軟體工程
(Software Engineering)

可靠的程式設計
(Reliable Programming)

1091SE10
MBA, IM, NTPU (M5118) (Fall 2020)
Tue 2, 3, 4 (9:10-12:00) (B8F40)

Min-Yuh Day
戴敏育
Associate Professor
副教授
Institute of Information Management, National Taipei University
國立臺北大學 資訊管理研究所
https://web.ntpu.edu.tw/~myday

2020-12-15

課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
1	2020/09/15	軟體工程概論 (Introduction to Software Engineering)
2	2020/09/22	軟體產品與專案管理：軟體產品管理，原型設計 (Software Products and Project Management: Software product management and prototyping)
3	2020/09/29	敏捷軟體工程：敏捷方法、Scrum、極限程式設計 (Agile Software Engineering: Agile methods, Scrum, and Extreme Programming)
4	2020/10/06	功能、場景和故事 (Features, Scenarios, and Stories)
5	2020/10/13	軟體架構：架構設計、系統分解、分散式架構 (Software Architecture: Architectural design, System decomposition, and Distribution architecture)
6	2020/10/20	軟體工程個案研究 I (Case Study on Software Engineering I)

課程大綱 (Syllabus)

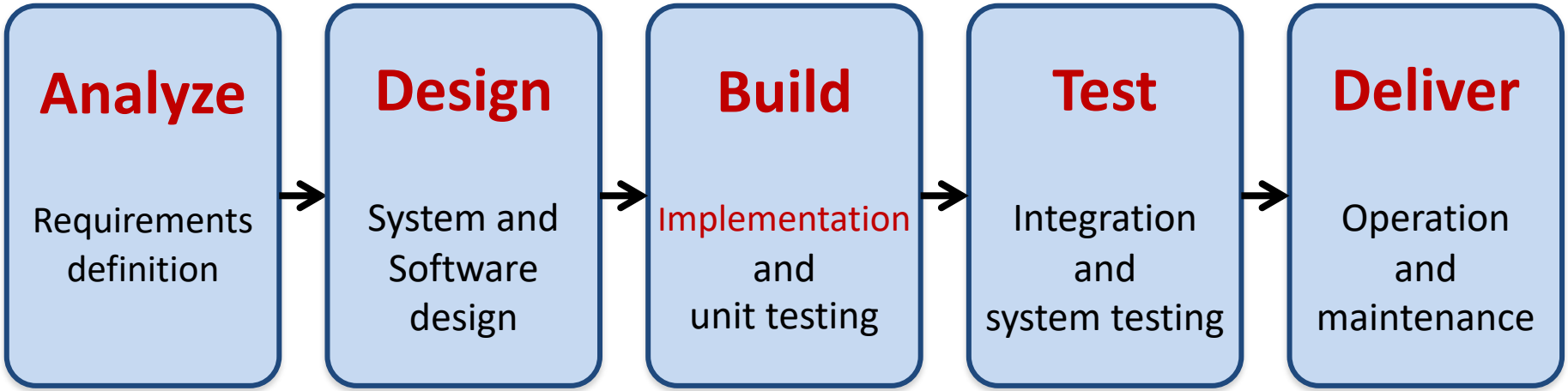
週次 (Week)	日期 (Date)	內容 (Subject/Topics)
7	2020/10/27	基於雲的軟體：虛擬化和容器、軟體即服務 (Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service)
8	2020/11/03	雲端運算與雲軟體架構 (Cloud Computing and Cloud Software Architecture)
9	2020/11/10	期中報告 (Midterm Project Report)
10	2020/11/17	微服務架構：RESTful服務、服務部署 (Microservices Architecture: RESTful services, Service deployment)
11	2020/11/24	軟體工程產業實務 (Industry Practices of Software Engineering)
12	2020/12/01	安全和隱私 (Security and Privacy)

課程大綱 (Syllabus)

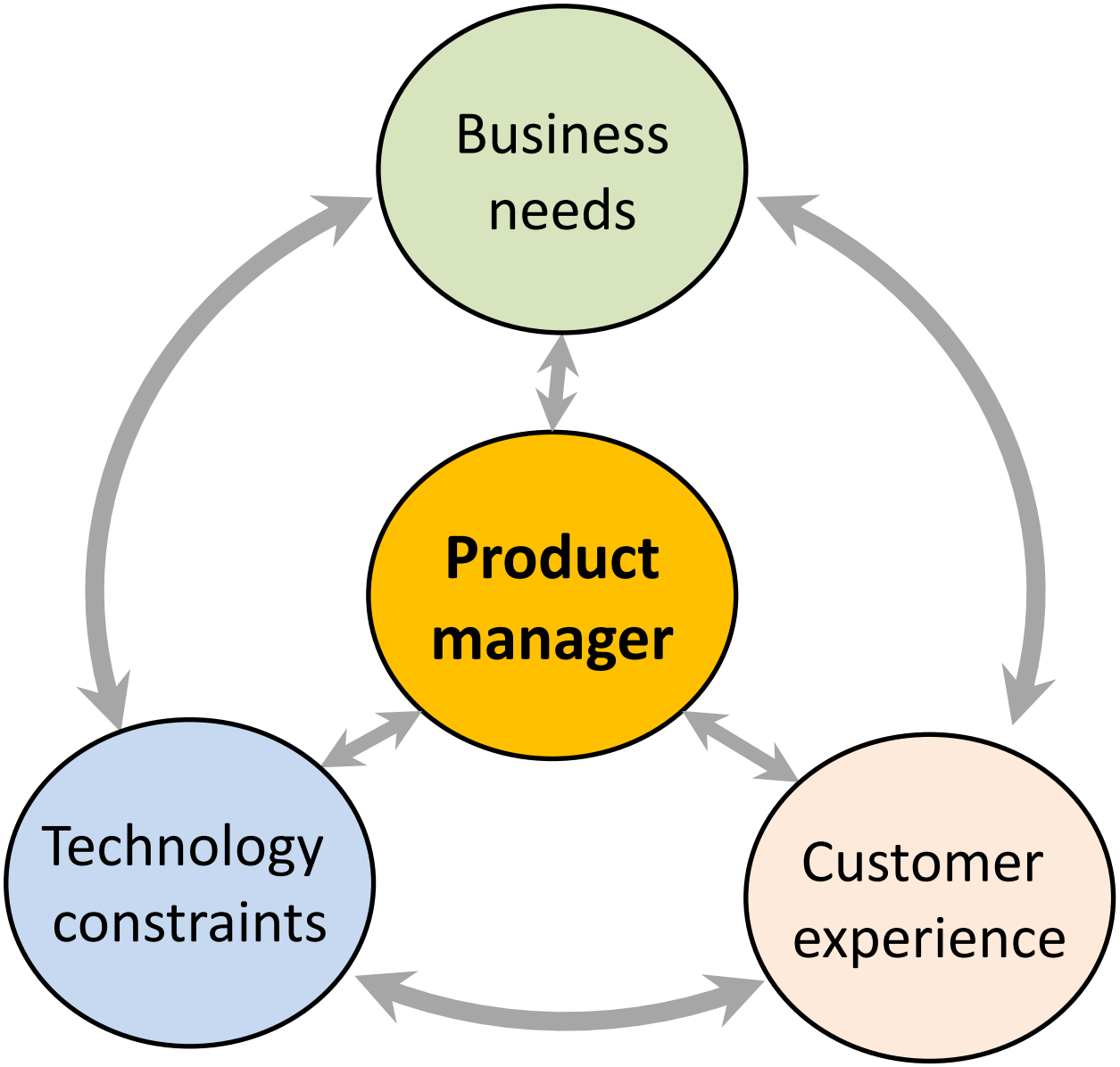
國立臺北大學 National Taipei University

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
13	2020/12/08	軟體工程個案研究 II (Case Study on Software Engineering II)
14	2020/12/15	可靠的程式設計 (Reliable Programming)
15	2020/12/22	測試：功能測試、測試自動化、測試驅動的開發、程式碼審查 (Testing: Functional testing, Test automation, Test-driven development, and Code reviews)
16	2020/12/29	DevOps和程式碼管理：程式碼管理和DevOps自動化 (DevOps and Code Management: Code management and DevOps automation)
17	2021/01/05	期末報告 I (Final Project Report I)
18	2021/01/12	期末報告 II (Final Project Report I)

Software Engineering and Project Management

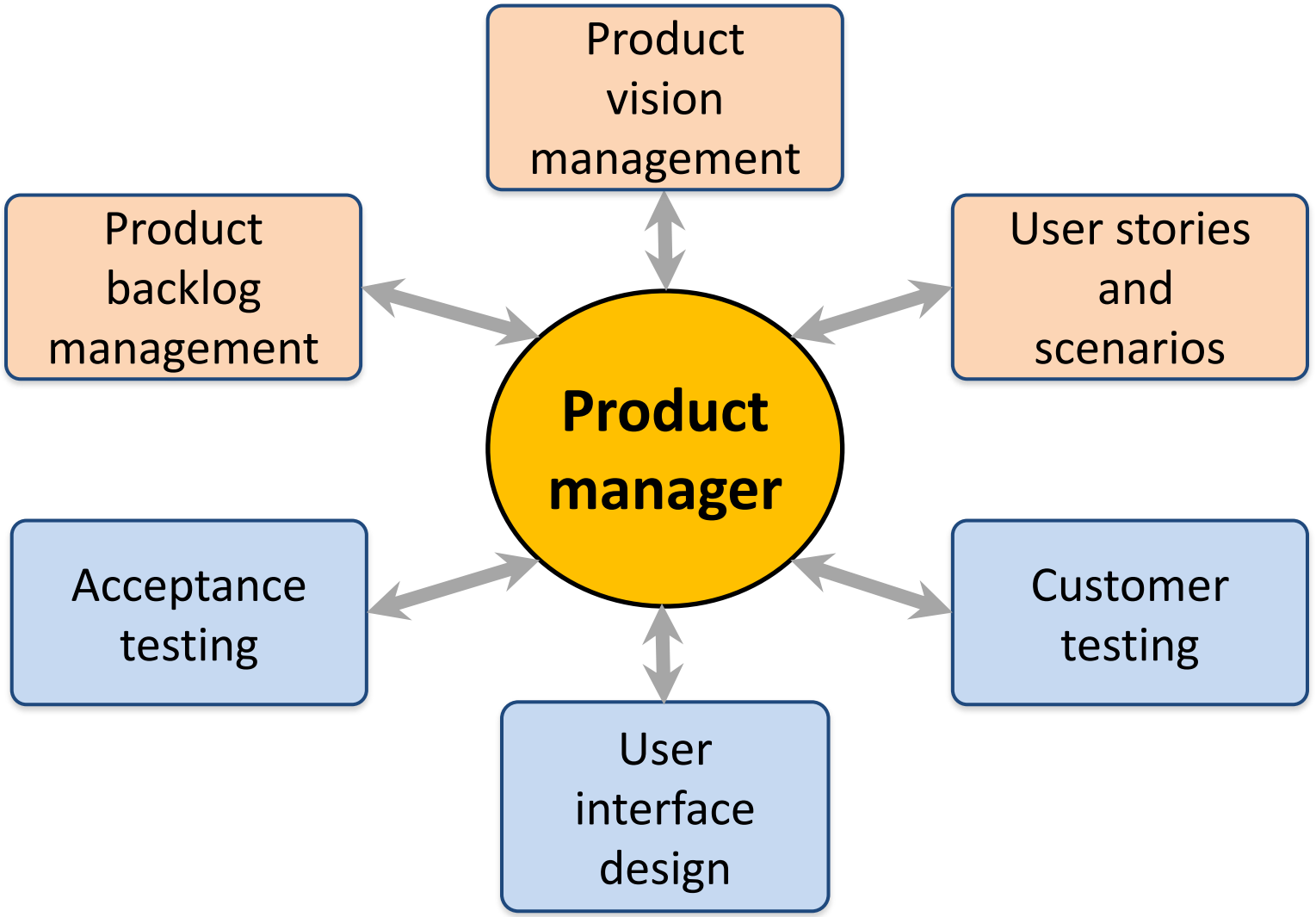


Product management concerns



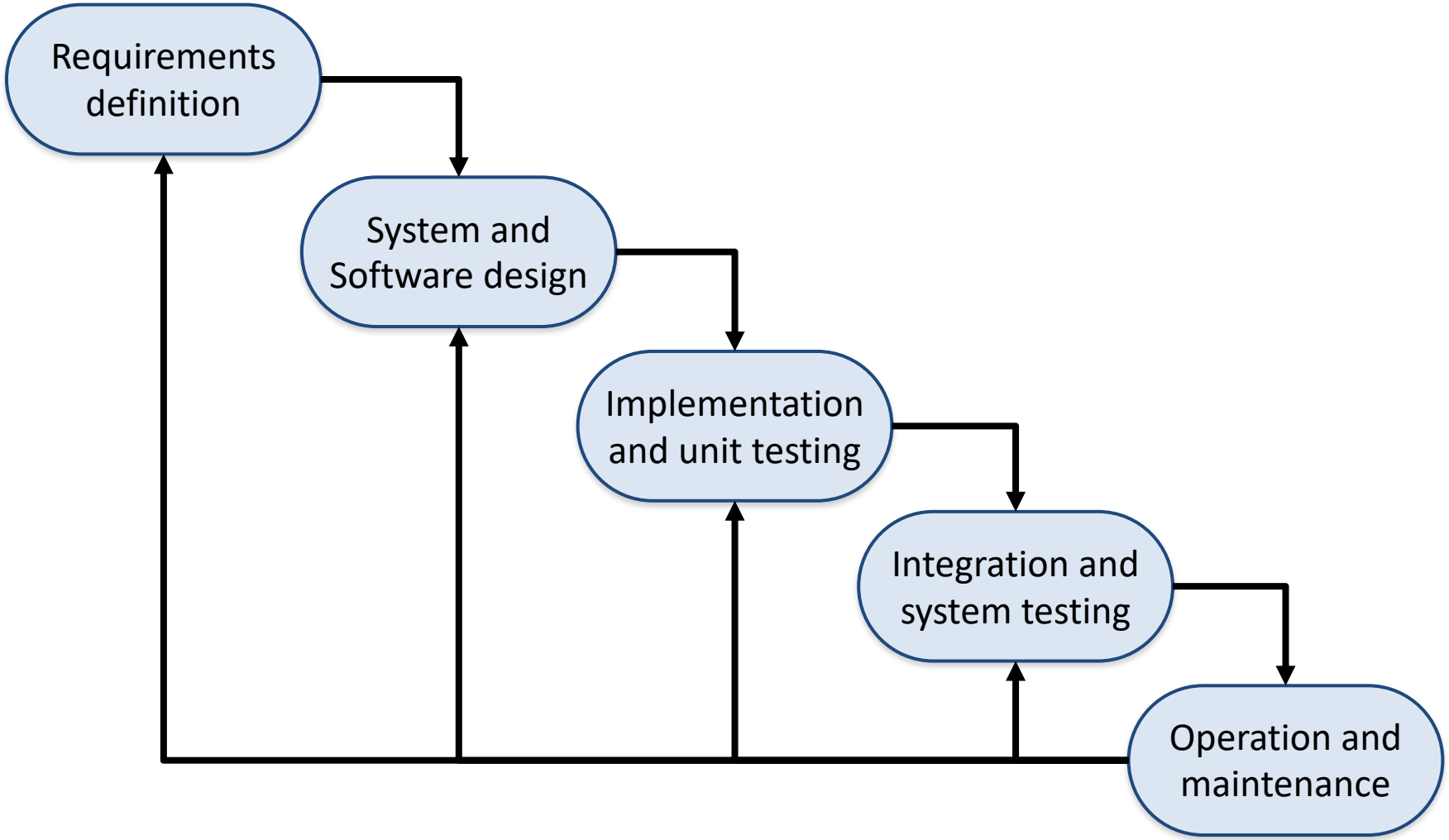
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Technical interactions of product managers



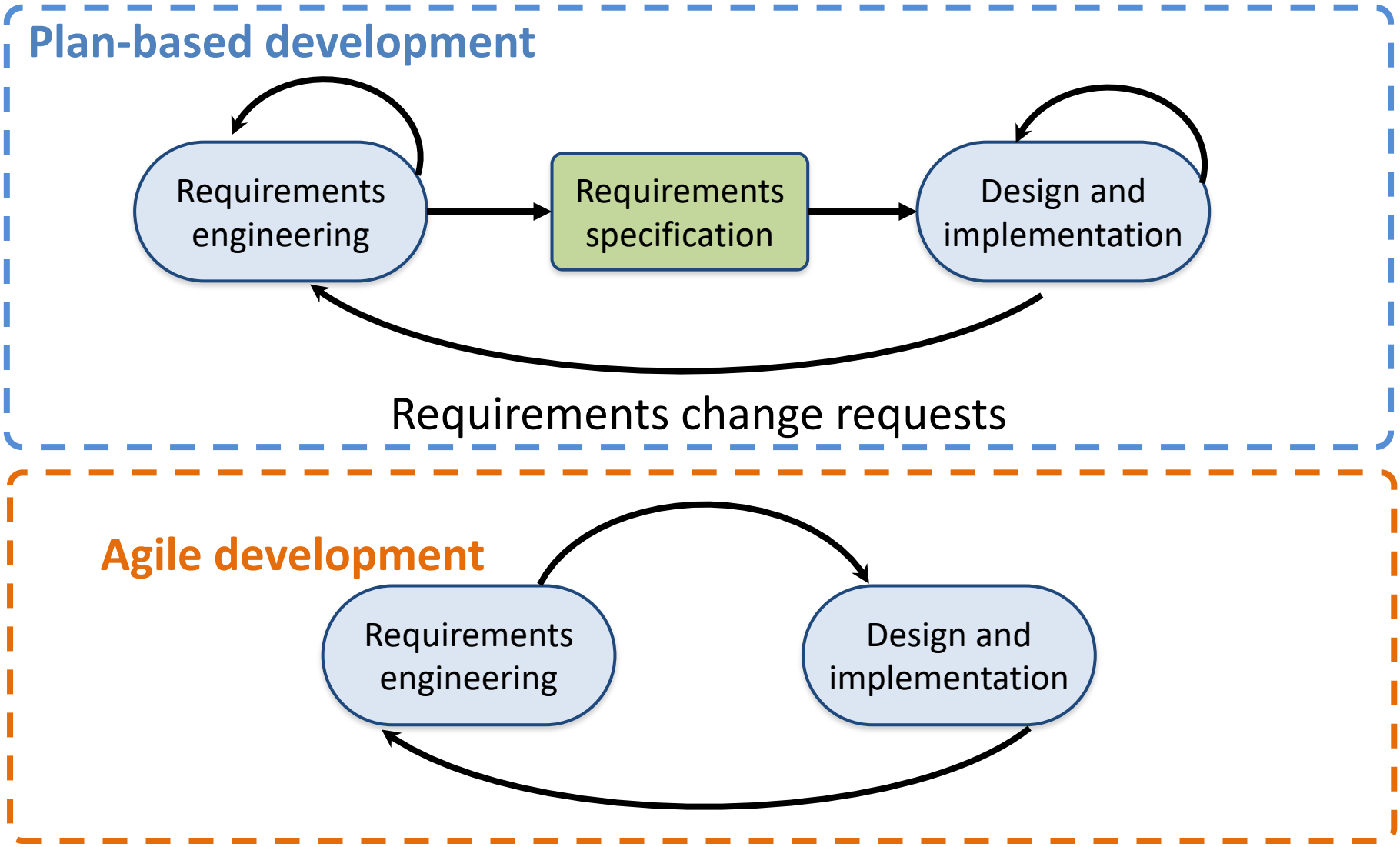
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software Development Life Cycle (SDLC)
The waterfall model



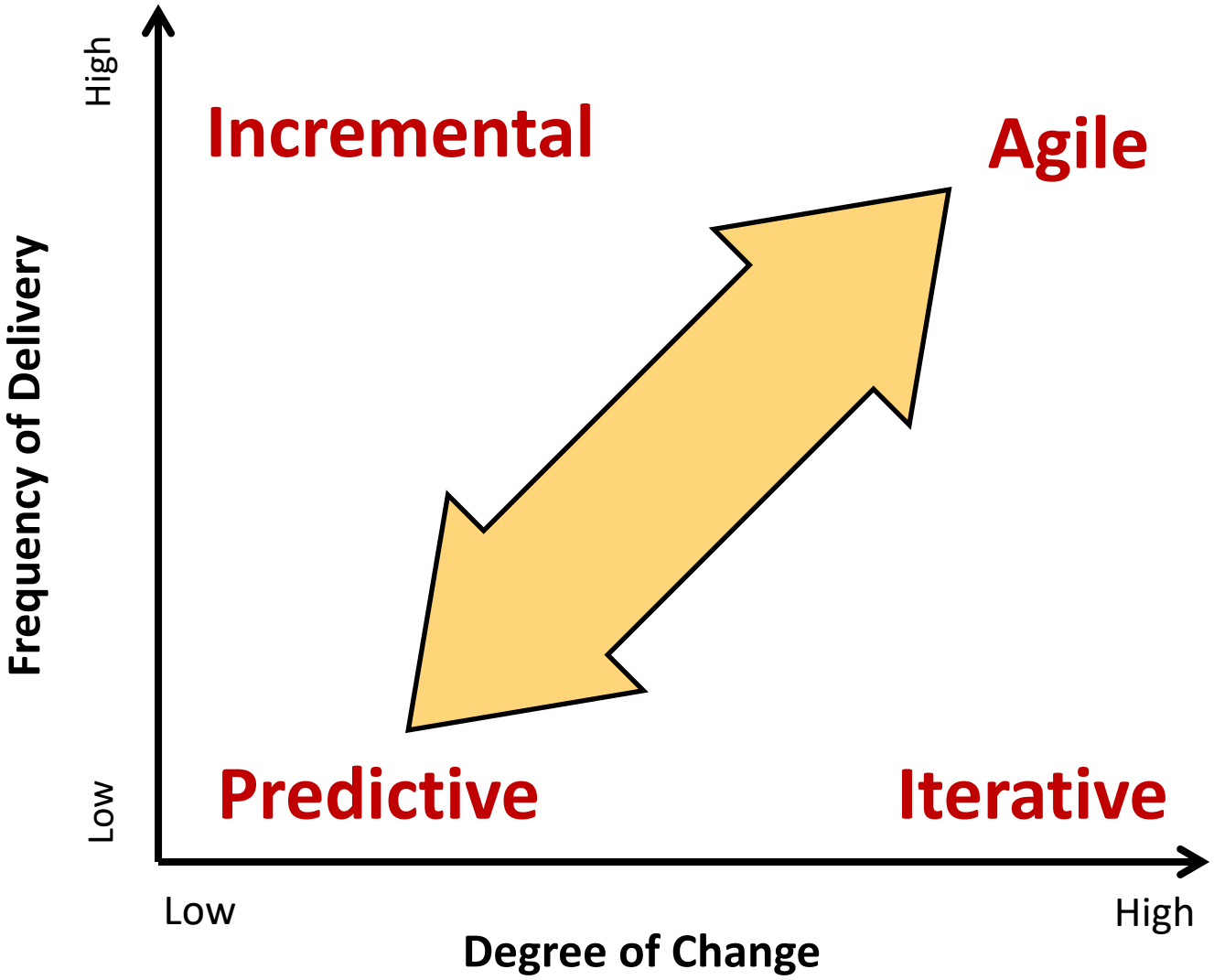
Source: Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.

Plan-based and Agile development



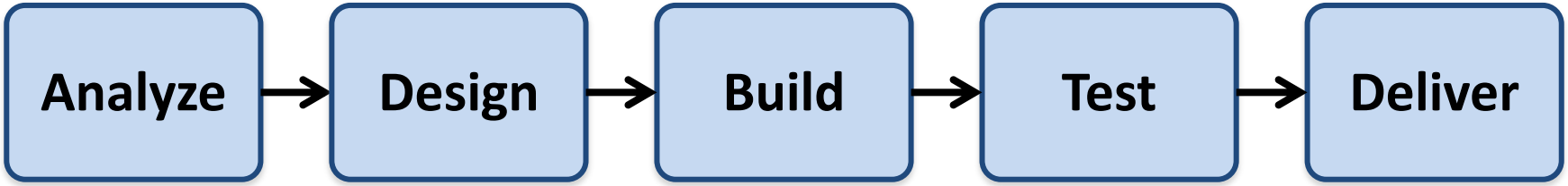
Source: Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.

The Continuum of Life Cycles



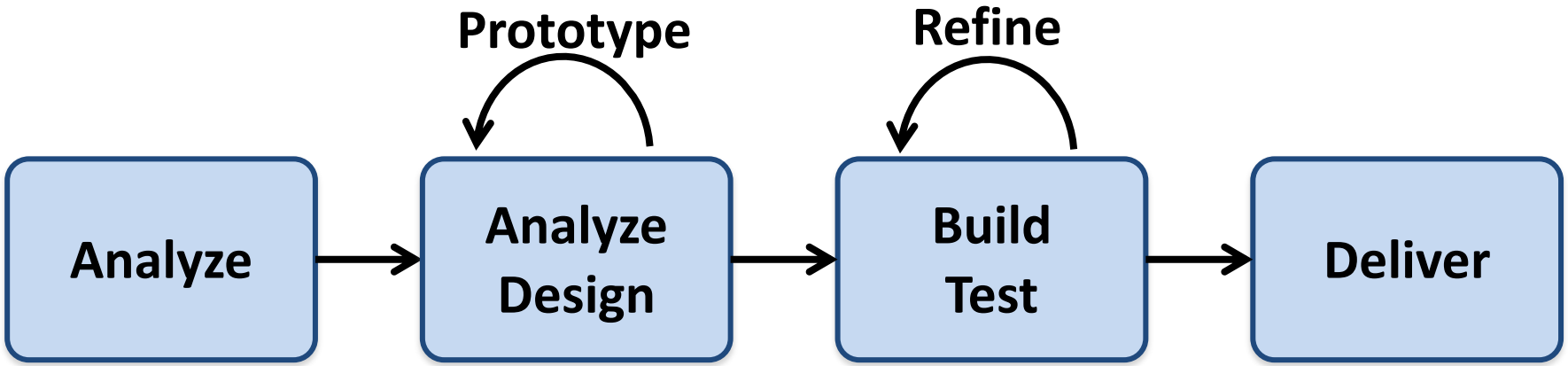
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Predictive Life Cycle



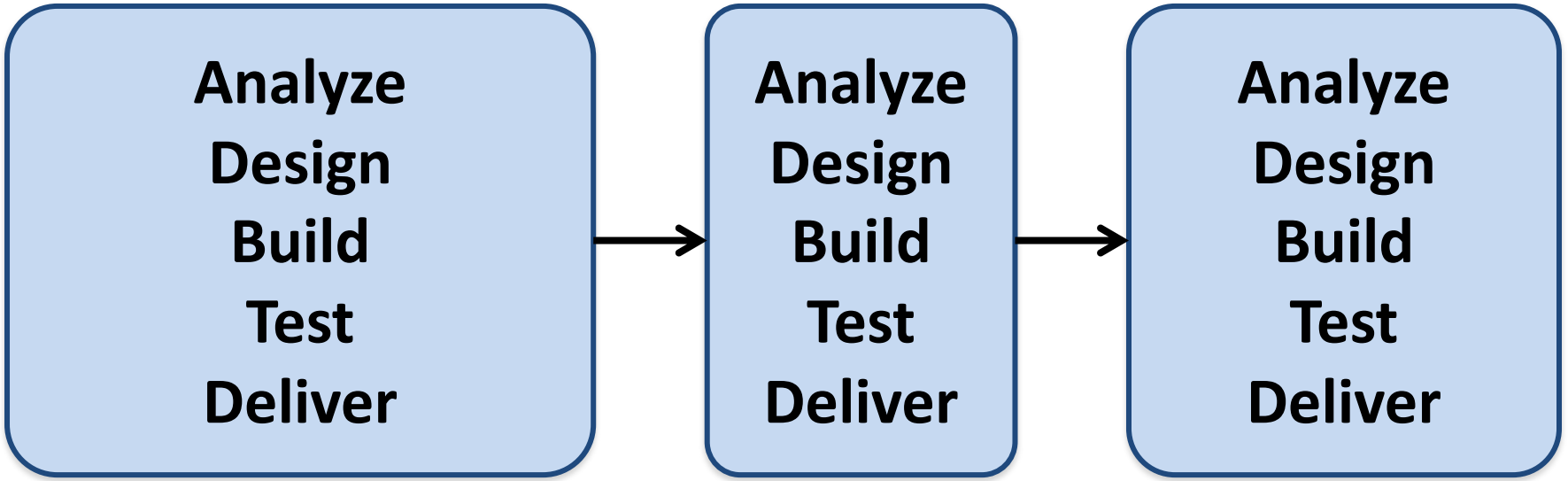
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

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

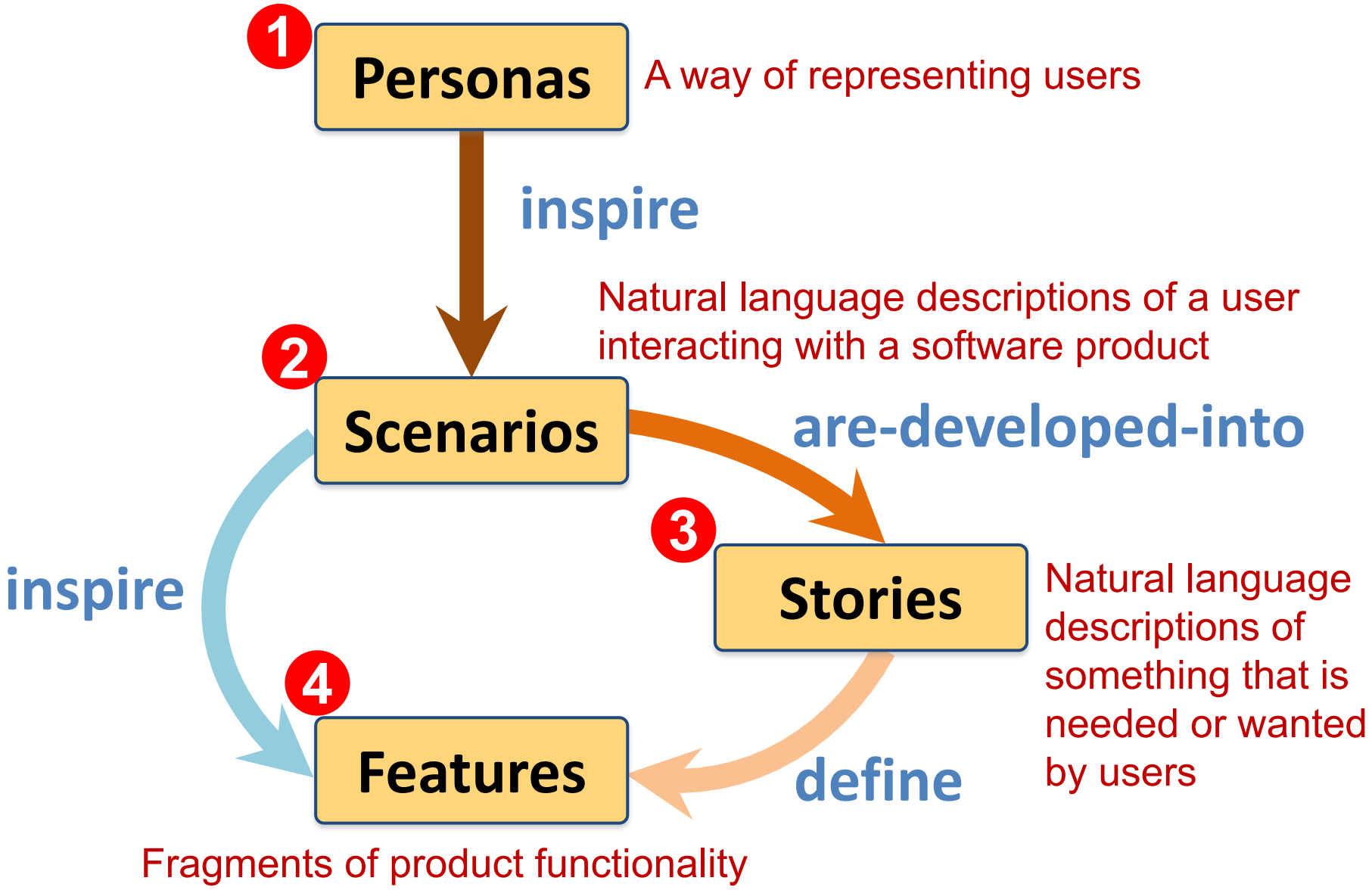
Requirements **Analysis** **Design** **Build** **Test**	**Requirements** **Analysis** **Design** **Build** **Test**	**Requirements** **Analysis** **Design** **Build** **Test**	**Requirements** **Analysis** **Design** **Build** **Test**	**Repeat** **as needed** **…**	**Requirements** **Analysis** **Design**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

Flow-Based Agile

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

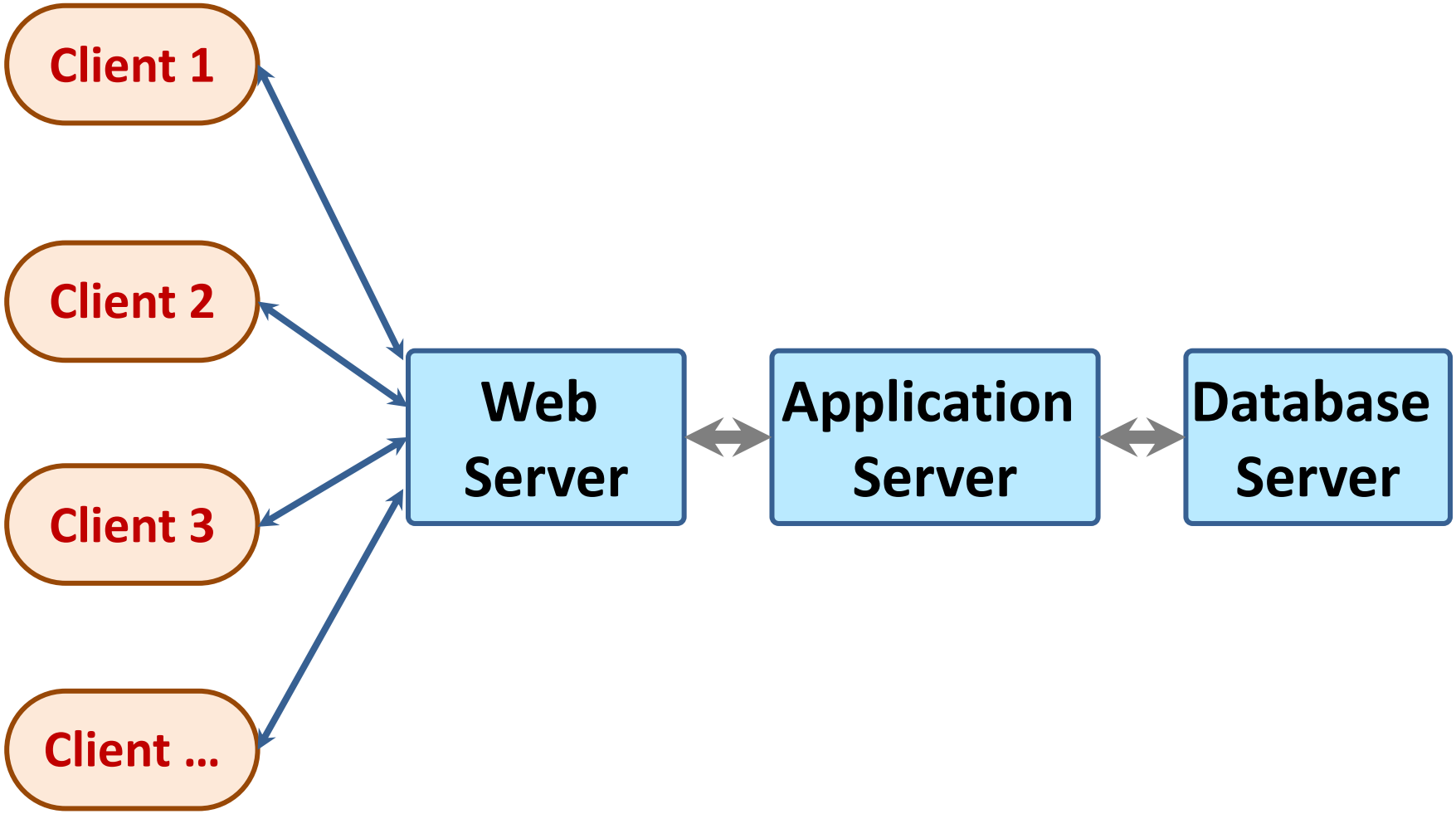
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

From personas to features



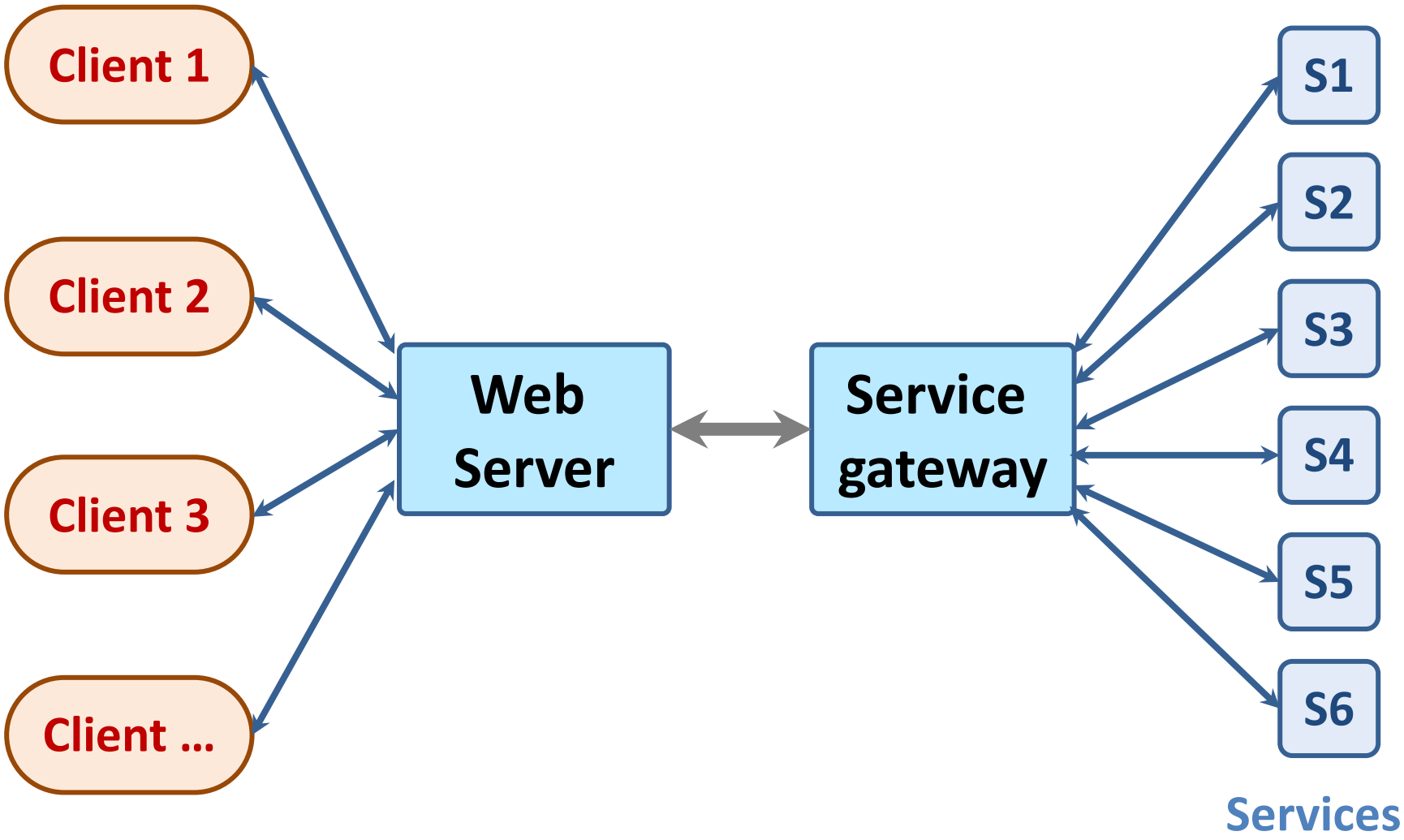
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Multi-tier client-server architecture

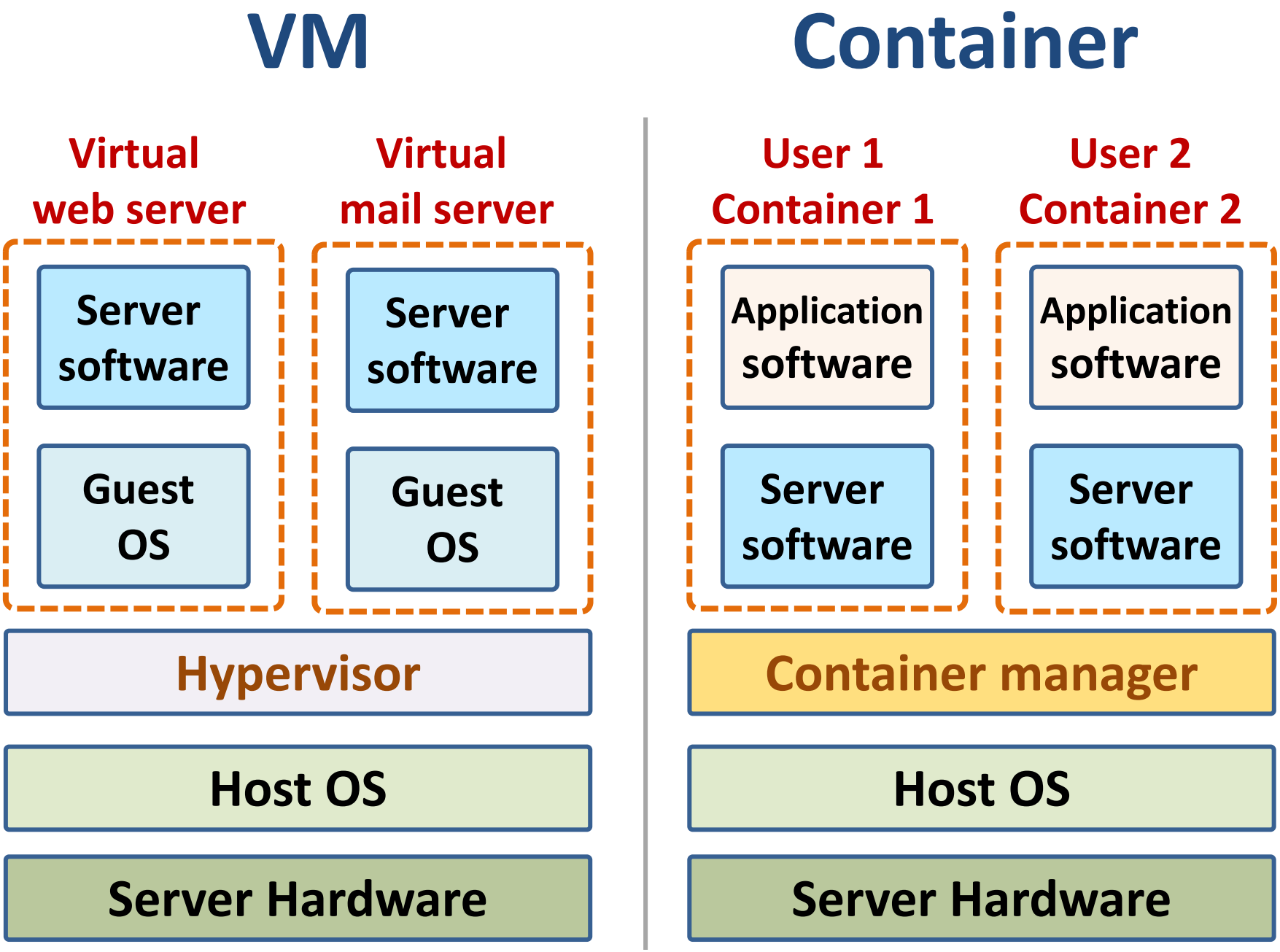


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Service-oriented Architecture

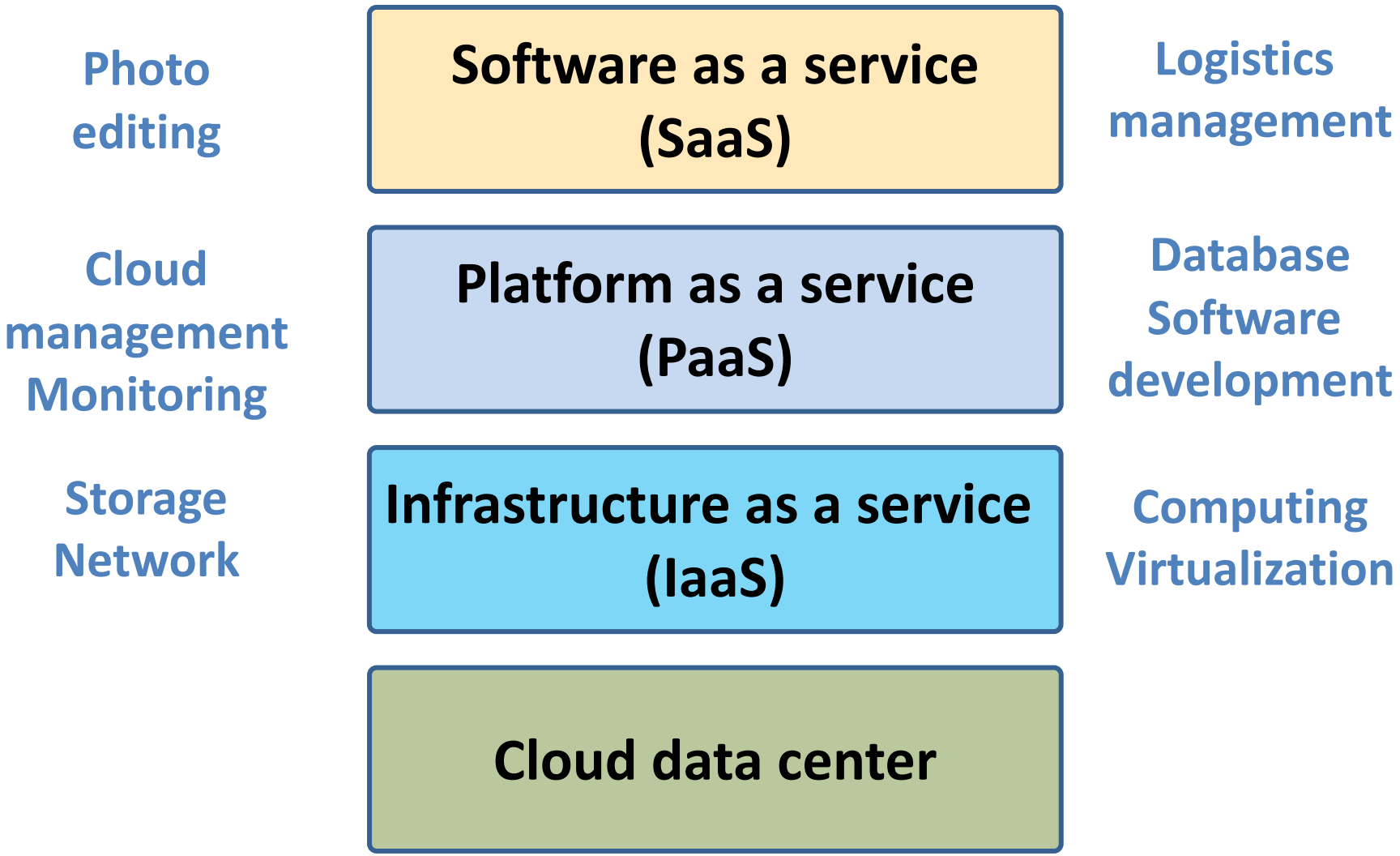


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.



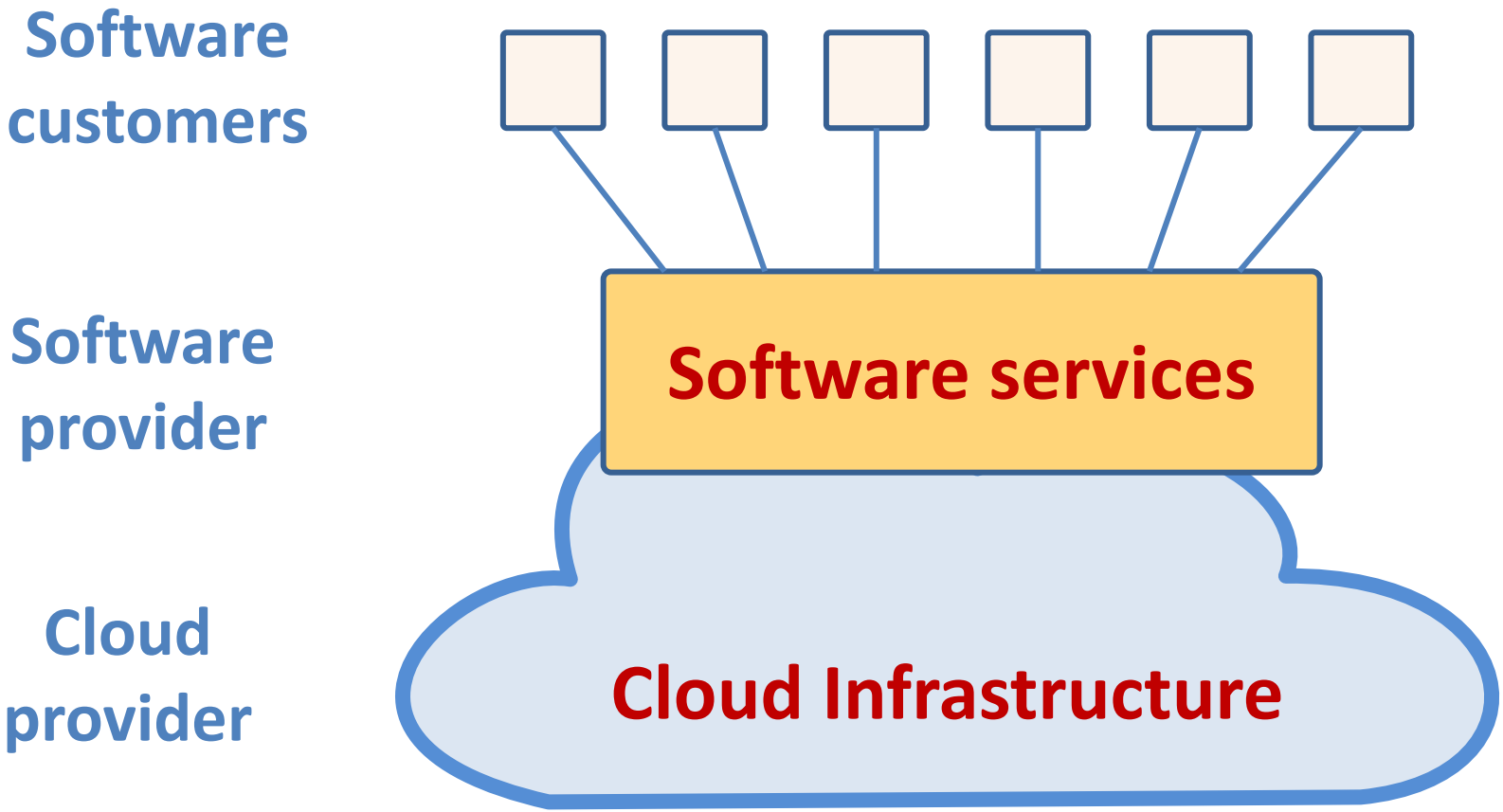
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Everything as a service



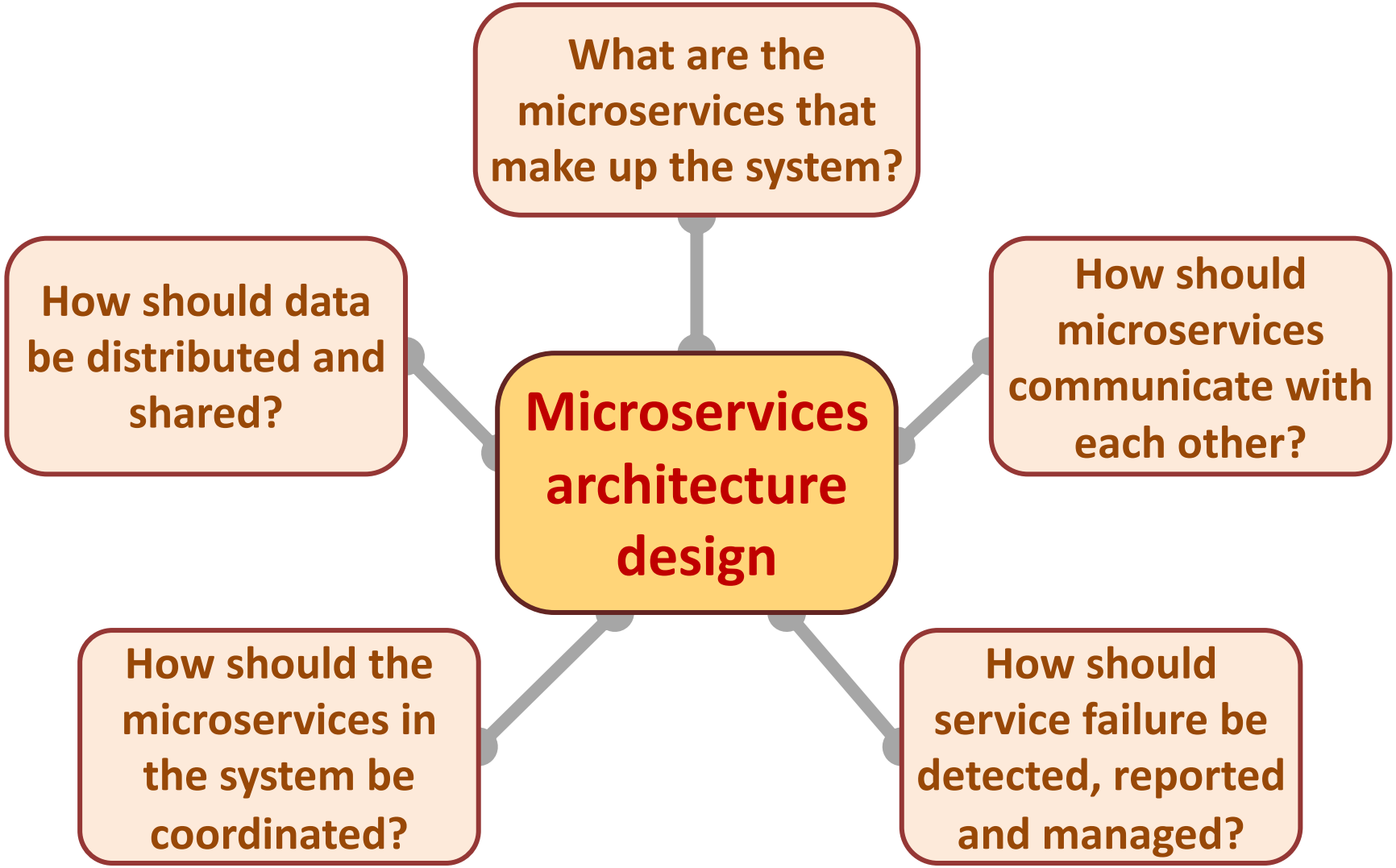
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software as a service



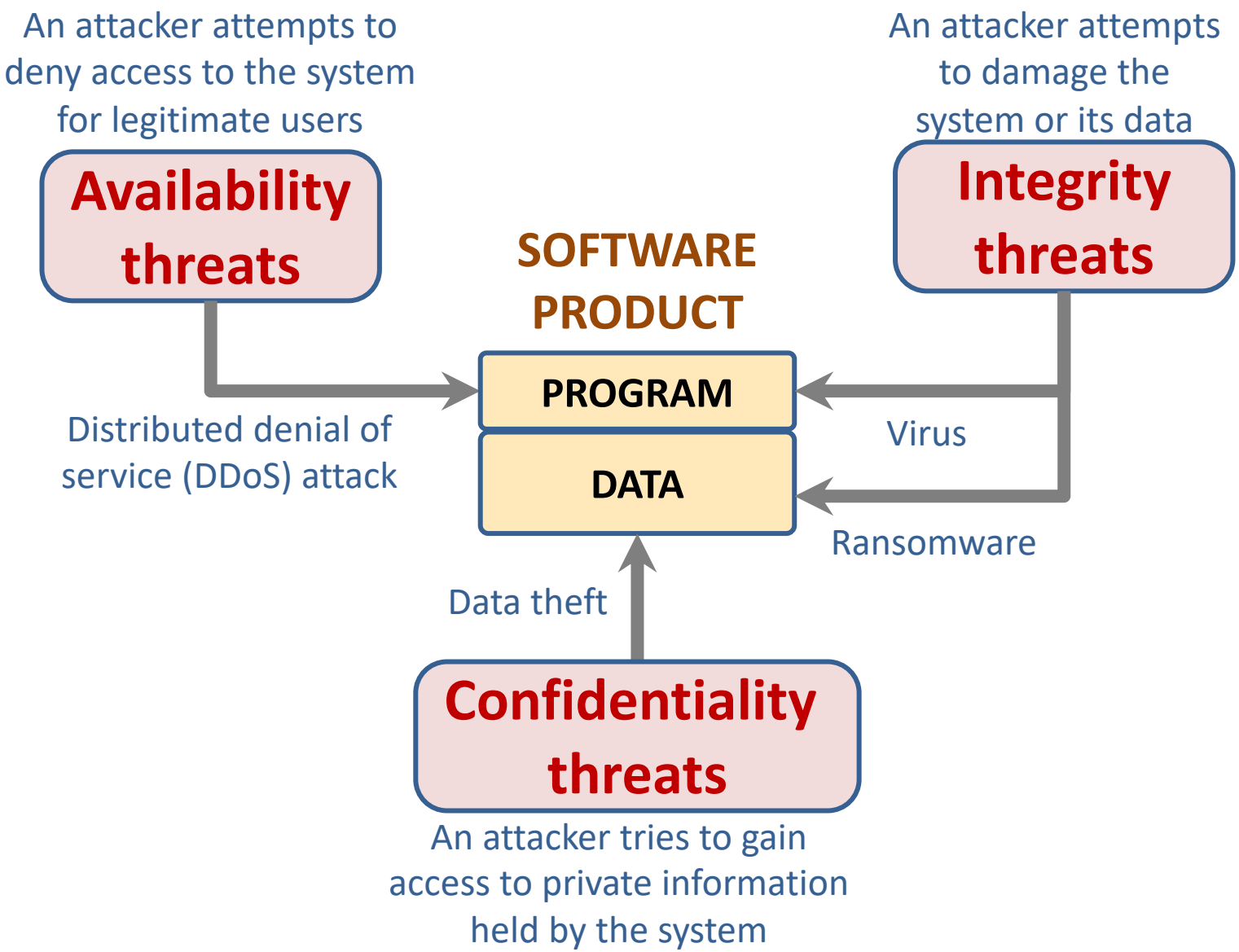
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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.

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.

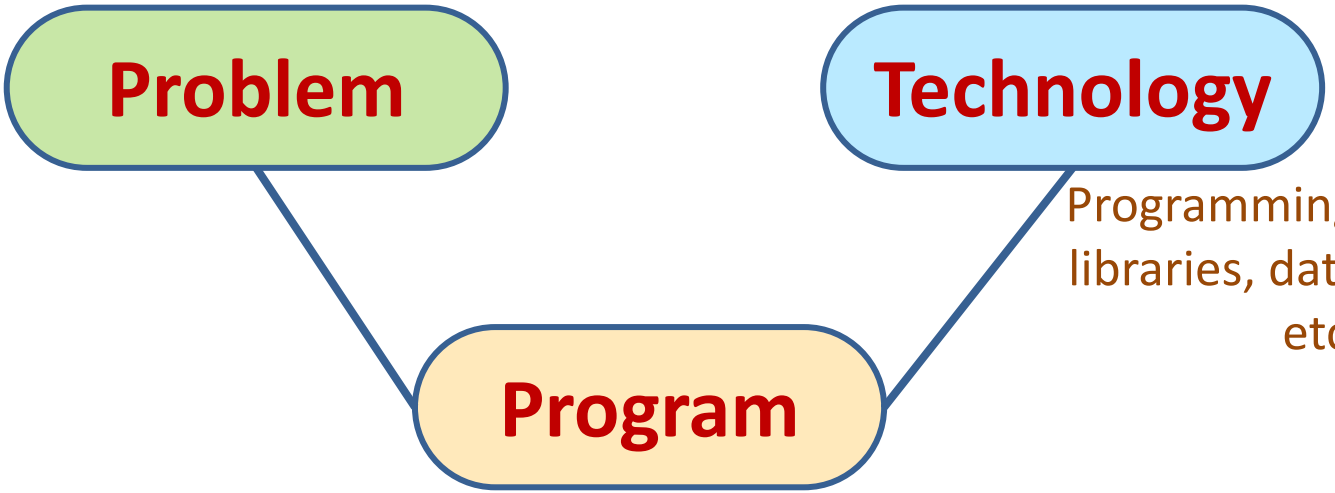
Programming for reliability

- 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.

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

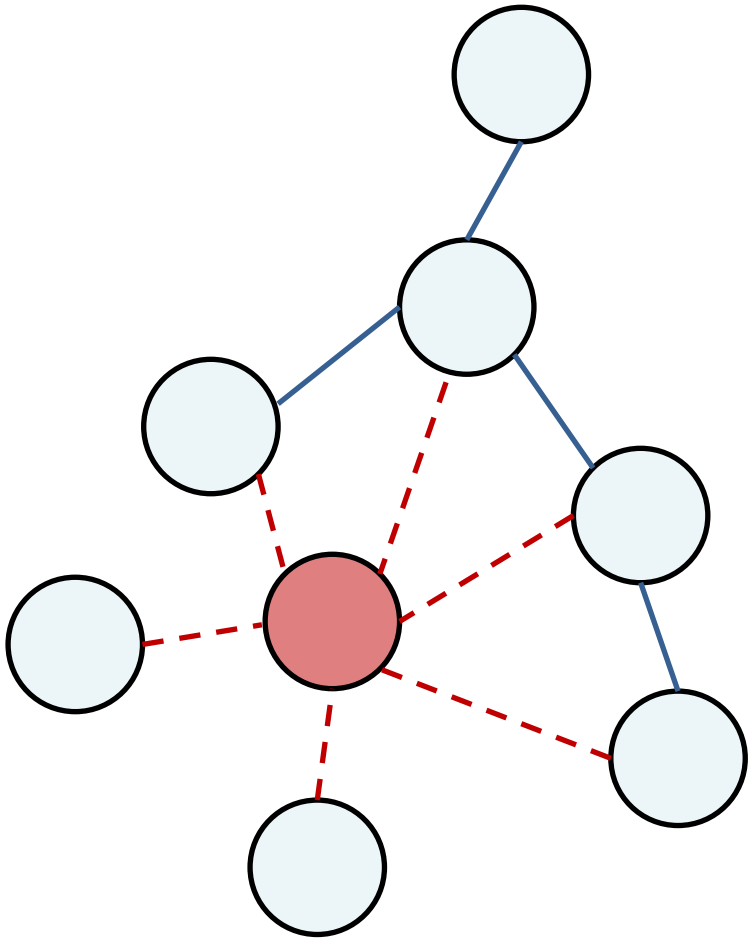


Programming language, libraries, database, IDE, etc.

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.

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

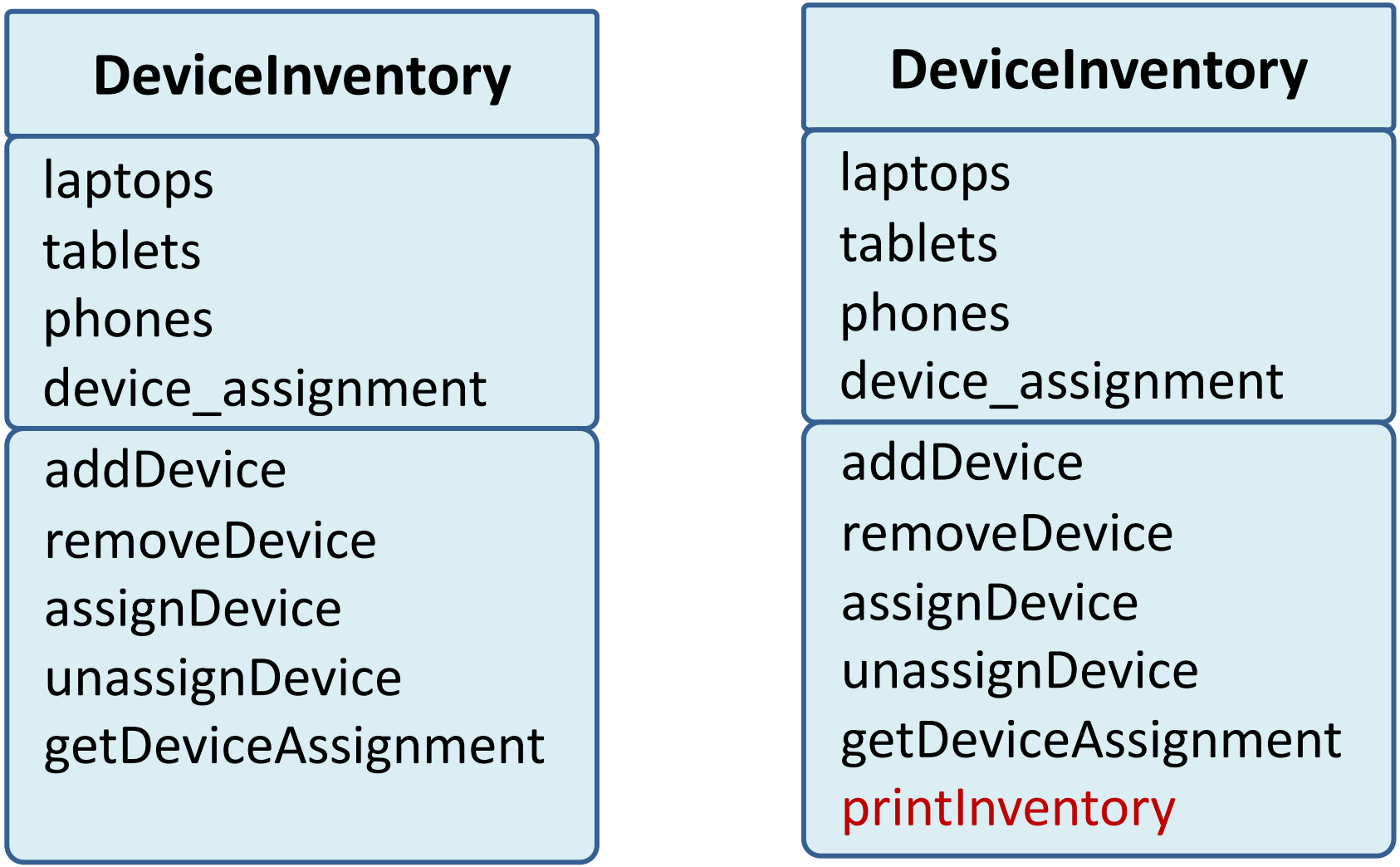
Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

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



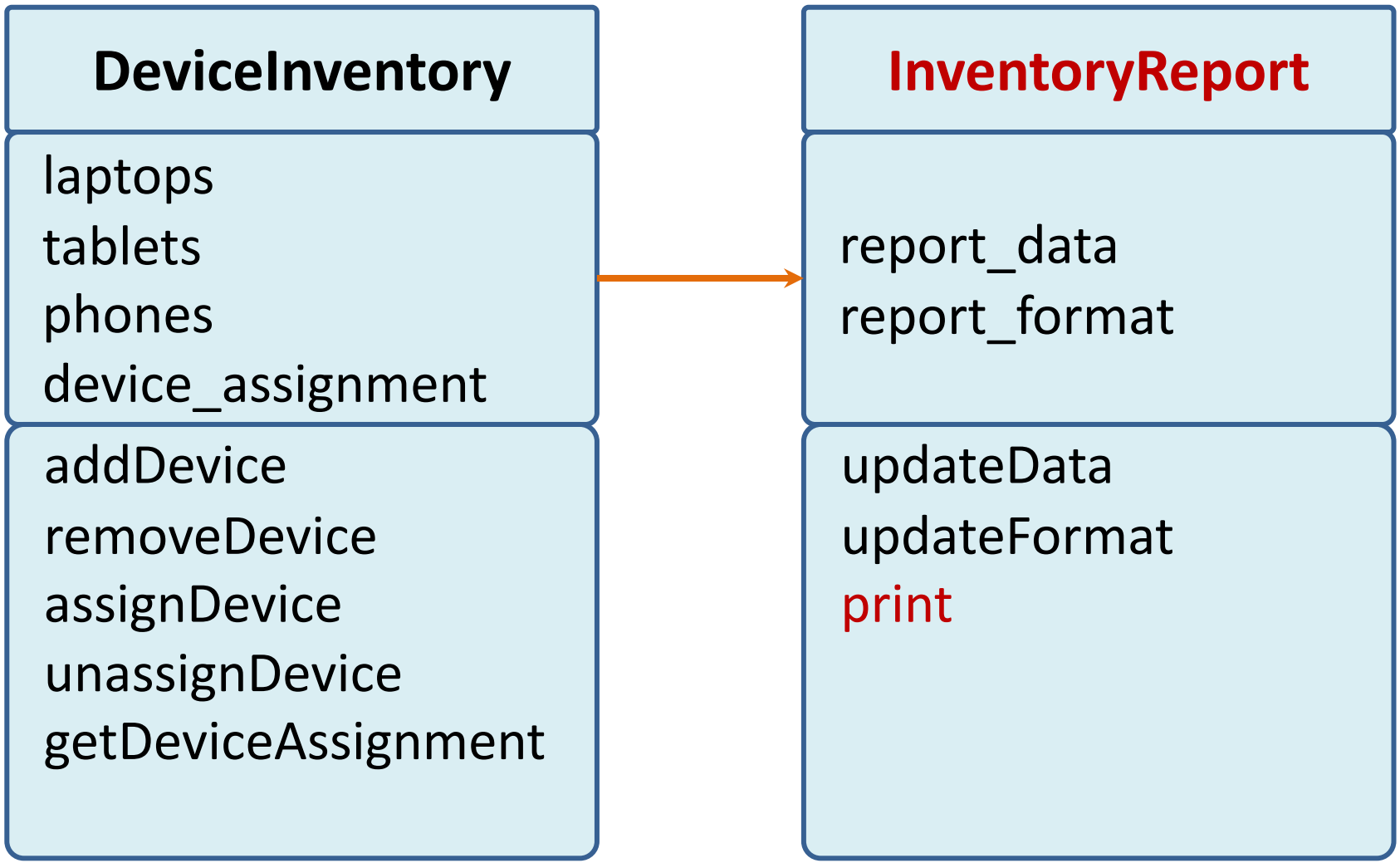
(a) (b)

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.

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.

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

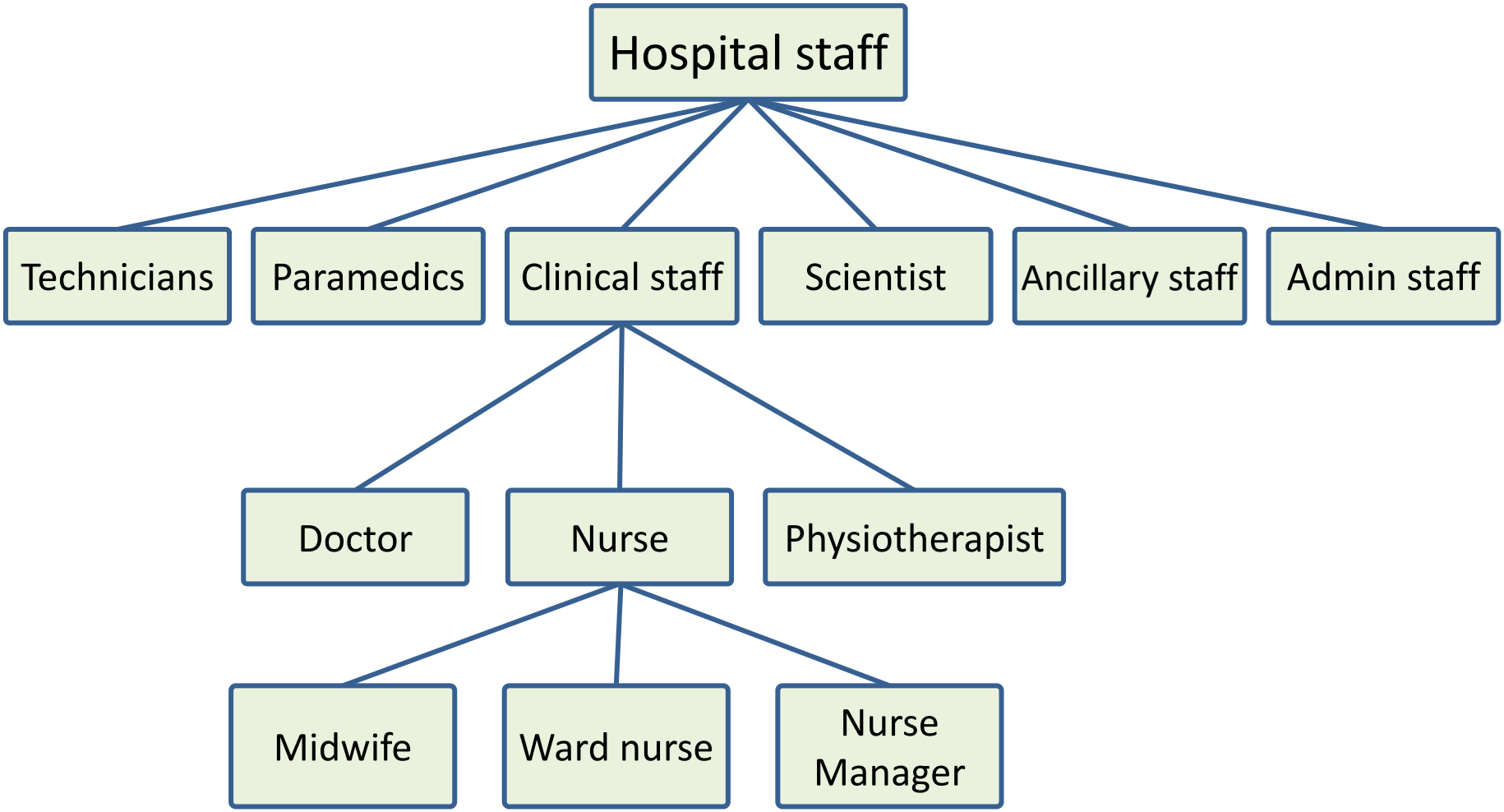
```
def agecheck_with_guards(age, experience):

    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
```

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.

Programming principles

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

Common types of design patterns

- **Creational patterns**
  - These are concerned with class and object creation. They define ways of instantiating and initializing objects and classes that are more abstract than the basic class and object creation mechanisms defined in a programming language.
- **Structural patterns**
  - These are concerned with class and object composition. Structural design patterns are a description of how classes and objects may be combined to create larger structures.
- **Behavioural patterns**
  - These are concerned with class and object communication. They show how objects interact by exchanging messages, the activities in a process and how these are distributed amongst the participating objects.

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.

Refactoring

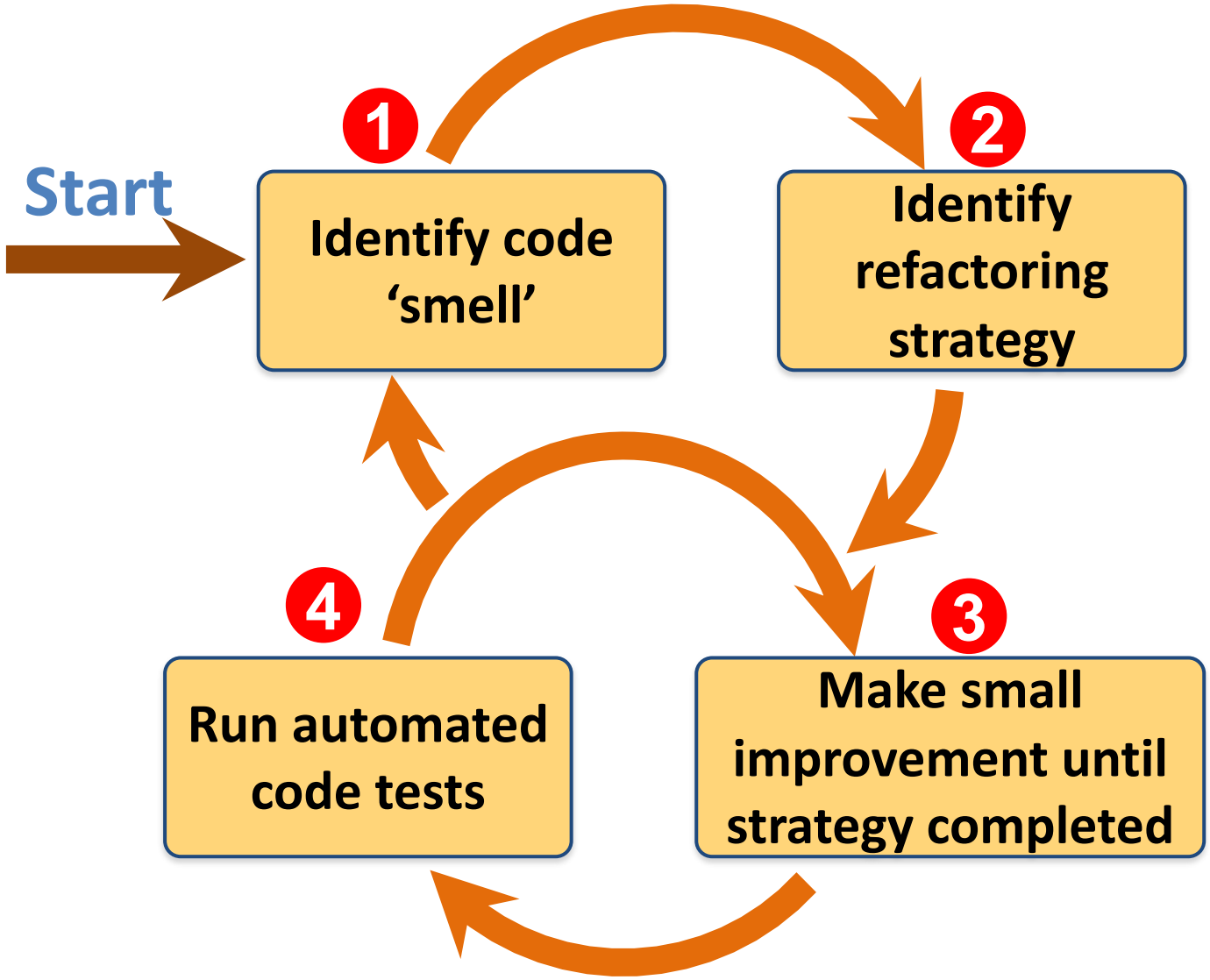
- **Refactoring** means changing a program to reduce its complexity without changing the external behaviour of that program.
- **Refactoring** makes a program more readable (so reducing the 'reading complexity') and more understandable.
- It also makes it easier to change, which means that you reduce 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.

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.

Examples of code smells

- **Large classes**
  Large classes may mean that the single responsibility principle is being violated. Break down large classes into easier-to-understand, smaller classes.
- **Long methods/functions**
  Long methods or functions may indicate that the function is doing more than one thing. Split into smaller, more specific functions or methods.

Examples of code smells

- **Duplicated code**

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

- **Meaningless names**

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

Examples of code smells

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

Examples of refactoring for complexity reduction

- **Reading complexity**
  You can rename variable, function and class names throughout your program to make their purpose more obvious.
- **Structural complexity**
  You can break long classes or functions into shorter units that are likely to be more cohesive than the original large class.

Examples of refactoring for complexity reduction

- **Data complexity**
  You can simplify data by changing your database schema or reducing its complexity. For example, you can merge related tables in your database to remove duplicated data held in these tables.
- **Decision complexity**
  You can replace a series of deeply nested if-then-else statements with guard clauses.

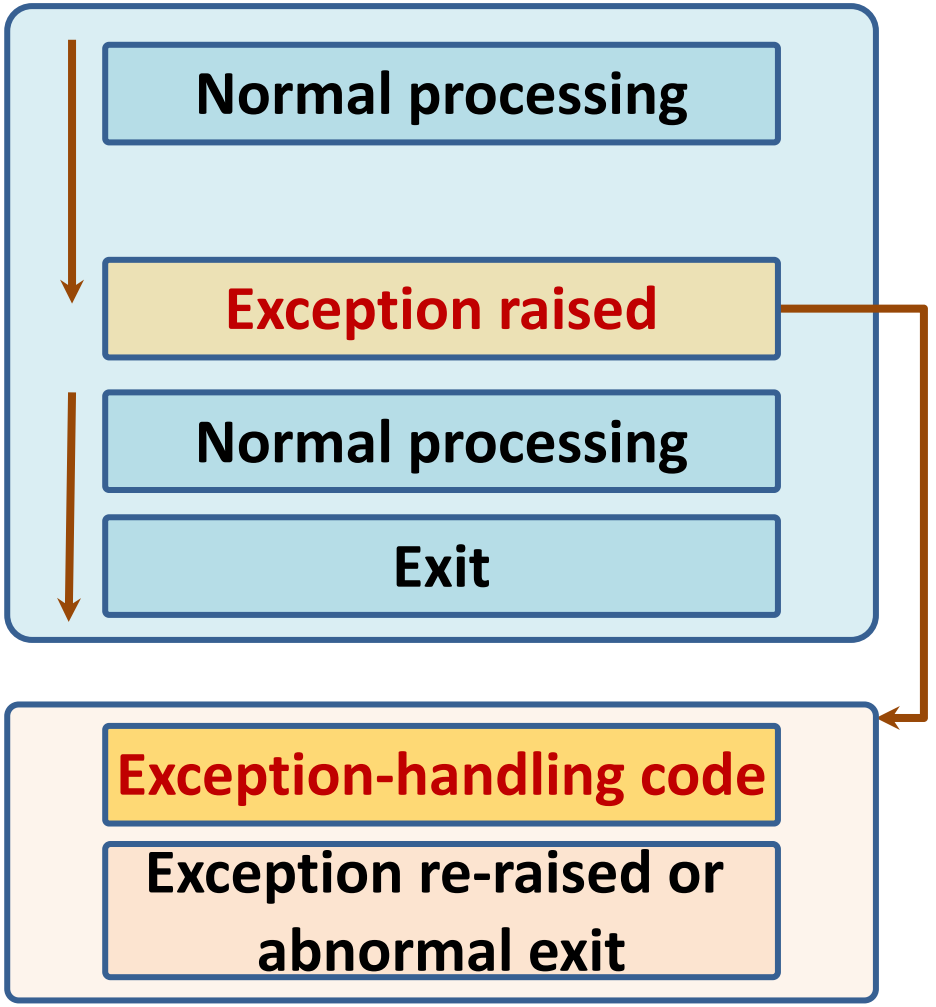
Exception handling

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

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

Exception handling

Executing code



Exception-handling block

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

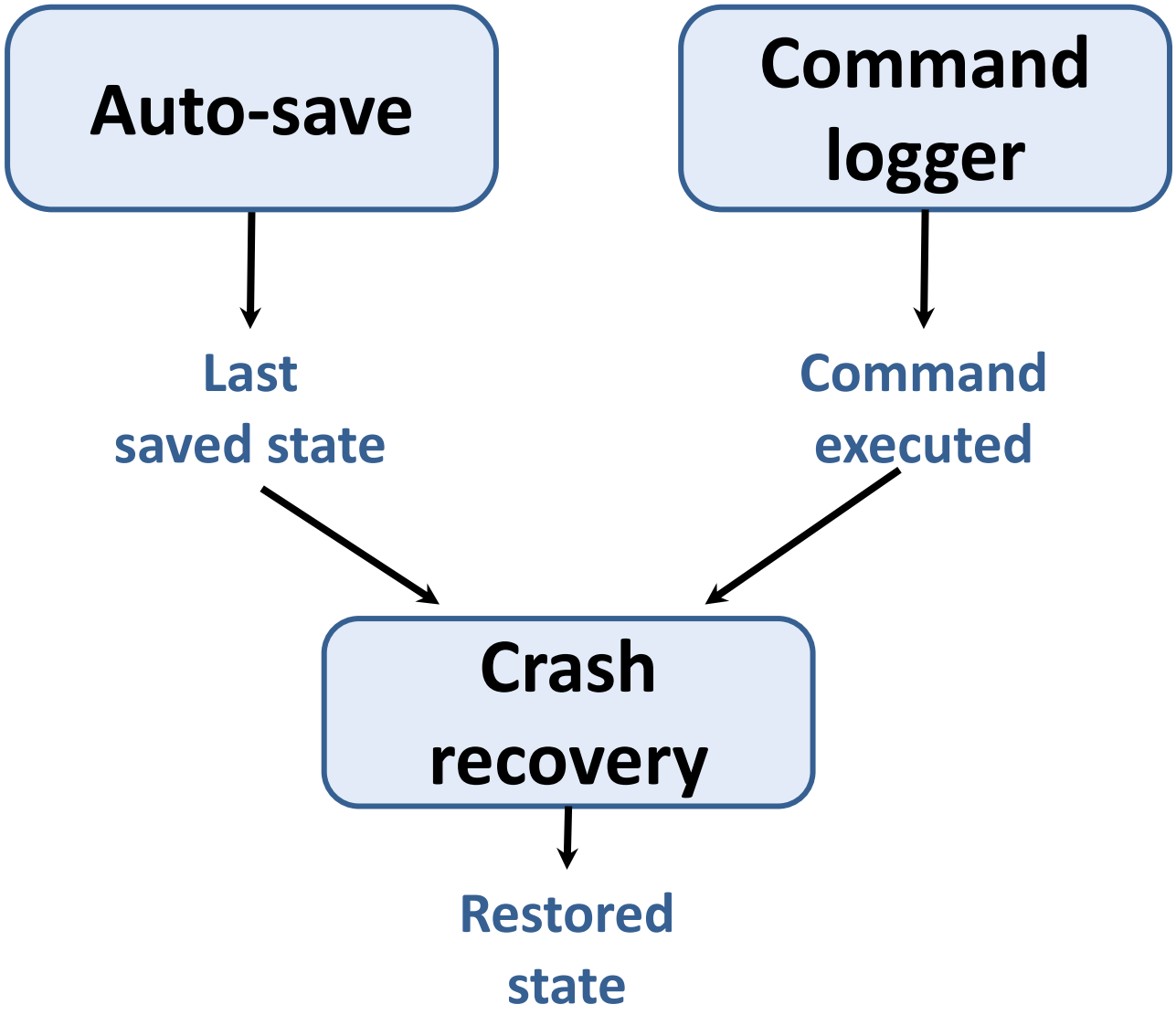
Python

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.

Summary

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

Summary

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

Summary

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

References

- Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.
- Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.
- Titus Winters, Tom Manshreck, and Hyrum Wright (2020), Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media.
- Project Management Institute (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