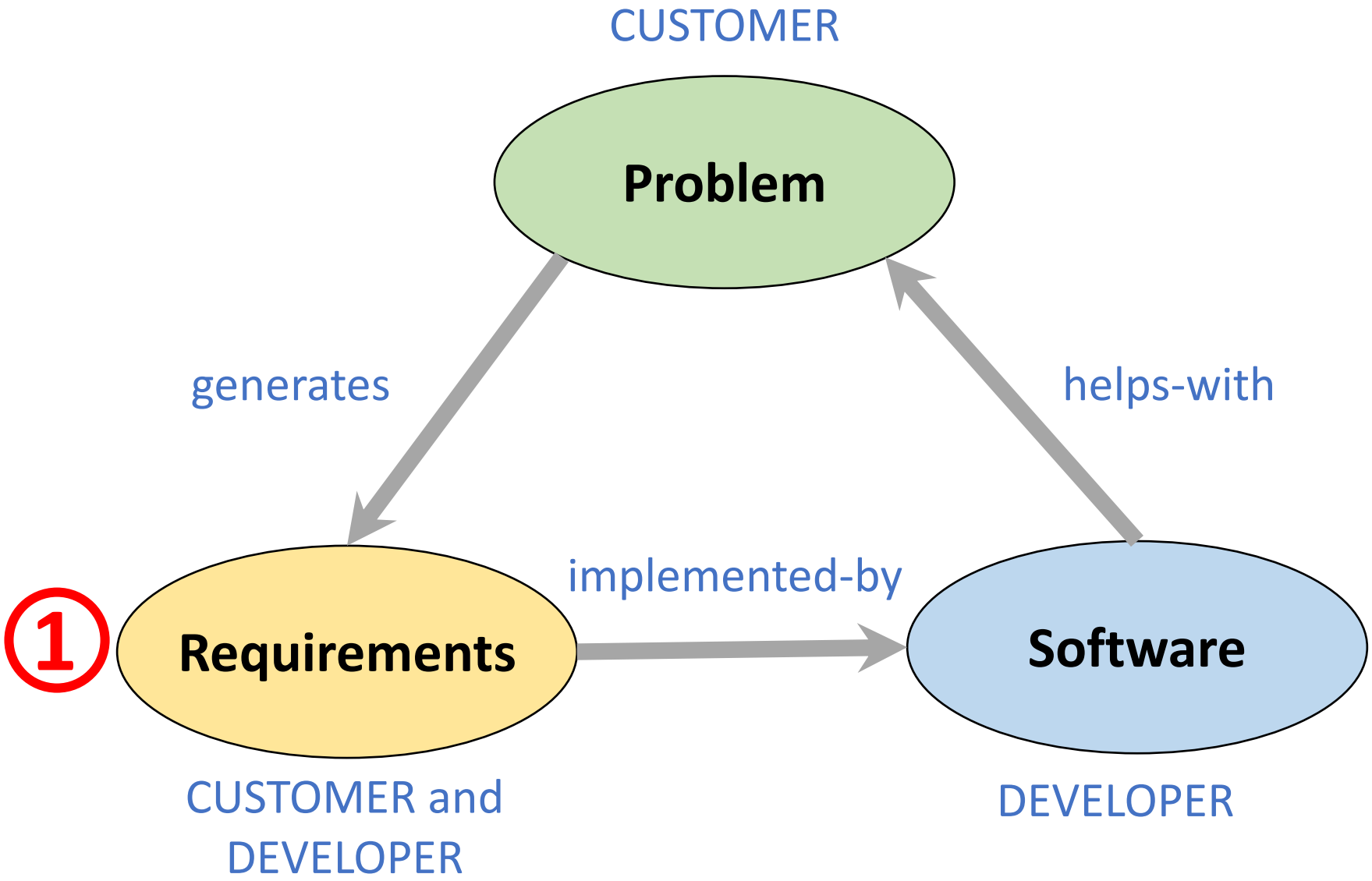
Source: Kenneth C. Laudon & Jane P. Laudon (2014), Management Information Systems: Managing the Digital Firm, Thirteenth Edition, Pearson.

Business Model



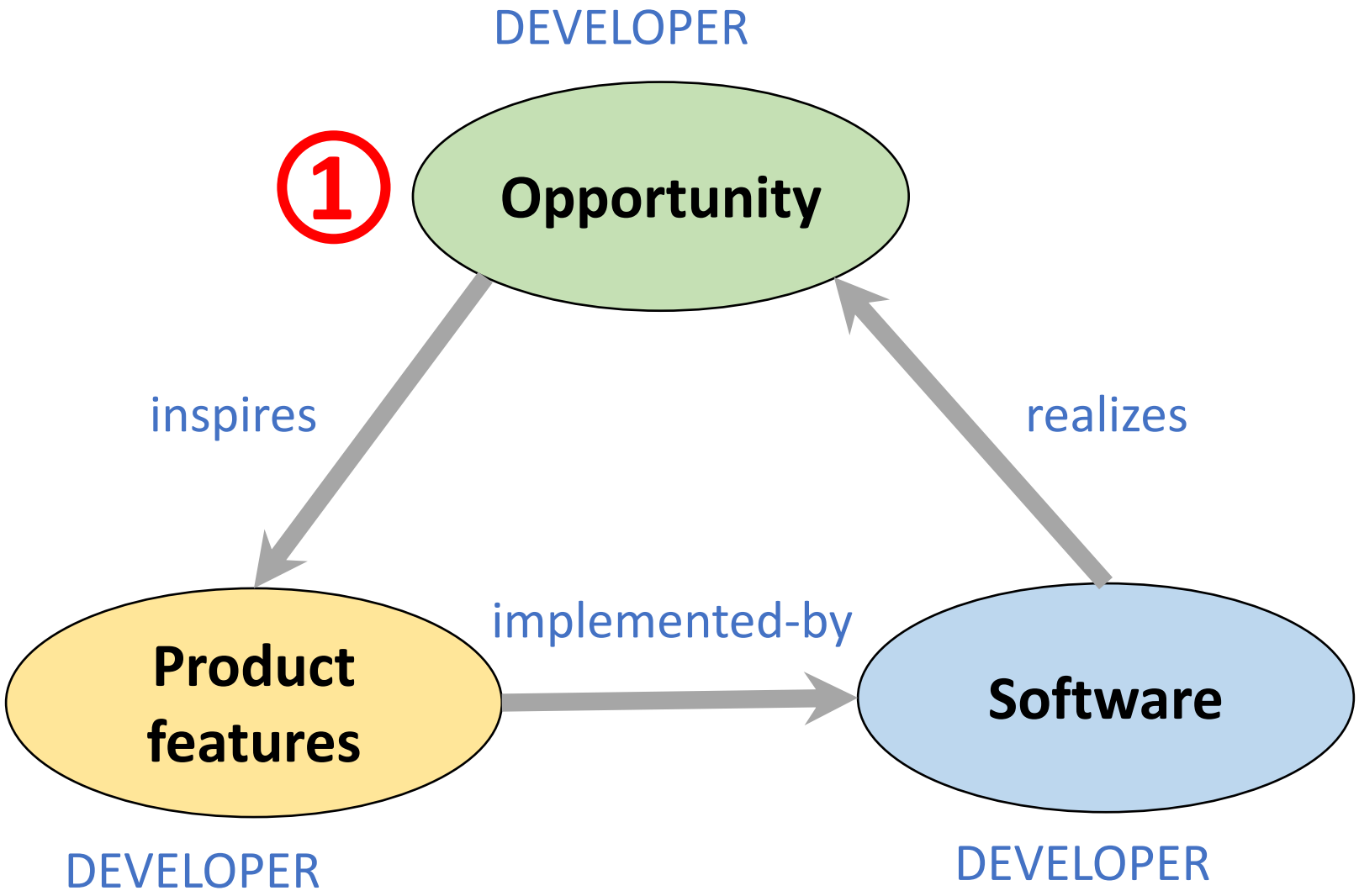
Source: Alexander Osterwalder & Yves Pigneur, Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers, Wiley, 2010.

Project-based software engineering



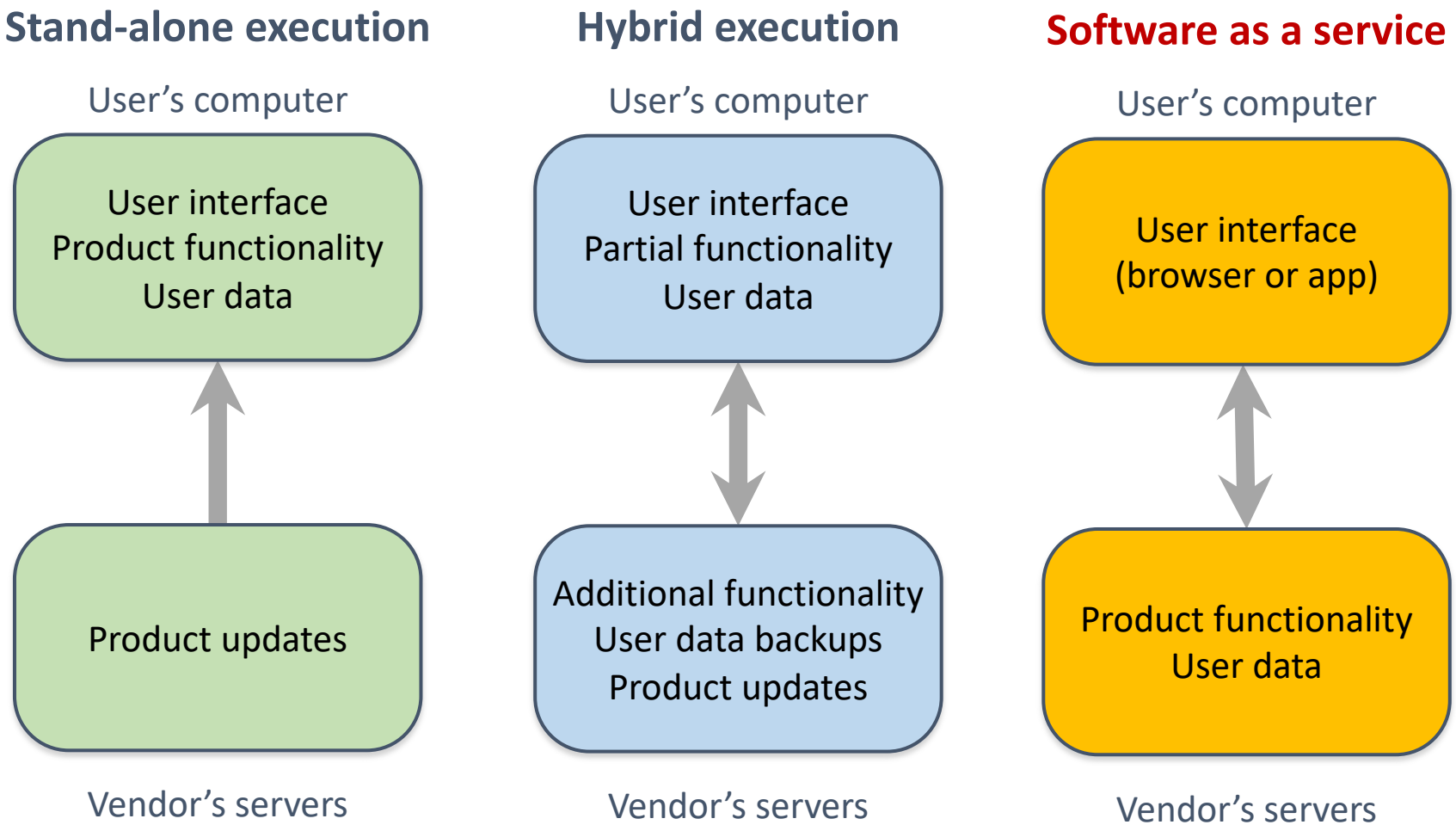
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Product software engineering



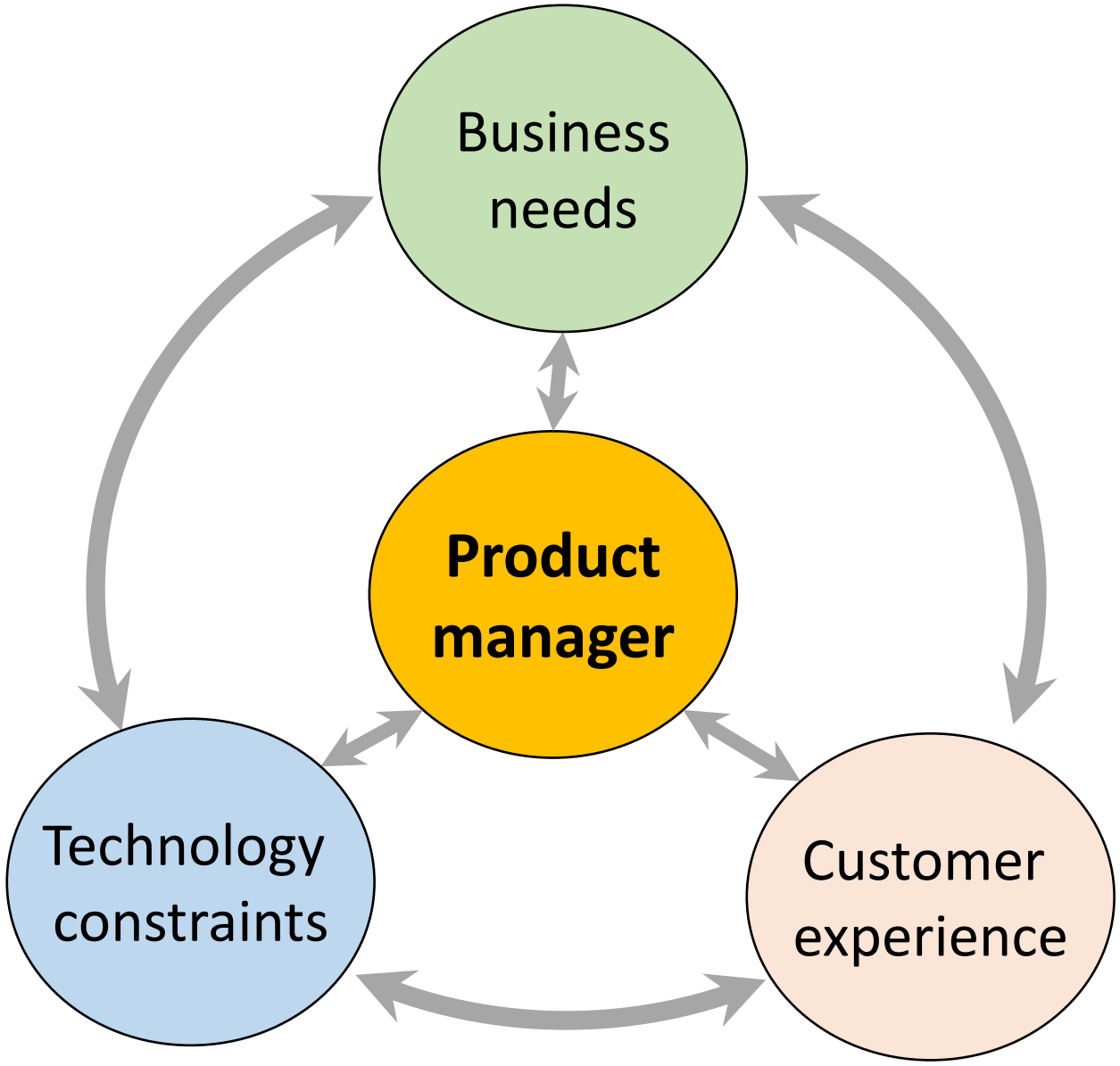
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software execution models



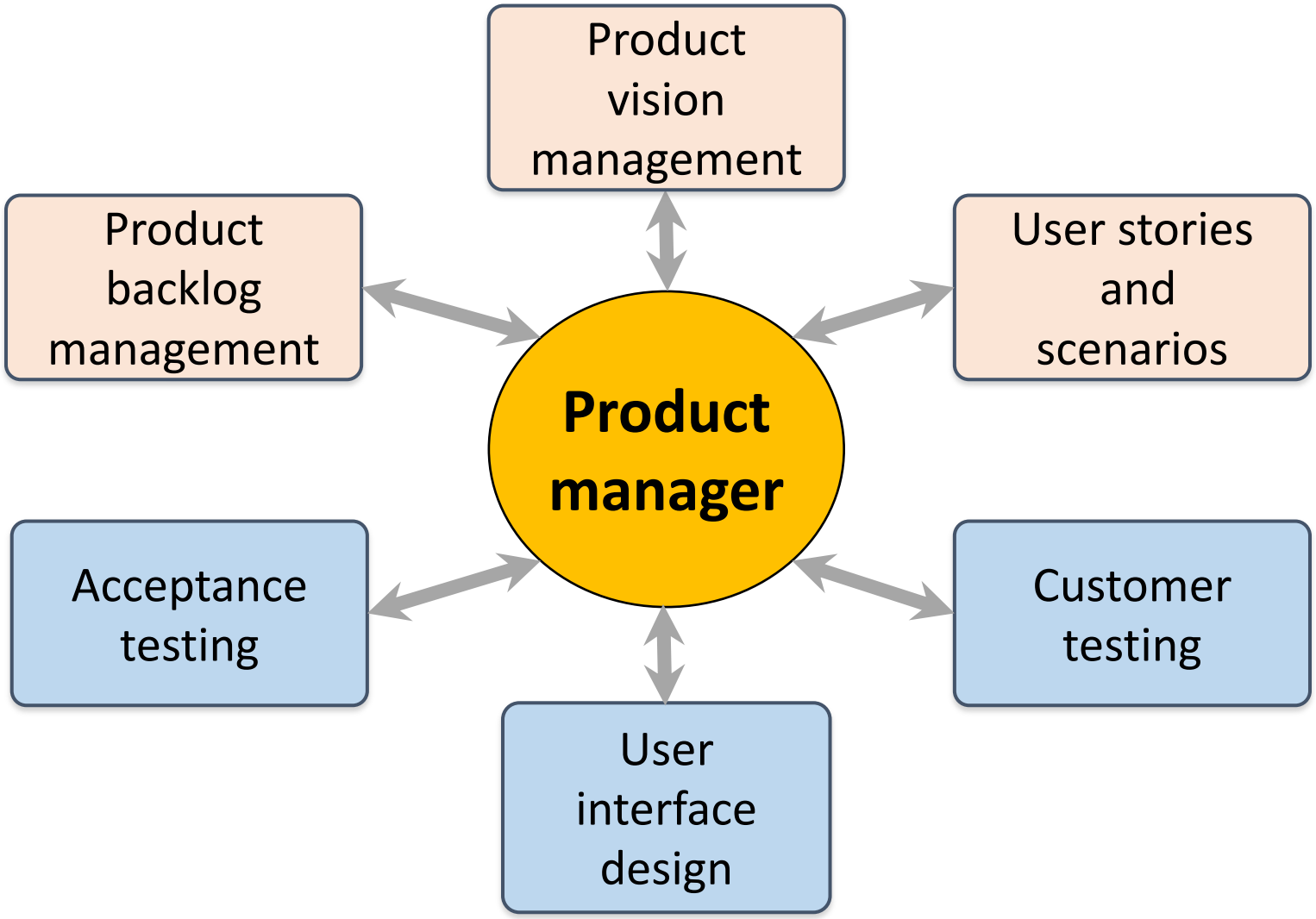
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Product management concerns



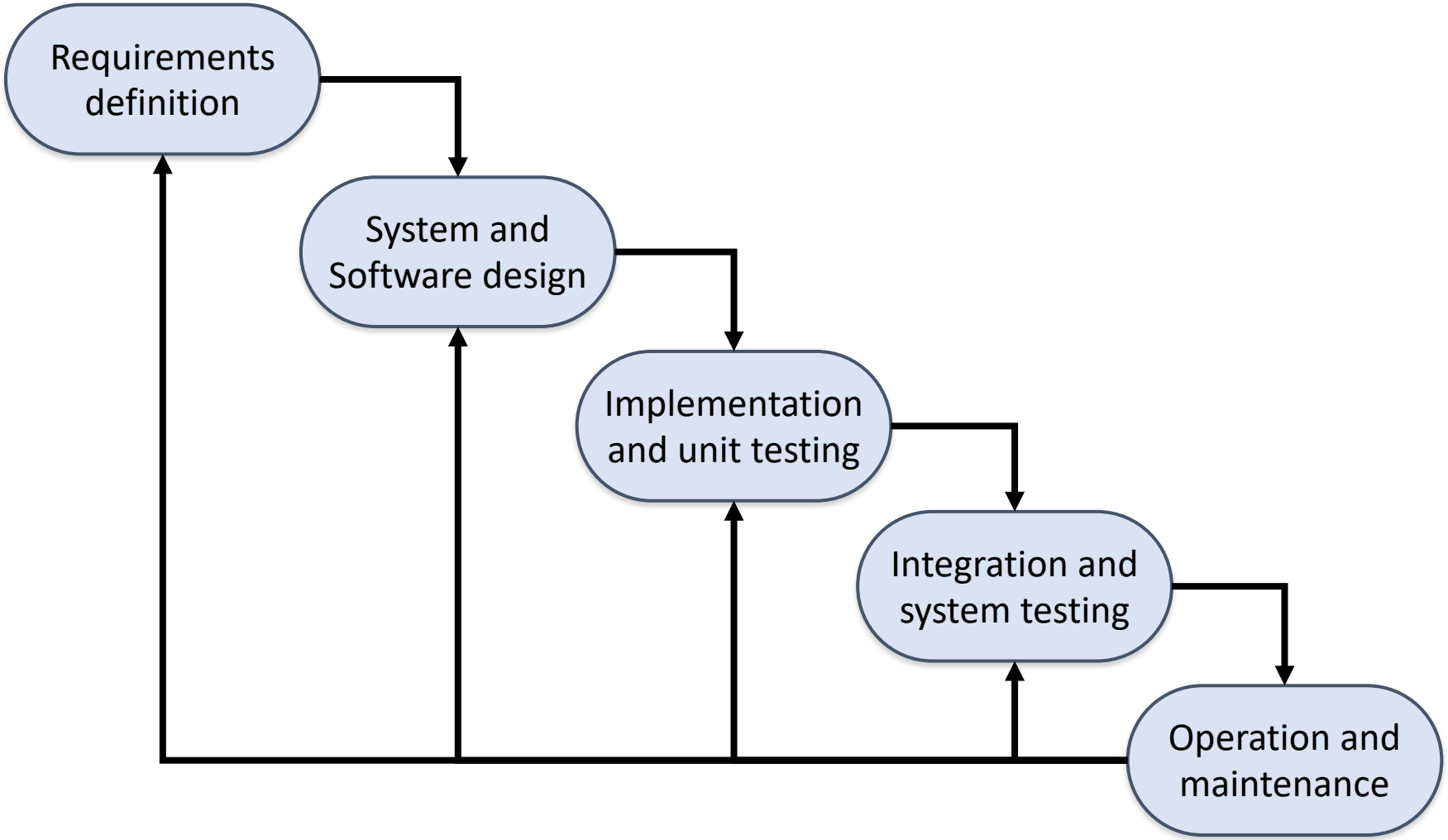
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Technical interactions of product managers



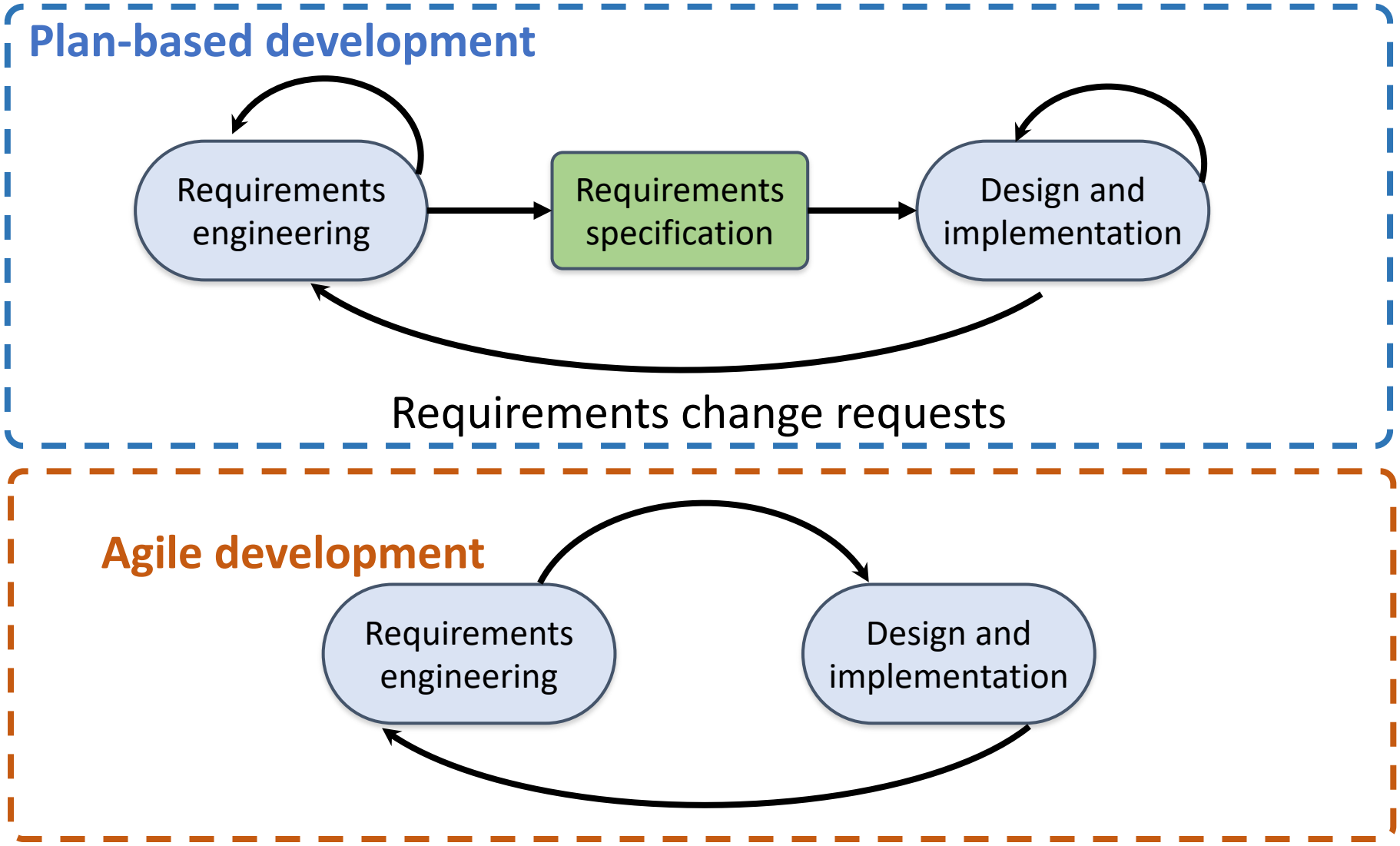
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software Development Life Cycle (SDLC)
The waterfall model



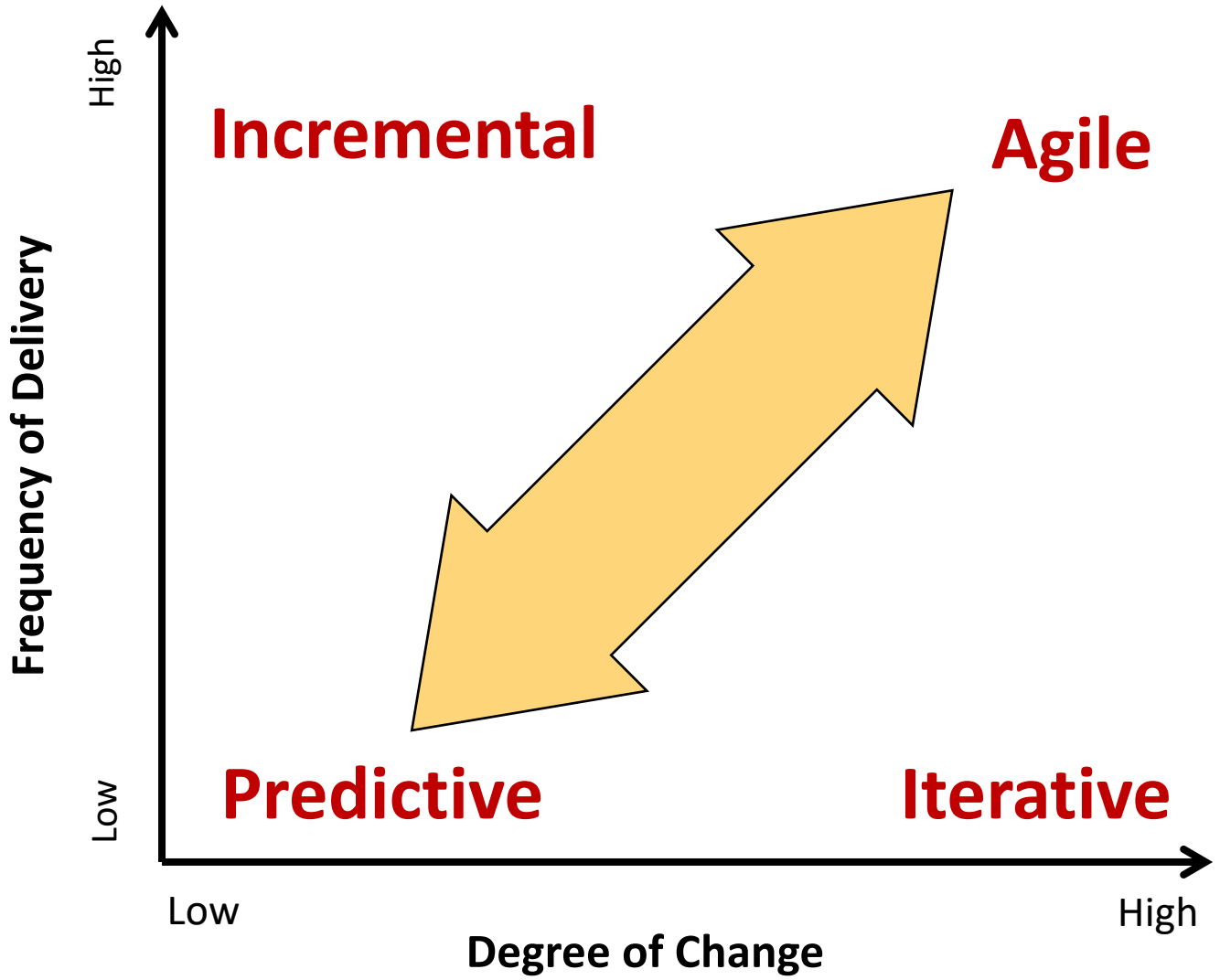
Source: Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.

Plan-based and Agile development



Source: Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.

The Continuum of Life Cycles

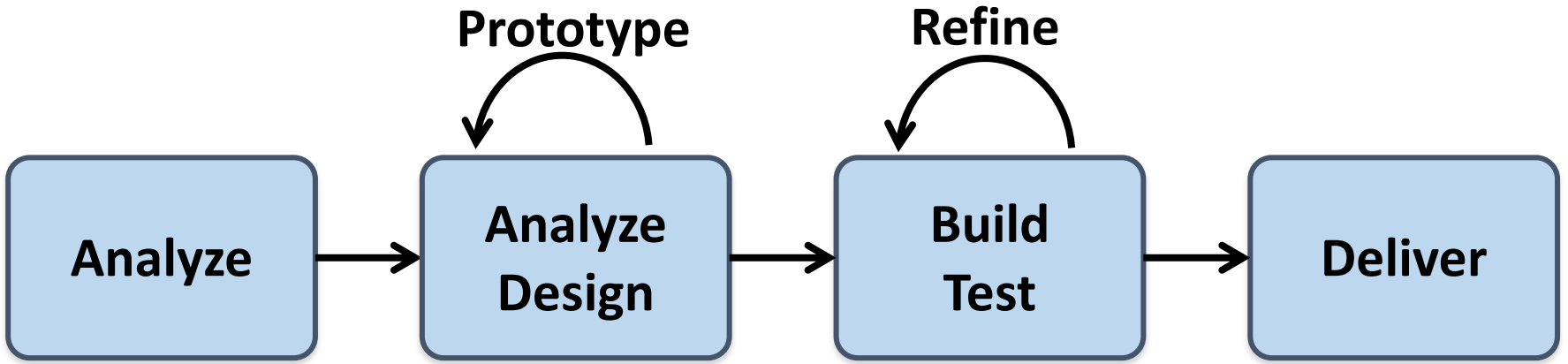


Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Predictive Life Cycle

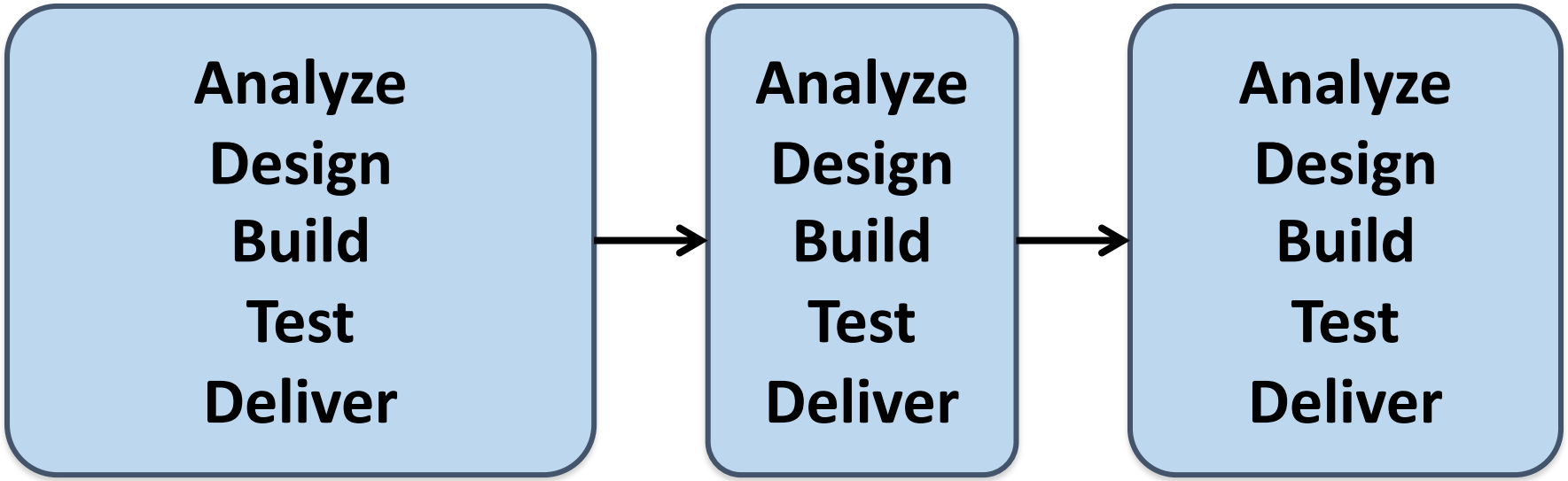
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Iterative Life Cycle



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

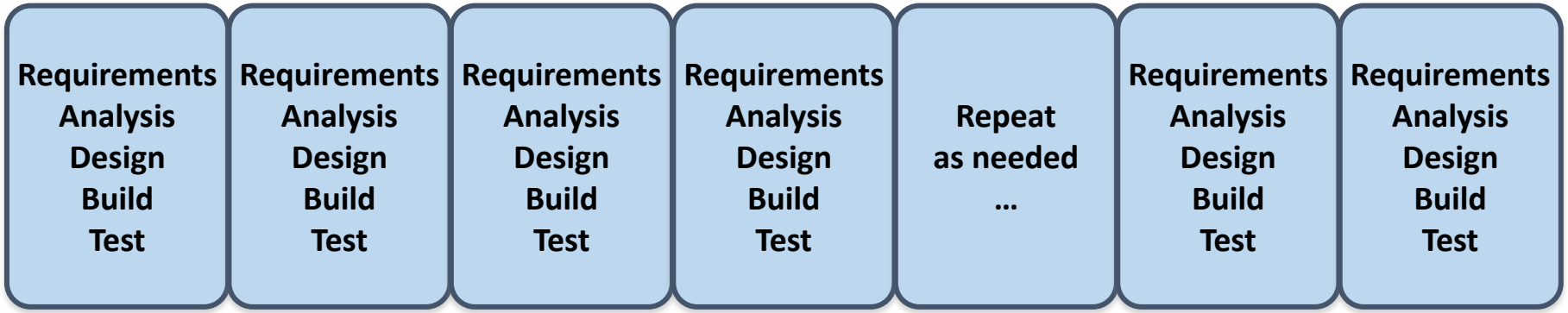
A Life Cycle of Varying-Sized Increments



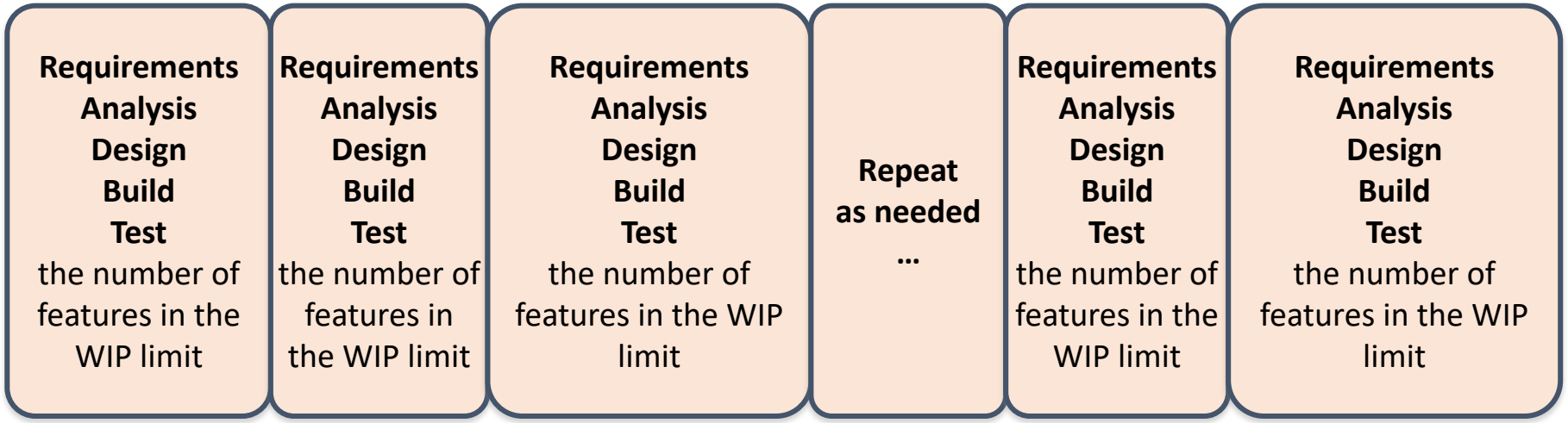
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Iteration-Based and Flow-Based Agile Life Cycles

Iteration-Based Agile

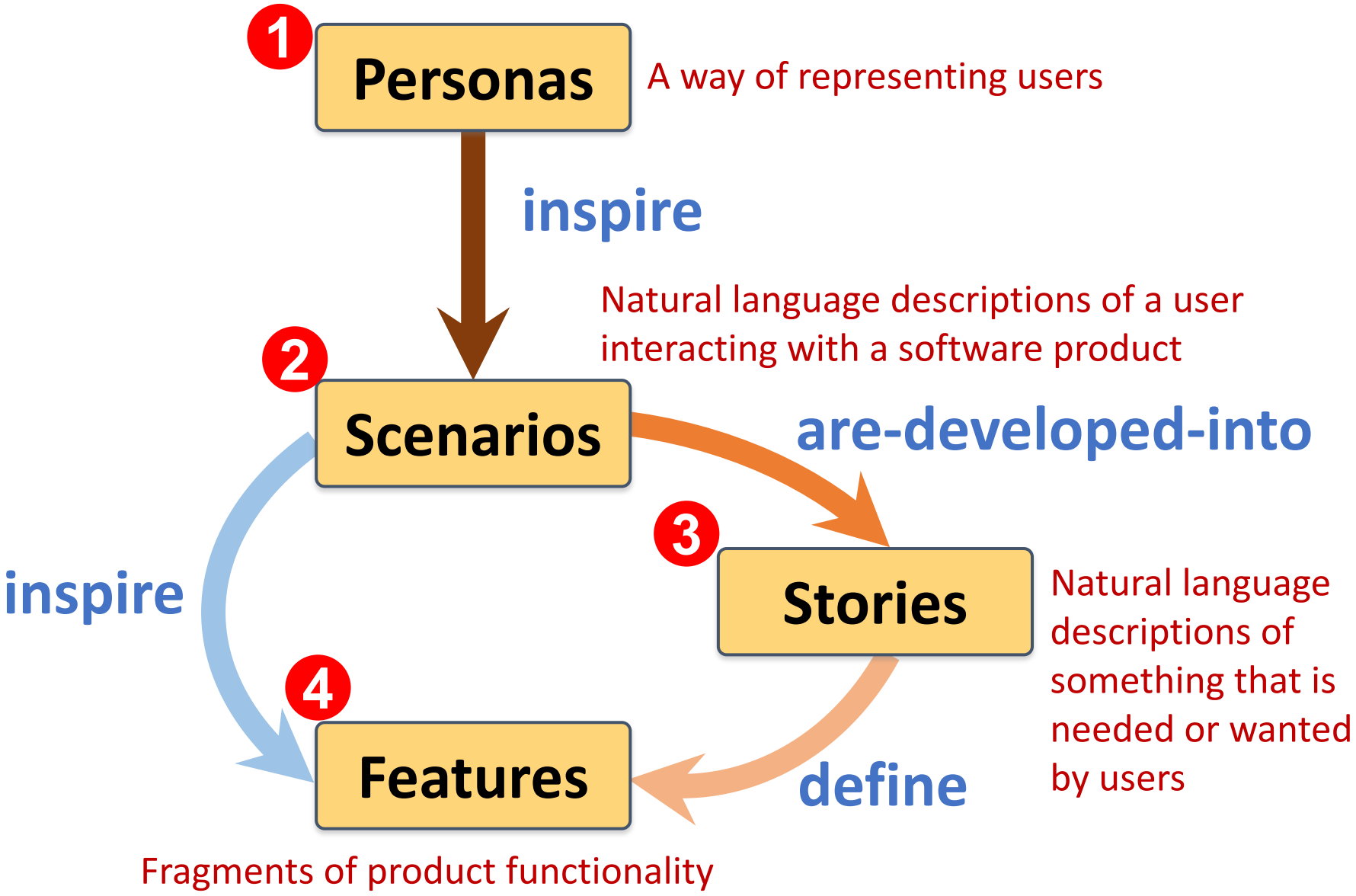


Flow-Based Agile



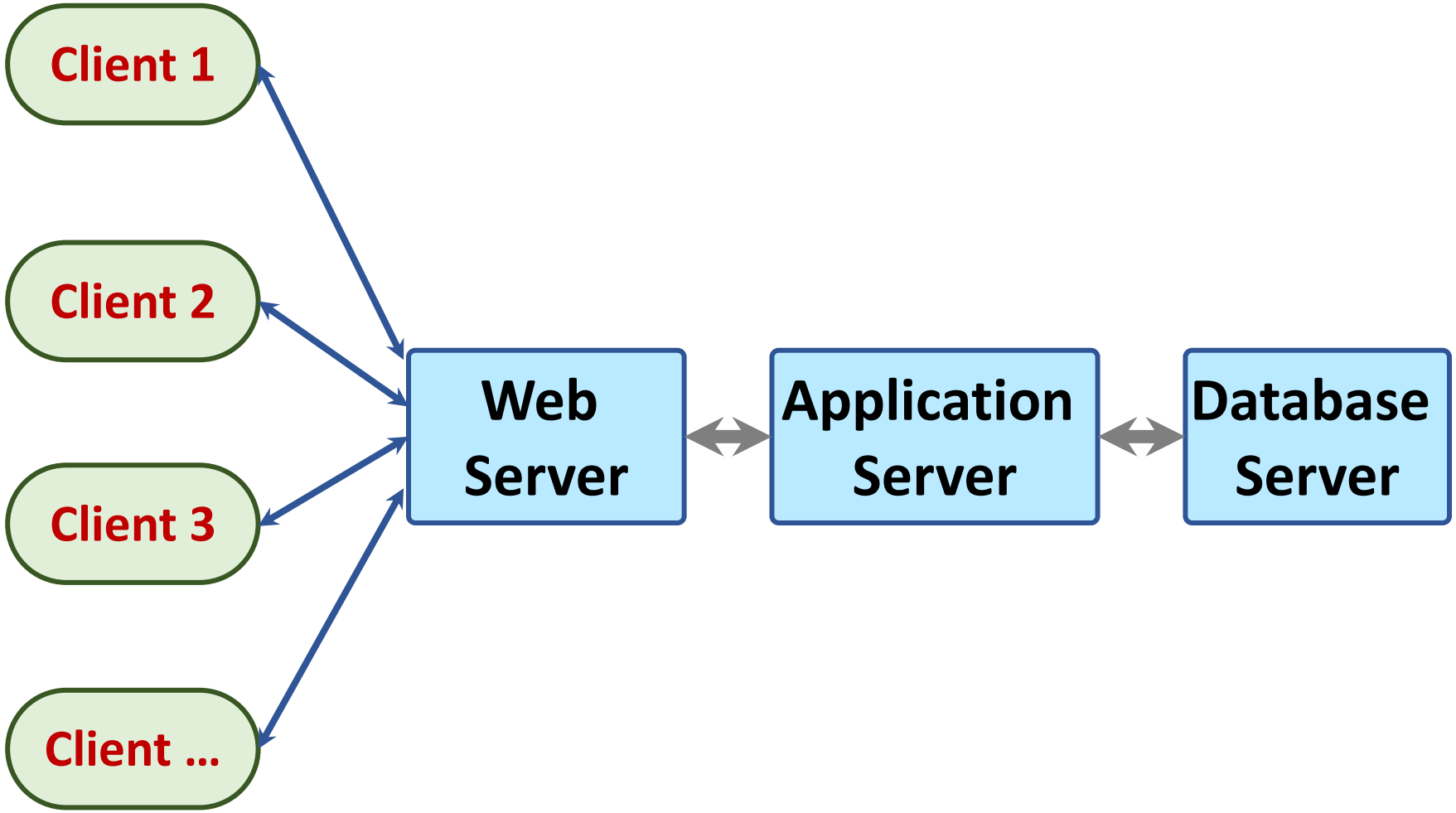
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

From personas to features



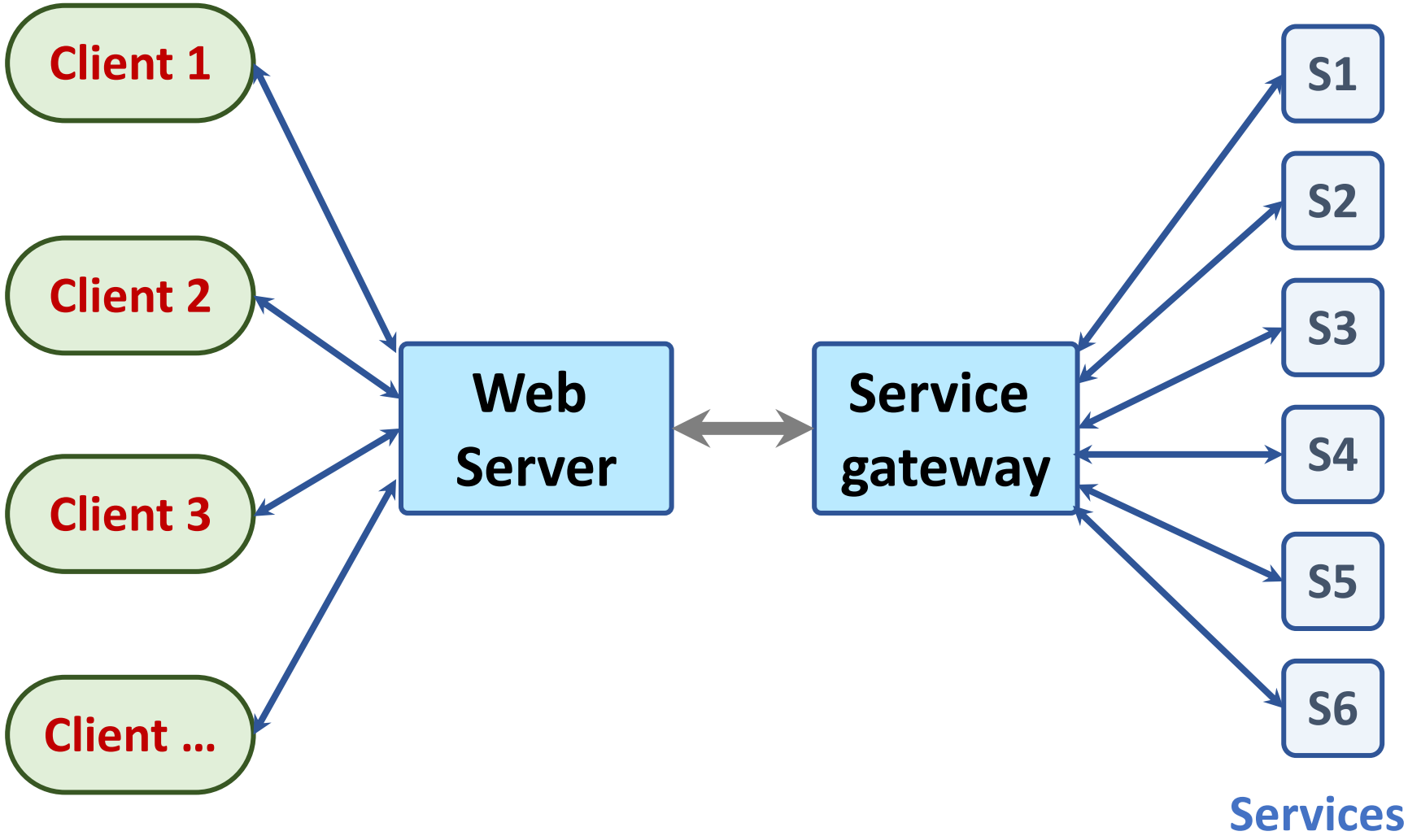
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Multi-tier client-server architecture

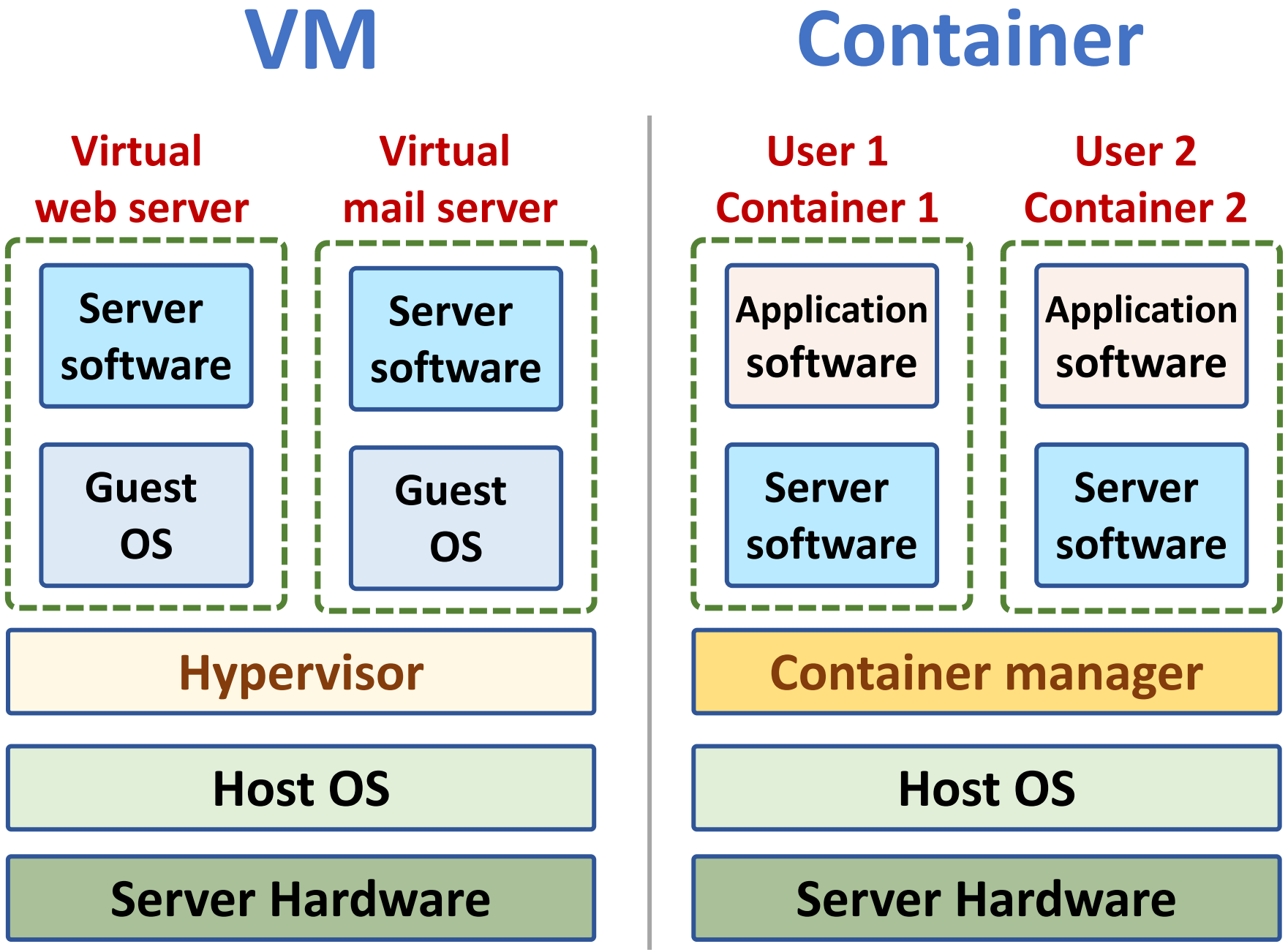


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Service-oriented Architecture

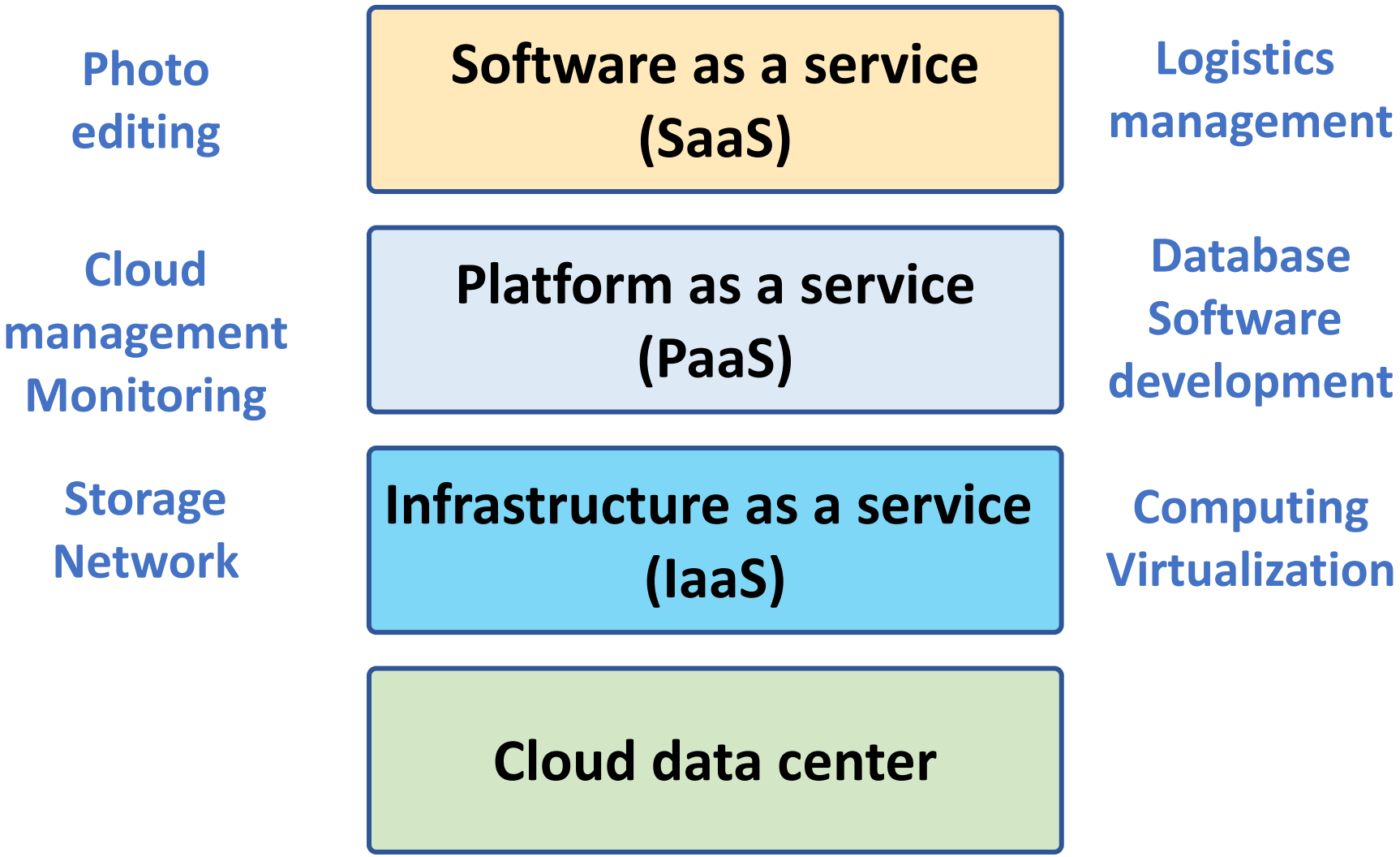


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.



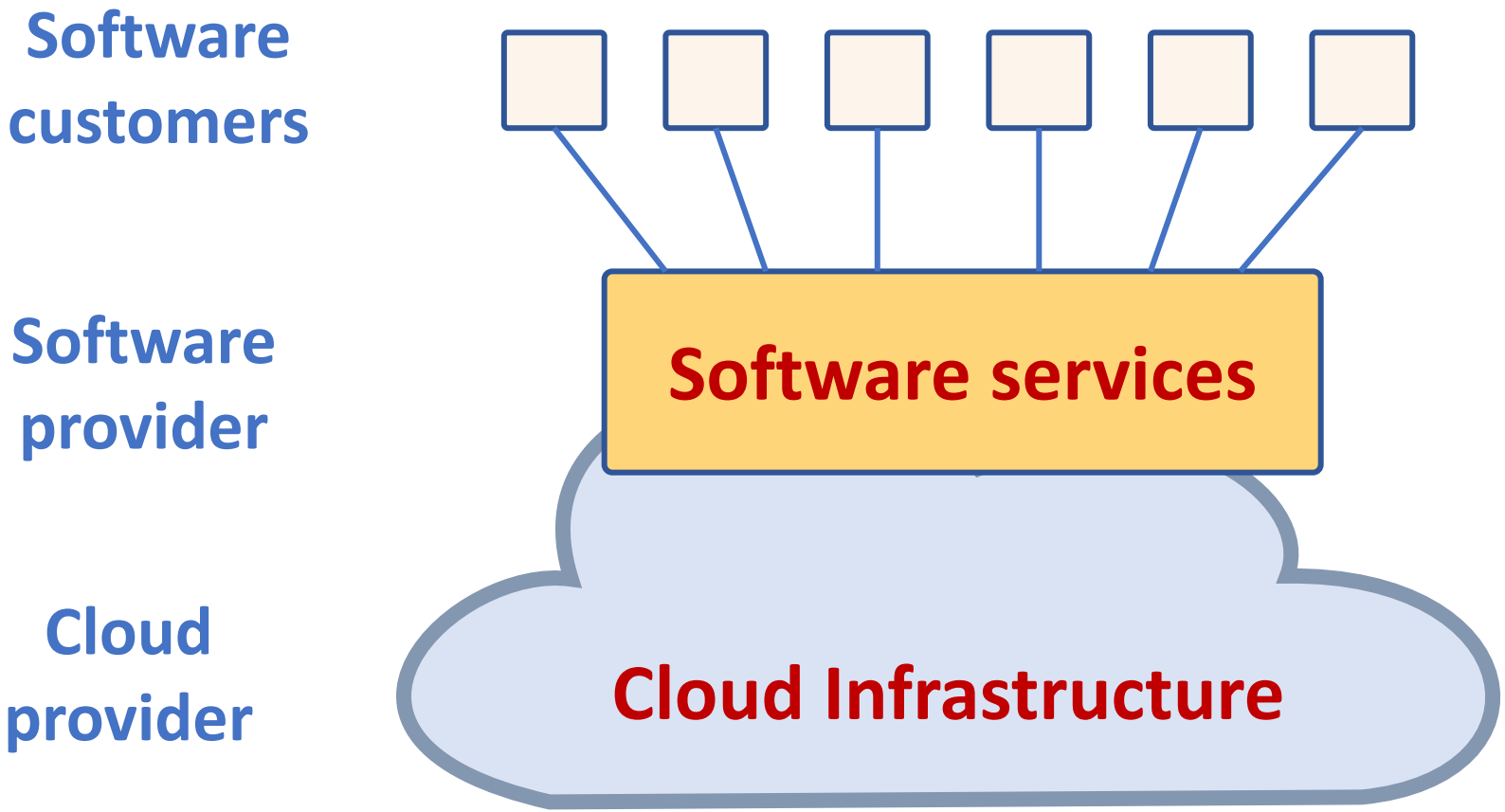
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Everything as a service



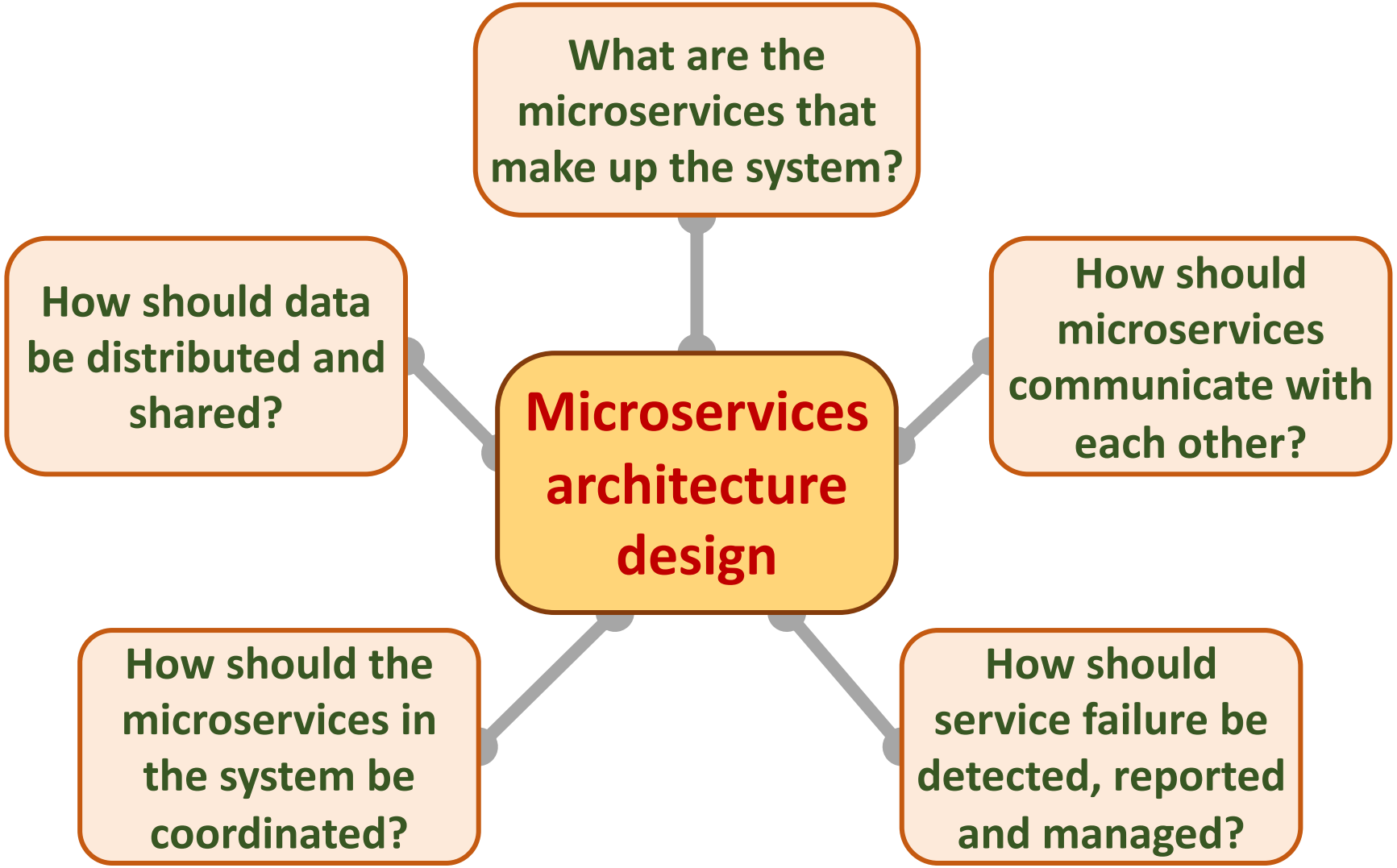
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software as a service



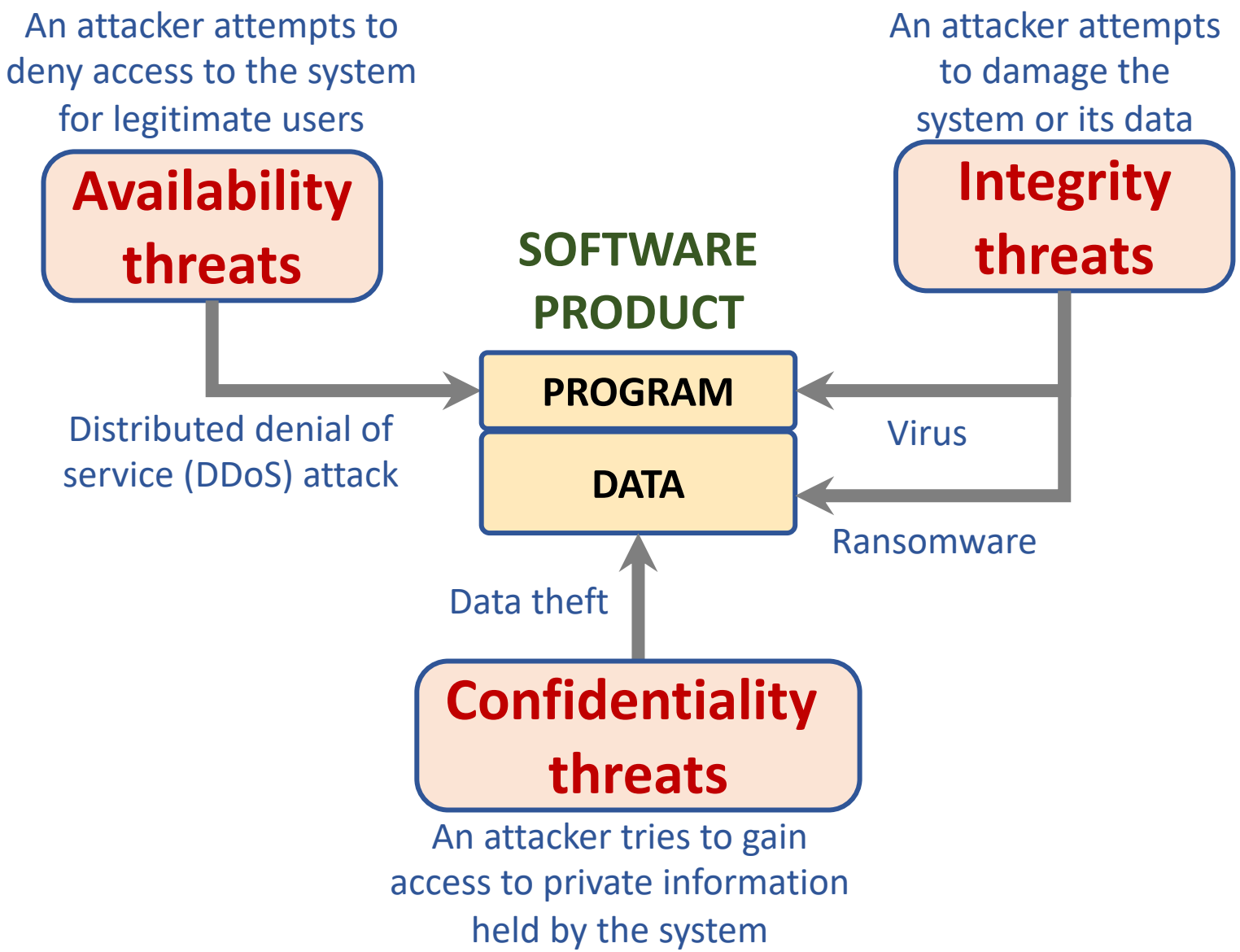
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Microservices architecture – key design questions



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

Types of security threat

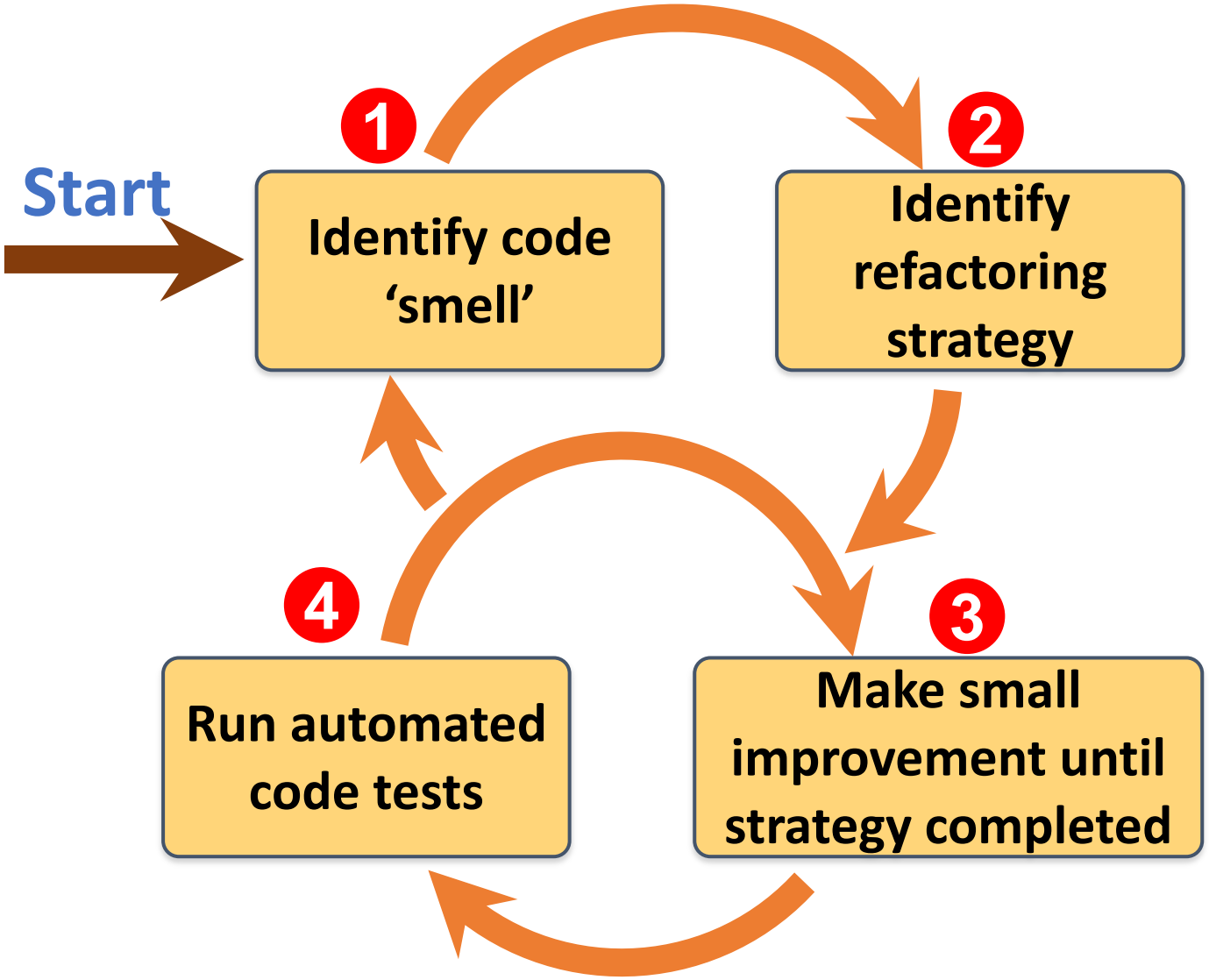


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software product quality attributes

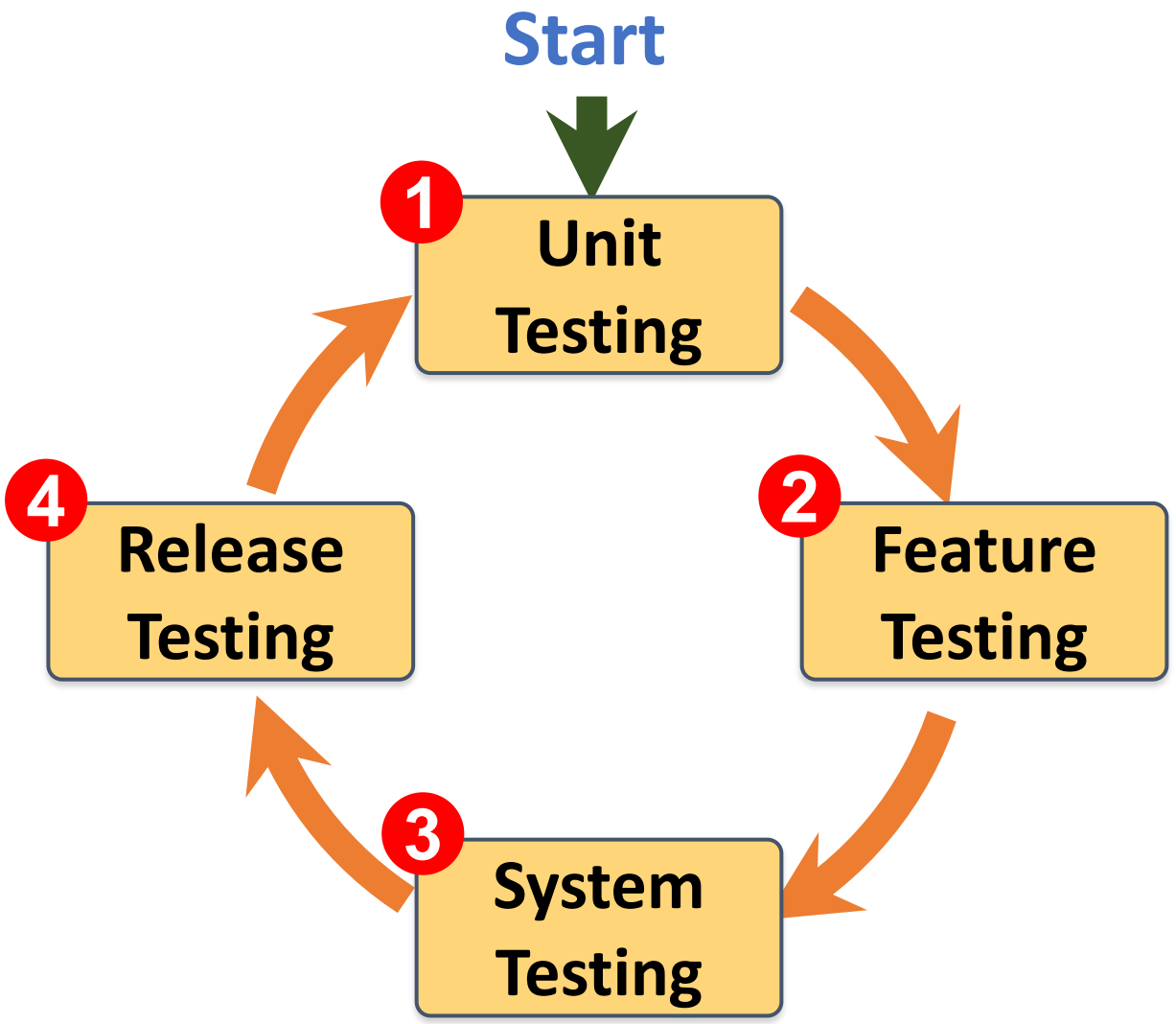
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

A refactoring process



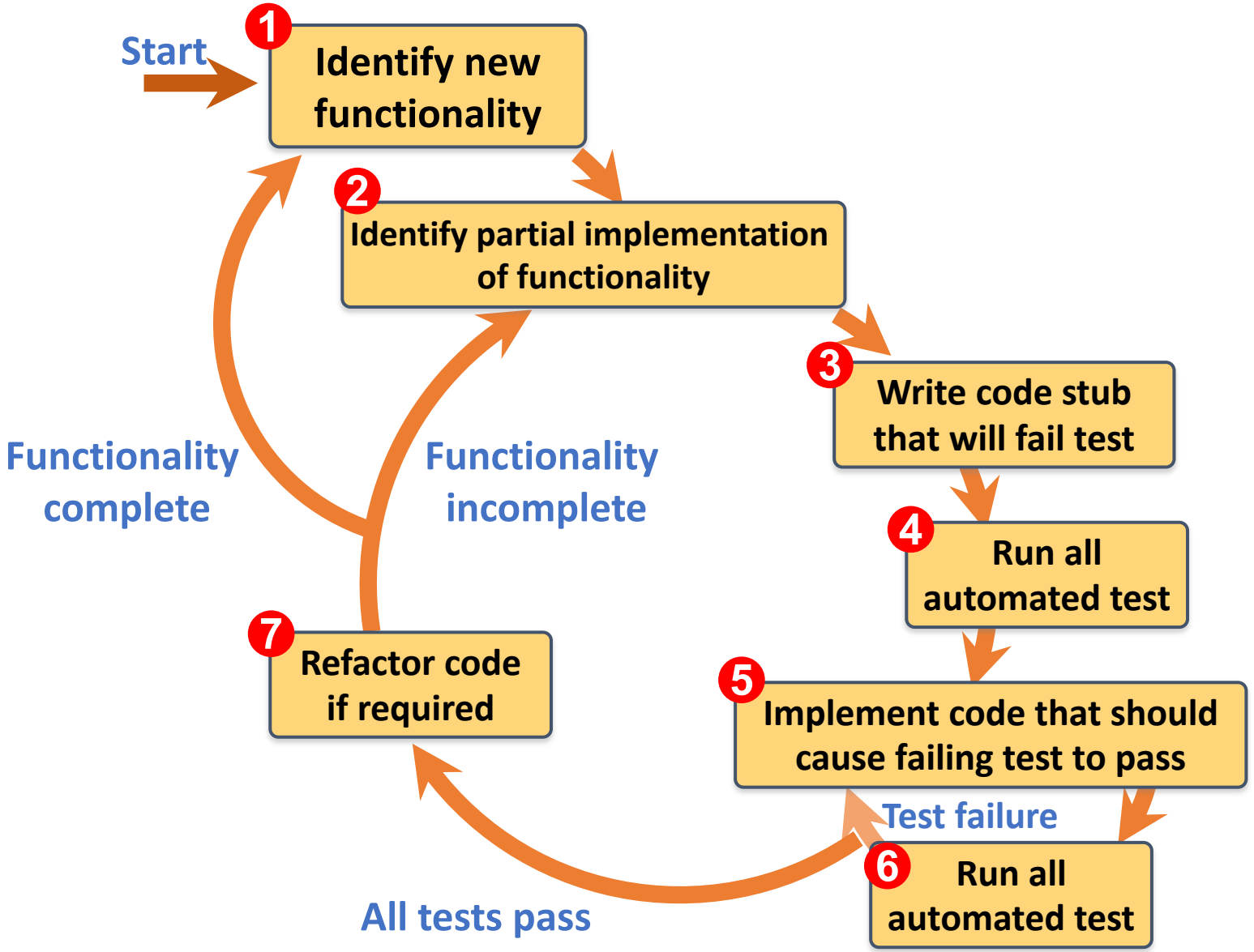
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Functional testing



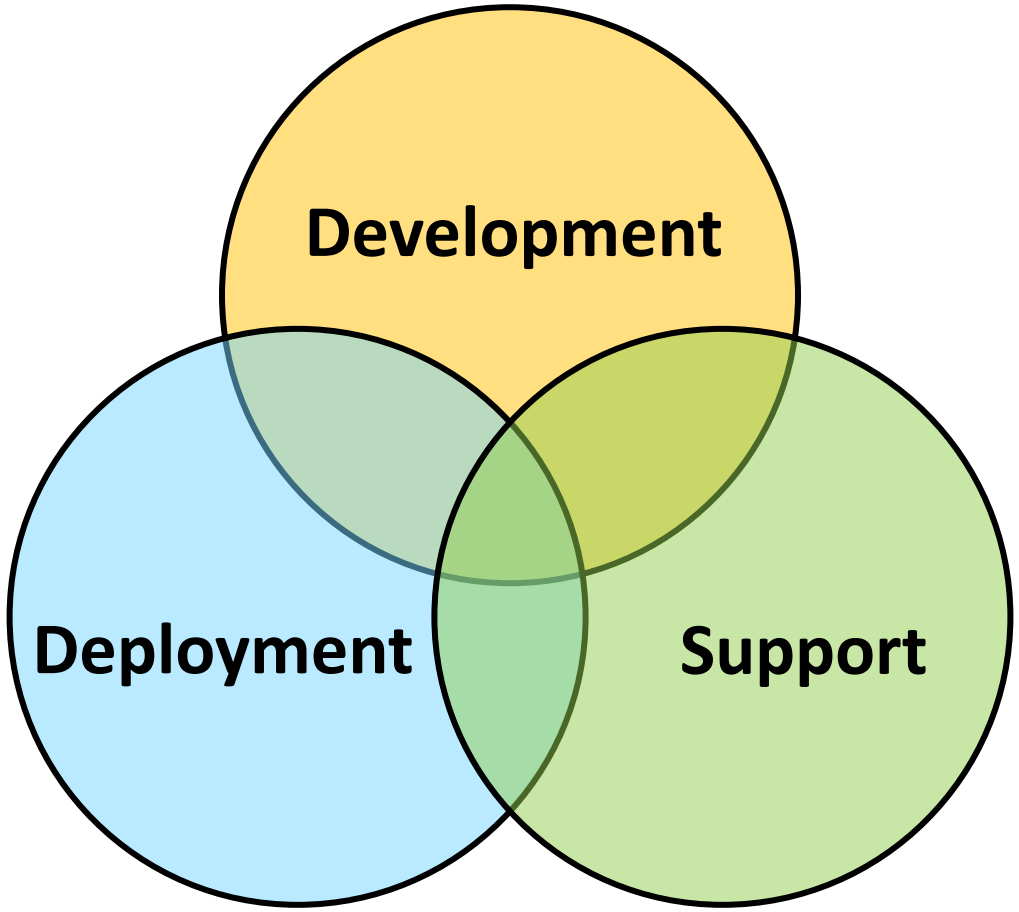
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Test-driven development (TDD)



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

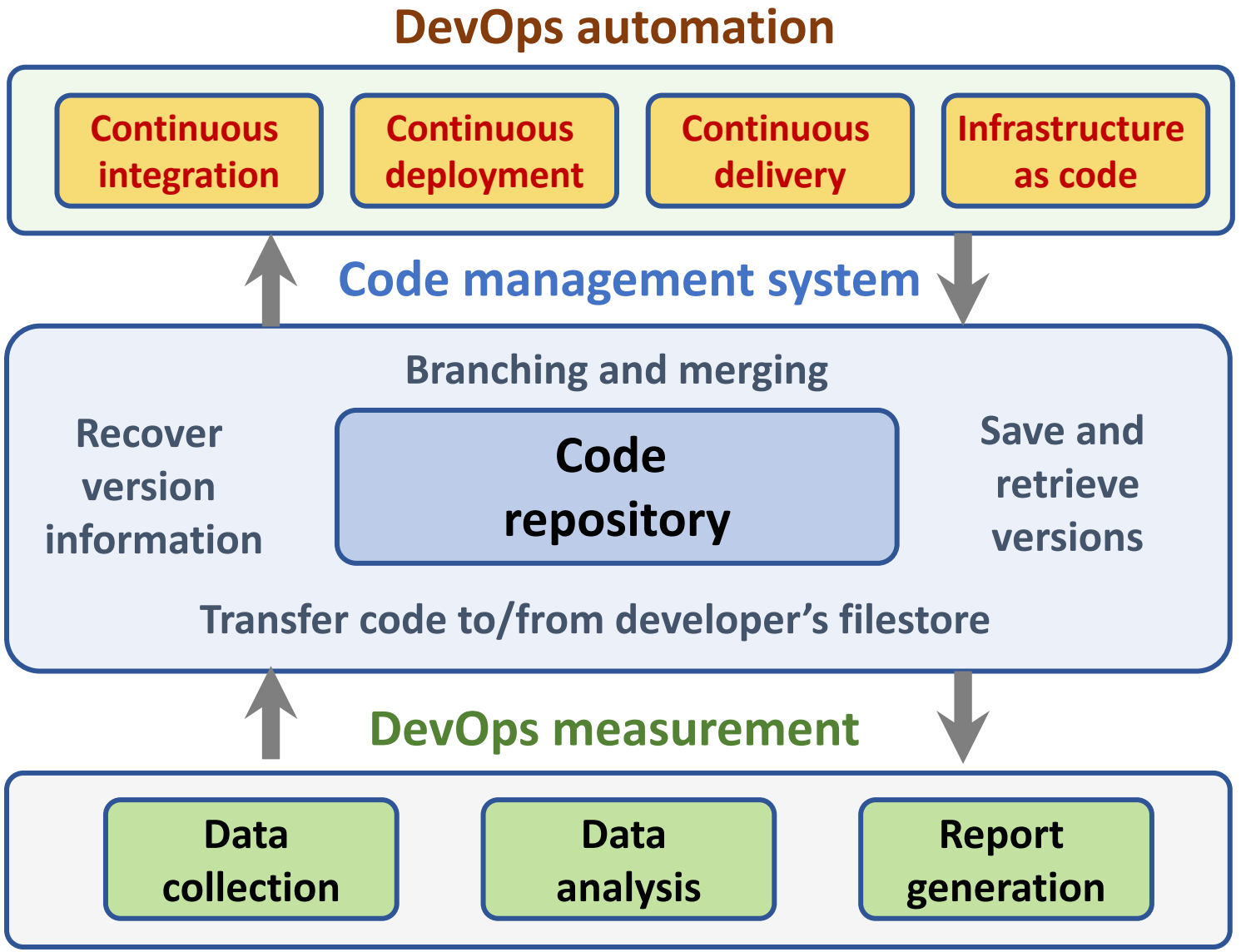
DevOps



Multi-skilled DevOps team

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

Code management and DevOps



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

Agentic Coding:
OpenAI Codex GPT-5.5 vs Claude Code Opus 4.7

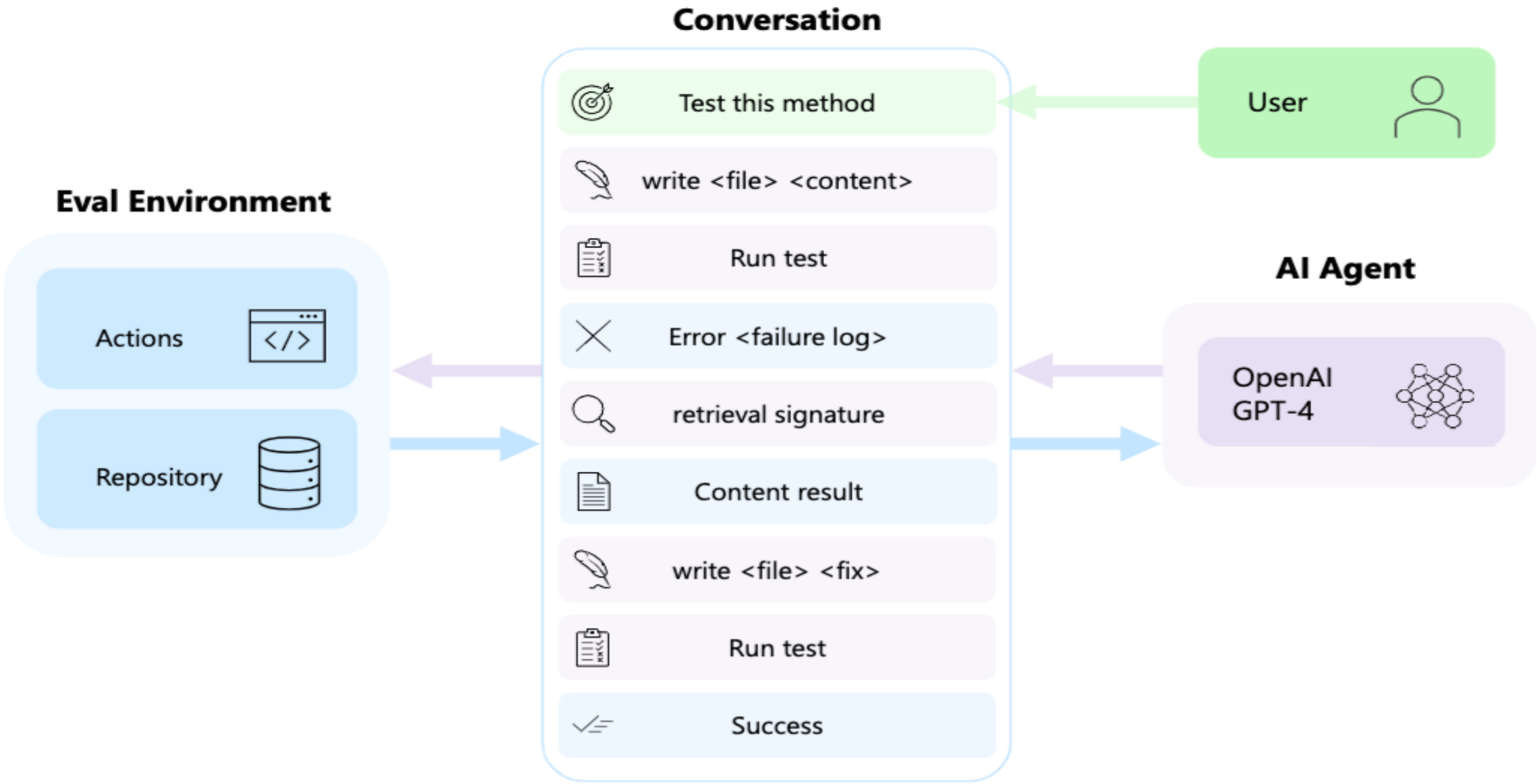
	GPT-5.5	GPT-5.4	GPT-5.5 Pro	GPT-5.4 Pro	Claude Opus 4.7	Gemini 3.1 Pro
Terminal-Bench 2.0	**82.7%**	75.1%	-	-	69.4%	68.5%
Expert-SWE (Internal)	**73.1%**	68.5%	-	-	-	-
GDPval (wins or ties)	**84.9%**	83.0%	82.3%	82.0%	80.3%	67.3%
OSWorld-Verified	**78.7%**	75.0%	-	-	78.0%	-
Toolathlon	**55.6%**	54.6%	-	-	-	48.8%
BrowseComp	84.4%	82.7%	**90.1%**	89.3%	79.3%	85.9%
FrontierMath Tier 1–3	51.7%	47.6%	**52.4%**	50.0%	43.8%	36.9%
FrontierMath Tier 4	35.4%	27.1%	**39.6%**	38.0%	22.9%	16.7%
CyberGym	**81.8%**	79.0%	-	-	73.1%	-

Source: https://openai.com/index/introducing-gpt-5-5/

GitHub Copilot CODING AGENT

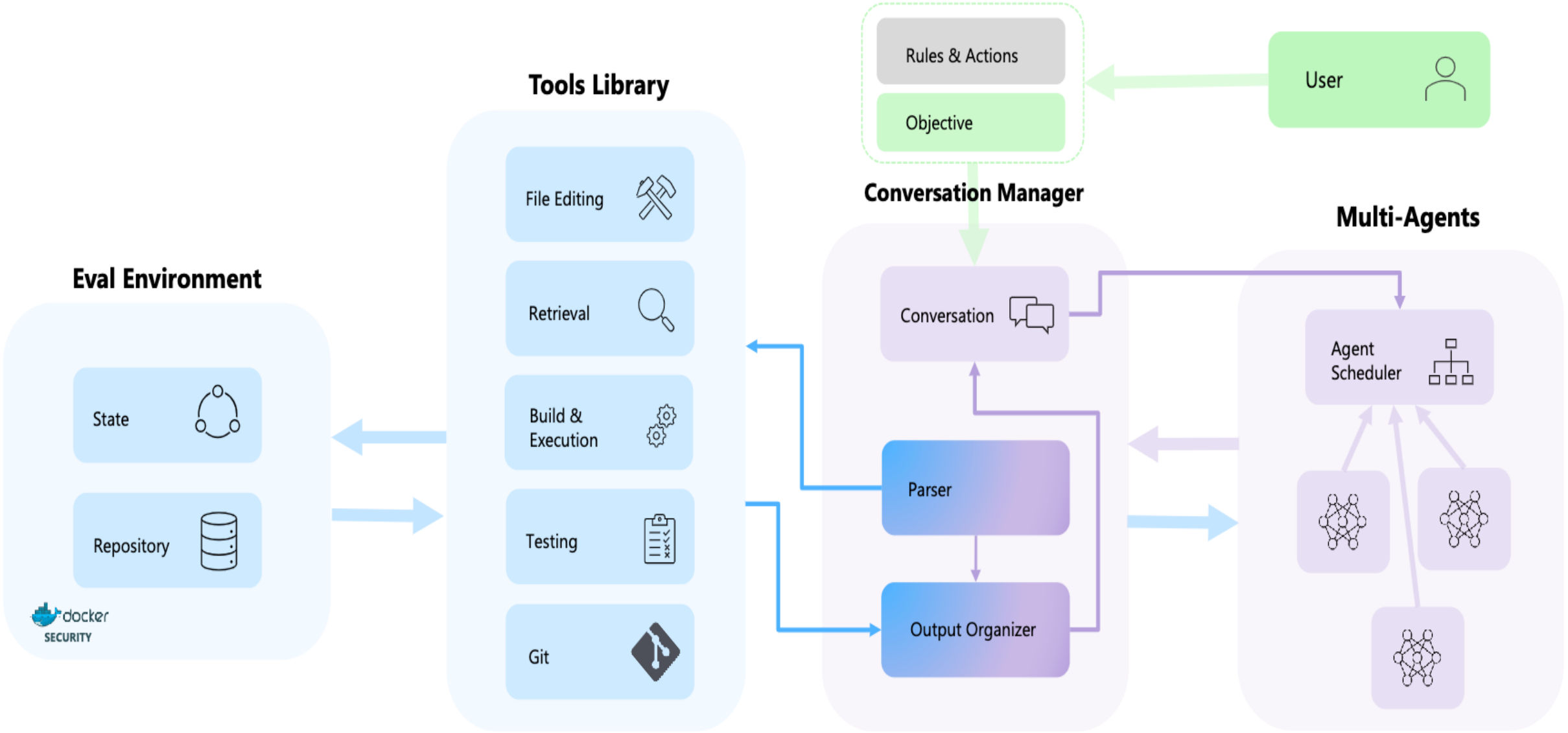
Source: https://github.blog/news-insights/product-news/github-copilot-meet-the-new-coding-agent/

AutoDev: Automated AI-Driven Development



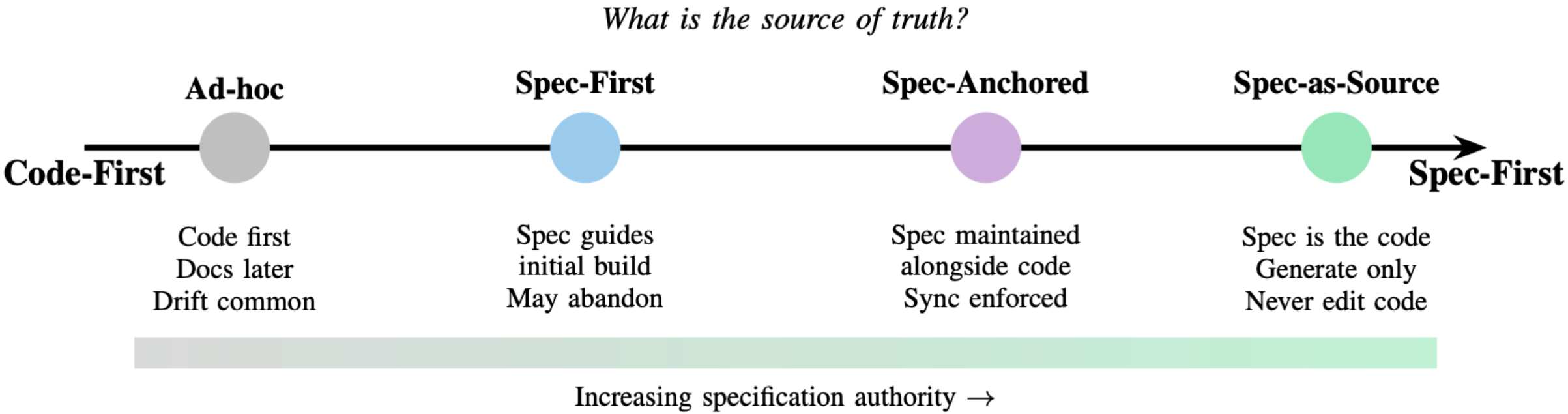
Source: Tufano, Michele, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. (2024) "AutoDev: Automated AI-Driven Development." arXiv preprint arXiv:2403.08299 (2024).

AutoDev: Automated AI-Driven Development



Source: Tufano, Michele, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. (2024) "AutoDev: Automated AI-Driven Development." arXiv preprint arXiv:2403.08299 (2024).

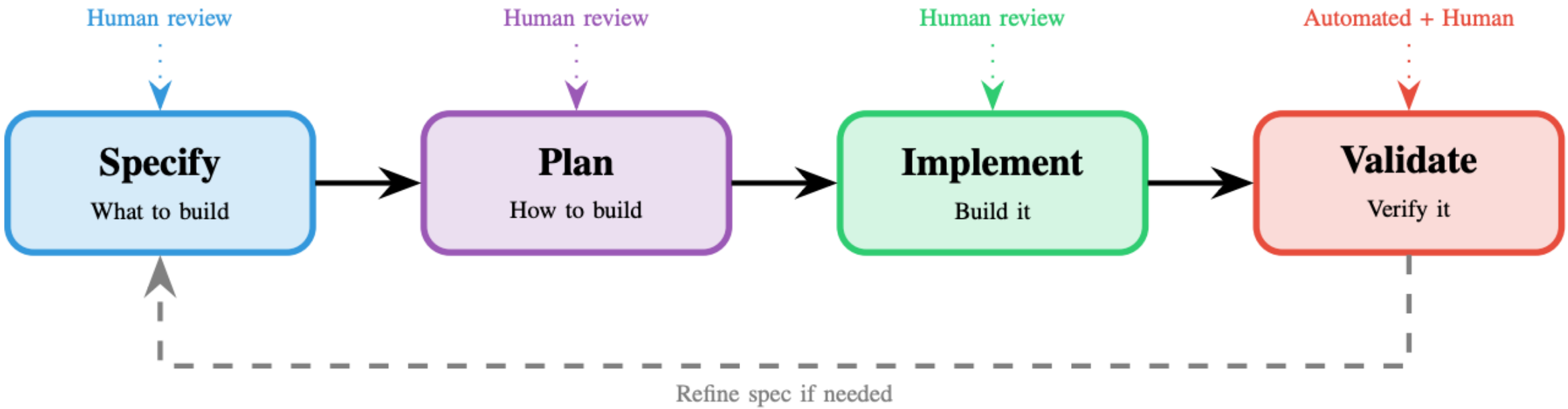
Specification-Driven Development (SDD)



Source: Deepak Babu Piskala. (2026) "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants." arXiv preprint arXiv:2602.00180 (2026).

Spec-Driven Development (SDD) workflow

Human review at each checkpoint ensures alignment with intent



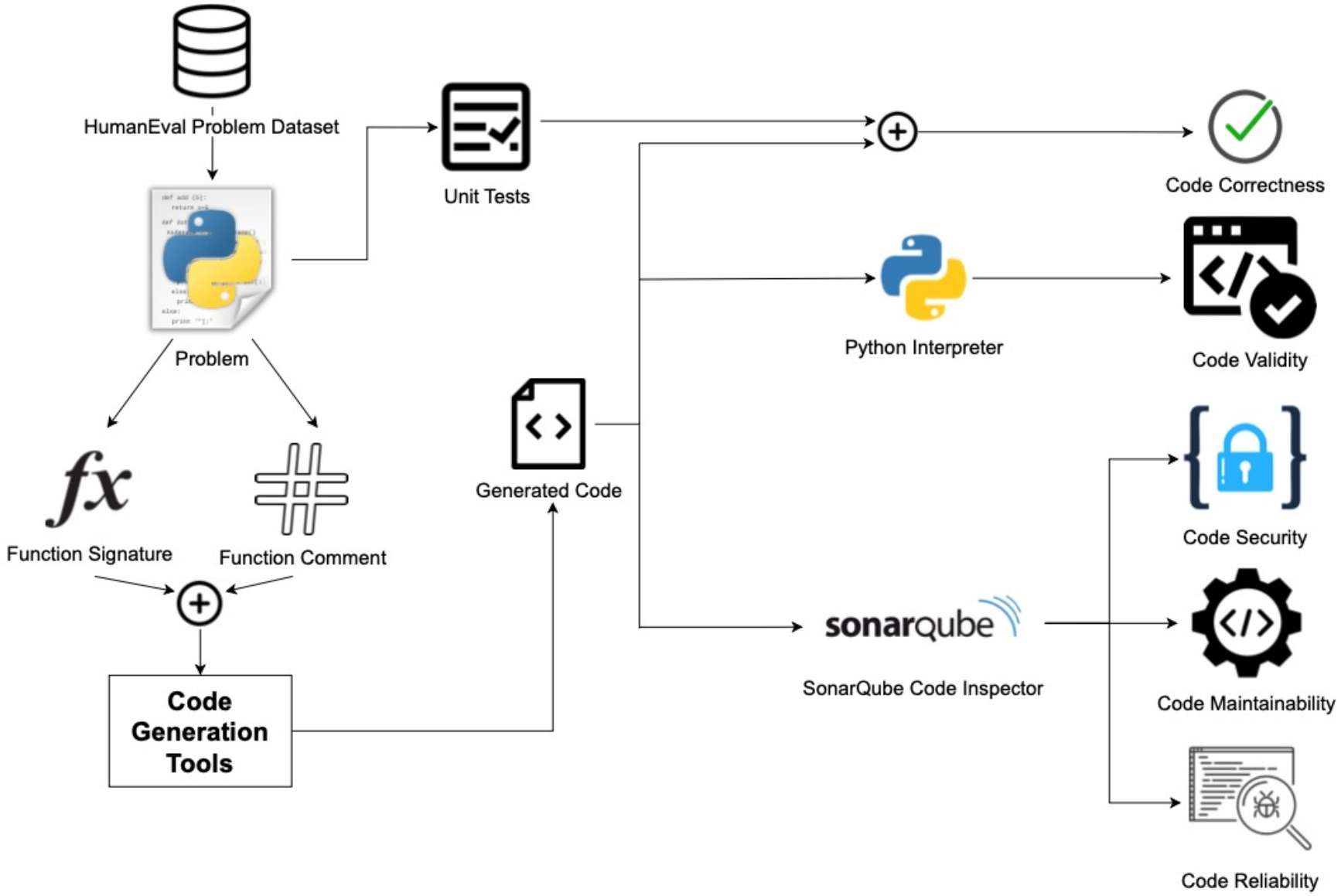
Source: Deepak Babu Piskala. (2026) "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants." arXiv preprint arXiv:2602.00180 (2026).

Spec-Driven Development (SDD) Supporting Tools and Frameworks

Category	Examples	Role in SDD
Behavior-Driven Development (BDD) Frameworks	Cucumber, SpecFlow, Behave	Write executable specs in plain language (Gherkin)
Test-Driven Development (TDD) Frameworks	RSpec, JUnit, pytest	Encode specifications as unit tests
API Specification	OpenAPI/Swagger, GraphQL SDL, Protocol Buffers	Define contracts; generate code and tests
Contract Testing	Pact, Specmatic	Verify implementations match specs
AI-Assisted SDD	GitHub Spec Kit, Amazon Kiro, Tessl	Structured AI workflows from spec to code
Model-Based Design	Simulink, SCADE	Visual specs that generate embedded code

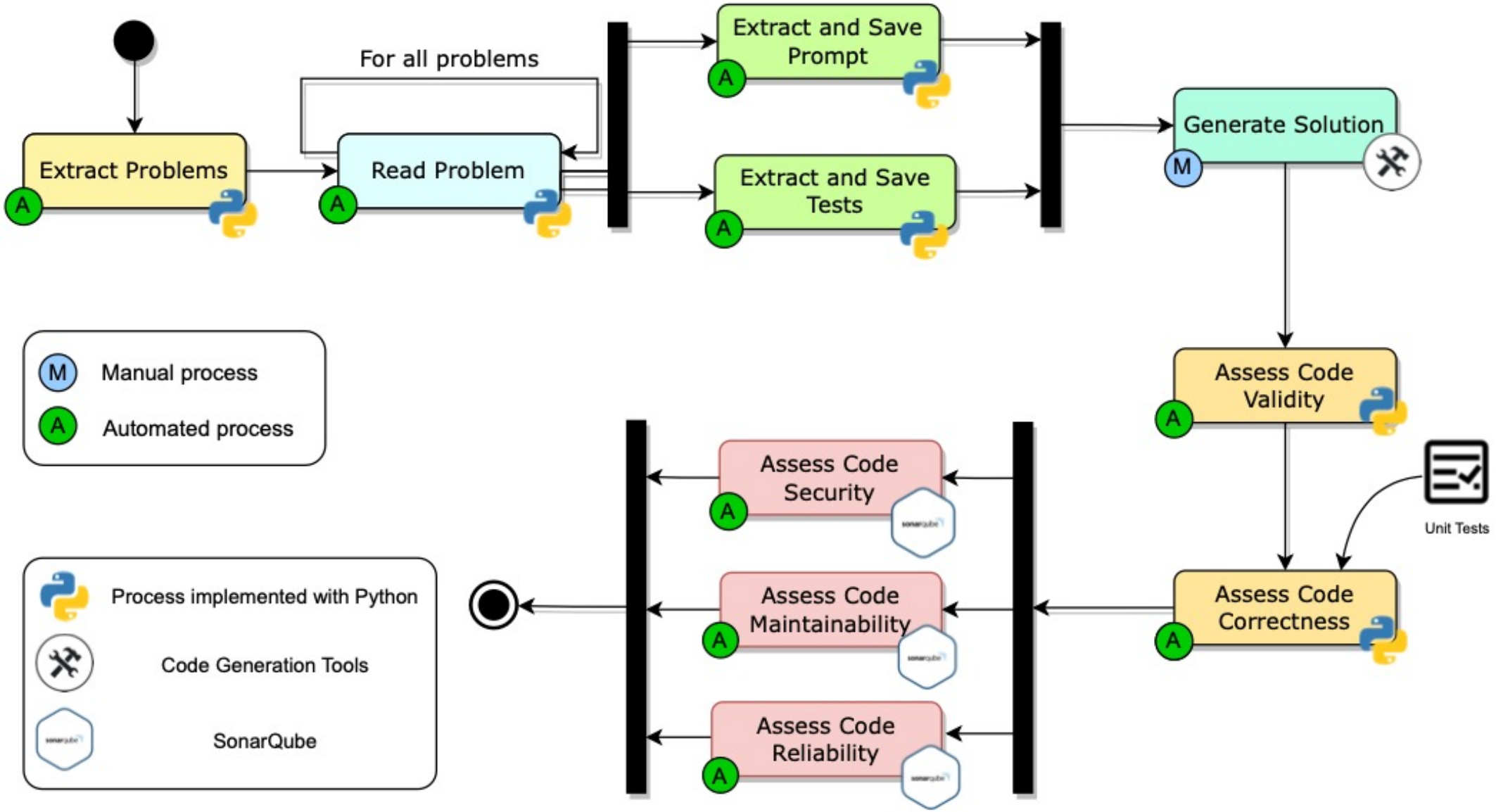
Source: Deepak Babu Piskala. (2026) "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants." arXiv preprint arXiv:2602.00180 (2026).

AI-Assisted Code Generation Tools: Experiment Setup



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 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).

Security and Privacy

Security and Privacy Outline

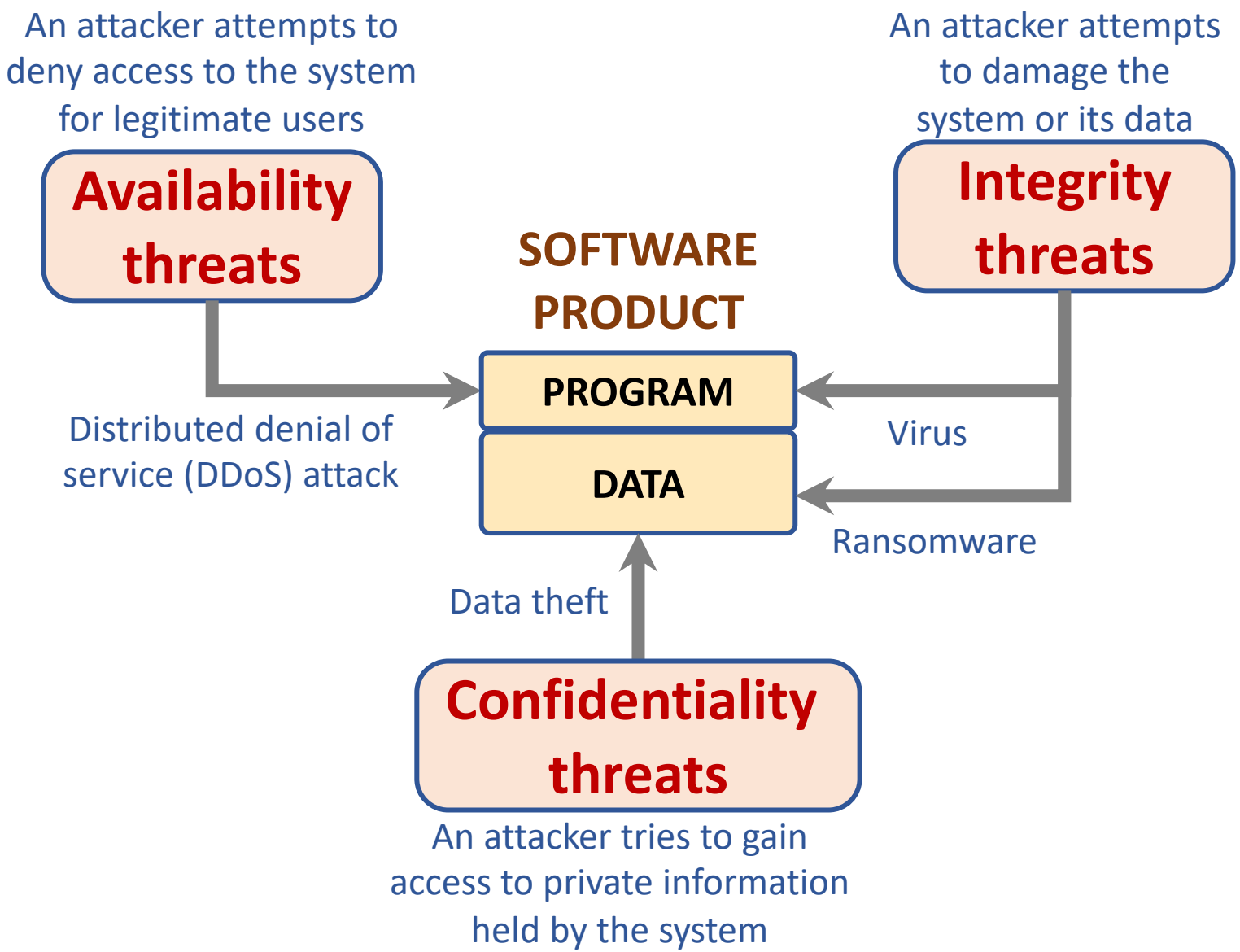
- **Security**
- **Privacy**

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.**

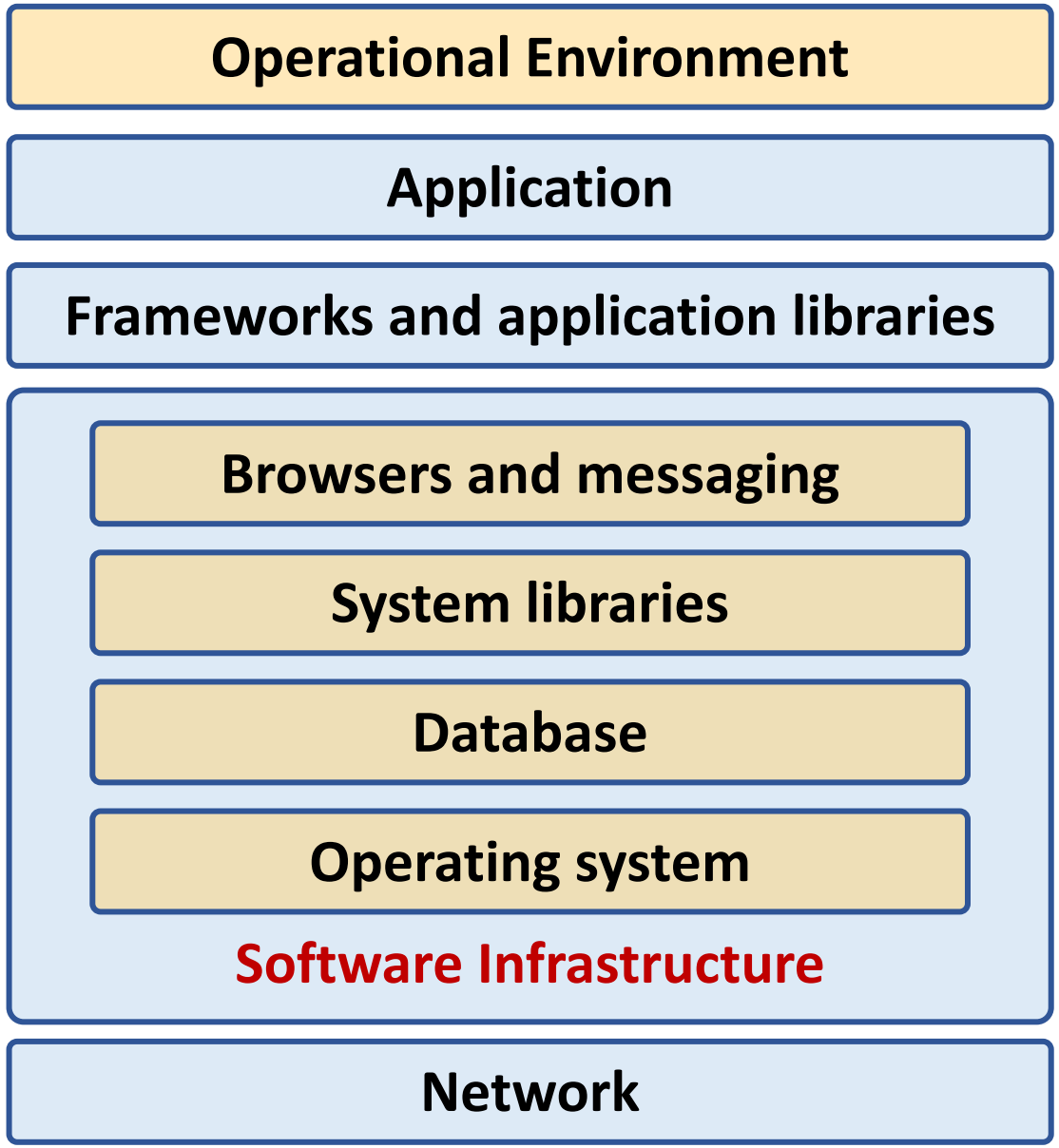
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Types of security threat



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

System infrastructure stack



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

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.

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

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.

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

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.**

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

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.**

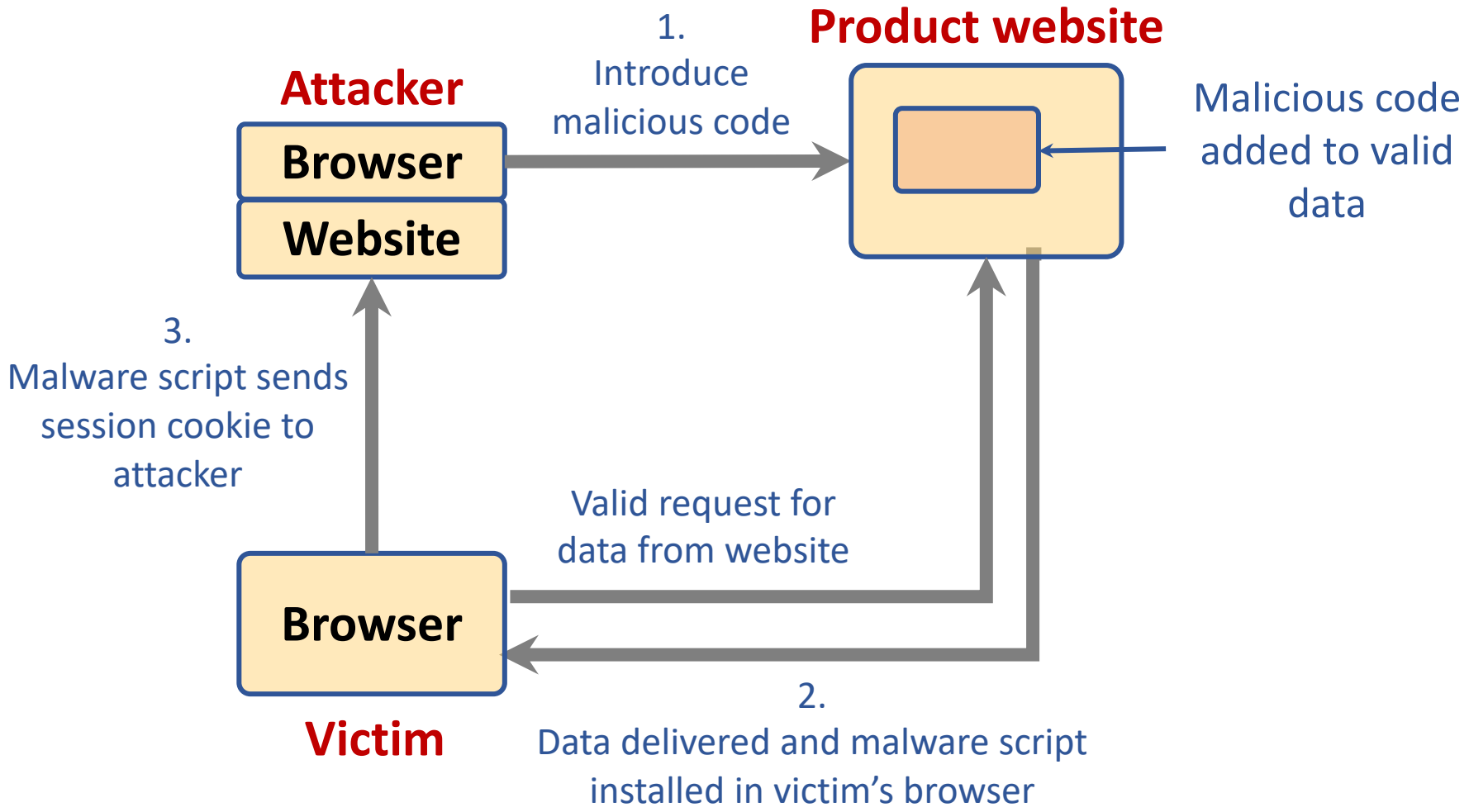
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Cross-site scripting attack



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

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.

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

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.

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

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.

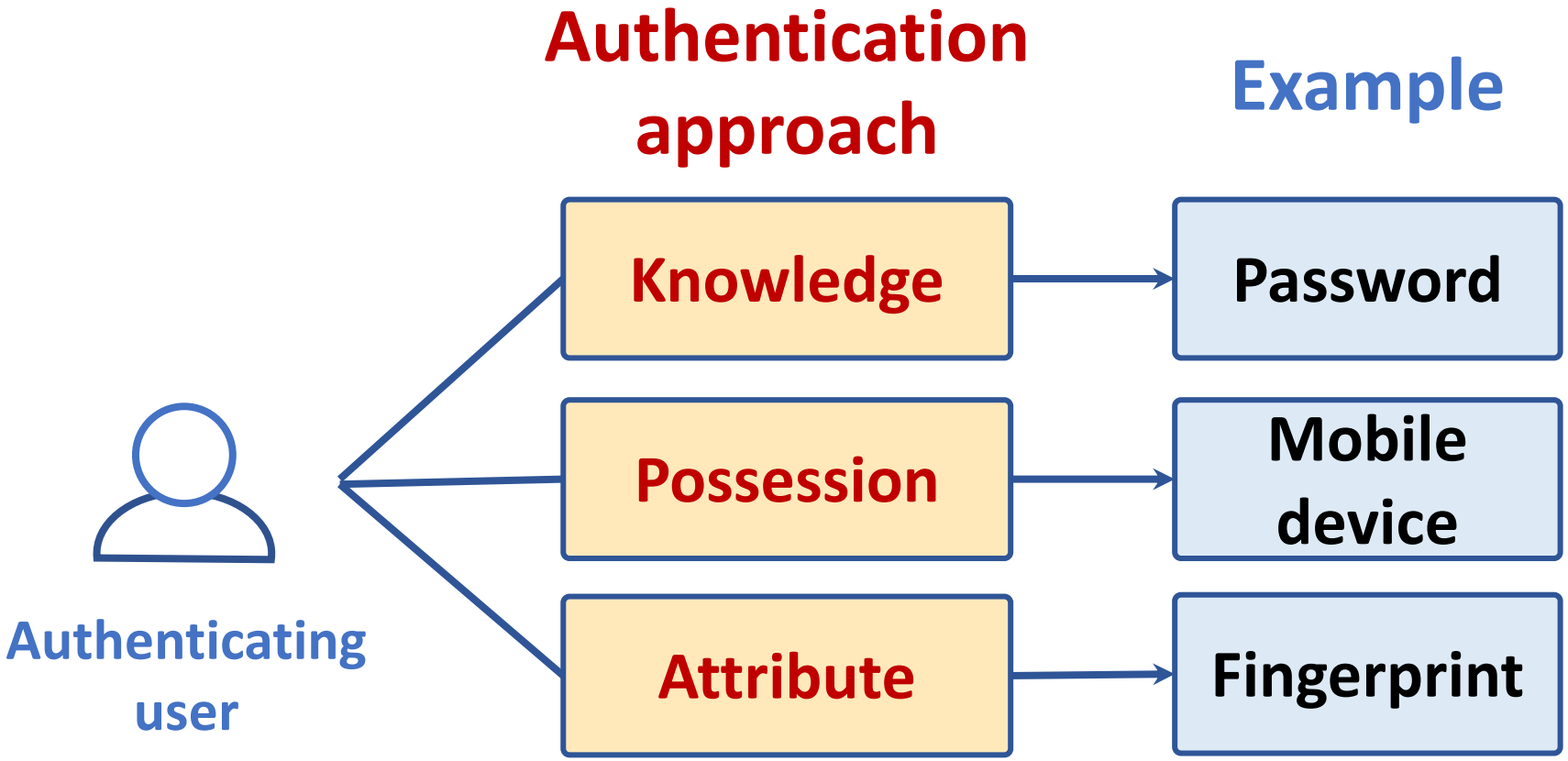
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Authentication approaches



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

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.

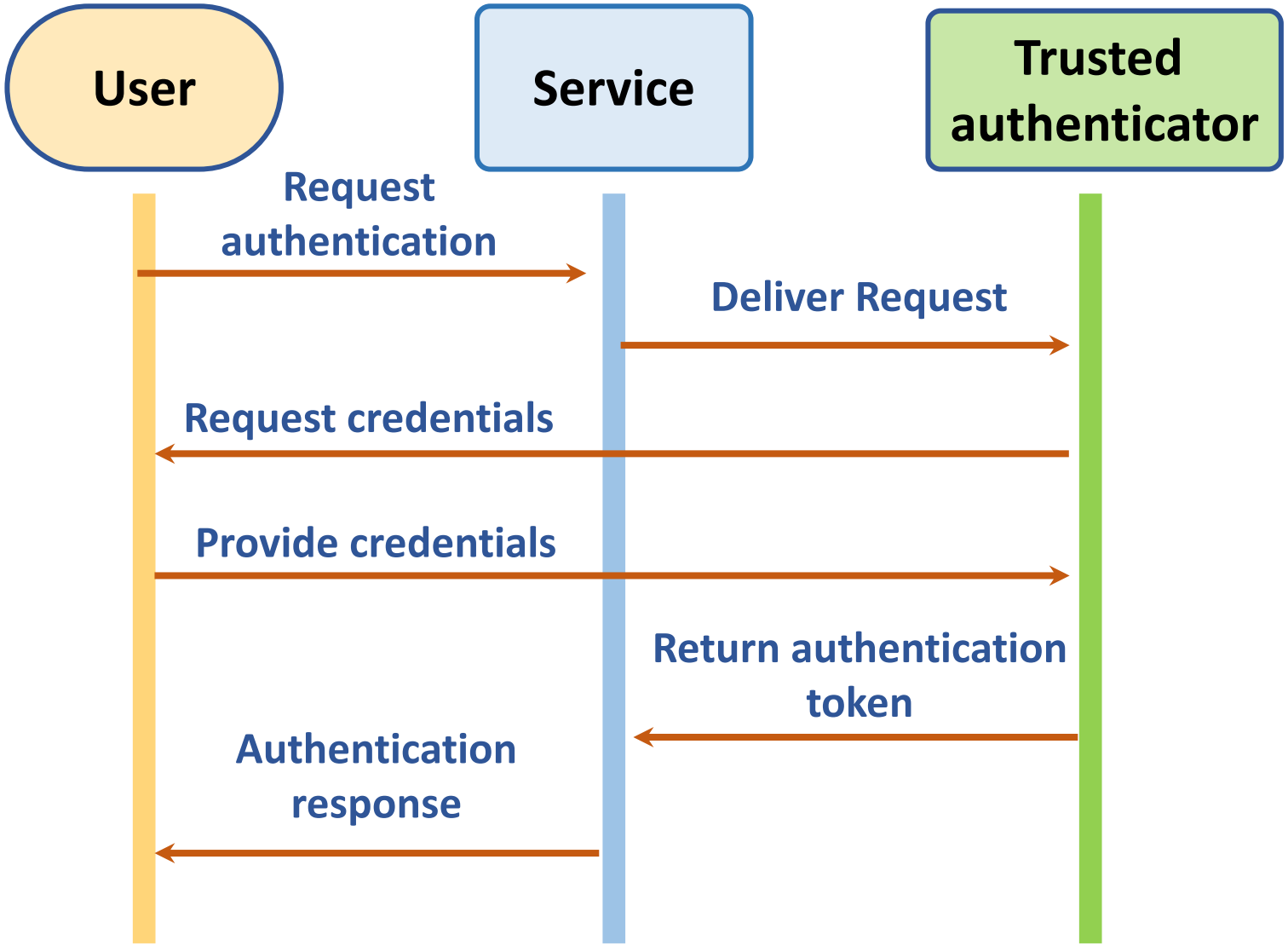
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Federated identity



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

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**

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

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.**

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

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**

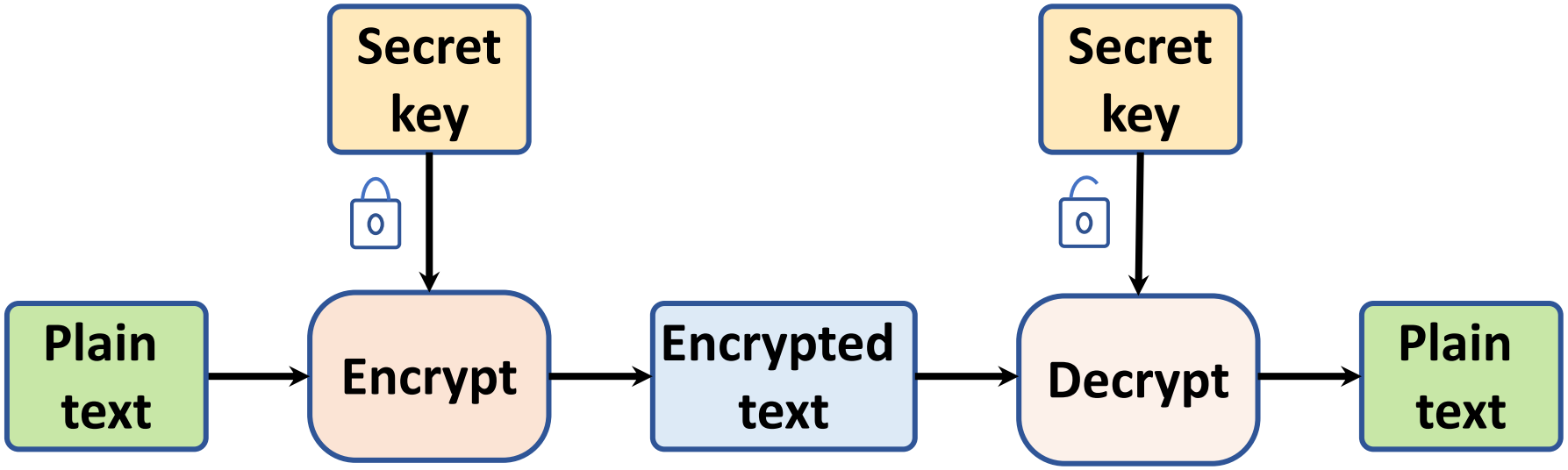
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

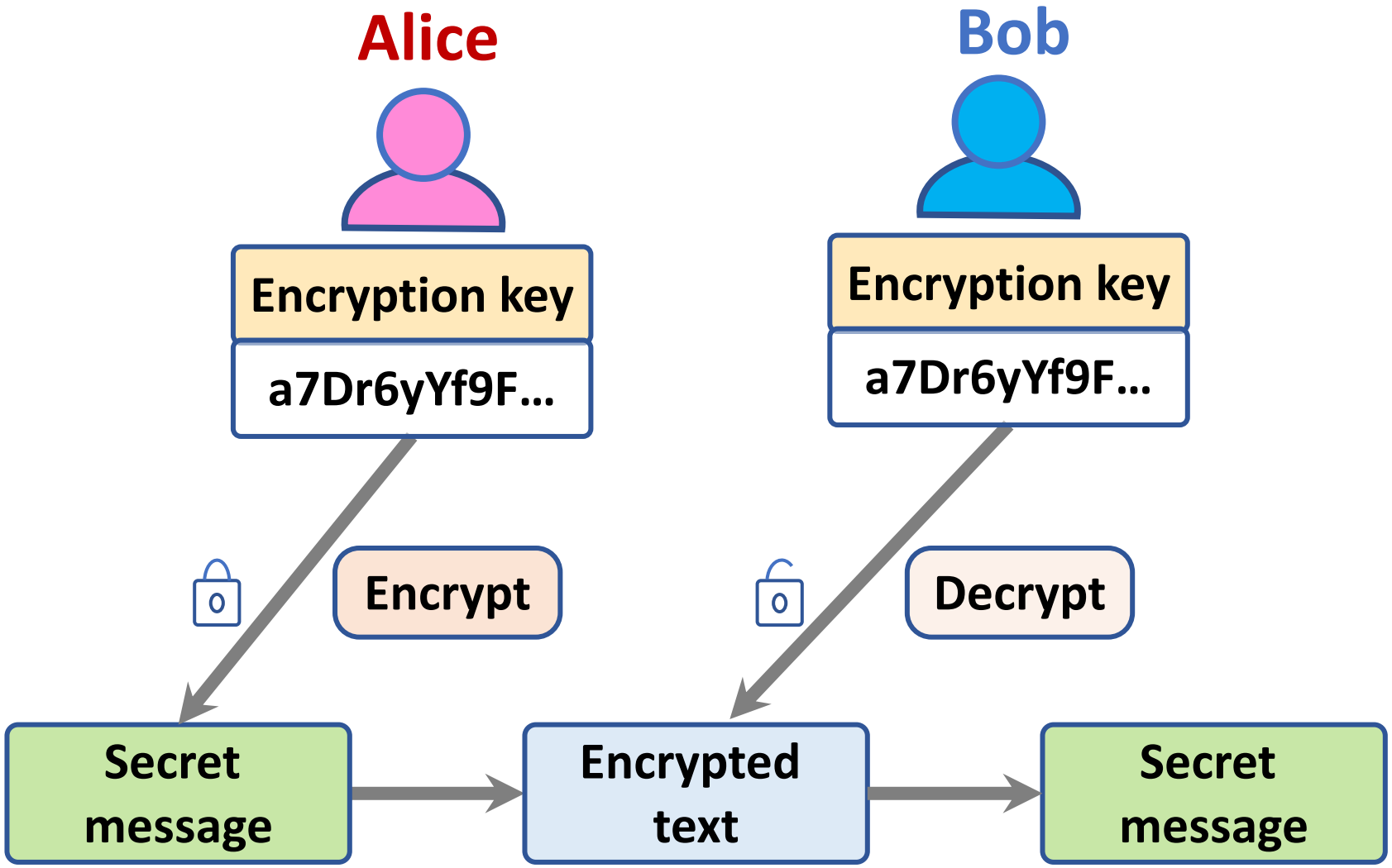
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Encryption and decryption



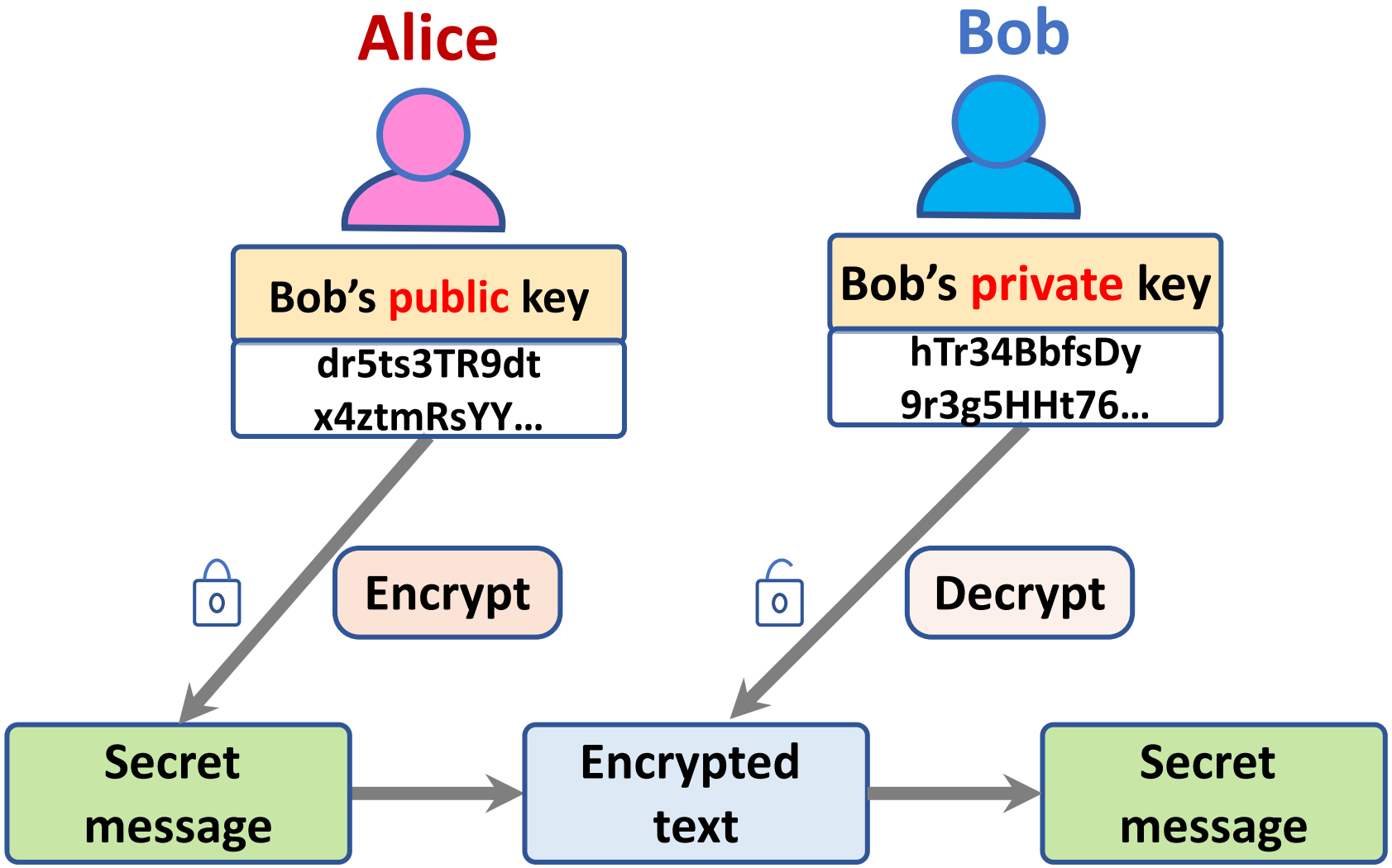
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Symmetric encryption



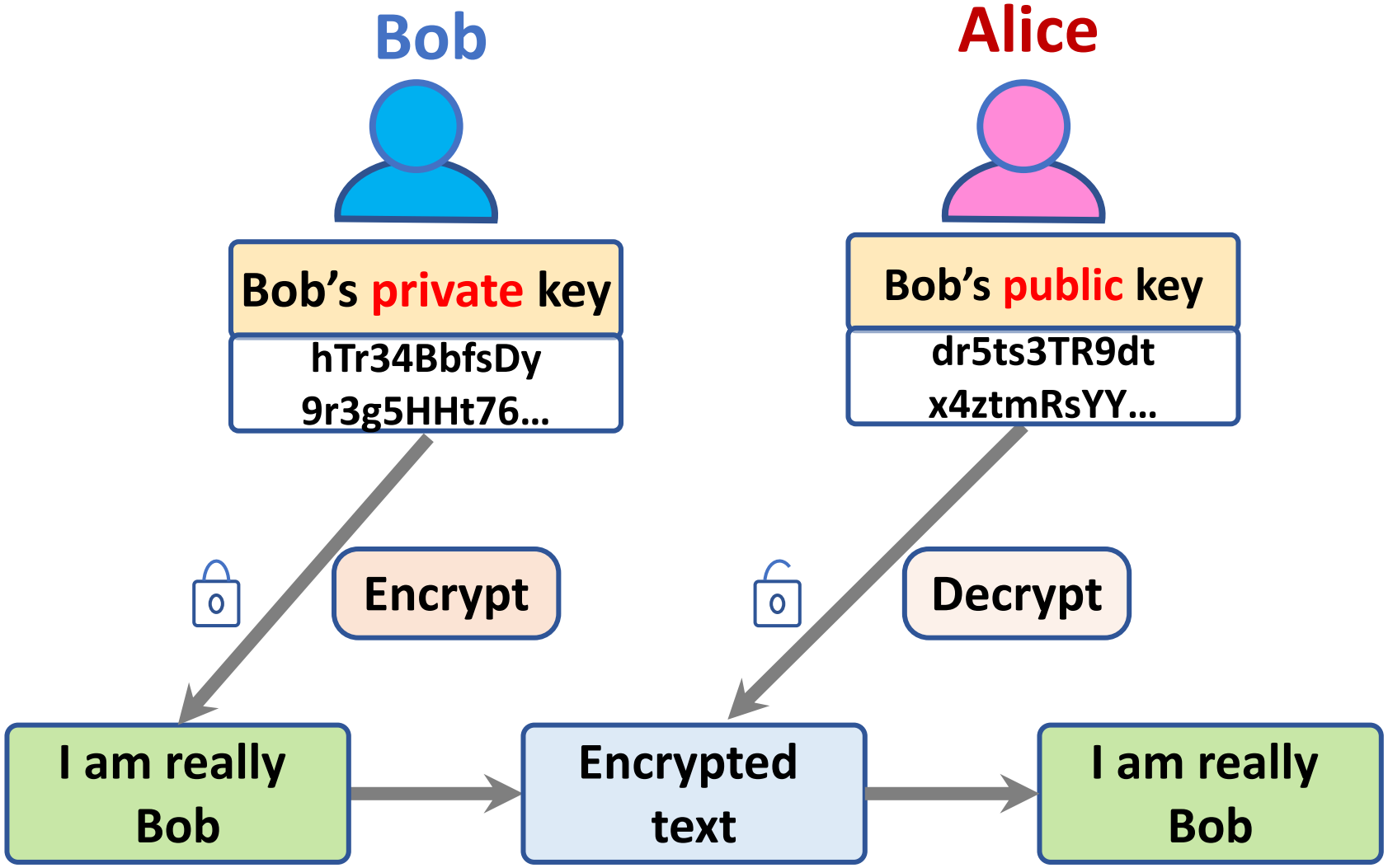
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Asymmetric encryption



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

Encryption for authentication



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

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

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

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.

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

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).**

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

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.

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

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.

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

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

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

Encryption levels

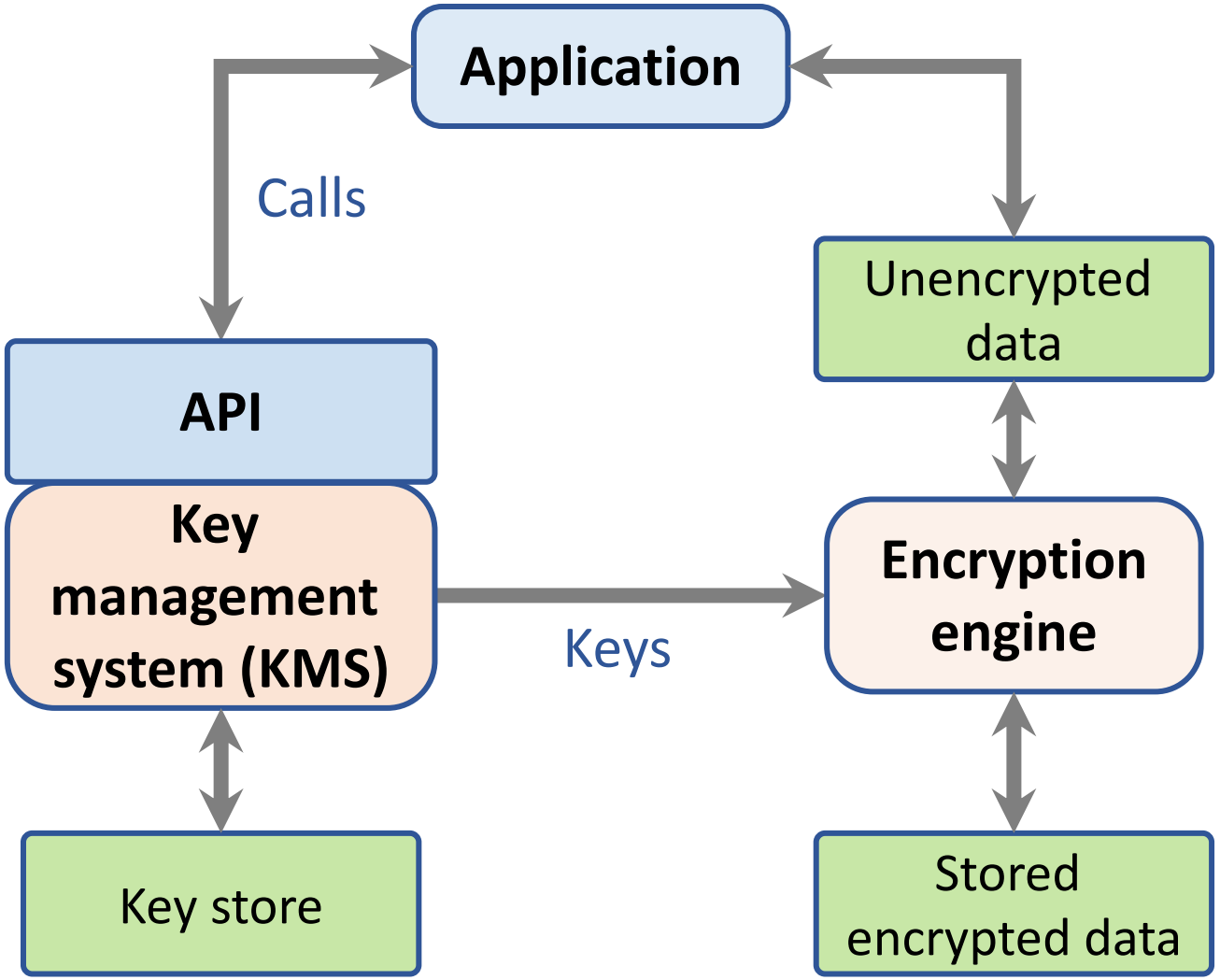
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Using a KMS for encryption management



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

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.

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

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.**

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

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.

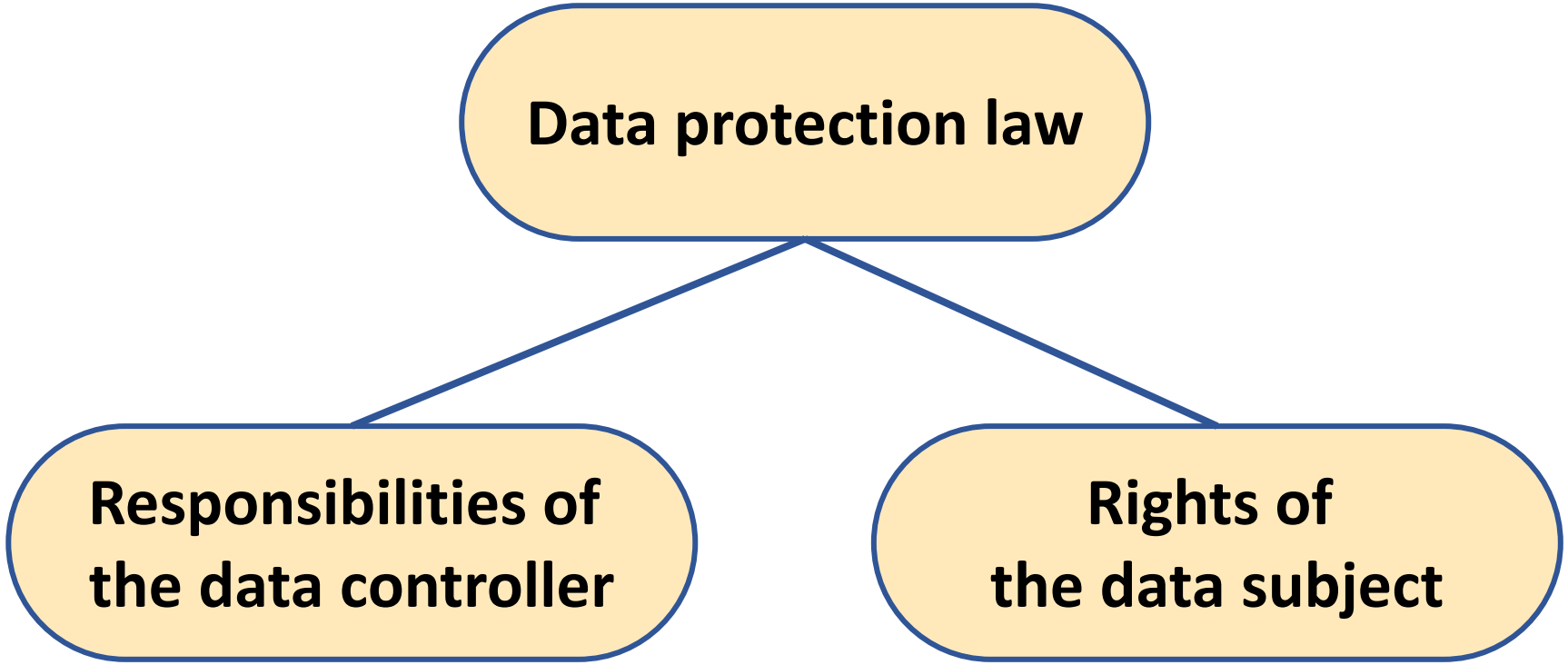
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.

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

Data protection laws



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

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.

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

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.

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

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.

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

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.

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

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.

Summary

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

Summary

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

Summary

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

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

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.

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

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.

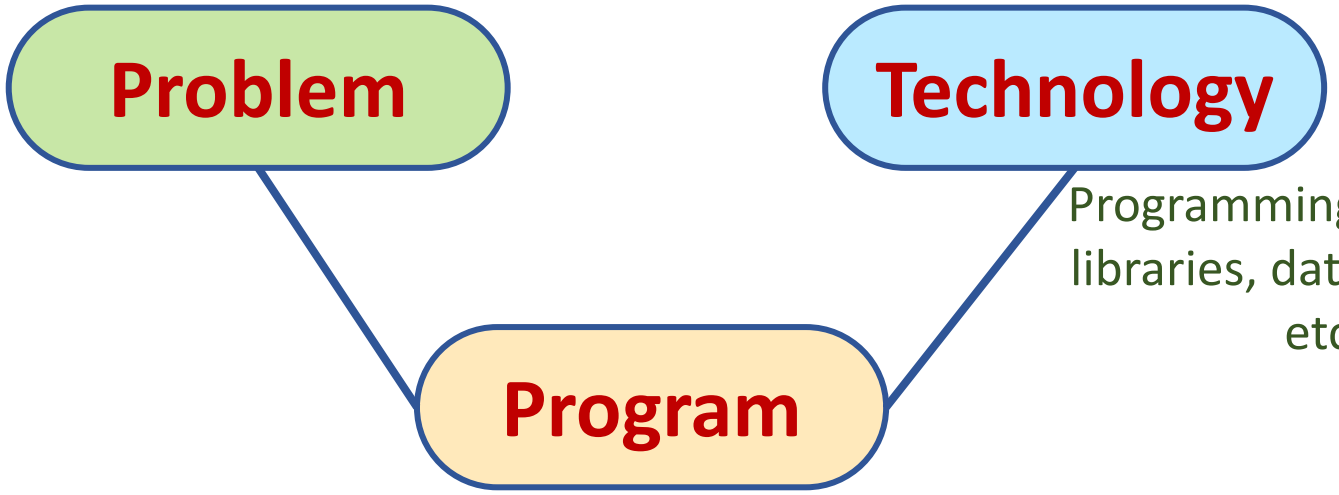
1. **Fault avoidance:** You should program in such a way that you avoid introducing faults into your program.
2. **Input validation:** You should define the expected format for user inputs and validate that all inputs conform to that format.
3. **Failure management:** You should implement your software so that program failures have minimal impact on product users.

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

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

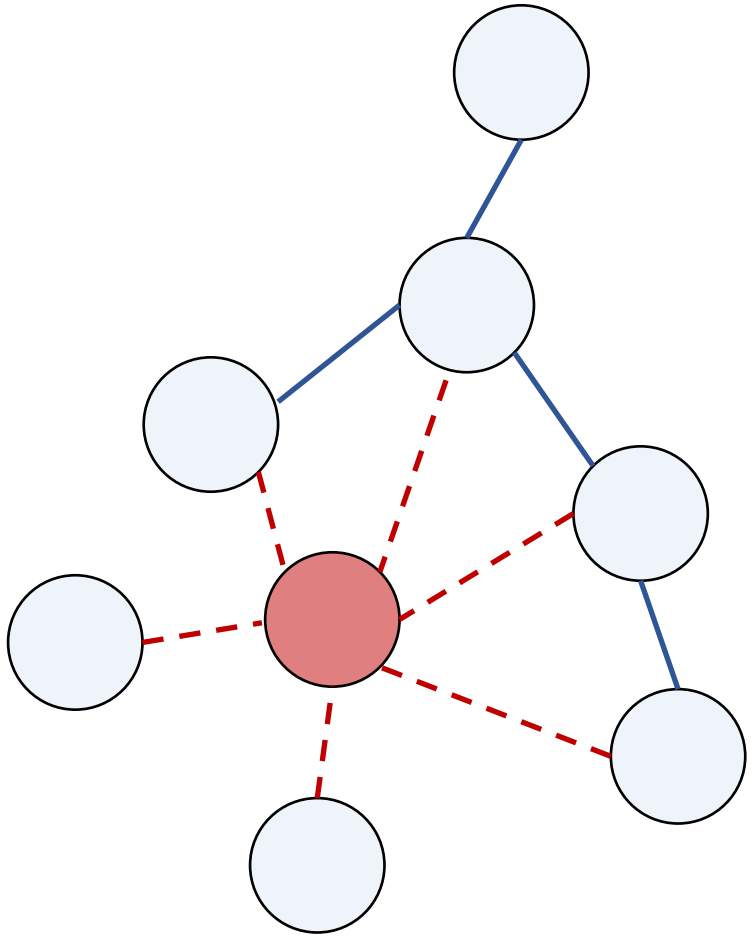


Programming language, libraries, database, IDE, etc.

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

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

Software complexity



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

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

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.

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.

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

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

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

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

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

Complexity reduction guidelines

Data complexity

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

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

Complexity reduction guidelines

Conditional complexity

- Avoid deeply nested conditional statements
- Avoid complex conditional expressions

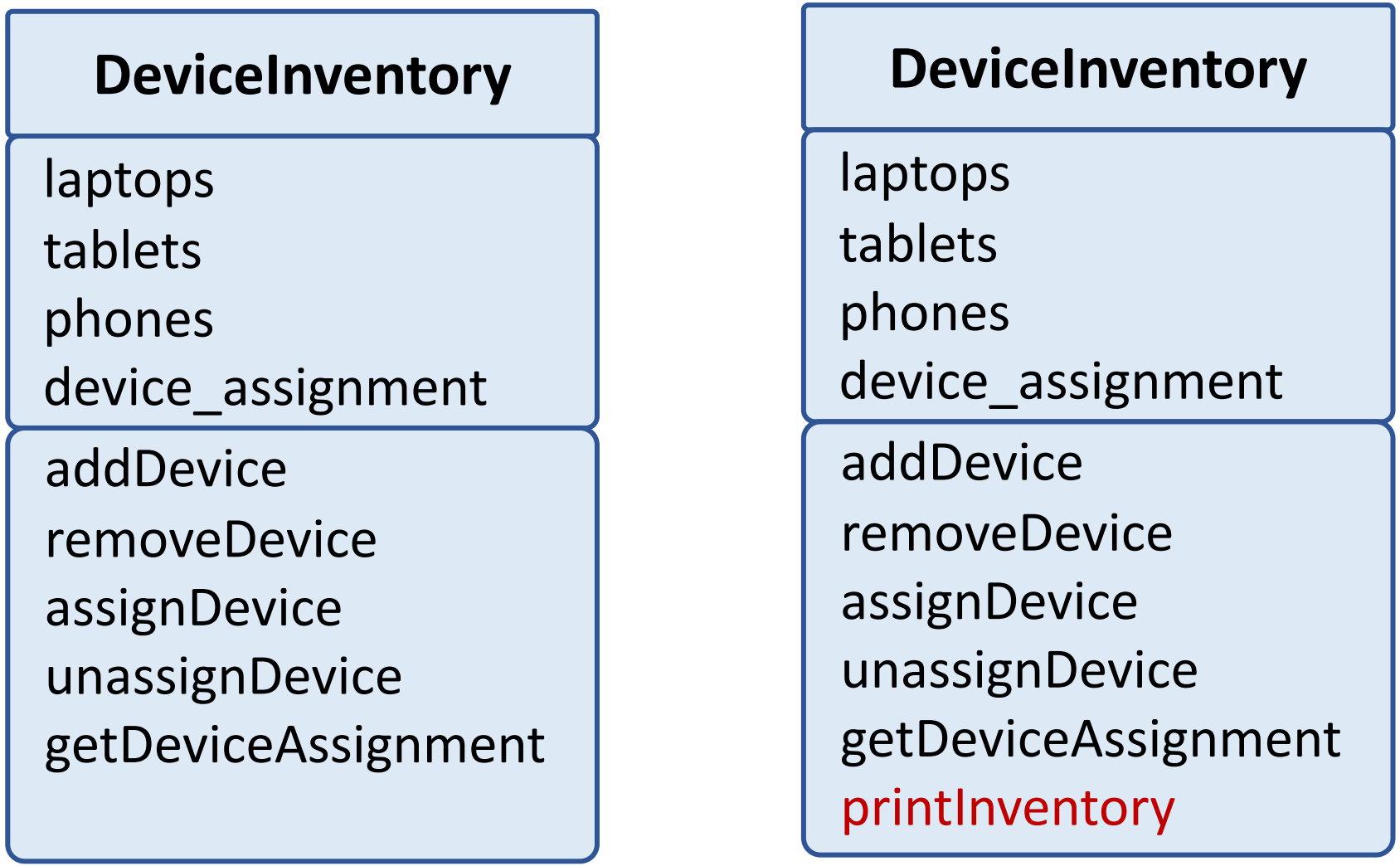
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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

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

The DeviceInventory class



(a) (b)

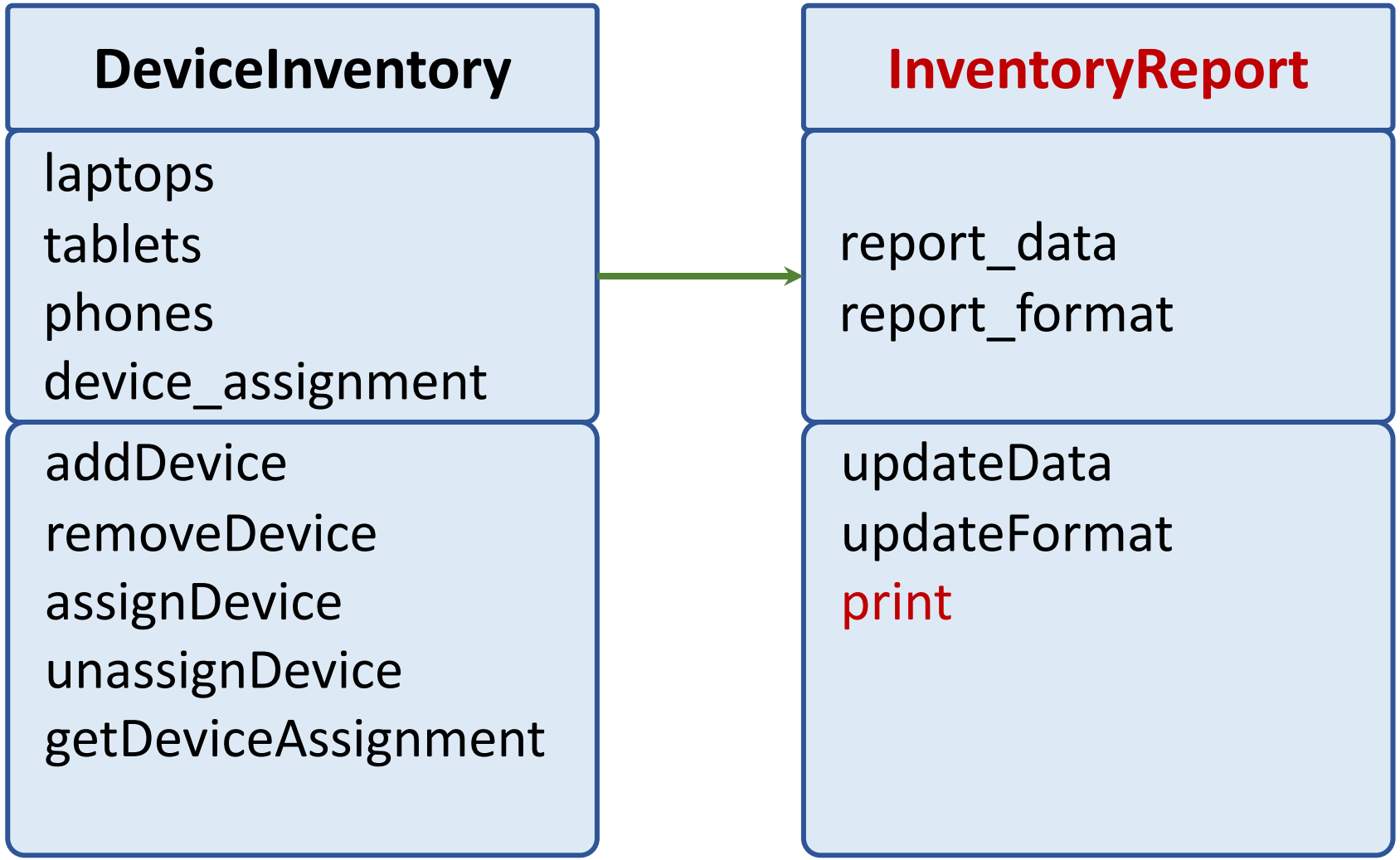
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.

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

The DeviceInventory and InventoryReport classes



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

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.**

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

Deeply nested if-then-else statements

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

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

Using guards to make a selection

```
def agecheck_with_guards(age, experience):

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

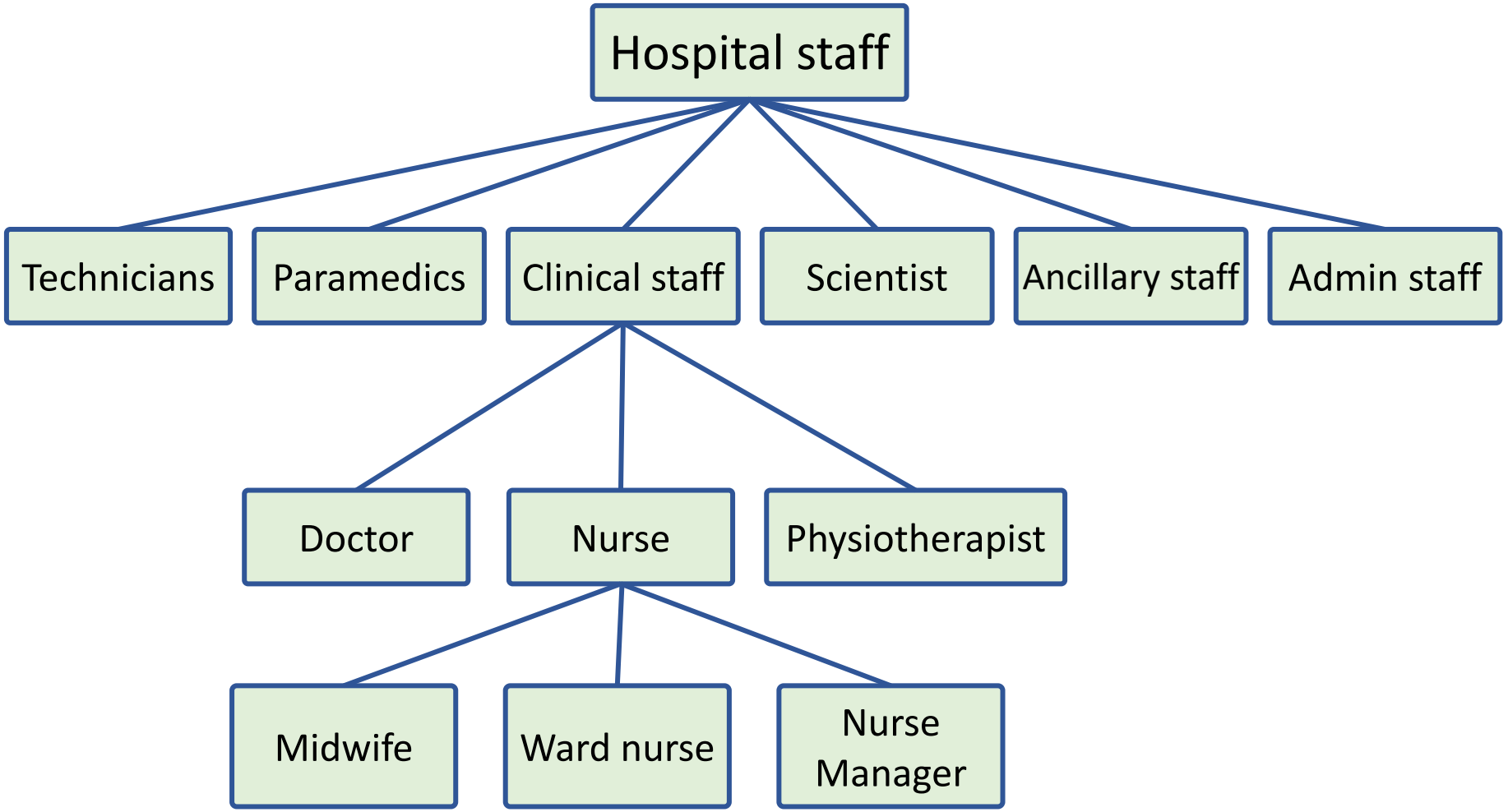
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Part of the inheritance hierarchy for hospital staff



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

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.

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.**

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

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.**

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

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.**

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

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.

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

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.**

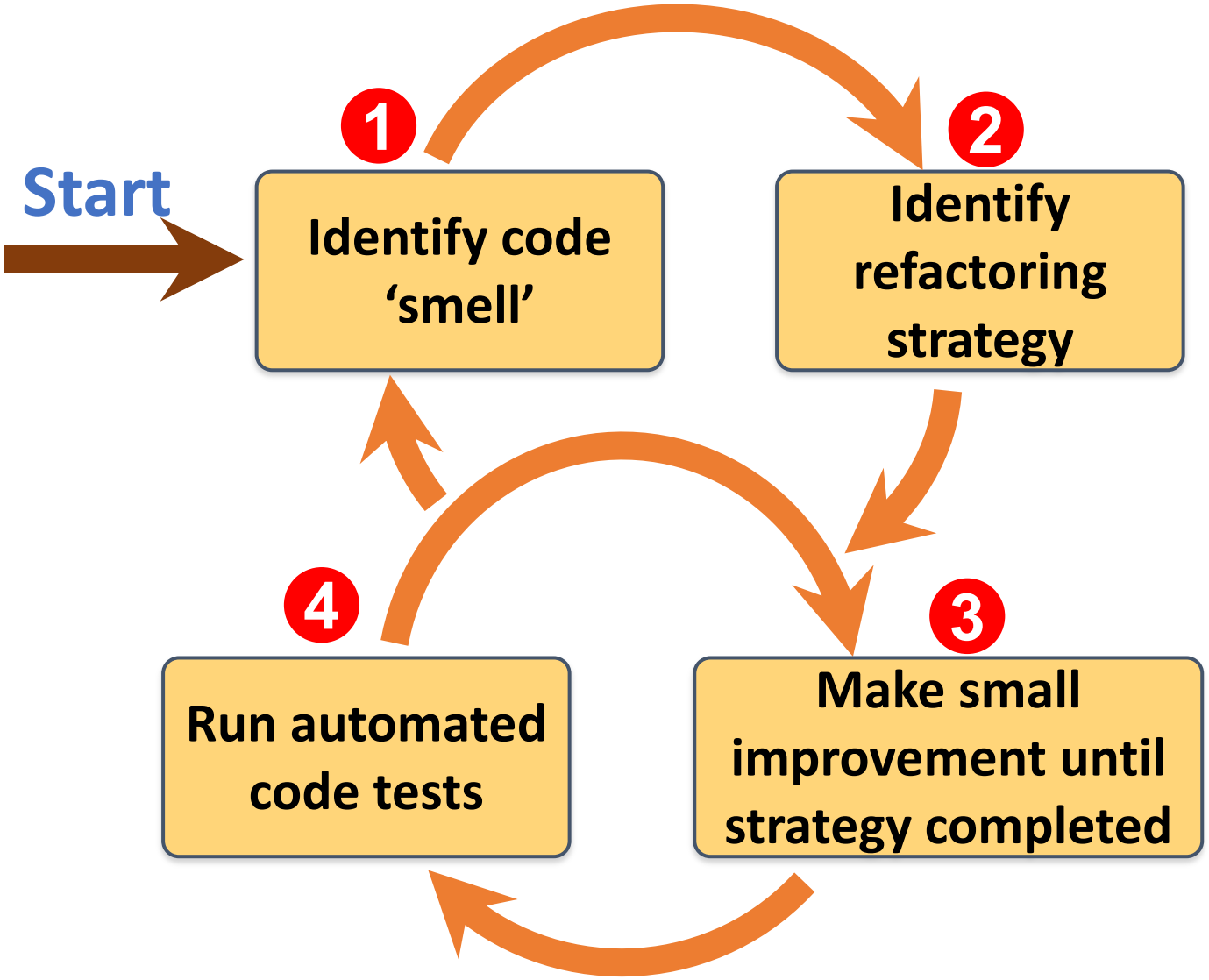
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

A refactoring process



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

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.**

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

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.

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

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.

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

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.

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

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.

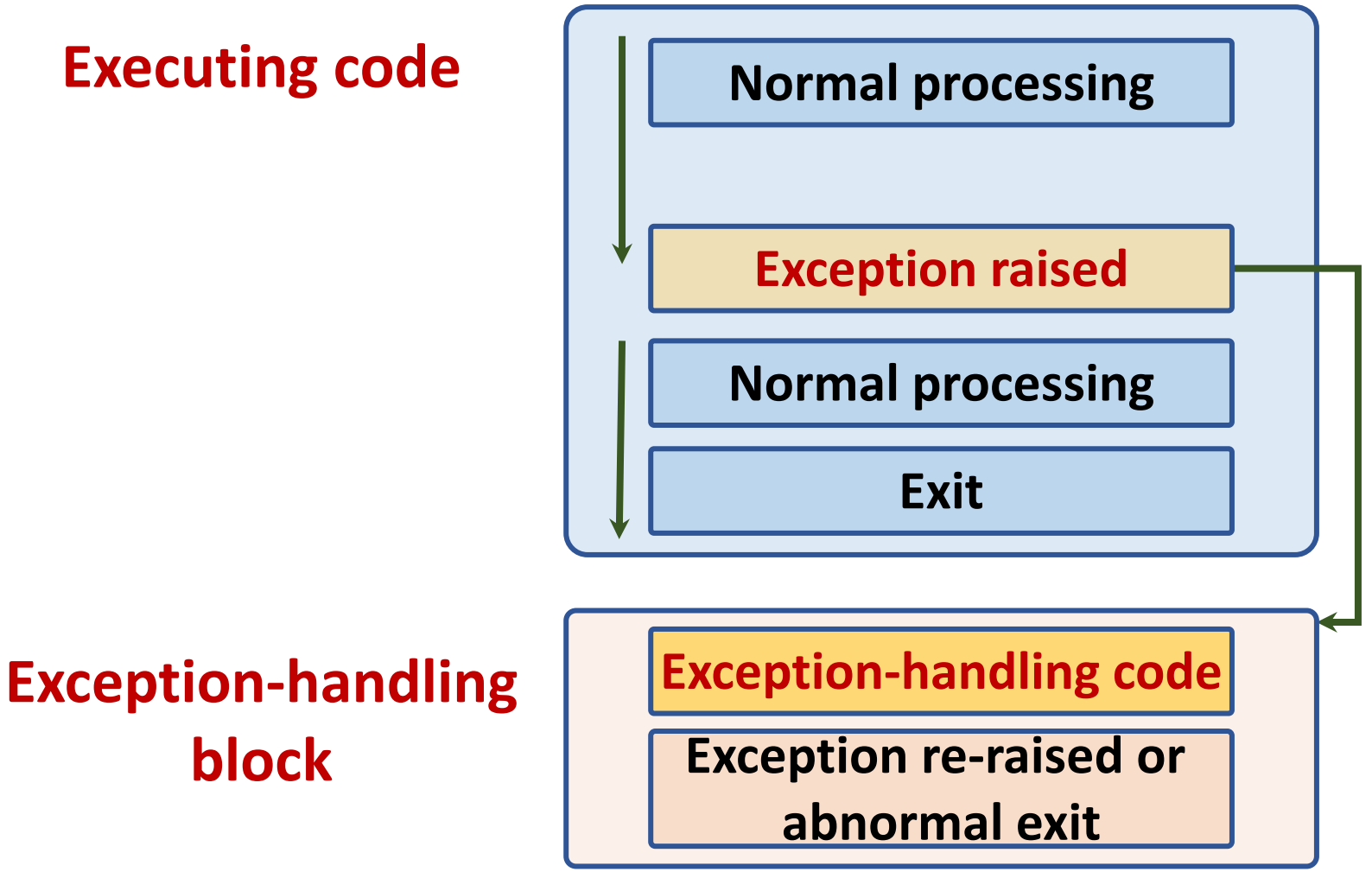
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Exception handling



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

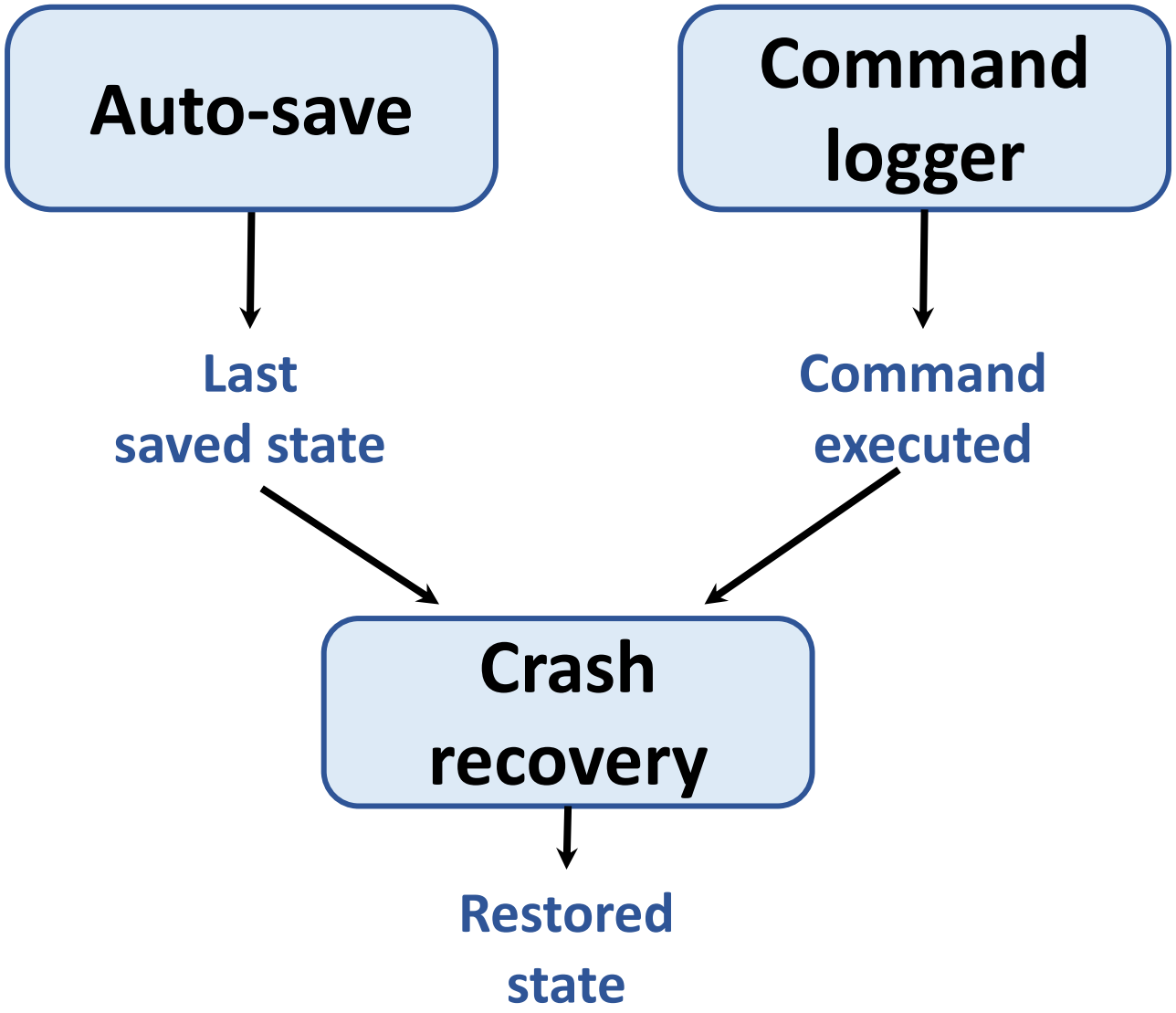
Python

try: except: finally:

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

Source: Python Try Except: https://www.w3schools.com/python/python_try_except.asp

Auto-save and activity logging



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

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.

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

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.

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.

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.

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

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.**

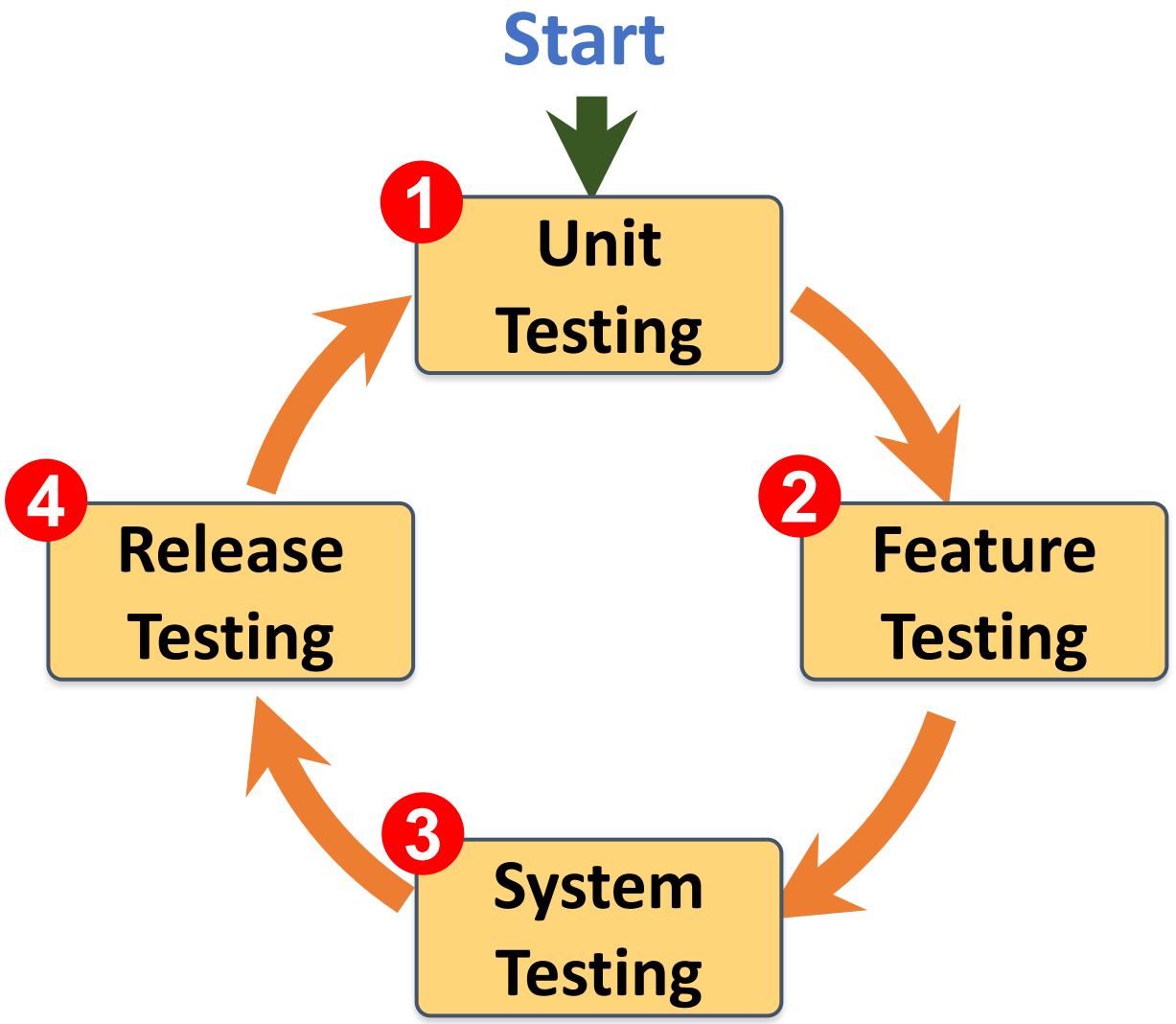
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Functional testing



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

A name checking function

```
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
```

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.

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

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.

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

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.

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.**

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

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.

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.

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

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.

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

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.

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

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.**

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

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.

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

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.**

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

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.**

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

Release testing and System testing

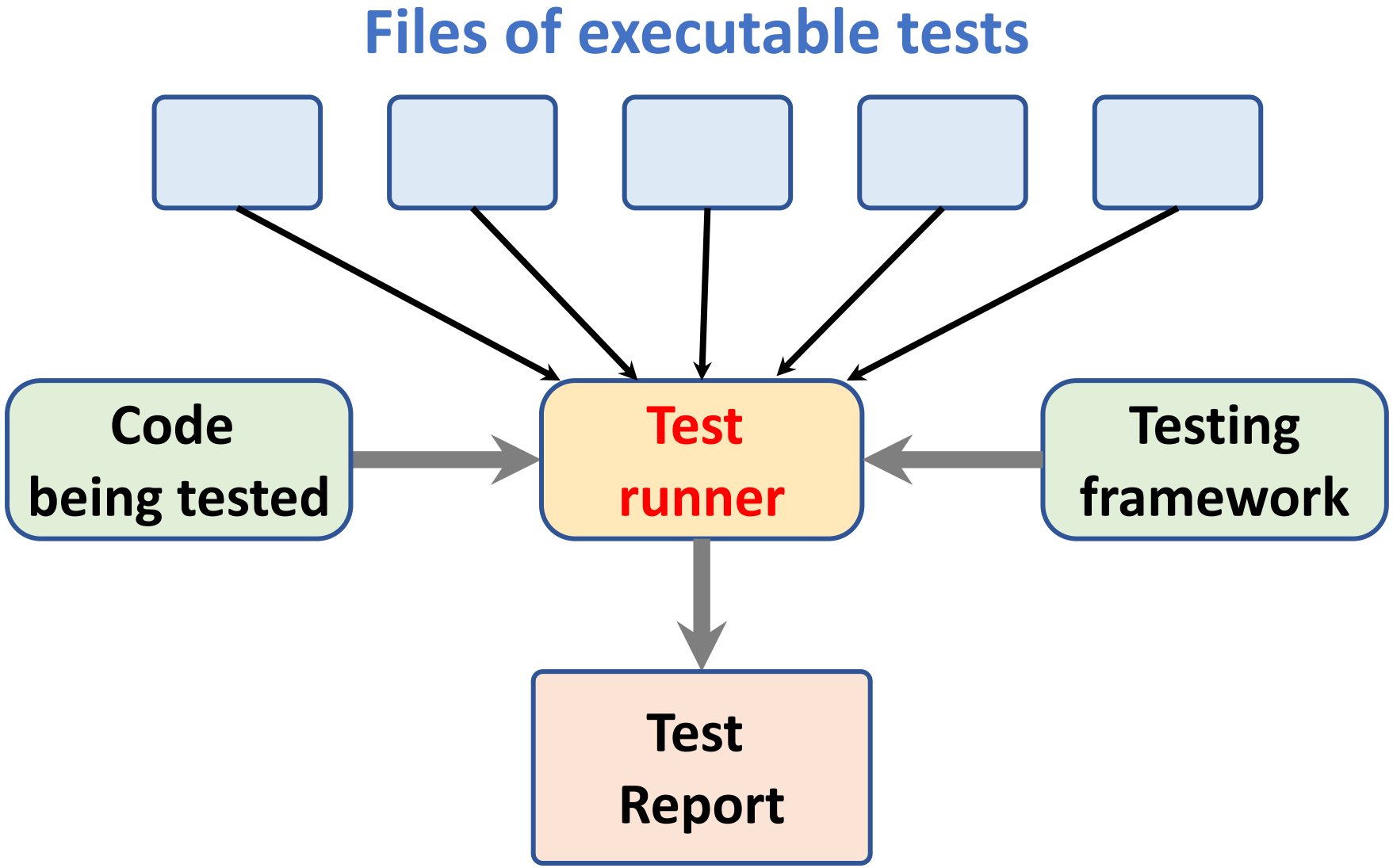
- **The fundamental differences between release testing and system testing are:**
  - **Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.**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

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

Automated testing



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

Test methods for an interest calculator

```
# 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)
```

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

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.

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

Executable tests for the namecheck function (1)

```
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)

```
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"))
```

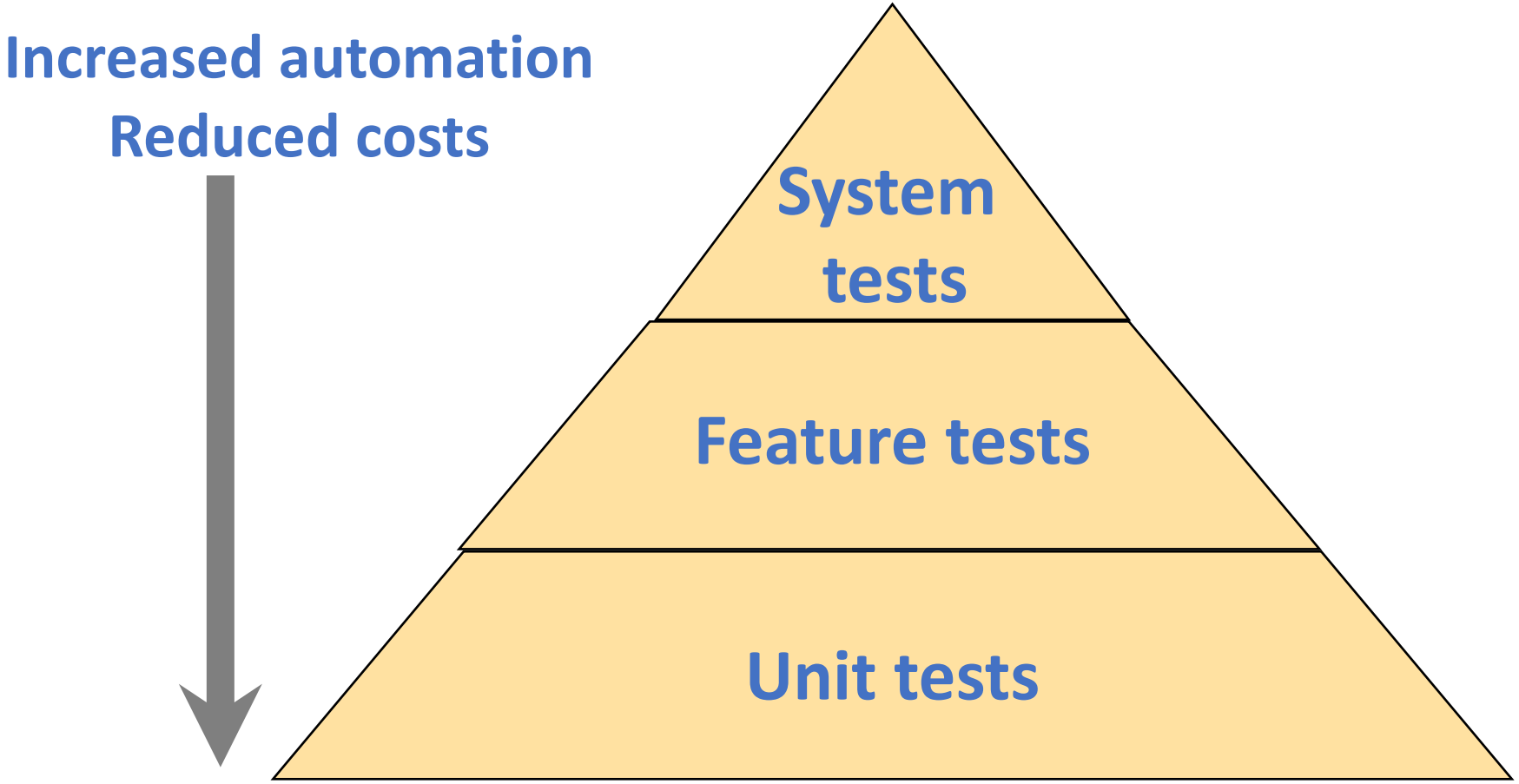
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Code to run unit tests from files

```
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)
```

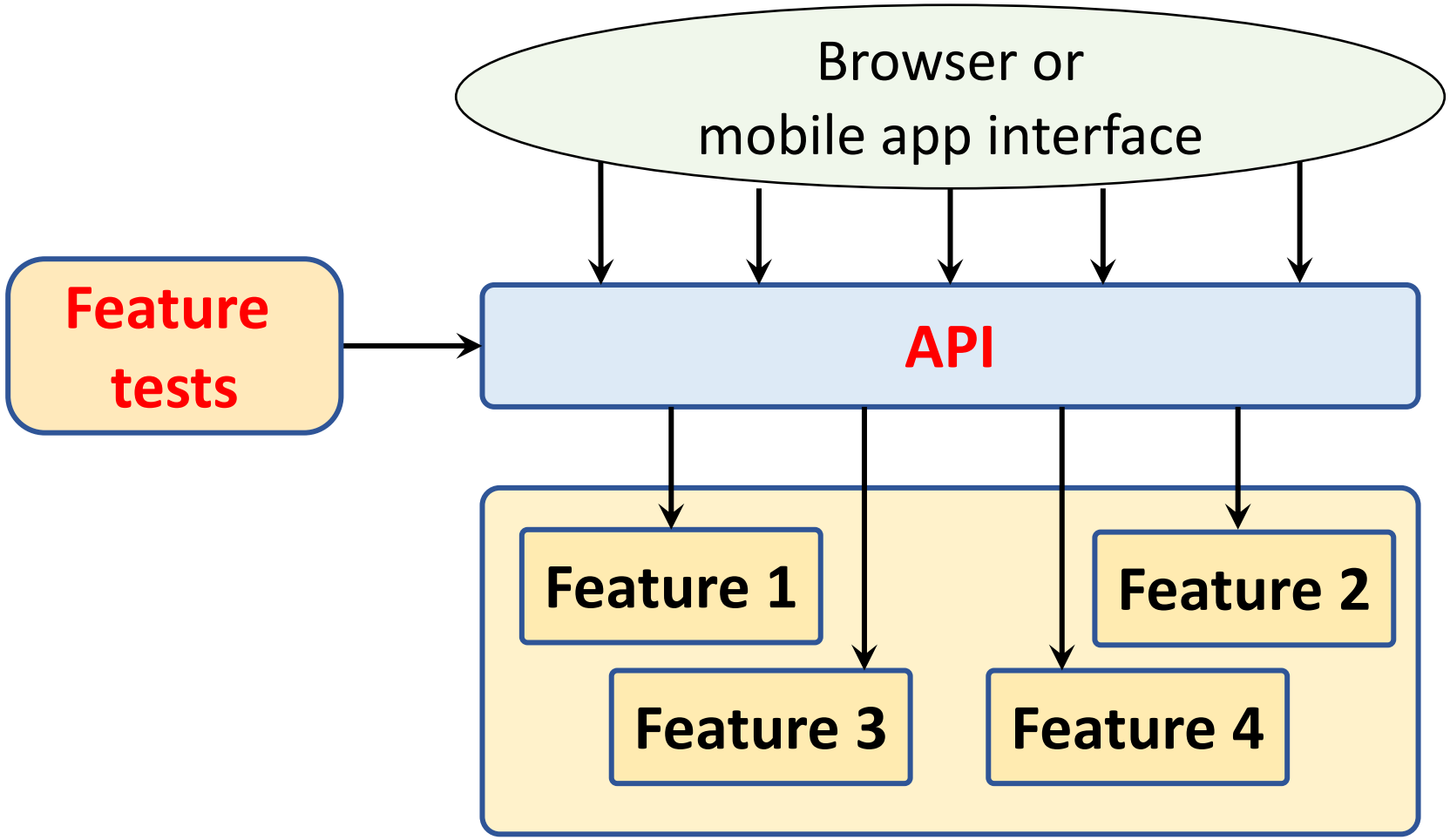
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

The test pyramid



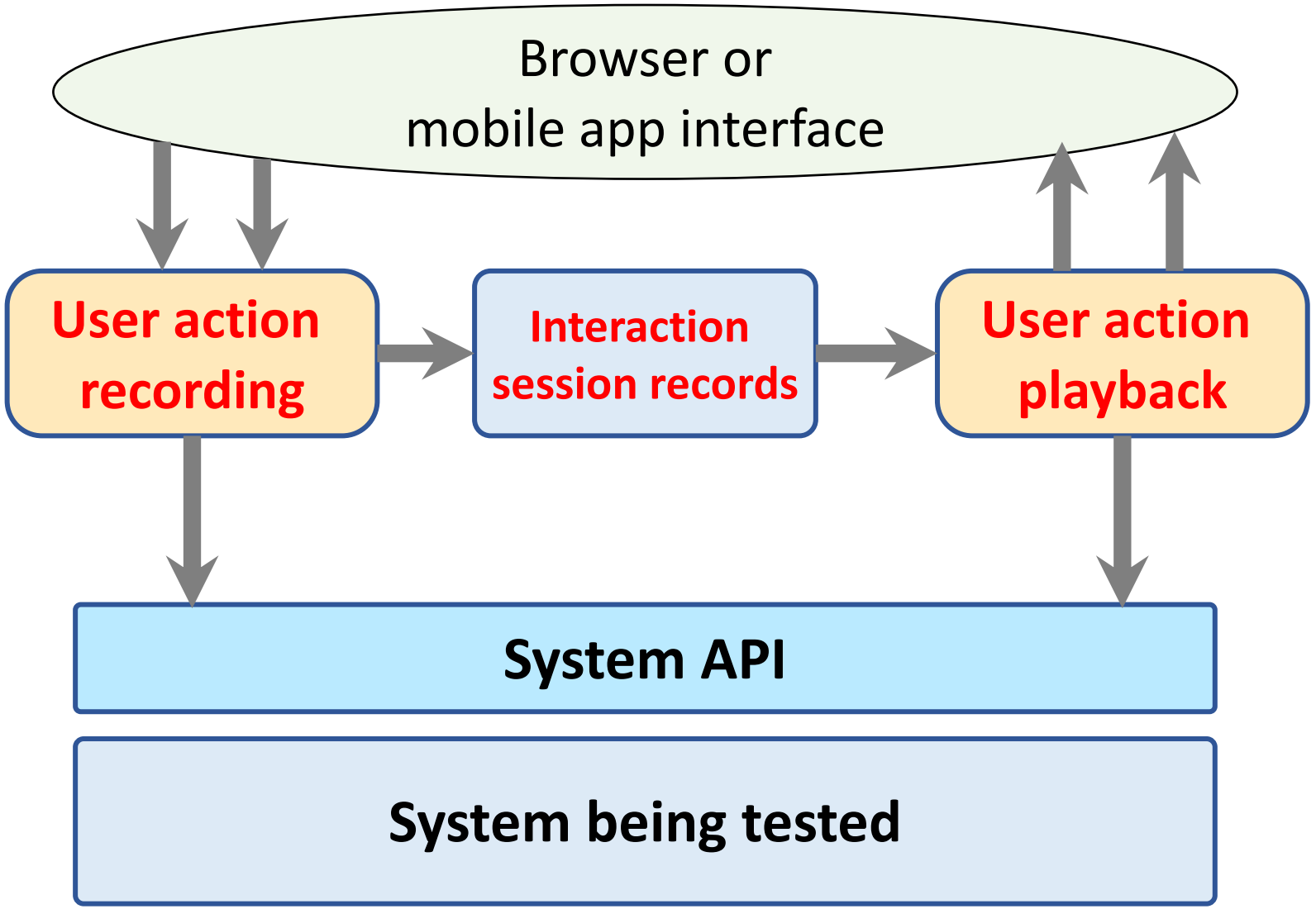
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Feature editing through an API



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

Interaction recording and playback



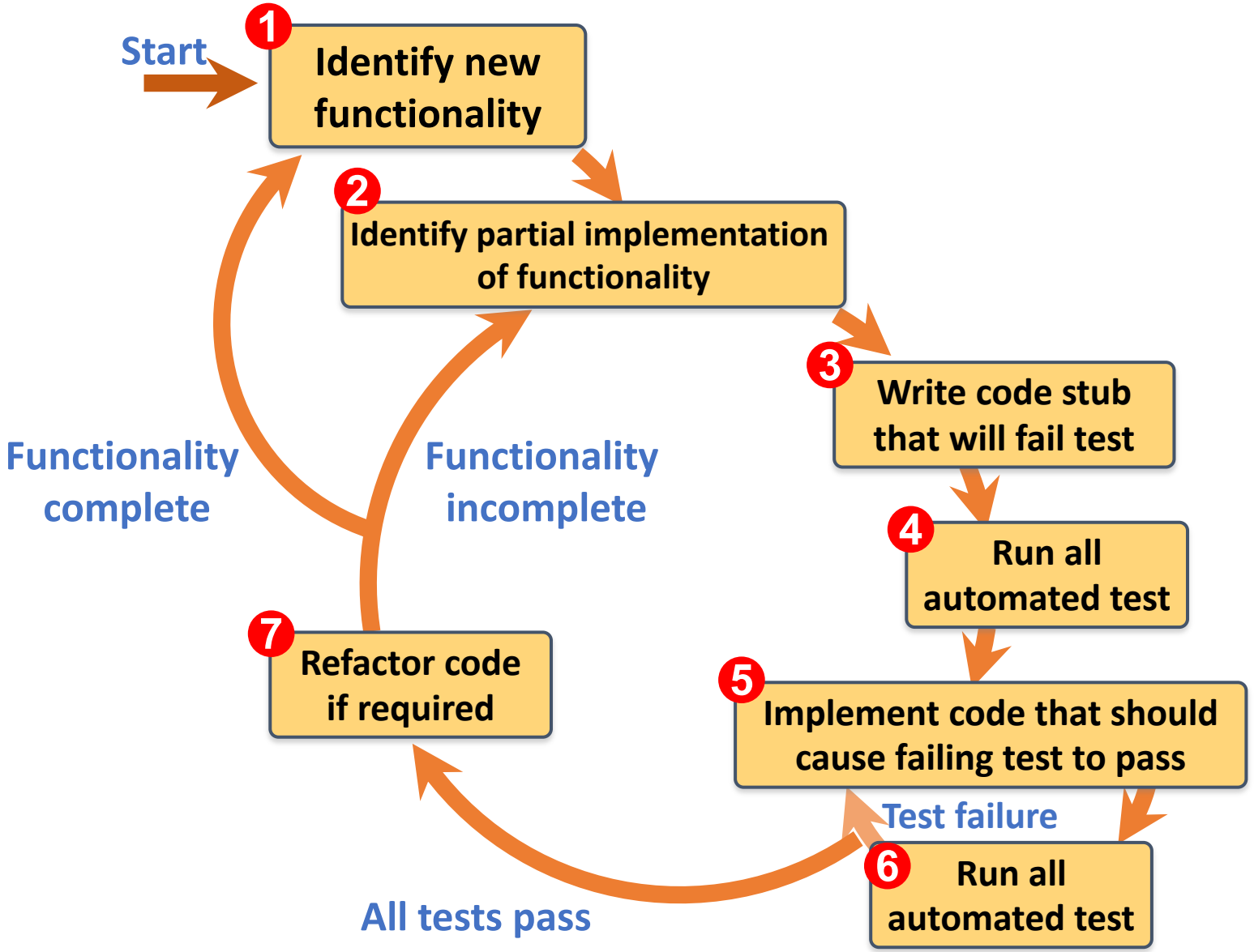
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.**

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

Test-driven development (TDD)



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

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.

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

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.

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

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.

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

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

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

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.**

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

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.

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

Risk analysis

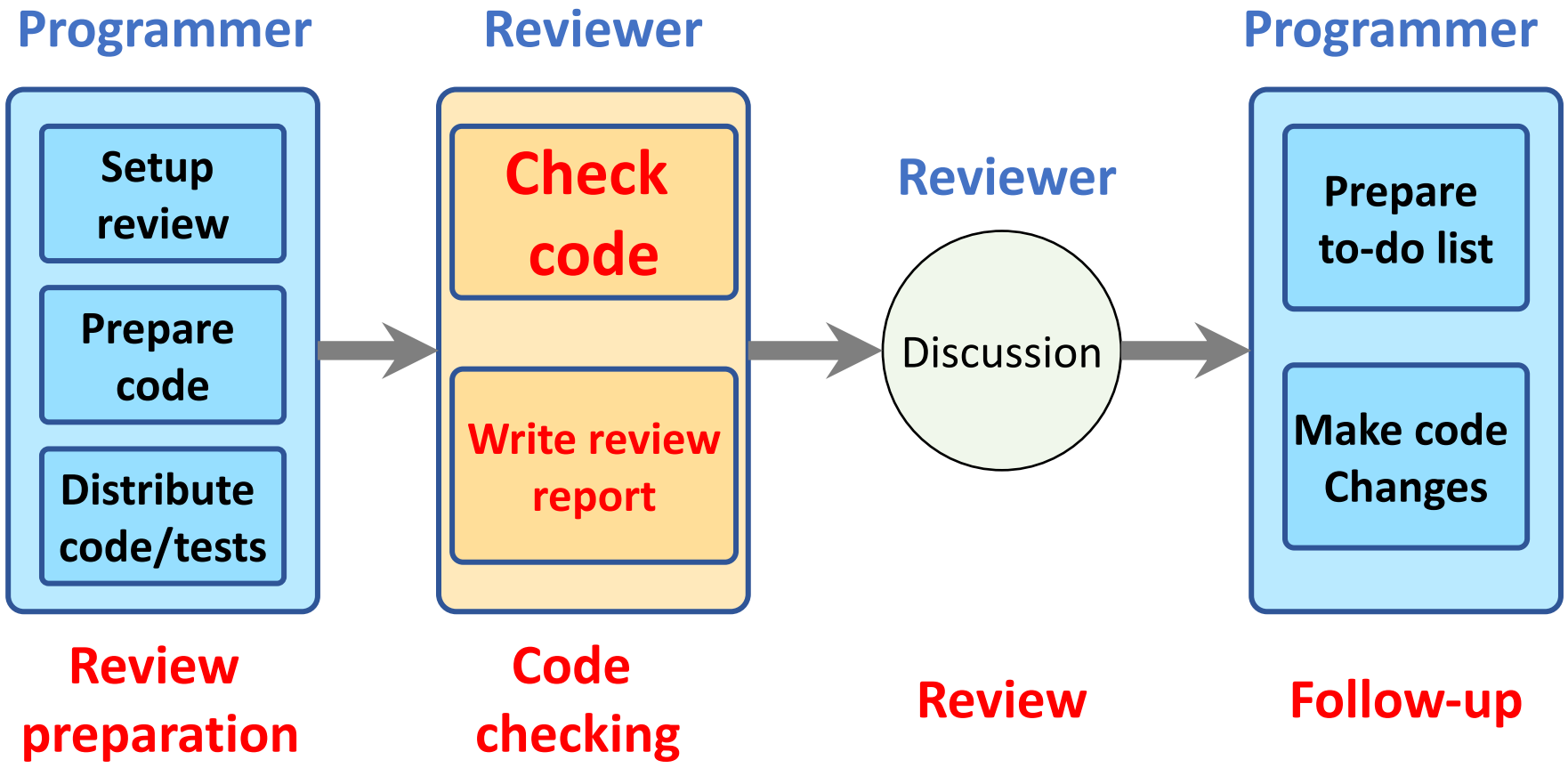
- Once you have identified security risks, you then analyze them to assess how they might arise.
  - The user has set weak passwords that can be guessed by an attacker.
  - The system's password file has been stolen and passwords discovered by attacker.
- Develop tests to check some of these possibilities.
  - 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.**

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

Code reviews



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

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.

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

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.

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.
- Nesreen Otoum and Nuha Elkhalili.(2026) "Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review." IEEE Access 14 (2026): 7443-7465.
- Lakshana Iruni Assalaarachchi, Zainab Masood, Rashina Hoda, and John Grundy. (2026) "Toward Agentic Software Project Management: A Vision and Roadmap." arXiv preprint arXiv:2601.16392 (2026).
- Deepak Babu Piskala. (2026) "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants." arXiv preprint arXiv:2602.00180 (2026).
- NVIDIA DLI (2026), Building RAG Agents with LLMs, https://learn.nvidia.com/courses/course-detail?course_id=course-v1:DLI+S-FX-15+V1
- NVIDIA DLI (2026), Generative AI with Diffusion Models, https://learn.nvidia.com/courses/course-detail?course_id=course-v1:DLI+S-FX-14+V1
- NVIDIA DLI (2026), Building Agentic AI Applications with LLMs, https://learn.nvidia.com/courses/course-detail?course_id=course-v1:DLI+S-FX-41+V1