Reliable Programming

CS 346 Application Development

Credits

- Some content adapted from Chapter 8: Reliable Programming, in: Sommerville. 2021. Engineering Software Products: An Introduction to Modern Software Engineering. Pearson. ISBN 978-1292376356.

Software quality

"The first requirement for an exemplary user experience is to meet the exact needs of the customer, without fuss or bother. Next comes simplicity and elegance that produce products that are a joy to own, a joy to use."

– Don Norman & Jacob Nielsen

- Creating a successful software product does not simply mean providing useful features for users.
- You also need to create a **high-quality product** that people want to use.
  - Customers must be confident that your product will not crash or lose information.
  - Its use should be simple and clear to use.

Reliability == quality improvements

- Achieving soft-goals generally means addressing quality attributes.
  - This is often what customers mean by “high quality” e.g., “won’t crash”, “just works”.
  - Addressing these is costly, time-consuming and can add to product complexity.
- We cannot realistically “address them all”.
  - Addressing any one area will usually require compromise e.g., improving security can sometimes hurt usability.
  - We usually need to prioritize 1-2 at most.
- How should we improve quality?
  - What’s the most impactful way to handle this situation?

Programming for reliability

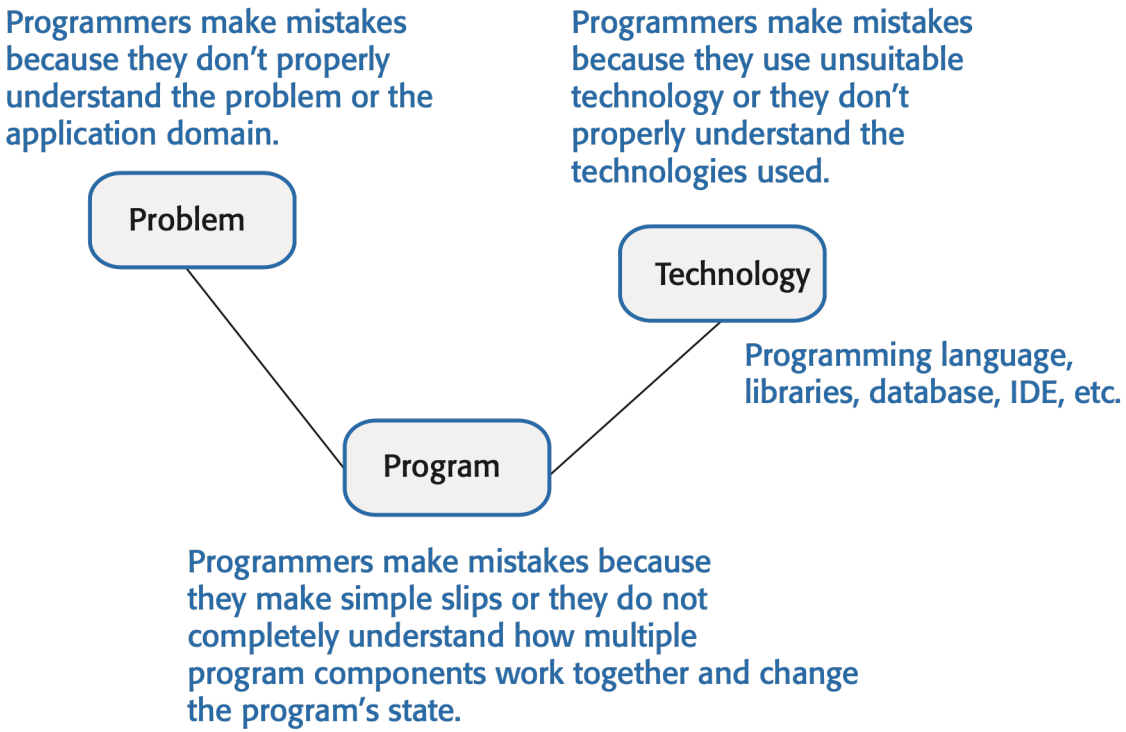
- There are three *simple* techniques for reliability improvement that can be applied in any software product.
  - **Fault avoidance:** You should program in such a way that you avoid introducing faults into your program.
  - **Input validation:** You should define the expected format for user inputs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

- You should do these *in addition to* any quality attributes that you might choose to prioritize.

1. Fault Avoidance

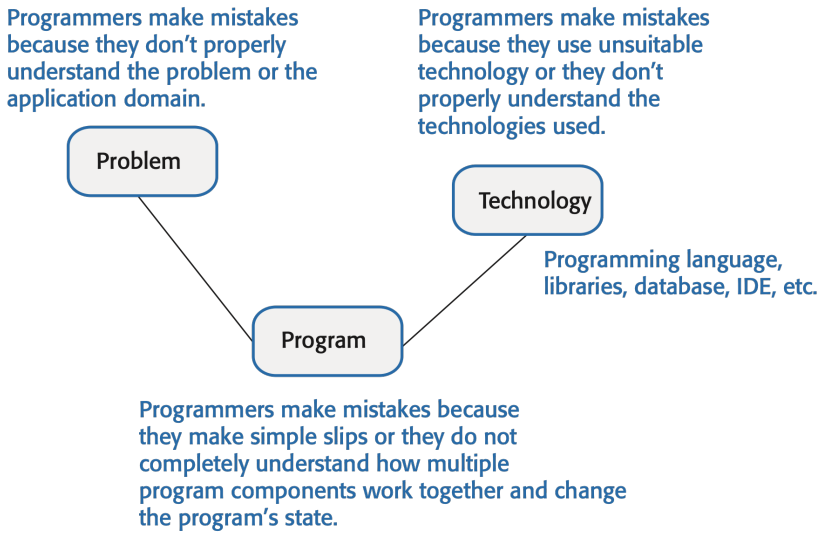
How to reduce the likelihood of errors being introduced in the first place!

Underlying causes of program errors



Software complexity

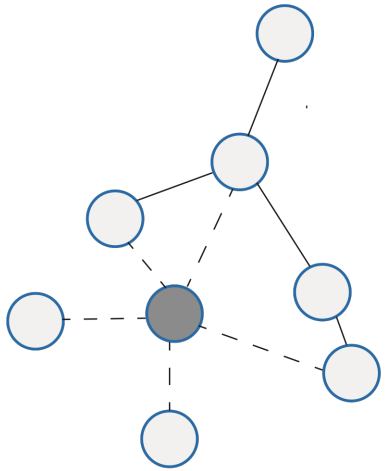
- Ousterhout (2018) points out that **complexity** is a leading cause of programming errors.
- Complexity contributes to problems in each of these areas:
  - **Problem errors** related to lack of understanding of the problem domain.
  - **Technology errors** related to the tech stack being used incorrectly.
  - **Program errors** related to the complexity of the code that you've written.
- Addressing complexity reduces errors.



Program errors occur because of mistakes that programmers make.

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**. *Higher coupling == More complexity.*
  - e.g., the shaded node is highly coupled with six other nodes.
- 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 change over time and are more complex than static relationships. *Dynamic == complex*.
  - An example of a dynamic relationship is the 'calls' relationship between functions.



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

Types of complexity

- **Reading complexity**
  This reflects how hard it is to read and understand the program.
- **Structural complexity**
  This reflects the number and types of relationship between the structures (classes, objects, methods or functions) in your program.
- **Data complexity**
  This reflects the representations of data used and relationships between the data elements in your program.
- **Decision complexity**
  This reflects the complexity of the decisions in your program

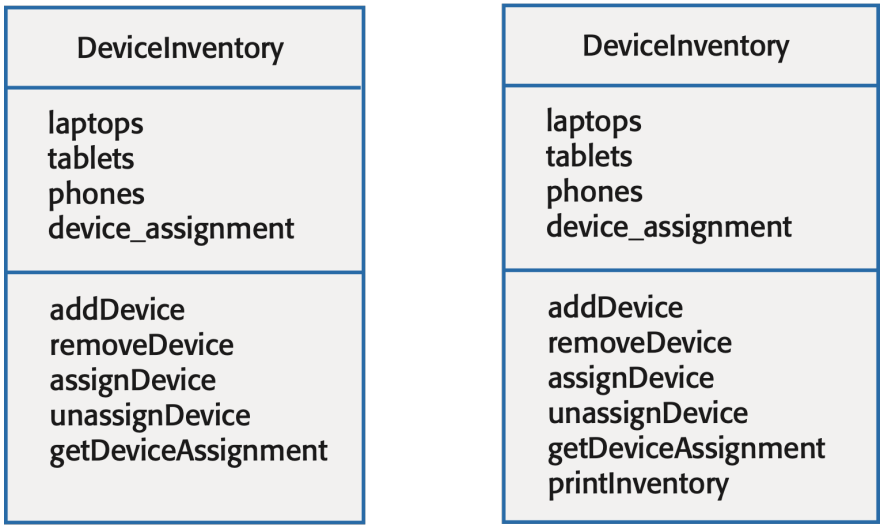
Complexity reduction guidelines

- **Structural complexity**
  - Functions should do one thing and one thing only
  - Functions should never have side-effects
  - Every class should have a single responsibility
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary
- **Data complexity**
  - Define interfaces for all abstractions
  - Define abstract data types
  - Avoid using floating-point numbers
  - Never use data aliases
- **Conditional complexity**
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions

Principle 1: Single Responsibility Principle

- 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 notion of ‘a single reason to change’ is, I think, quite hard to understand.
- However, in a blog post, Bob Martin explains the single responsibility principle in a much better way:
  - Gather together the things that change for the same reasons.
  - Separate those things that change for different reasons.*

Example: `DeviceInventory` class

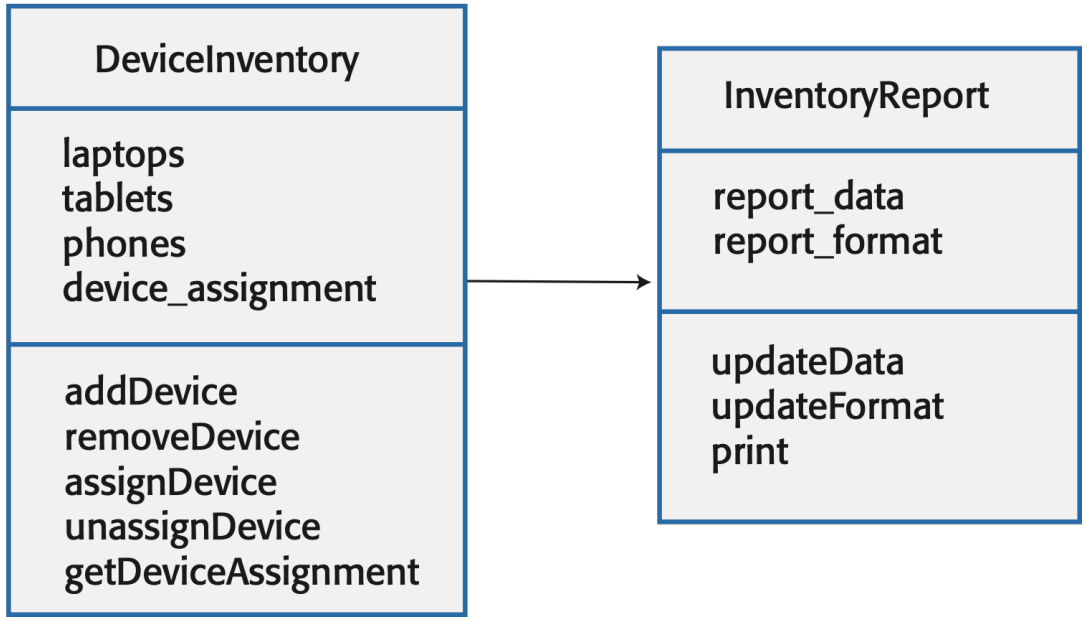


We have a class (a) and we want to add the ability to print a report from this data (b), so we add a new method. This violates our **Single Responsibility** principle. Why?

Adding a `printInventory` method

- Adding a method breaks the single responsibility principle as it then adds an additional ‘reason to change’ the class.
  - Without the `printInventory` method, the reason to change the class is that there has been some fundamental change in the inventory, such as recording who is using their personal phone for business purposes.
  - However, if you add a print method, you are associating another data type (a report) with the class. Another reason for changing this class might then be to change the format of the printed report.
- Instead of adding a `printInventory` method to DeviceInventory, it is better to add a new class to represent the printed report.

Example: `DeviceInventory` & `InventoryReport`



We now have two classes, each with their own specific responsibilities.
We did not have to change the `DeviceInventory` class.

Principle 2: Avoid deeply nested conditionals

- 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 the program names all constants that are used.

Example: deeply nested if-then-else

What's wrong with this?

- How easy is this code to debug?
- How easily can you figure out the return value?

```
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
```

Example: Using guards to make a selection

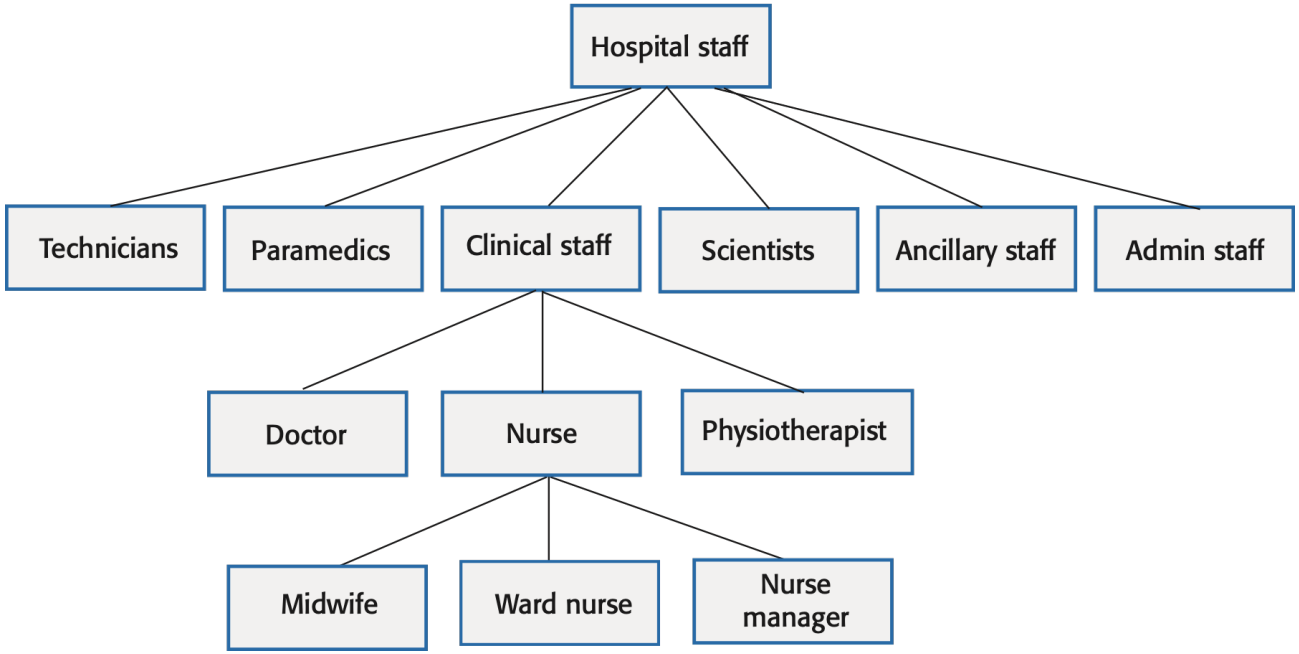
```
def agecheck_with_guards (age, experience):

        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
```

This is much more readable, since you can scan down the LHS until you find a match for Age.
switch > if… then… else. NOTE that Python added a `match` statement in 2021 so this is a dated example.

Principle 3: 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.

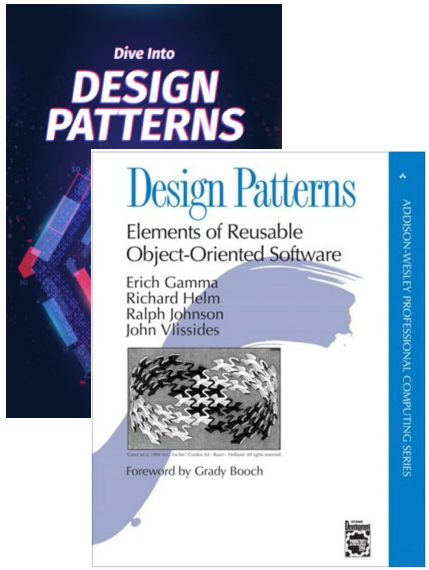
Example: deep inheritance hierarchy



Part of a 4-level inheritance hierarchy that could be defined for staff in a hospital.

Design pattern definition

A design pattern is defined as a "*a general reusable solution to a commonly-occurring problem within a given context in software design*" -- Gamma et al. 1994



- Design patterns describe solutions in terms of objects and classes. They are not off-the-shelf solutions.
- They describe the structure of a problem solution but must be adapted to suit your application and programming language.
- They trade increased complexity now for the promise of benefits later.

Reinforcing 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. For example, if authentication is a concern in your program, then everything to do with authentication should be in one place, rather than distributed throughout your code.
- 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.

Types of patterns

The original set of patterns were subdivided based on the types of problems they addressed.

- Creational Patterns: dynamic creation of objects.
- Structural Patterns: organizing classes to form new structures.
- Behavioral Patterns : identifying communication patterns between objects.

The expectation is that you might encounter a small number of these in any given application.

Some problems are commonly encountered (e.g. decoupling using Observer) and others are rarely used (e.g. Abstract Factory).

Singleton Pattern (Creational)

A **singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Why is this pattern useful?

- It can be used to control access to a shared resource—for example, a database or a file. Singleton guarantees that there is only a single object instance managing that resource.
- Provides a global access point to that instance. Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program.

Singleton Example (Java)

Here's an implementation:

1. Make the default constructor private, to prevent other objects from creating an instance of it.

2. Create a static method to instantiate the class. This method ensures that the object is only instantiated once.

3. Create a static method to return a static reference to the object.

```
public class Singleton {
    private Singleton() {}
    private static instance: Singleton = null

    public static getInstance(): Singleton {
        if (instance == null) {
            instance = Singleton()
        }
        return instance
    }
}

// get a reference to it
Singleton s = Singleton.getInstance()
```

Singleton Example (Kotlin)

In Kotlin, implementation is easier.

The ‘object’ keyword in Kotlin defines a static instance of a class. Effectively, an object is a Singleton and we can just call its methods statically.

Like any other class, you can add properties and methods if you wish.

You do not need initialize it — it’s lazy initialized as needed.

```
object Singleton {
    init {
        println("Singleton class invoked.")
    }
    fun print(){
        println("Print method called")
    }
}


fun main(args: Array<String>) {
    Singleton.print()
    // echos “Singleton class invoked.”
    // echos "Print method called"
}
```

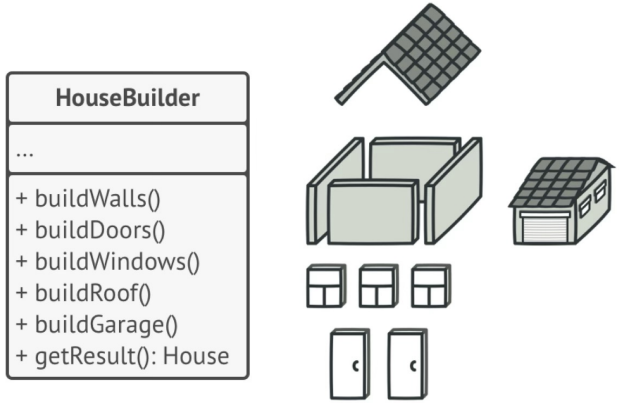
Builder Pattern (Creational)

How do you build complex objects with multiple (optional) initialization steps?

Builder lets you construct complex objects step by step. Produce different types and representations of an object using the same construction code.

A builder class generates the initial object, and subsequent methods can be called to customize it.

- After calling the constructor, call methods to invoke the steps in the correct order.
- You only call the steps that you require, which are relevant to what you are building.



The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Builder Pattern Example (Java)

How do you implement it in other languages? e.g., Java

```
val dialog = AlertDialog.Builder(this)
      .setTitle(“File Save Error”)
      .setText(“Error encountered. Continue?”)
      .setIcon(ERROR_ICON)
      .setType(YES_NO_BUTTONS)
      .show()
```

Builder Pattern Example (Kotlin)

Kotlin supports named and default arguments, simplifying this.

```
val dialog = AlertDialog(
      title = “File Save Error”,
      error = Error encountered. Continue?”,
      icon = ERROR_ICON,
      type = YES_NO_BUTTONS
)
```

Command Pattern (Behavioural)

Imagine that you are writing a user interface, and you want to support a common action like Save. You might invoke Save from the menu, or a toolbar, or a button. Where do you put the code, without duplicating it?

The **command pattern** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request (a command could also be thought of as an action to perform).

Several classes implement the same functionality.

Command Pattern Example (Kotlin)

```kotlin
// Entry point
fun main(args: Array<String>) {
    val command = CommandFactory.createFromArgs(args)
    command.execute()
}

// Factory Method
object CommandFactory {
    fun createFromArgs(args: Array<String>): Command =
        if (args.isEmpty()) {
            when (args[0]) {
                "add" -> AddCommand(args)
                "del" -> DelCommand(args)
                "show" -> ShowCommand(args)
                else -> HelpCommand(args)
            }
        }
}
```

Command Pattern Example (Kotlin)

```kotlin
interface Command {
    fun execute()
}

class AddCommand(val args: Array<String>) : Command {
    override fun execute() {
        assert(args.size == 2)
        println("Add: ${args[1]}")
    }
}

class DelCommand(val args: Array<String>) : Command {
    override fun execute() {
        assert(args.size == 2)
        println("Delete: ${args[1]}")
    }
}
```

Refactoring

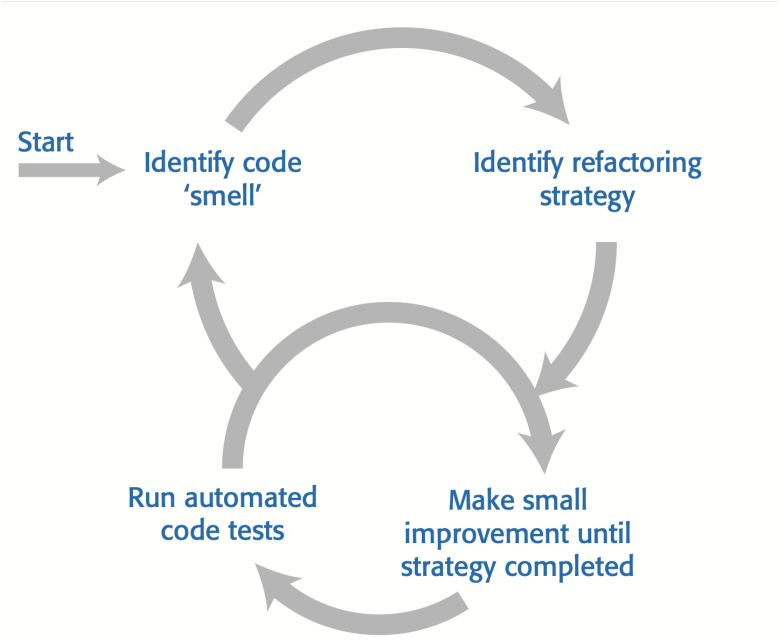
The reality of programming is that as you make changes and additions to existing code, you inevitably increase its complexity. How do we overcome this tendency?

Refactoring is the process of changing a program to reduce its complexity without changing the external behaviour of that program.

- It makes a program more readable (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.

Code smells? Time to refactor!

- Martin Fowler, a refactoring pioneer, suggests that 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.



Example code smells

Large classes

- This suggests 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.

Duplicated code

- Rewrite to create a single instance of the duplicated code that is used as required

Meaningless names

- Theses make the code harder to understand. Replace with meaningful names and check for other shortcuts that the programmer may have taken.

Unused code

- This increases the reading complexity of the code. Delete it! It's in your Git history right?

Example 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.

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, as I explained earlier in this chapter.

IntelliJ supports refactoring! (Ctrl-T)

- **Rename (Shift+F6):** Rename a variable, method, class, or other element and updates all references to it throughout the codebase.
- **Extract Method (Ctrl+Alt+M):** Convert a block of code into a new method.
- **Inline (Ctrl+Alt+N):** Replace method calls with the method's code.
- **Change Signature (Ctrl+F6):** Modify the signature of a method, including parameters, return type, and visibility.
- **Move (F6):** Move classes, methods, or variables to a different package or class.
- **Extract Variable (Ctrl+Alt+V):** Extract a selected expression into a new variable.
- **Extract Field (Ctrl+Alt+F):** Extract a selected expression into a new field.
- **Introduce Parameter (Ctrl+Alt+P):** Introduce a new parameter to a method/constructor.
- **Safe Delete (Alt+Delete):** Delete a file/element without breaking references.

2. Input Validation

Make sure that invalid data doesn’t cause problems.

Input validation

- Input validation involves checking that a user's input is in the correct format and that its value is within the range defined by input rules.
  - This is critical for security and reliability. As well as inputs from attackers that are deliberately invalid, input validation catches accidentally invalid inputs that could crash your program or pollute your database.
- User input errors are the most common cause of database pollution.
  - You should define rules for every type of input field and you should include code that applies these rules to check the field's validity.
  - If it does not conform to the rules, the input should be rejected.

Example: Rules for name checking

Sample rule for checking a person's name:

- The length of a name should be between 2 and 40 characters.
- The characters in the name must be alphabetic or alphabetic characters with an accent, plus a small number of special separator characters. Names must start with a letter.
- The only non-alphabetic separator characters allowed are hyphen, and apostrophe.
- If you use rules like these, it becomes impossible to input very long strings that might lead to buffer overflow, or to embed SQL commands in a name field.

What's wrong with this example?

Methods of implementing input validation

Built-in validation functions

- You can use input validator functions provided by your web/UI development framework. For example, most frameworks include a validator function that will check an email address.

Type coercion functions

- You can use type coercion functions, such as int() in Python, that convert the input string into the desired type. If the input is not a sequence of digits, the conversion will fail.

Explicit comparisons

- You can define a list of allowed values and possible abbreviations and check inputs against this list. For example, check against a list of all months or valid abbreviations.

Regular expressions

- You can use regular expressions to define a pattern that the input should match and reject inputs that do not match that pattern.

Regular expressions

- Regular expressions (REs) are a way of defining patterns.
- A search can be defined as a pattern and all items matching that pattern are returned. For example, the following Unix command will list all the JPEG files in a directory:

```
ls | grep ..*\.jpg$
```

- A single dot means ‘match any character’ and `\*` means zero or more repetitions of the previous character. Therefore `..\*` means ‘one or more characters’. The file prefix is .jpg and the $ character means that it must occur at the end of a line.

Example: use regex to check names

```
def namecheck (s):

        # checks that a name only includes alphabetic characters, -, or 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
```

Number checking

- Number checking is used with numeric inputs to check that these are not too large or small and that they are sensible values for the type of input.
  - e.g., if the user is expected to input their height in meters then you should expect a value between 0.6m (a very small adult) and 2.6m (a very tall adult).
- Number checking is important for two reasons:
  - If numbers are too large or too small to be represented, this may lead to unpredictable results and numeric overflow or underflow exceptions. If these exceptions are not properly handled, very large or very small inputs can cause a program to crash.
  - The information in a database may be used by several other programs and these may make assumptions about the numeric values stored. If the numbers are not as expected, this may lead to unpredictable results.

Input range checks

- As well as checking the ranges of inputs, you may also perform checks on these inputs to ensure that these represent sensible values.
- These protect your system from accidental input errors and may also stop intruders who have gained access using a legitimate user's credentials from seriously damaging their account.
- For example, if a user is expected to enter the reading from an electricity meter, then you should
  - (a) check this is equal to or larger than the previous meter reading and
  - (b) consistent with the user's normal consumption.

3. Failure Management

Despite all your efforts, something went wrong. Now what?

Failure management

- Software is so complex that, irrespective of how much effort you put into fault avoidance, you will make mistakes. You will introduce faults into your program that will sometimes cause it to fail.
- Program failures may also be a consequence of the failure of an external service or component that your software depends on.
- Whatever the cause, you have to plan for failure and make provisions in your software for that failure to be as graceful as possible.

Failure categories

Data failures

- The outputs of a computation are incorrect. For example, if someone's year of birth is 1981 and you calculate their age by subtracting 1981 from the current year, you may get an incorrect result. Finding this kind of error relies on users reporting data anomalies that they have noticed.

Program exceptions

- The program enters a state where normal continuation is impossible. If these exceptions are not handled, then control is transferred to the run-time system which halts execution. For example, if a request is made to open a file that does not exist then an IOexception has occurred.

Timing failures

- Interacting components fail to respond on time or where the responses of concurrently-executing components are not properly synchronized. For example, if service S1 depends on service S2 and S2 does not respond to a request, then S1 will fail.

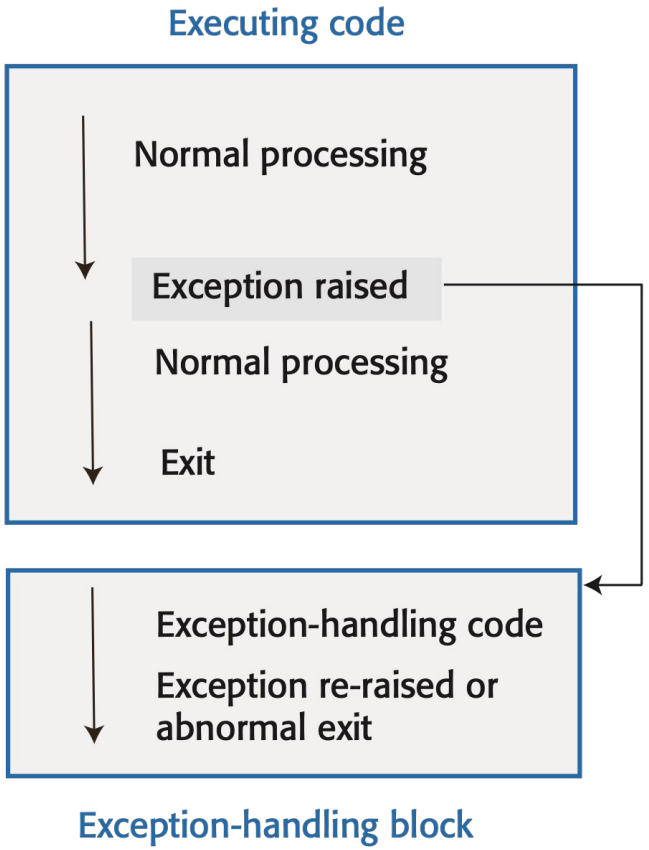
Failure effect minimisation

- Persistent data (i.e. data in a database or files) should not be lost or corrupted;
- The user should be able to recover the work that they've done before the failure occurred;
- Your software should not hang or crash;
- You should always 'fail secure' so that confidential data is not left in a state where an attacker can gain access to it.

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 Kotlin, the equivalent keywords are **try-catch.**

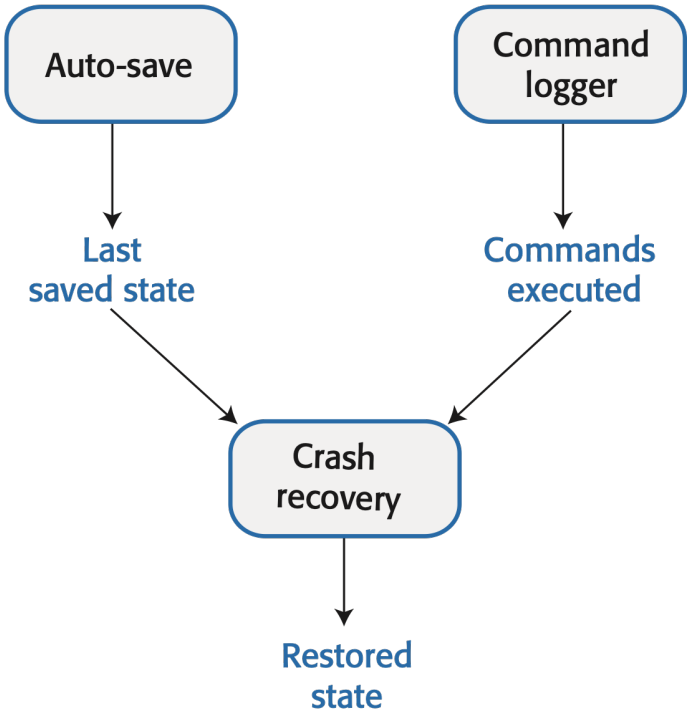
Exception handling



Auto-save and activity logging

- Activity logging
  - You keep a log of what the user has done and provide a way to replay that against their data. You don't need to keep a complete session record, simply a list of actions since the last time the data was saved to persistent store.
- Auto-save
  - You automatically save the user's data at set intervals - say every 5 minutes. This means that, in the event of a failure, you can restore the saved data with the loss of only a small amount of work.
  - Usually, you don't have to save all of the data but simply save the changes that have been made since the last explicit save.

Auto-save and activity logging



External service failure

- If your software uses external services, you have no control over these services and the only information that you have on service failure is whatever is provided in the service's API.
- As services may be written in different programming languages, these errors can't be returned as exception types but are usually returned as a numeric code.
- When you are calling an external service, you should always check that the return code of the called service indicates that it has operated successfully.
- You should, also, if possible, check the validity of the result of the service call as you cannot be certain that the external service has carried out its computation correctly.

Summary 1

- 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 increases the chances of programmer errors and makes the program more difficult to change.
- 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 2

- 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.
- 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.

Reference

- Fowler. 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. ISBN 978-0134757599.
- Gamma et al. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 978-0321700698.
- Ousterhout. 2018. A Philosophy of Software Design. Yaknyam Press. ISBN 978-1732102200.
- Shvets. 2021. Refactoring Guru: Design Patterns. Online.
- Sommerville. 2021. Engineering Software Products: An Introduction to Modern Software Engineering. Pearson. ISBN 978-1292376356.