

Engenharia de Software (14341, 16230, 15386)

Software Construction, Code Quality, Reviews & Refactoring

(adapted from lecture notes of the "DIT 635 - Software Quality and Testing" unit,
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

Topics covered



01

Software evolution & maintenance

Understand why software continuously evolves and why maintenance dominates the software lifecycle.



02

Refactoring fundamentals

Learn what refactoring is, why it matters and how it preserves software behaviour.



03

Code smells & refactoring techniques

Recognize common code smells and discover practical techniques for improving software design.



04

Modern refactoring practice

Explore how professional development environments support continuous refactoring through static analysis, automation and AI-assisted development.



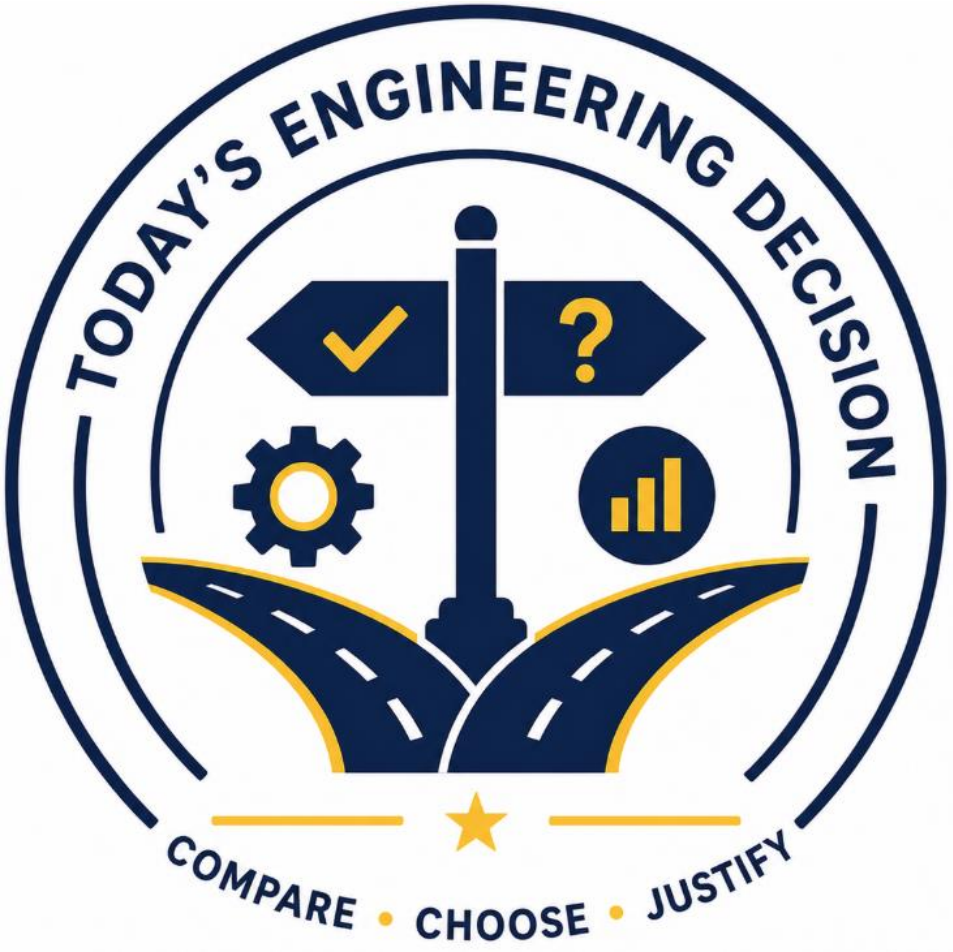
Goal: Learn how professional software engineers continuously improve software while preserving its external behaviour.




PROFESSIONAL SKILL
SPOTLIGHT
DEVELOP • APPLY • GROW

Critical Thinking




TODAY'S ENGINEERING DECISION
?
COMPARE • CHOOSE • JUSTIFY

Today’s engineering decision

A bridge is safe to cross today, but a recent inspection reveals **corrosion** in several structural elements. The bridge still supports traffic safely.



What should the engineering team do?

A **Keep using it and inspect later.**



Risk increases over time. Problems may get worse.

B **Close the bridge and rebuild everything.**



Very costly and disruptive. Overkill for the current condition.

C **Repair the affected parts and continue monitoring.**



Addresses the risk, keeps the bridge safe and maintains value.

D **Ignore the inspection because the bridge hasn’t failed.**



Unsafe and irresponsible. Failure could be catastrophic.

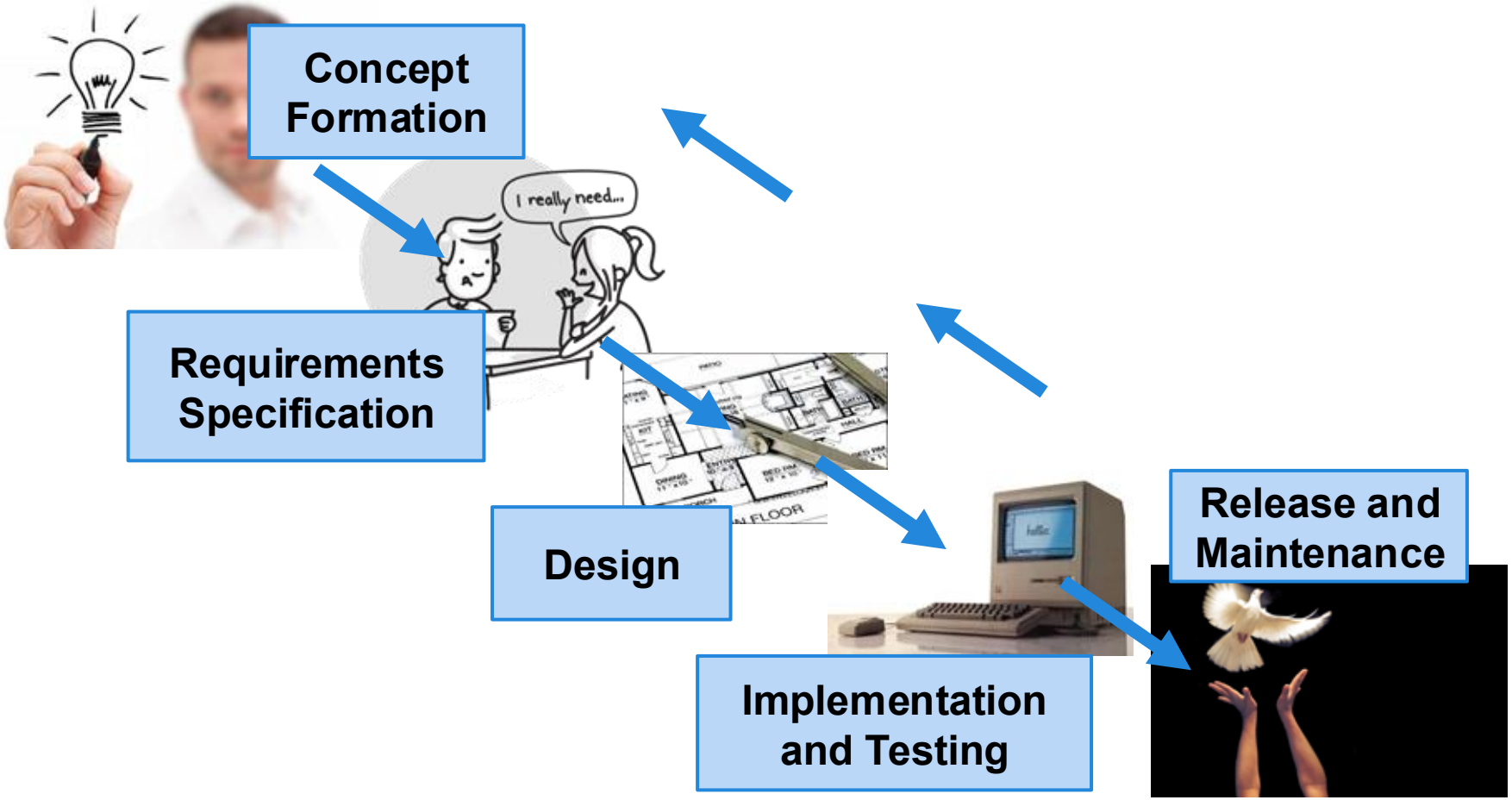
DISCUSSION PROMPTS

- Which option best balances safety, cost and long-term reliability?
- How do you manage risk without overreacting?
- When is it better to fix incrementally rather than rebuild?
- How do inspection and monitoring support good decisions?

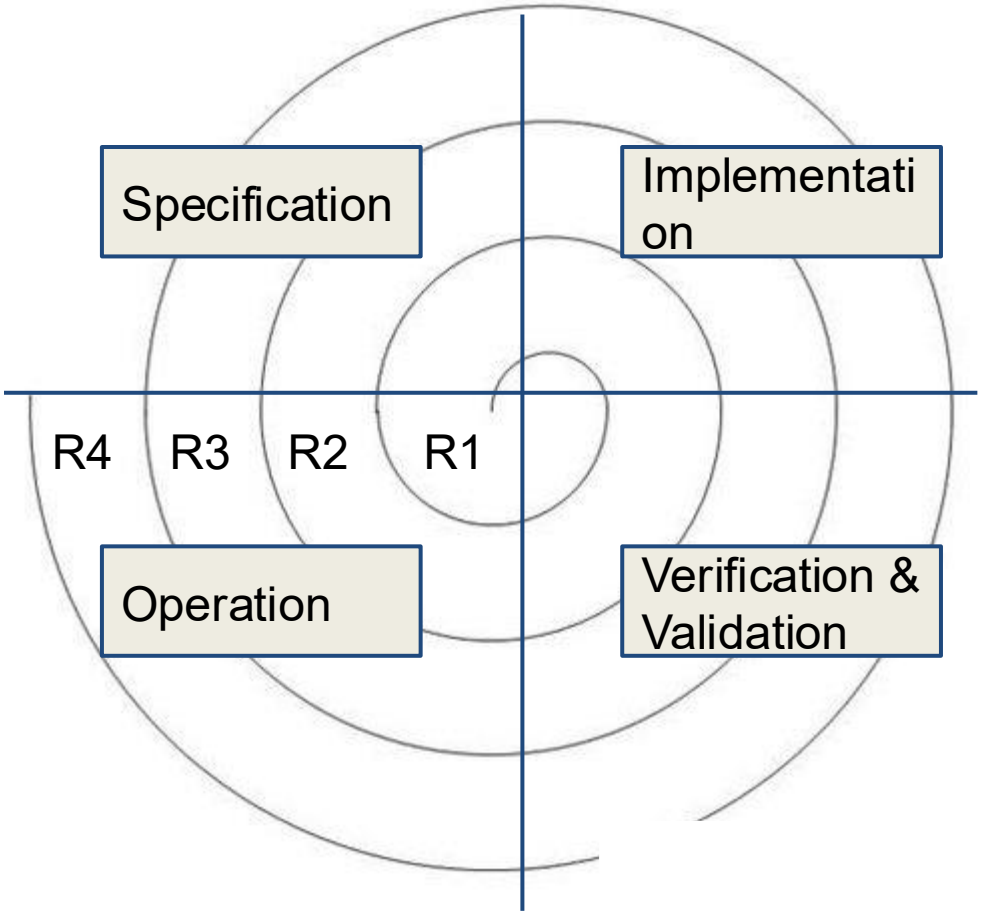
We’ll revisit this decision at the end of the lecture.

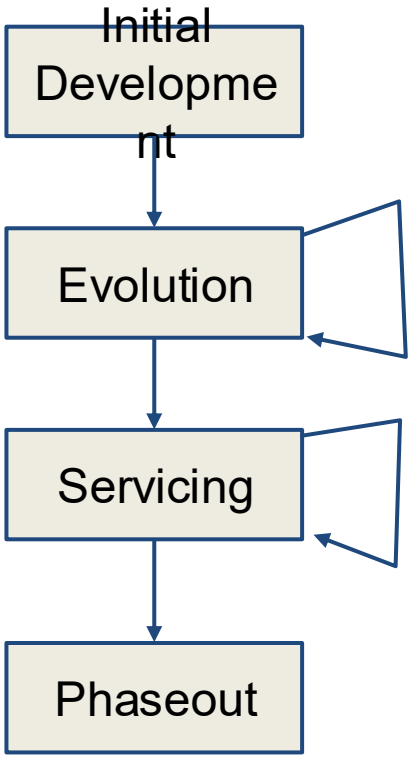
Good engineers make informed trade-offs!

The software lifecycle



The real lifecycle





Software maintenance

- Fault Repairs
  - Changes made in order to correct coding, design, or requirements errors.
- Environmental Adaptations
  - Changes made to accommodate changes to the hardware, OS platform, or external systems.
- Functionality Addition
  - New features are added to the system to meet new user requirements.

Software maintenance effort

- Maintenance costs more than the initial development.
  - 2/3rds of budget goes to maintenance on average.
  - Up to four times the development cost to maintain critical systems.
- General breakdown:
  - 65% of effort goes to functionality addition
  - 18% to environmental adaptation
  - 17% to fault repair

Maintenance is hard

It is harder to maintain than to write new code.

- Must understand code written by another developer, or code that you wrote long ago.
- Creates a “house of cards” effect.
- Developers tend to prioritize new development.

Smooth maintenance requires planning and design that supports maintainability.

The laws of software evolution

- Maintenance is an inevitable process.
  - Requirements change as the environment changes.
  - Changing the software causes environmental changes, which leads to more requirement changes.
- As changes occur, the structure degrades.
  - When changes are made, the structure becomes more complex.
  - To prevent this, resources must go into *preventative maintenance* - refactoring to preserve and simplify the structure without adding to functionality.

The laws of software evolution

- The amount of change in each release is approximately constant.
  - The more functionality introduced, the more faults.
  - A large functionality patch tends to be followed by a patch that fixes faults without adding additional functionality. Small functionality changes do not require a fault-correcting patch.
- Functionality must continually increase to maintain user satisfaction.

The laws of software evolution

- The quality of the system will decline unless updated to work with changing environment.
- To improve quality, evolution must be treated as a feedback system.
  - Stakeholders must be continually involved in evolution, and changes should be influenced by their needs.

Refactoring

- Process of revising the code or design to improve its structure, reduce complexity, or otherwise accommodate change.
- When refactoring, you do not add functionality.
- Continuous process of improvement throughout the evolution of the system.

Why refactor?

Why fix what isn't broken?

- Components have three purposes:
  - To perform a service.
  - To allow change.
  - To be understood by developers reading it.
- If it does not do any of these, it is "broken".
- Enables change and improves understandability.

Refactoring is an iterative process

- Refactoring should take place as an iterative cycle of small transformations.
  - Choose a small part of the system, redesign it, and make sure it still works.
  - Choose a new section of the system and refactor it.
- Refactoring requires unit tests.
  - Make sure the code works before and after.

Modern refactoring ecosystem



Refactoring in modern development



SUPPORTED BY MODERN TOOLS

- **IDEs**: VS Code, IntelliJ IDEA
- **Static Analysis**: SonarQube, ESLint, PMD
- **AI Assistants**: GitHub Copilot, ChatGPT, Gemini
- **Quality Gates**: SonarQube Quality Gate, CodeQL, Checkstyle
- **CI/CD**: GitHub Actions, Jenkins, GitLab CI
- **Monitoring**: Dashboards, Alerts, Metrics

Continuous Improvement

Refactoring is not a one-time activity.
It is a continuous practice that keeps the codebase healthy, maintainable and ready for change.

> Good design is not about perfect code today, but about easy change tomorrow.

Choosing what to refactor

- Refactor any piece of the system that:
  - Seems to work,
  - But isn't well designed,
  - And now needs new functionality.
- There are stereotypical situations that indicate the need for refactoring.
  - These are called **"bad smells"**.

Code smells

- **Code is duplicated** in multiple places.
- A method is **too long**.
- Conditional statements control behavior based on an **object type**.
- Groups of data **attributes are duplicated**.
- A class has **poor cohesion** or **high coupling**.
- A method has **too many parameters**.
- **Speculative generality** - adding functionality that “we might need someday.”

More code smells

- Changes must be made in **several places**.
- **Poor encapsulation** of data that should be private.
- If a **weak subclass** does not use inherited functionality.
- If a class contains **unused code**.
- If a class contains **potentially unused attributes** that are only set in particular circumstances.
- There are data classes containing only attributes, getters, and setters, but nothing else - **objects should encapsulate data and behaviors.**
  - Unless that data is used by multiple classes.

Common refactorings (more at http://www.refactoring.com)

Composing Methods

- Extract Method
- Inline Method; Inline Temp
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Substitute Algorithm

Moving Features Between Objects

- Move Method; Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middleman
- Introduce Foreign Method

Organizing Data

- Replace Data Value with Object
- Change Value to Reference; Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to to Unidirectional

Simplifying Conditional Expressions

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Making Method Calls Simpler

- Rename Method
- Add/Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

Dealing with Generalization

- Pull Up Field; Method; Constructor Body
- Push Down Method; Push Down Field
- Extract Subclass; Extract Superclass; Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation (or vice versa)

Big Refactorings

- Nature of the Game
- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy

Refactorings - composing methods

- If you have a complex code fragment that can exist independently, **extract it into its own method**.
- If you have a method that is extremely simple, **inline** it into locations where it is used.
- If you assign values to a temporary variable more than once, **split it into additional temporary variables**.
- If assignments are made to parameter variables in a method, instead **assign to a temporary variable**.
- If an algorithm is hard to understand, **swap it for a version that is clearer**.

Refactorings - moving features between objects

- If a method or field is used more by a calling class than the class it is placed in, **move** it.
- If a class is doing more work than it should (or has low cohesion), **extract** a subset of related methods into a new class.
- If a class is doing too little, **combine** it with another.
- If a class delegates too many calls to a middleman class, **get rid of the middleman** and call the client directly.
- If an imported class needs an additional method, but you can't modify it directly, **create a method in the client class with the imported object as a parameter**.

Refactorings - conditional expressions & data

- If your conditional statements are too complex, **extract methods from the if, then, and else conditions**.
- If you have a sequence of conditional tests with the same result or repeated conditions in each branch, **consolidate them into fewer conditional statements**.
- If you have conditional statements to choose behavior based on object type, instead **use polymorphism**.
- If you have an attribute that needs additional data or operations, turn it into a new type of **data object**.
- If certain array values have special meaning, use **a class to store items instead**.

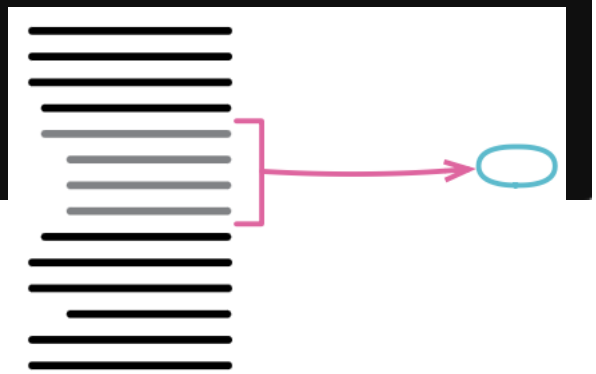
Refactorings - simplifying method calls and generalization

- If a method both returns a value and changes the state of a passed object, split into two methods and **separate the query from the modifier**.
- If several methods do similar things - differentiated by value - create one method that takes the **value as a parameter**.
- If two classes have the same attribute/method/constructor body, **pull it up into the parent**. If an item is only used by some subclasses, **push it into the children**.
- If a class has features only used situationally, **extract subclasses** for those situations.

Example 1: Extract method

✧ Before: Long method doing multiple things.

```
def print_owing(amount):
    print('*****')
    print('Amount: ' + str(amount))
```



Example 1: Extract method

- ✧ After: Break the method into smaller, well-named methods.
- ✧ Explanation: Improves readability and reusability.

```
def print_banner():
    print('*****')

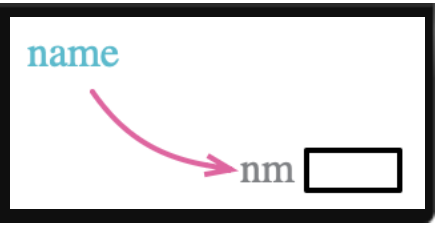
def print_details(amount):
    print('Amount: ' + str(amount))

def print_owing(amount):
    print_banner()
    print_details(amount)
```

Example 2: Rename variable

✧ Before: Unclear or misleading variable names.

```
int d = 0;
int t = 100;
int a = d + t;
```



Example 2: Rename variable

- ✧ After: Replace with meaningful, descriptive names.
- ✧ Explanation: Enhances code readability and maintainability.

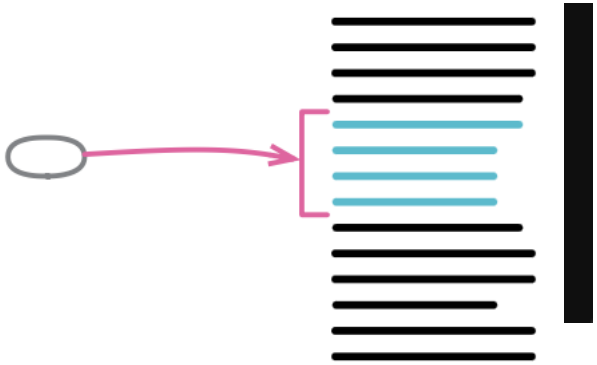
```
int distance = 0;
int time = 100;
int total = distance + time;
```

Example 3: Inline method

✧ Before: Method that is too simple and used only once.

```
def get_rating():
    return more_than_five_late_deliveries()

def more_than_five_late_deliveries():
    return self.number_of_late_deliveries > 5
```



Example 3: Inline method

✧ After: Inline the method directly into the caller.

✧ Explanation: Reduces unnecessary indirection.

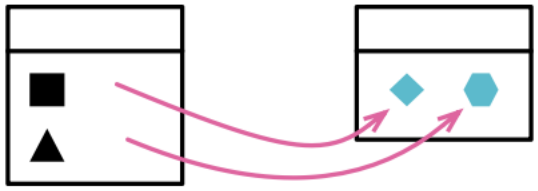
```
def get_rating():
    return self.number_of_late_deliveries > 5
```

Example 4: Extract class

✧ Before: Class doing too much (God class).

```
class Person:
    def __init__(self, name, office_area_code, office_number):
        self.name = name
        self.office_area_code = office_area_code
        self.office_number = office_number

    def get_telephone_number(self):
        return '(' + self.office_area_code + ') ' + self.office_number
```



Example 4: Extract class

- ✧ After: Split responsibilities into multiple classes.
- ✧ Explanation: Promotes Single Responsibility Principle (SRP).

```
class Person:
    def __init__(self, name, telephone_number):
        self.name = name
        self._telephone_number = telephone_number

    def get_telephone_number(self):
        return self._telephone_number

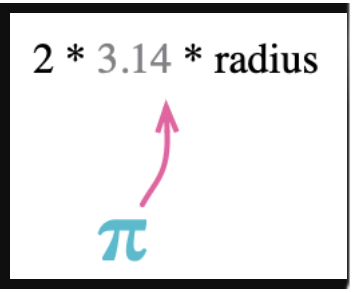
class TelephoneNumber:
    def __init__(self, office_area_code, office_number):
        self._office_area_code = office_area_code
        self._office_number = office_number

    def get_telephone_number(self):
        return '(' + self._office_area_code + ') ' + self._office_number
```

Example 5: Replace magic number with symbolic constant

✧ Before: Hard-coded numbers in code.

```
if (salary > 100000)
    tax = salary * 0.4;
else
    tax = salary * 0.3;
```



Example 5: Replace magic number with symbolic constant

✧ After: Replace with named constants.

✧ Explanation: Makes the code self-explanatory.

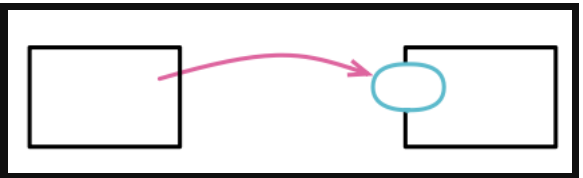
```
const TAX_THRESHOLD = 100000;
const HIGH_TAX_RATE = 0.4;
const LOW_TAX_RATE = 0.3;

if (salary > TAX_THRESHOLD)
    tax = salary * HIGH_TAX_RATE;
else
    tax = salary * LOW_TAX_RATE;
```

Example 6: Move method

✧ Before: Method more related to another class.

```
class Account:
    def overdraft_charge(self):
        if self._type.is_premium():
            result = 10
            if self._days_overdrawn > 7:
                result += (self._days_overdrawn - 7) * 0.85
            return result
        else:
            return self._days_overdrawn * 1.75
```



Example 6: Move method

- ✧ After: Move the method to appropriate class.
- ✧ Explanation: Improves code organization and relevance.

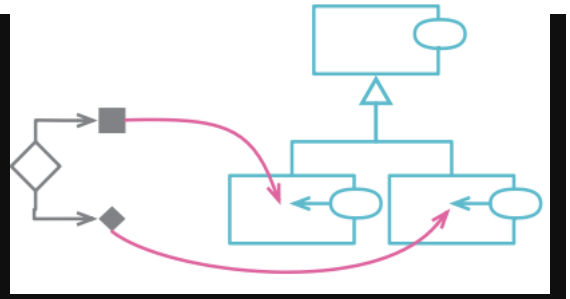
```
class AccountType:
    def overdraft_charge(self, days_overdrawn):
        if self.is_premium():
            result = 10
            if days_overdrawn > 7:
                result += (days_overdrawn - 7) * 0.85
            return result
        else:
            return days_overdrawn * 1.75

class Account:
    def overdraft_charge(self):
        return self._type.overdraft_charge(self._days_overdrawn)
```

Example 7: Replace conditional with polymorphism

✧ Before: Complex conditional logic.

```
class Bird:
    def get_speed(self):
        if self.type == 'EUROPEAN':
            return self.get_base_speed()
        elif self.type == 'AFRICAN':
            return self.get_base_speed() - self.get_load_factor() * self.number_
        elif self.type == 'NORWEGIAN_BLUE':
            return 0 if self.is_nailed else self.get_base_speed() * self.voltage
```



Example 7: Replace conditional with polymorphism

- ✧ After: Replace with polymorphic classes or strategies.
- ✧ Explanation: Simplifies logic and enhances flexibility.

```
class Bird:
    def get_speed(self):
        return self.type.get_speed(self)

class European(Bird):
    def get_speed(self):
        return self.get_base_speed()

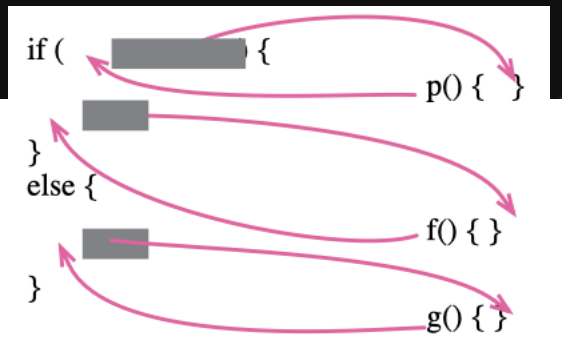
class African(Bird):
    def get_speed(self):
        return self.get_base_speed() - self.get_load_factor() * self.number_of_c

class NorwegianBlue(Bird):
    def get_speed(self):
        return 0 if self.is_nailed else self.get_base_speed() * self.voltage
```

Example 8: Decompose conditional

✧ Before: Complex and nested conditional statements

```
if date.before(SUMMER_START) or date.after(SUMMER_END):
    charge = quantity * winter_rate + winter_service_charge
else:
    charge = quantity * summer_rate
```



Example 8: Decompose conditional

- ✧ After: Break down into methods with clear names.
- ✧ Explanation: Increases clarity and reduces code complexity.

```
if is_summer(date):
    charge = summer_charge(quantity)
else:
    charge = winter_charge(quantity)

# Methods extracted:
def is_summer(date):
    return not (date.before(SUMMER_START) or date.after(SUMMER_END))

def summer_charge(quantity):
    return quantity * summer_rate

def winter_charge(quantity):
    return quantity * winter_rate + winter_service_charge
```

Example 9: Replace inheritance with composition

✧ Before: Inheritance leads to rigid and brittle code.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Salesman(Employee):
    def __init__(self, name, salary, commission):
        super().__init__(name, salary)
        self.commission = commission
```

Example 9: Replace inheritance with composition

✧ After: Use composition instead of inheritance.

✧ Explanation: Improves flexibility and reusability.

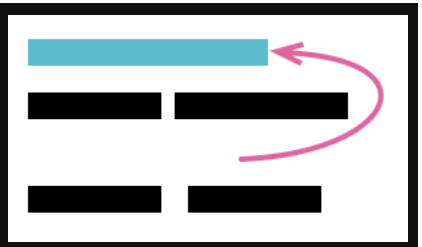
```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Salesman:
    def __init__(self, employee, commission):
        self.employee = employee
        self.commission = commission
```

Example 10: Consolidate duplicate conditional fragments

✧ Before: Duplicate code within conditional branches.

```
if is_special_deal():
    total = price * 0.95
    send()
else:
    total = price * 0.98
    send()
```



Example 10: Consolidate duplicate conditional fragments

✧ After: Consolidate duplicate code outside the conditional.

✧ Explanation: Reduces redundancy and enhances maintainability.

```
if is_special_deal():
    total = price * 0.95
else:
    total = price * 0.98

send()
```

Dangers of refactoring

- Code that used to be well commented, well tested, and fully reviewed might not be any of these things after refactoring.
- You might have inserted faults into code that previously worked.
  - This is why unit tests are important. If the new code is broken, revert back to the old code.
- What if the new design is not better?

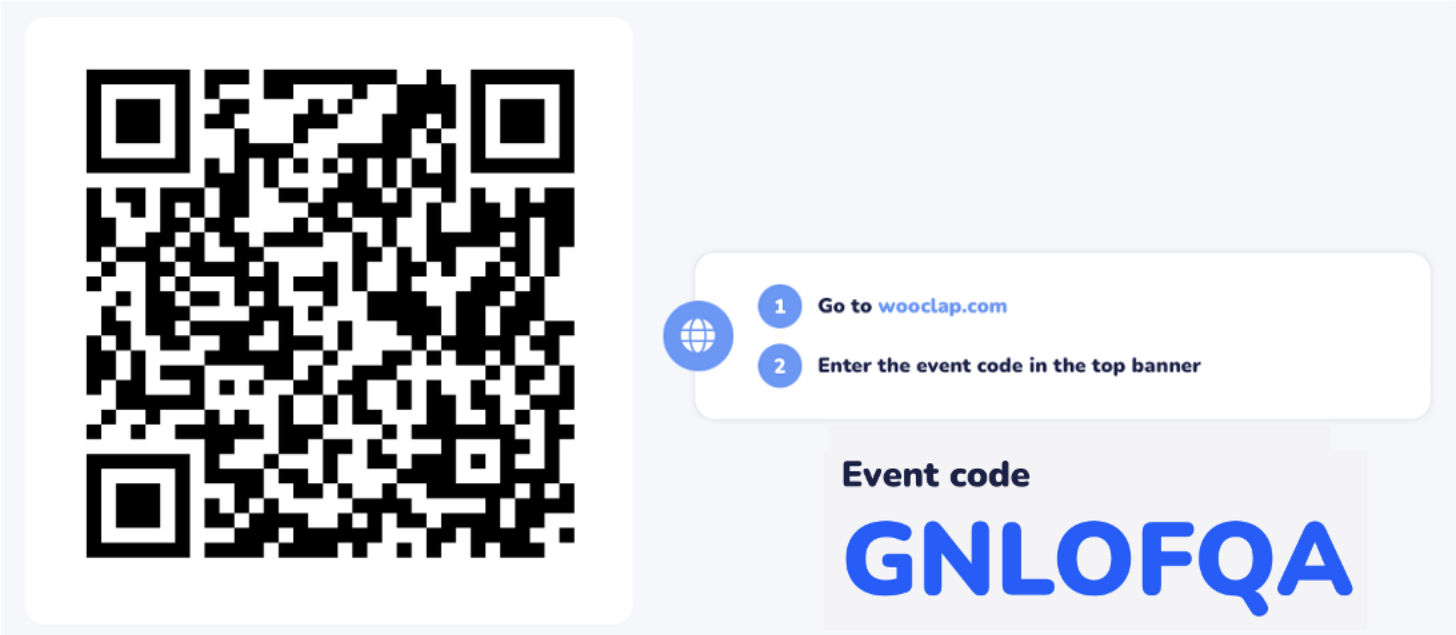
“I Don’t Have Time”

- Most common excuse for not refactoring.
- Refactoring incurs an up-front cost.
  - Developers don’t want to do it.
  - Neither do managers - they lose time and get “nothing” (no new features)
- Small companies (start-ups) avoid it.
  - “We can’t afford it.” “We don’t need it.”
- So do large companies.
  - “We’d rather add new features.”
  - “No one gets promoted for refactoring.”

“I Don’t Have Time”

- Refactoring is the key to effective evolution.
  - Enables rapid addition of new features, with fewer faults (up to a 500% ROI).
  - Good for programmer morale.
- Refactoring is an investment in a company’s prime asset - its code base.
- Many start-ups use cutting-edge tech and agile processes that evolve rapidly. So should the code.
- Some of the most successful companies (Google) reward and require refactoring.

Prioritize the refactoring



Practice#1: Identify the *code smell* and suggest a possible refactoring

```
def compute_area(r):
    a = 3.14 * r * r
    return a
```

Practice#1: Solution (Rename Variable)

```
def compute_area(radius):
    area = 3.14 * radius * radius
    return area
```

Practice#2: Identify the *code smell* and suggest a possible refactoring

```python
def calculate_total(order):
    total = 0
    for item in order['items']:
        total += item['price'] * item['quantity']
    total += order['shipping_cost']
    total += order['tax']
    return total
```

Practice#2: Solution (Extract Method)

```python
def calculate_total(order):
    total = sum(item_total(item) for item in order['items'])
    total += order['shipping_cost']
    total += order['tax']
    return total

def item_total(item):
    return item['price'] * item['quantity']
```

Practice#3: Identify the *code smell* and suggest a possible refactoring

```
def process_order(order):
    if order['is_expedited']:
        send_expedited(order)
        notify_customer(order)
    else:
        send_standard(order)
        notify_customer(order)
```

Practice#3: Solution (Consolidate Duplicate Conditional Fragments)

```
def process_order(order):
    if order['is_expedited']:
        send_expedited(order)
    else:
        send_standard(order)

    notify_customer(order)
```

Practice#4: Identify the *code smell* and suggest a possible refactoring

```python
class Employee:
    def __init__(self, employee_type, salary):
        self.employee_type = employee_type
        self.salary = salary

    def calculate_bonus(self):
        if self.employee_type == 'Manager':
            return self.salary * 0.10
        elif self.employee_type == 'Engineer':
            return self.salary * 0.05
        elif self.employee_type == 'Intern':
            return self.salary * 0.01
```

Practice#4: Solution (Replace Conditional with Polymorphism)

```python
class Employee:
    def __init__(self, salary):
        self.salary = salary

    def calculate_bonus(self):
        raise NotImplementedError("Subclasses should implement this!")

class Manager(Employee):
    def calculate_bonus(self):
        return self.salary * 0.10

class Engineer(Employee):
    def calculate_bonus(self):
        return self.salary * 0.05

class Intern(Employee):
    def calculate_bonus(self):
        return self.salary * 0.01
```

Practice#5: Identify the *code smell* and suggest a possible refactoring

```python
class Bird:
    def __init__(self, color, wing_span):
        self.color = color
        self.wing_span = wing_span

class Penguin(Bird):
    def __init__(self, color, wing_span, swimming_speed):
        super().__init__(color, wing_span)
        self.swimming_speed = swimming_speed

    def swim(self):
        return f"Swimming at {self.swimming_speed} speed"
```

Practice#5: Solution (Replace Inheritance with Composition)

```python
class Bird:
    def __init__(self, color, wing_span):
        self.color = color
        self.wing_span = wing_span

class SwimmingAbility:
    def __init__(self, swimming_speed):
        self.swimming_speed = swimming_speed

    def swim(self):
        return f"Swimming at {self.swimming_speed} speed"

class Penguin:
    def __init__(self, bird, swimming_ability):
        self.bird = bird
        self.swimming_ability = swimming_ability
```

Automated smell detection

✧ Why Automate?

- Manual detection is **time-consuming**, **subjective**, and prone to inconsistency.
- Automated tools:
  - Scan entire codebases quickly.
  - Apply consistent rules.
  - Detect complex metrics-based smells (complexity, duplication, coupling).

Automated smell detection approaches (1 of 2)

✧ Rule-based detection

- Predefined smell patterns matched via AST (Abstract Syntax Tree) analysis.
- Example: Methods > 20 lines trigger a “Long Method” smell.
- Tools: PMD, Checkstyle, ESLint, Pylint.

✧ Metrics-based detection

- Use thresholds on quantitative metrics:
  - Cyclomatic complexity
  - Cognitive complexity
  - Coupling Between Objects (CBO)
  - Lines of Code (LOC)
- Tools: SonarQube, NDepend, CodeClimate.

Automated smell detection approaches (2 of 2)

✧ Duplication detection

- Token-based or tree-based comparison for copy-pasted code.
- Tools: DupFinder, PMD CPD.

✧ ML/AI-assisted detection

- Learn smell patterns from large corpora.
- Can suggest possible refactorings.
- Examples: Deep learning classifiers for “God Class” or “Feature Envy” detection.

Quality Gates in software engineering

- A **Quality Gate** is a set of *non-functional requirements* that code must meet before it is considered acceptable for integration into the main branch.
- **Goals**
  - Enforce a *minimum quality standard*.
  - Prevent regression in maintainability, security, and reliability.
  - Encourage *incremental improvement* by focusing on new code ("Clean as You Code").

Principles of quality gates

- **Shift-left**: Catch issues earlier in the development lifecycle.
- **Automation**: Integrate into CI/CD to run on every pull/merge request.
- **Transparency**: Visible pass/fail status with detailed feedback.
- **Objectivity**: Metrics-driven rather than opinion-driven.

Typical quality gate criteria

Criterion	Rationale
No Blocker/Critical issues	Blockers indicate critical defects or vulnerabilities that must be fixed.
Coverage ≥ 80%	Ensures adequate testing of new features.
Duplication ≤ 3%	Reduces redundancy and maintenance effort.
Maintainability ≥ A	Controls technical debt growth.
Reliability ≥ A	Reduces the risk of production failures.
Security ≥ A	Prevents known vulnerabilities from entering production.
No new security hotspots	Forces review of potentially risky code paths.

Benefits of automated smell detection & quality gates

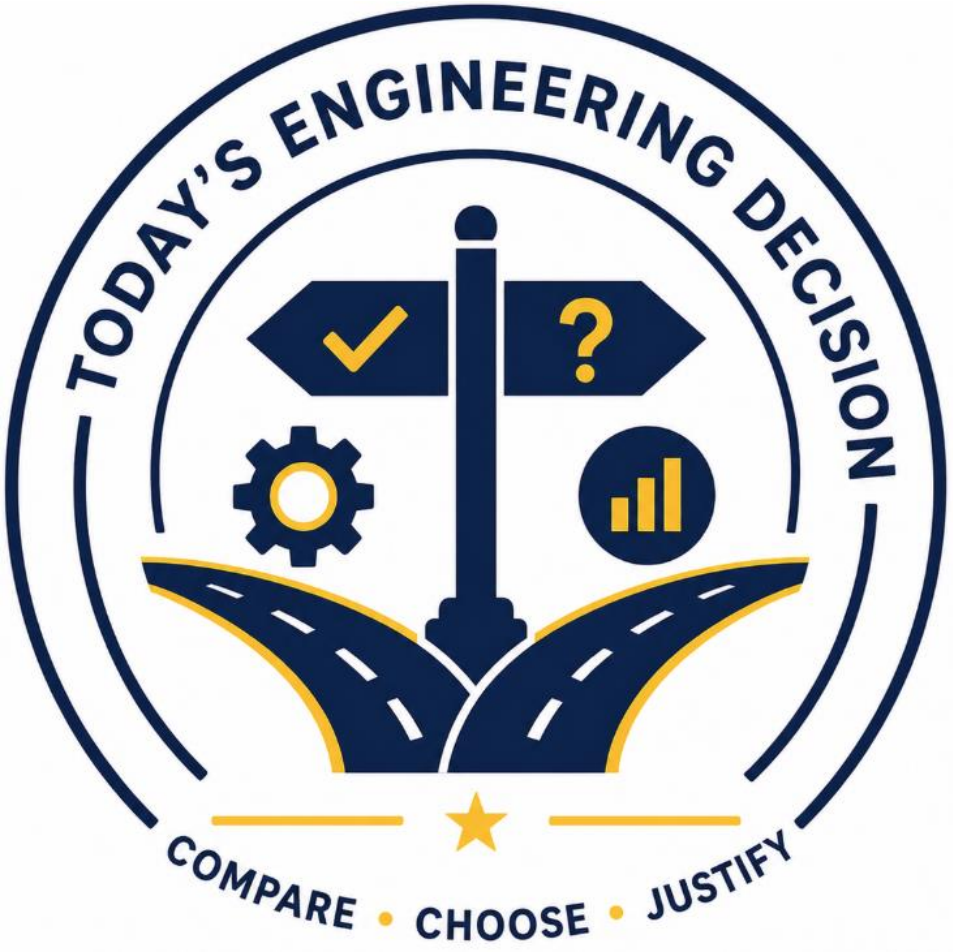
- **Consistency**: Rules are enforced uniformly across the team.
- **Early feedback**: Issues are caught before merging to main.
- **Reduced technical debt**: Incremental cleaning of codebase.
- **Higher maintainability**: Smells are addressed before they proliferate.
- **Improved collaboration**: Objective metrics reduce subjective debates.

Challenges and risks

- **False positives**: Overly strict or poorly tuned rules can frustrate developers.
- **Legacy code drag**: Applying gates to old code can block all progress unless “new code only” mode is used.
- **Bypass culture**: Frequent waivers without accountability weaken the process.
- **Metric gaming**: Teams might optimize for passing the gate rather than improving code quality.

Best practices

- ✧ **Start with new code only**: Avoid paralysis from legacy issues.
- ✧ **Pick a small, impactful rule set**: Begin with security and critical maintainability rules.
- ✧ **Integrate into Pull-Request workflow**: Make the gate a required status check.
- ✧ **Regularly tune rules and thresholds**: Evolve as codebase and team mature.
- ✧ **Pair with education**: Train developers on smell detection and refactoring techniques.


TODAY'S ENGINEERING DECISION
?
COMPARE • CHOOSE • JUSTIFY

Today’s engineering decision

Let's reconsider the decision we made at the beginning of the lecture.

? The scenario

A bridge is safe to cross today, but a recent inspection reveals corrosion in several structural elements.

The bridge still supports traffic safely.



WHAT DID WE LEARN TODAY?

- Software degrades as it evolves.
- Maintenance is continuous.
- Balancing improves informed quality without changing behaviour.
- Small, incremental improvements reduce technical debt.
- Modern teams rely on automated tests and CI/CD.

A Keep using it and inspect later.



Risk increases over time.
Problems may get worse.

B Close the bridge and rebuild everything.



Very costly and disruptive.
Overkill for the current condition.

C Repair the affected parts and continue monitoring.



Addresses the risk, keeps the bridge safe and maintains value.

RECOMMENDED CHOICE

D Ignore the inspection because the bridge hasn't failed.



Unsafe and irresponsible.
Failure could be catastrophic.

ENGINEERING TAKEAWAY

“Great engineers don’t wait for failure. They continuously improve systems before failure occurs.”

What would you do now?

- What is your answer?
- Why or why not?
- What are the trade-offs?

THE ANALOGY: BRIDGE ↔ SOFTWARE

BRIDGE (OUR SCENARIO)		SOFTWARE (OUR SYSTEM)
Bridge connection	⟷	Code & modules
Structural degradation	⟷	Technical debt
Preventive maintenance	⟷	Refactoring
Incremental repairs	⟷	Automated tests
Safe operation	⟷	Continuous delivery

Refactoring is preventive maintenance.

The best time to improve software is before it starts to fail.

Next lecture:

How do we focus our refactoring efforts and test software?

→ **SOFTWARE TESTING**

Engineering insight

Refactoring is preventive maintenance.
It improves the internal structure of the software so it can continue to evolve safely and affordably.

Reduce risk before problems become failures.

Lower long-term costs and increase agility.

Deliver more value with higher quality and confidence.

WHAT THIS MEANS

Early attention pays off.
Address issues when they are small and easier to fix.

Iterate, don't overhaul.
Small, incremental refactoring steps reduce risk and preserve momentum.

Tests are our safety net.
Automated tests give us confidence that behaviour stays the same.

Quality is a team sport.
Clean code, reviews and shared standards sustain long-term quality.

Continuous improvement is the goal.
Better structure today enables faster and safer change tomorrow.

THE ANALOGY: BRIDGE ↔ SOFTWARE





BRIDGE ENGINEERING		SOFTWARE ENGINEERING
Inspect regularly	↔	Detect code smells early
Identify areas of deterioration	↔	Identify technical debt
Plan small repairs	↔	Refactor incrementally
Ensure safety while in use	↔	Protect behaviour with tests
Keep the bridge reliable over time	↔	Enable continuous delivery

THE BIG PICTURE

> Great engineers don't wait for failure. They continuously improve systems before failure occurs.

REMEMBER
Good structure today is the foundation for innovation tomorrow.

THINK AHEAD

We can refactor safely because we trust our tests.

KEY QUESTION FOR NEXT LECTURE

How do we gain that confidence?

→ **NEXT LECTURE: SOFTWARE TESTING**

Key points

1 Software evolves

Change is inevitable. As software grows, its internal structure tends to degrade.

2 Maintenance is continuous

Maintenance is not a phase, it is an ongoing activity necessary to keep software healthy.

3 Code smells are early warnings

They indicate potential problems that can lead to higher costs and defects if ignored.

4 Refactoring improves the code

Refactoring changes the internal structure without changing external behaviour.

5 Refactor iteratively

Small, incremental changes reduce risk and are easier to review and integrate.

6 Tests enable safe change

A solid suite of automated tests gives confidence to refactor and evolve the system.

7 Use modern tooling

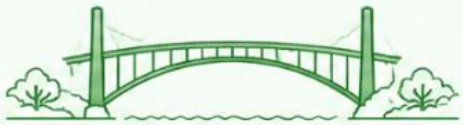
Automated smell detection, refactoring tools and quality gates help teams act earlier.

8 Quality is a team responsibility

Clean code, tests and reviews are everyone's responsibility, every day.

THE BIG TAKEAWAY

> Great engineers don't wait for failure.
> They continuously improve systems before failure occurs.



Just like bridges, software needs inspection, maintenance and incremental improvements to remain safe and reliable.

WHAT YOU CAN DO NOW

- Look for code smells in your projects.
- Refactor in small, safe steps.
- Run tests often and trust the feedback.
- Use tools and quality gates.
- Strive for continuous improvement.

NEXT LECTURE

SOFTWARE TESTING & AI-AUGMENTED TESTING

How do we gain confidence that our changes preserve the system's behaviour?

QUESTIONS?

Let's clarify, explore and learn together!

Concepts

Anything you didn't understand?

Applications

How do we apply these ideas?



Decisions

Want to discuss any trade-offs?

Next Steps

What would you like to learn next?

No question is too small!

Your questions help everyone learn and grow.

Ask freely, challenge ideas, and let's build understanding **together!**