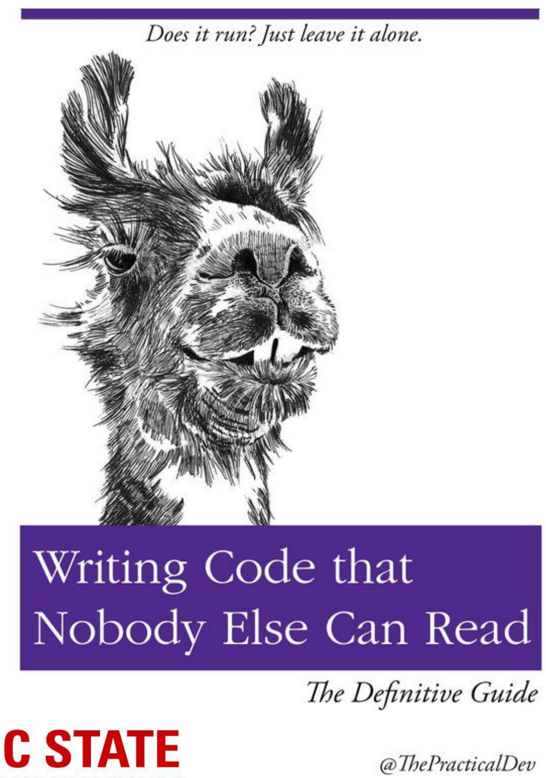
NC STATE UNIVERSITY

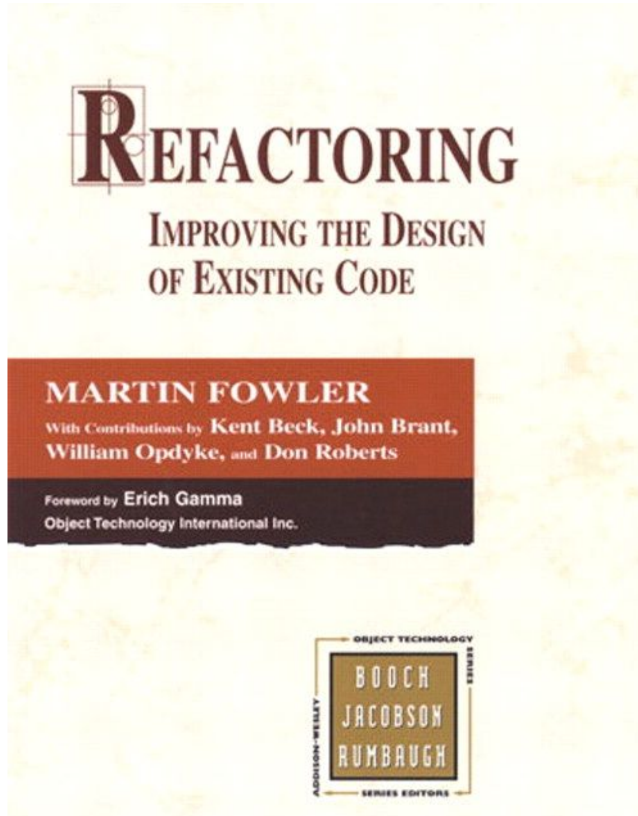
CSC 510
Software Engineering

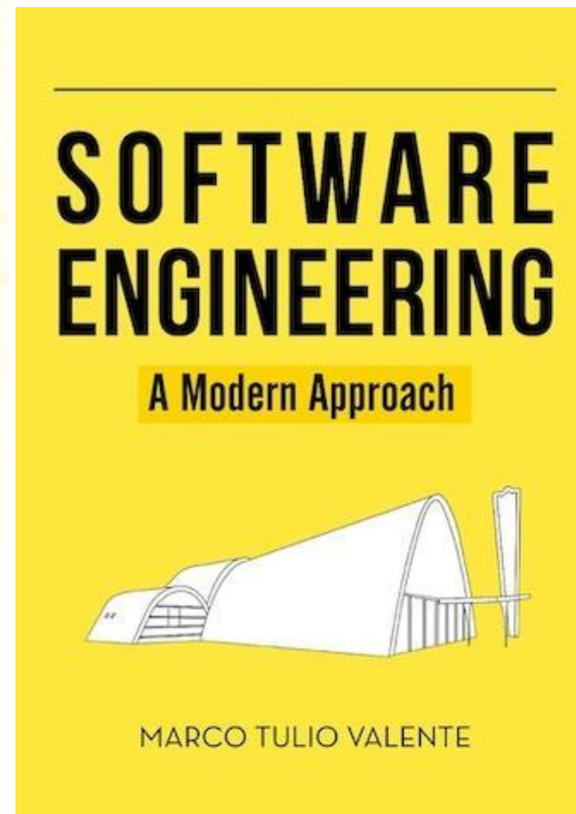
Refactoring

Katerina Vilkomir

Book Referenced







NC STATE
UNIVERSITY

Overview

- Introduction to Refactoring
- Origins of Refactoring
- Example of Refactorings
- Refactoring Practice
- Challenges for Refactoring
- Code Smells
- The Importance of Testing

Software Maintenance & Lehman’s Laws

- Software systems require maintenance: corrective, adaptive, and evolutionary.
- Like living organisms, software ages over time.
- Lehman’s Laws (1970s):
- 1. Systems must be continuously adapted.
- 2. Maintenance increases complexity unless action is taken.
- Without intervention, code quality deteriorates naturally.

Example:
A system with years of patches becomes fragile, even if it still 'works.'

NC STATE UNIVERSITY

What is refactoring?

"Code refactoring is the process of restructuring existing computer code – changing the factoring – without changing its external behavior.", from the Wikipedia article on Code Refactoring

What is refactoring?

Refactoring = improving maintainability without changing behavior

Key goals:

- Improve modularity, design, readability
- Increase testability and simplify modification

Types of transformations:

- Renaming, splitting methods, moving functions/classes

Behavior must be preserved!
System should work the same after refactoring.

NC STATE UNIVERSITY

What isn’t refactoring?

- Adding new functionality to a program
- Fixing bugs
- Is changing a program to use safer libraries a refactoring? Why or why not?

Origins of refactoring

- Unclear where the term first came from
- Also unclear when it first started — to some extent, people have been refactoring for as long as they have been programming
- Two researchers focused on this, made it prominent:
  - Griswold and Notkin (University of Washington)
  - Opdyke and Johnson (University of Illinois at Urbana-Champaign)

Origin 1: Griswold and Notkin

- Motivating problem: how do we modify deployed applications?
  - Initial solution allowed functionality to be added to extend app
  - Results in a different architecture than what would have been created if extensions known in advance (why?)
- Question: can a program be restructured in a way that changes the design but doesn't change the behavior?
- Result was the Restructure tool for refactoring Scheme

Restructure: Some points

- Restructure worked over Program Dependence Graphs (PDGs), which encode control and data flow dependencies
  - Control flow: loops, conditionals, etc
  - Data flow: how data flows through program statements, which data reaches what places
- Guarded transformations: must meet preconditions first
- Refactoring as graph transformation

Origin 2: Opdyke and Johnson

- Motivating problem: how do we evolve real software?
- Opdyke wanted a "macro" focus, Johnson had a "micro" focus
- This resulted in evolution (macro) through a sequence of small (micro) changes
- One big question: how do we reuse abstractions that are tangled with specifics of the application domain?
- Approach: mine actual changes from existing software

Origin 2: Some points

- Decided to focus on C++, not Smalltalk (why?)
- Came up with a collection of transformations, some bigger and some very focused
- Investigated safe application: refactoring should not change meaning of code (deciding which to apply still being investigated)
- Approach based on code equivalence, cannot change meaning — not decidable in general, but often is in real systems
- Equivalence based on invariants of transformation

Differences in approaches

- Griswold: focused on the semantics, what it means for a refactoring to be meaning-preserving
- Opdyke: focused more on OO aspects and real-world systems, e.g., pushing methods up and down the inheritance hierarchy
- Many other tools and systems for this now, with support for lots of languages and some fairly involved refactoring scripts

History of Refactoring

- Popularized by Extreme Programming (XP) in 1999.
- Martin Fowler’s 2000 book created a refactoring vocabulary:
  - Dozens of common refactorings
  - Benefits, examples, and trade-offs
- Kent Beck: 'I’m not a great programmer; I’m just a good programmer with great habits.'

Extract Method

Before Refactoring:

```python
def process_order(order):
    print("Processing order")
    print(f"Customer: {order['customer']}")
    print(f"Items: {order['items']}")
    print(f"Total: ${order['total']}")
```

After Refactoring:

```python
def print_order_details(order):
    print(f"Customer: {order['customer']}")
    print(f"Items: {order['items']}")
    print(f"Total: ${order['total']}")

def process_order(order):
    print("Processing order")
    print_order_details(order)
```

This separates the logic of printing order details into its own method. This improves **readability**, supports **code reuse**, and makes the code easier to **test and maintain**.

NC STATE UNIVERSITY

Motivations for Extract Method

Study of GitHub projects showed top motivations:

- Code reuse (43 cases)
- Alternative signature (25)
- Improve understanding (21)
- Remove duplication (15)
- Facilitate features or bug fixes (14)

Quote: 'I always try to reuse code... duplication complicates maintenance.'

“Make one change in one place.”

Inline Method

- Opposite of Extract Method.
- Replace a call to a short method with its body.
- Used when a method is very short and rarely reused.

Use when method adds no meaningful abstraction.

NC STATE UNIVERSITY

Inline Method

Before Refactoring:

```python
def calculate_tax(price):
    return price * 0.08

def checkout(price):
    tax = calculate_tax(price)
    return price + tax

```

The method `calculate_tax` was small and only used once. Inlining makes the code more **straightforward** when the method doesn't add meaningful abstraction.

After Refactoring:

```python
def checkout(price):
    tax = price * 0.08  # inlined the method
    return price + tax
```

Move Method

- Move a method to the class it most logically belongs to.

- Improves cohesion and reduces coupling.

- If the method accesses more of another class's data, it probably belongs there.

Move method 'f' from Class A to Class B

Before:
A.f() { ... uses B }

After:
B.f() { ... }

Move Method

Before Refactoring:

```python
class ShoppingCart:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return sum(item.price for item in self.items)
```

After Refactoring:

```python
class ShoppingCart:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return sum(item.get_price() for item in self.items)

class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_price(self):
        return self.price
```

`calculate_total` relied on the data from `Item`, so moving price access logic into the `Item` class improves **cohesion** and makes each class responsible for its own behavior.

Extract Class

- When a class has too many responsibilities, extract related fields and methods.

Example:

- Person class had 4 phone fields.
- Extracted new class Phone; Person now uses Phone objects.

Improves modularity and reuse.

NC STATE UNIVERSITY

Extract Class

Before Refactoring:

```python
class Person:
    def __init__(self, name, home_phone, mobile_phone):
        self.name = name
        self.home_phone = home_phone
        self.mobile_phone = mobile_phone
```

After Refactoring:

```python
class Phone:
    def __init__(self, home, mobile):
        self.home = home
        self.mobile = mobile

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

The `Person` class had too many responsibilities. Extracting the phone info into a Phone class improves **modularity**, reduces **duplication**, and supports **reuse**.

Rename

- Rename elements (methods, classes, variables) to more meaningful names.

- Helps readability and maintainability.

- Keep deprecated version of old method to allow gradual migration.

"Naming is one of the two hard things in CS."

NC STATE UNIVERSITY

Rename

Before Refactoring:

```python
def f():
    return "Hello"
```

After Refactoring:

```python
def greet():
    return "Hello"
```

Renaming `f` to `greet` improves **readability** and helps other developers understand the method’s purpose. Good naming is essential for long-term **maintainability**.

Other Useful Refactorings

- Extract Variable – simplify expressions (e.g., extract delta).
- Remove Flag – use return/break instead of boolean flags.
- Replace Conditional with Polymorphism – move logic to subclasses.
- Remove Dead Code – eliminate unused code to improve clarity and reduce risk.

Simple changes → big readability improvements.

Extract Variable

Before Refactoring:

```
total_price = price * (1 + tax_rate)
```

After Refactoring:

```
tax = price * tax_rate
total_price = price + tax
```

Extracting `tax` improves **clarity** and helps when debugging or modifying tax logic.

Remove flag

Before Refactoring:

```python
found = False
for item in items:
    if item == target:
        found = True
        break
```

After Refactoring:

```python
if target in items:
    print("Found!")
```

Simplifies logic by eliminating unnecessary flags and using built-in Python capabilities.

NC STATE
UNIVERSITY

Replace Conditional with Polymorphism

Before Refactoring:

```python
def get_discount(customer_type):
    if customer_type == "Student":
        return 0.10
    elif customer_type == "Senior":
        return 0.15
    else:
        return 0.0
```

Instead of using conditionals to determine behavior, polymorphism delegates it to subclasses. This makes the system easier to **extend**, test, and **modify** without modifying existing code.

After Refactoring:

```python
class Customer:
    def get_discount(self):
        return 0.0

class Student(Customer):
    def get_discount(self):
        return 0.10

class Senior(Customer):
    def get_discount(self):
        return 0.15
```

Remove Dead Code

Before Refactoring:

```python
def calculate_area(radius):
    pi = 3.14159
    unused_variable = 42  # <- dead code
    result = pi * radius * radius
    return result

def unused_function():
    print("This function is never called.")  # <- dead code
```

After Refactoring:

```python
def calculate_area(radius):
    pi = 3.14159
    return pi * radius * radius
```

`unused_variable` and `unused_function()` were never used or called.

Removing such code improves **clarity**, reduces **cognitive load**, and minimizes the **risk of confusion or bugs** later.

Keeping dead code can mislead future developers or waste time during maintenance.

NC STATE UNIVERSITY

Example refactorings

- Extract Method: https://sourcemaking.com/refactoring/extract-method
- Rename Method: https://sourcemaking.com/refactoring/rename-method
- Move Method: https://sourcemaking.com/refactoring/move-method
- Replace Temp with Query: https://sourcemaking.com/refactoring/replace-temp-with-query
- Replace Type Code with State/Strategy: https://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Conditional with Polymorphism: https://sourcemaking.com/refactoring/replace-conditional-with-polymorphism
- Self Encapsulate Field: https://sourcemaking.com/refactoring/self-encapsulate-field

Refactoring Practice: The importance of testing

- Why is having automated tests important?

NC STATE
UNIVERSITY

The importance of testing

- Remember back to concept of regression testing, what was this?

NC STATE UNIVERSITY

The importance of testing

- Remember back to our concept of regression testing, what was this?
    - We need a way to ensure our changes don't break anything…
    - …and we will be making lots of changes!
- Automated testing gives us confidence that we aren't breaking anything when we change the code

Testing: Foundation for Refactoring

- Unit tests are critical for safe refactoring.
- Refactorings don't add features — only modify structure.
- Without tests, changes are risky and often avoided.
- Refactoring should be safe and confidence-driven.
- John Ousterhout: Refactoring without tests leads to design decay.
- 'Make the change easy… then make the easy change.' – Kent Beck

> “Without tests, major structural changes are dangerous.”

NC STATE UNIVERSITY

Testing and refactoring

- What do you do if you are working on an existing system that doesn’t have tests?
- When should you add tests? What approaches can you use?

Refactoring Practice

- Opportunistic Refactoring
  - Done during regular development tasks (fixing bugs or adding features).
  - Triggered by code smells: long methods, bad names, duplication, etc.
  - Invest ~20% of task time to improve code while working.
- Strategic Refactoring
  - Planned, larger-scale refactoring efforts.
  - Examples: splitting large packages, reorganizing modules.
  - Scheduled when technical debt has grown over time.
  - Essential when multiple legacy issues accumulate.

The two challenges

- What two challenges do you see with actually doing refactoring?
- (Helpful hint, they were mentioned earlier in the talk…)

NC STATE
UNIVERSITY

The two challenges

- What two challenges do you see with actually doing refactoring?
- (Helpful hint, they were mentioned earlier in the talk…)
- Challenge 1: How do we know where to refactor (and can we automate this)?
- Challenge 2: How do we know a refactoring is safe (and can we automate this)?

Dealing with challenge 1: code smells

- A funny phrase for structures or patterns in the code that "suggest…the possibility of refactoring"
- Note: these are not bugs!
- They are structural problems in the code that could make it harder to ensure quality, evolve the system over time

— What? How can code "smell"??
— Well, it doesn't have a nose... but it definitely can stink!

How do we detect code smells?

- Manually: this is a matter of gaining experience
- Automatically: many tools now attempt to do this, but this isn’t an easy problem (why?)
- One example: FaultBuster
- Many others, if you go to https://scholar.google.com and search for “code smell detection” you will get lots of links (more than 55,000 as of February 2022)

NC STATE UNIVERSITY

Code Smells

- Indicators of low-quality code.
- Difficult to maintain, understand, modify, or test.
- Not every smell demands immediate refactoring.
- Decision to refactor depends on importance and frequency of changes.

NC STATE
UNIVERSITY

Duplicated Code

- Increases maintenance complexity and risk of inconsistent changes.

Types of clones:

1. Identical (except whitespace/comments)
2. Same logic, different variable names
3. Same logic, minor statement differences
4. Semantically equivalent, different implementations

Solutions: Extract Method, Extract Class, Pull Up Method.

Duplicated Code

Before:

```python
def calculate_invoice_total(invoice):
    total = 0
    for item in invoice['items']:
        total += item['price'] * item['quantity']
    return total

def print_invoice(invoice):
    total = 0
    for item in invoice['items']:
        total += item['price'] * item['quantity']
    print(f"Total: {total}")
```

After:

```python
def calculate_invoice_total(invoice):
    return sum(item['price'] * item['quantity'] for item in invoice['items'])

def print_invoice(invoice):
    print(f"Total: {calculate_invoice_total(invoice)}")
```

Refactor by Extract Method to remove repeated logic.

NC STATE
UNIVERSITY

Long Method & Large Class

Long Method:

- Hard to read and maintain.
- Solution: Extract Method.

Large Class (God Class/Blob):

- Multiple responsibilities, low cohesion.
- Solution: Extract Class.

Long Method

Before:

```python
def generate_report(data):
    print("Report Header")
    for item in data:
        print(f"{item['name']} - {item['value']}")
    print("End of Report")
    print("Summary:")
    total = sum(i['value'] for i in data)
    print(f"Total: {total}")
```

After:

```python
def print_header():
    print("Report Header")

def print_items(data):
    for item in data:
        print(f"{item['name']} - {item['value']}")

def print_summary(data):
    total = sum(i['value'] for i in data)
    print("Summary:")
    print(f"Total: {total}")

def generate_report(data):
    print_header()
    print_items(data)
    print("End of Report")
    print_summary(data)
```

Refactor by breaking into smaller methods for readability and reuse.

NC STATE UNIVERSITY

Large Class (God Class / Blob)

Before:

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

    def calculate_pay(self):
        return self.salary * self.hours

    def print_badge(self):
        print(f"Employee: {self.name}")
```

After:

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

    def calculate_pay(self):
        return self.salary * self.hours

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

    def print_badge(self):
        print(f"Employee: {self.name}")
```

Refactor by Extract Class
for separation of concerns.

Feature Envy & Long Parameters List

Feature Envy:

- Method heavily uses another class's data/methods.
- Solution: Move Method.

Long Parameters List:

- Method with excessive parameters.
- Solutions: obtain params internally, or group params into objects.

NC STATE
UNIVERSITY

Feature Envy & Long Parameters List

Before:

```python
def apply_discount(order, price, tax, customer_level):
    if customer_level == "gold":
        return price * 0.9 + tax
    return price + tax
```

After:

```python
#Encapsulate Logic in Order
class Order:
    def __init__(self, price, tax, customer_level):
        self.price = price
        self.tax = tax
        self.customer_level = customer_level

    def apply_discount(self):
        if self.customer_level == "gold":
            return self.price * 0.9 + self.tax
        return self.price + self.tax
```

Feature Envy: move logic to the class whose data is being used.

Long Parameters: use object instead of separate parameters.

NC STATE UNIVERSITY

Global Variables & Primitive Obsession

Global Variables:

- Difficult to track changes; cause unintended side-effects.
- Solution: Encapsulate state within classes.

Primitive Obsession:

- Overuse of basic types (int, String).
- Solution: Create dedicated classes (e.g., ZIPCode, Currency).

Global Variables & Primitive Obsession

Before:

```python
discount_rate = 0.1  # global
def apply_discount(price):
    return price - (price * discount_rate)
```

After:

```python
#Encapsulate State & Use Domain Types
class Discount:
    def __init__(self, rate):
        self.rate = rate

    def apply(self, price):
        return price - (price * self.rate)

discount = Discount(0.1)
price_after_discount = discount.apply(100)
```

Avoid global variables; use objects to hold state.

Replace **primitive obsession** (e.g., raw `float`) with domain-specific types.

NC STATE UNIVERSITY

Mutable Objects & Data Classes

Mutable Objects:

- Objects whose state changes post-creation.
- Prefer immutable objects to ensure thread-safety.

Data Classes:

- Only attributes, no significant behavior.
- Solution: Move relevant methods into these classes.

Mutable Objects & Data Classes

Before:

```python
class User:
    def __init__(self, name):
        self.name = name

user = User("Alice")
user.name = "Bob"  # Mutable object
```

After:

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class User:
    name: str

user = User("Alice")
# user.name = "Bob"  # Will raise an error
```

Use `@dataclass(frozen=True)` for immutability — safer for concurrency and testing.

NC STATE UNIVERSITY

Comments & Technical Debt

Comments:

- Often indicate unclear code.
- Solution: Refactor code for clarity, reduce need for comments.

Technical Debt:

- Technical issues that accumulate maintenance costs.
- Examples: lack of tests, poor architecture, unresolved smells.
- Paying off debt improves long-term maintainability.

Comments & Technical Debt

Before:

```
def calculate():
    # calculate square root
    result = x ** 0.5  # ← unclear variable, bad naming
```

After:

```
def calculate_square_root(number):
    return number ** 0.5
```

Clear code reduces the need for explanatory comments.

Paying off **technical debt** improves maintainability.

NC STATE UNIVERSITY

Examples of code smells

- Duplicate code: https://sourcemaking.com/refactoring/smells/duplicate-code
- Long method: https://sourcemaking.com/refactoring/smells/long-method
- Large class: https://sourcemaking.com/refactoring/smells/large-class
- **Divergent change**: https://sourcemaking.com/refactoring/smells/divergent-change
- **Shotgun surgery**: https://sourcemaking.com/refactoring/smells/shotgun-surgery
- Feature envy: https://sourcemaking.com/refactoring/smells/feature-envy

Dealing with challenge 2: program analysis

- Program analysis lets us encode rules about when a refactoring is safe (preconditions, invariants) and check these on the code

- Not easy! And some languages are harder than others

- For interesting examples, see work of Max Schäfer (http://dblp.uni-trier.de/pers/hd/s/Sch=auml=fer:Max) (e.g., “Sound and Extensible Renaming for Java”)

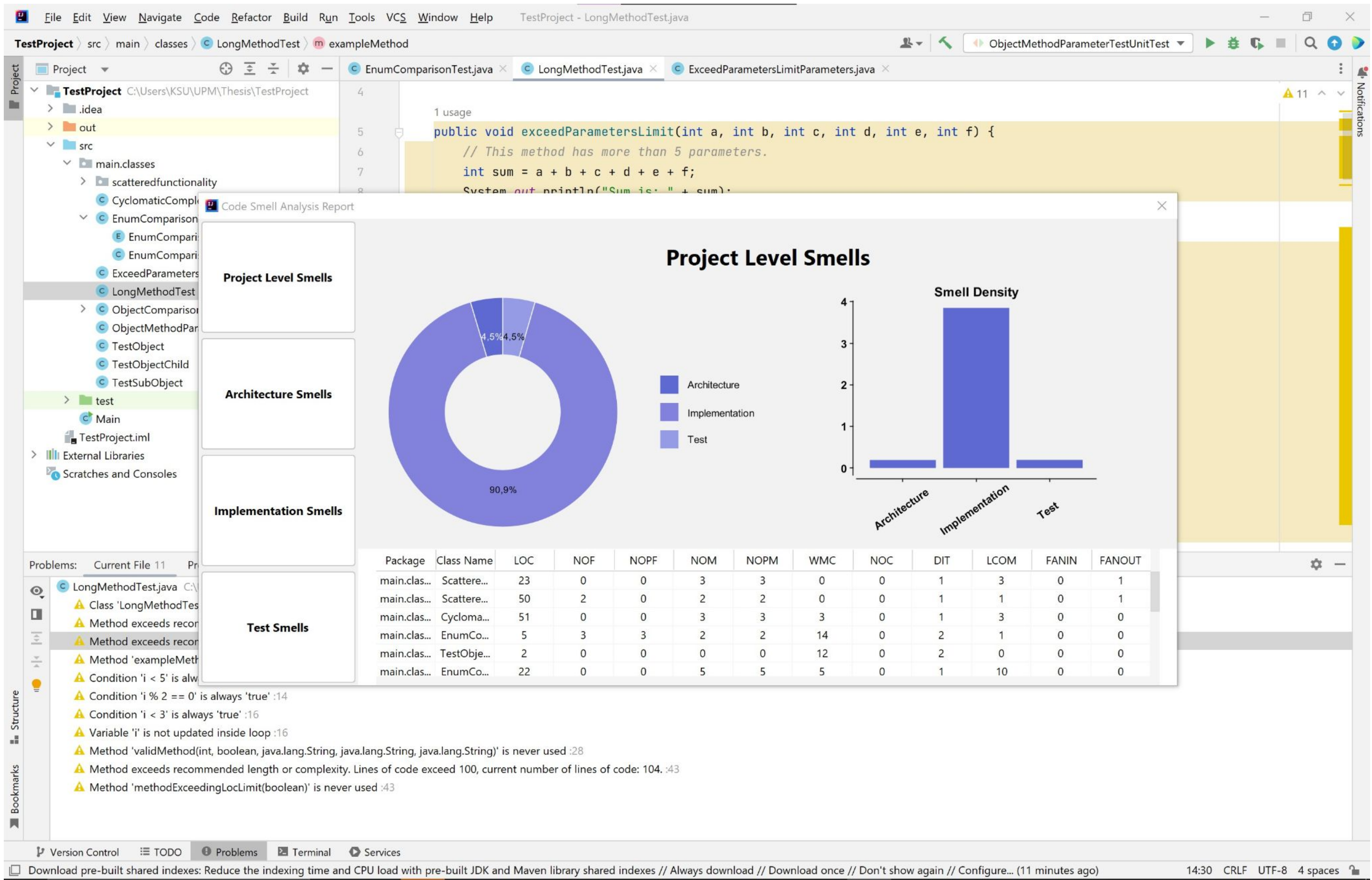
NC STATE UNIVERSITY

Checking Refactoring Preconditions

- **Tests exist and pass before and after refactoring.** Ensure behavior hasn't changed.
- **Behavior is stable and testable.** Avoid refactoring unknown or unpredictable code.
- **Code compiles or runs cleanly.** Don't refactor broken code unless intentionally fixing it.
- **Dependencies are understood.** Know what classes/functions use the code being changed.
- **No side effects or shared state.** Refactored code shouldn't rely on hidden mutable state.
- **Access and visibility rules are respected.** Ensure moved or renamed methods remain accessible where needed.
- **No name conflicts.** New names should be unique in the current scope.
- **Code is not duplicated after refactoring.** Prevent introducing repetition (e.g., from inlining).
- **Contracts and APIs are preserved.** External behavior, return types, and public interfaces must stay consistent.
- **Changes are isolated.** Refactor in small steps to avoid cascading errors.

IDE warnings prevent subtle errors by checking visibility & behavior.

Automated Refactoring in IDEs

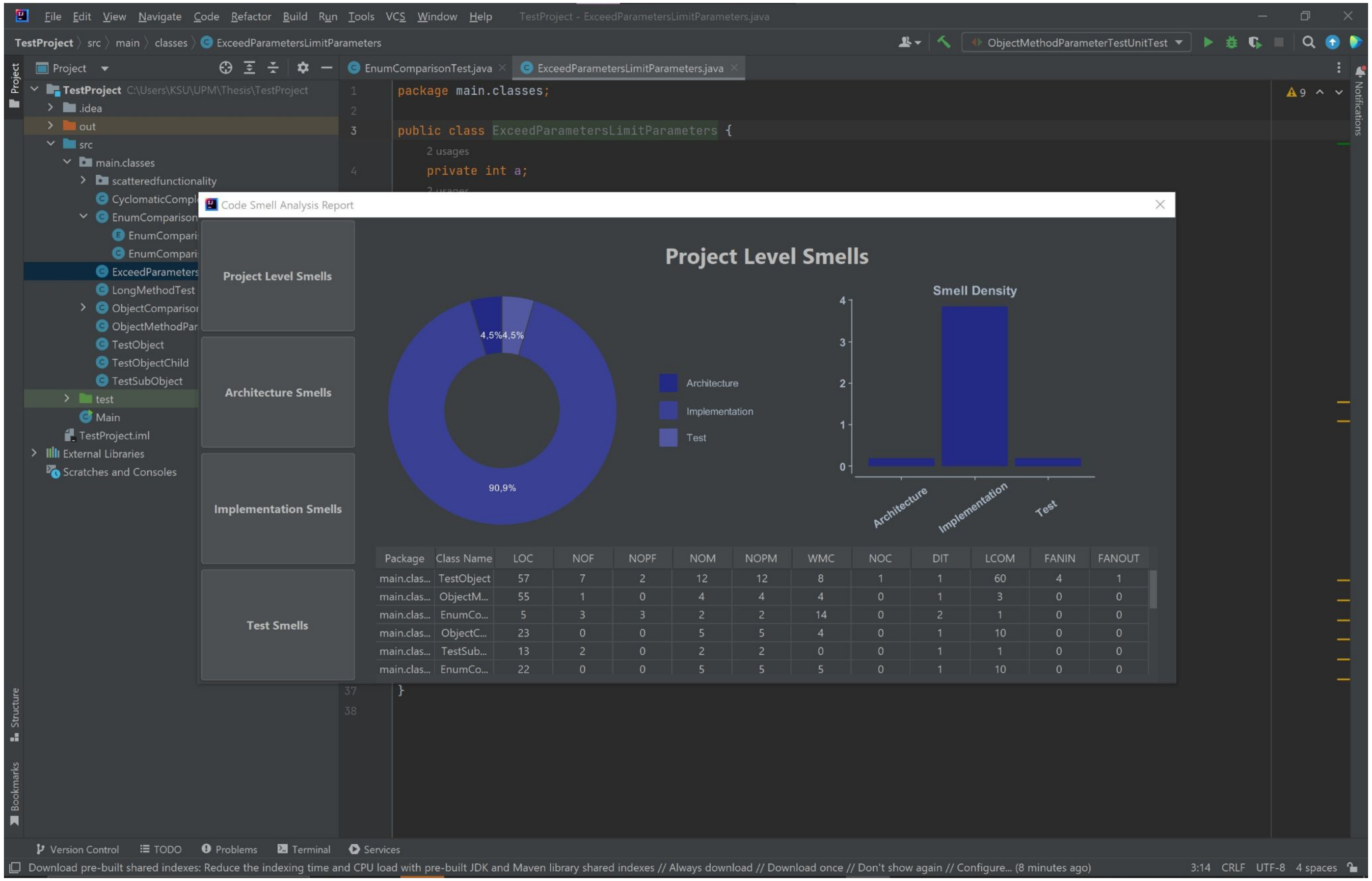


Picture source: https://plugins.jetbrains.com/plugin/24498-autorefactor

NC STATE UNIVERSITY

IDE helps automate refactoring, but developers remain in control.

Automated Refactoring in IDEs

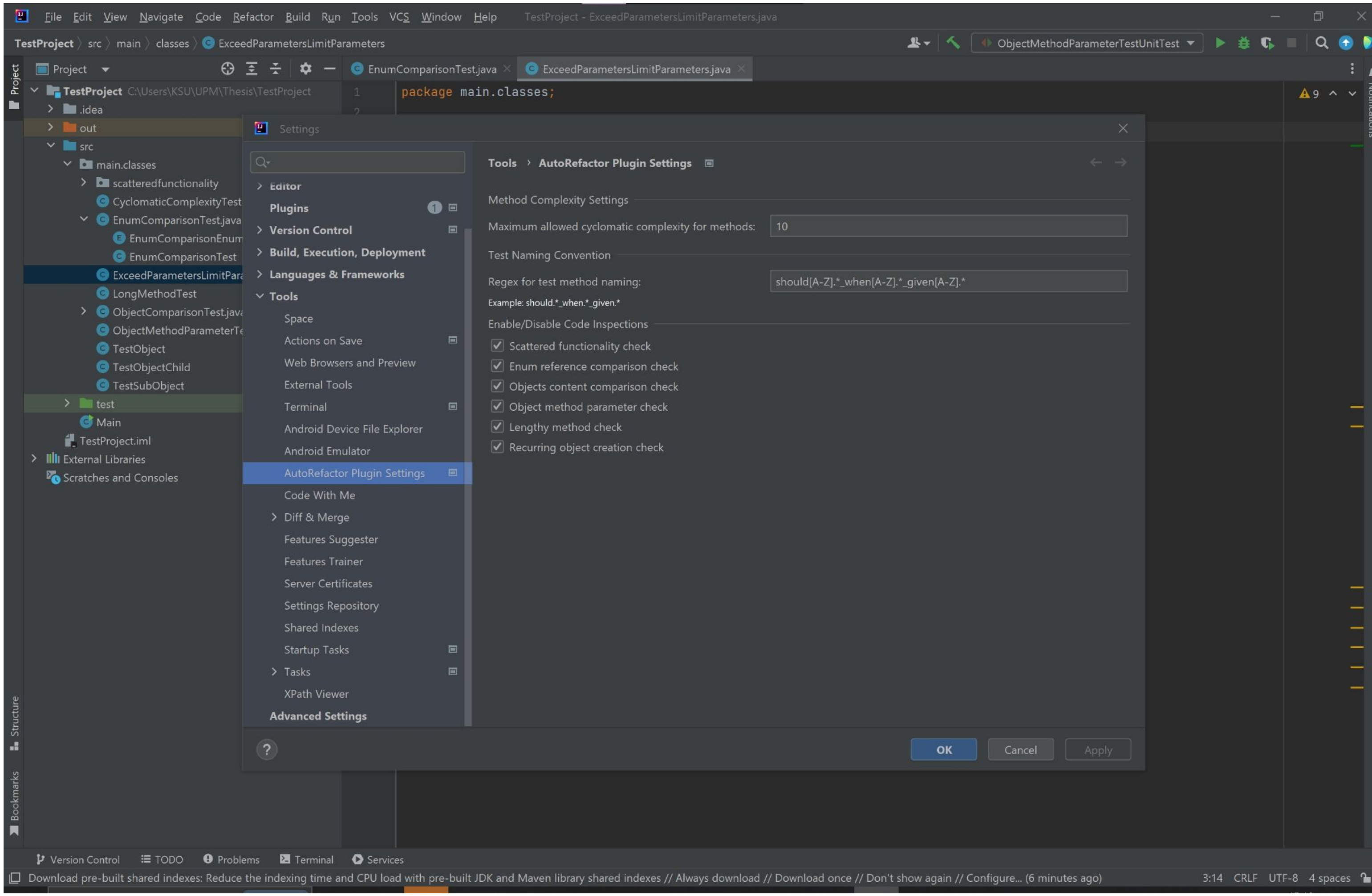


Picture source: https://plugins.jetbrains.com/plugin/24498-autorefactor

NC STATE UNIVERSITY

IDE helps automate refactoring, but developers remain in control.

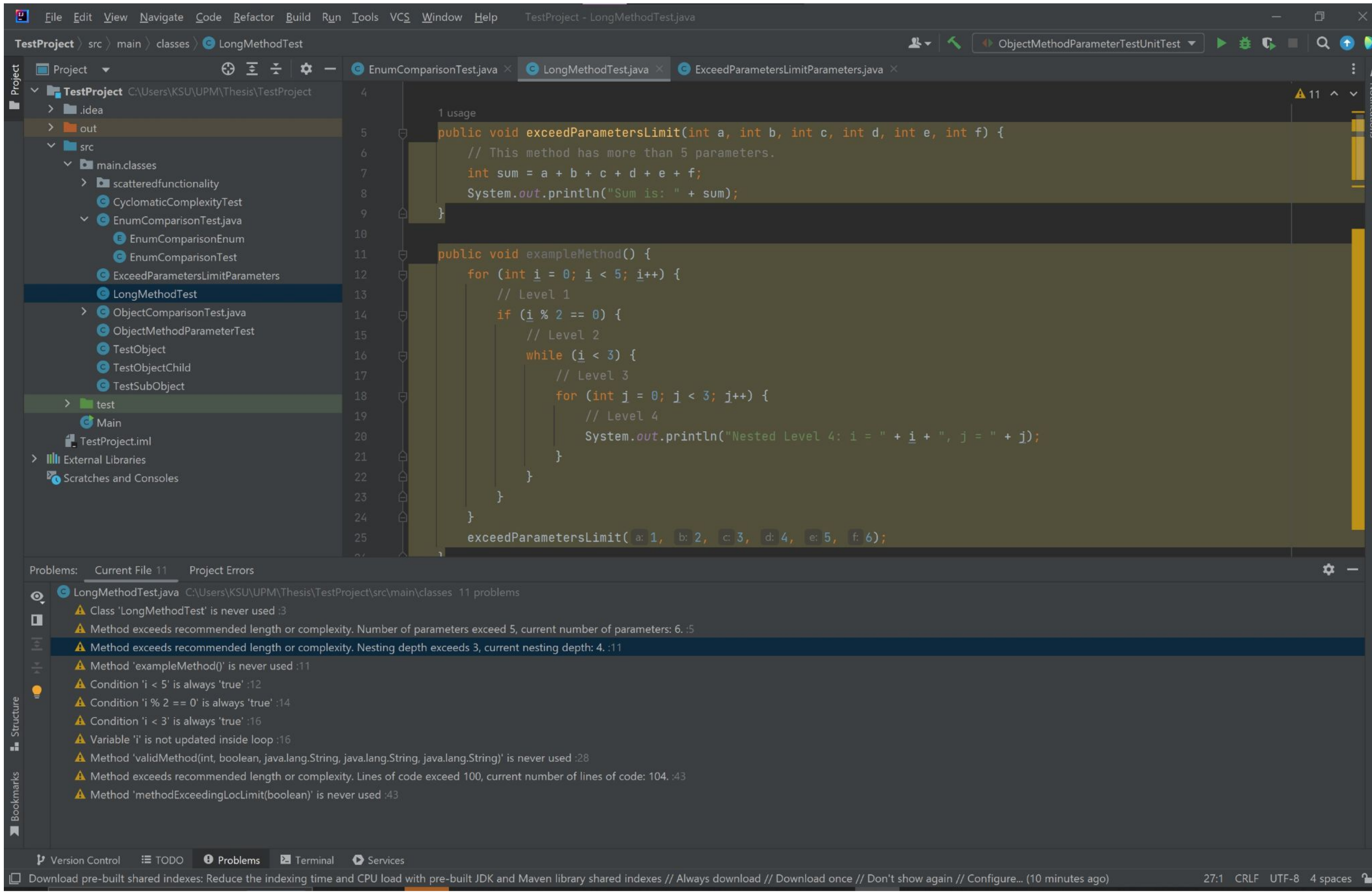
Automated Refactoring in IDEs



NC STATE UNIVERSITY

IDE helps automate refactoring, but developers remain in control.

Automated Refactoring in IDEs



Picture source: https://plugins.jetbrains.com/plugin/24498-autorefactor

NC STATE UNIVERSITY

IDE helps automate refactoring, but developers remain in control.

```
for u in range (0, 1000):
    print ("Thank you!")
```

NC STATE
UNIVERSITY