Preventive Change: Automated Refactoring

Original Project link:

The project I used for this assignment contains sample code with various code smells and refactoring examples:

https://github.com/Trina-SE/Software-Maintenance/tree/main/Automated_Refactoring/refactoring_demo

After Refactoring:

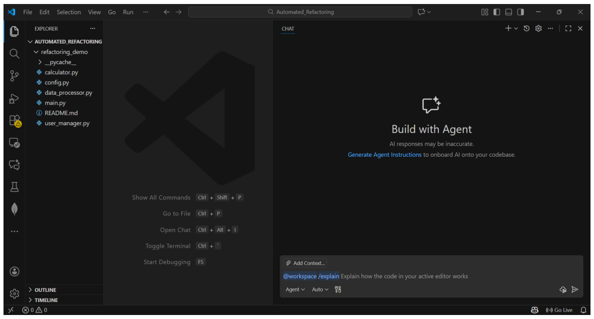
I followed automated refactoring tutorials, and the refactored version of the project is available here:

https://github.com/Trina-SE/Software-Maintenance/tree/main/Automated_Refactoring/After_Refactoring

Demonstration

- ### Understanding Code:

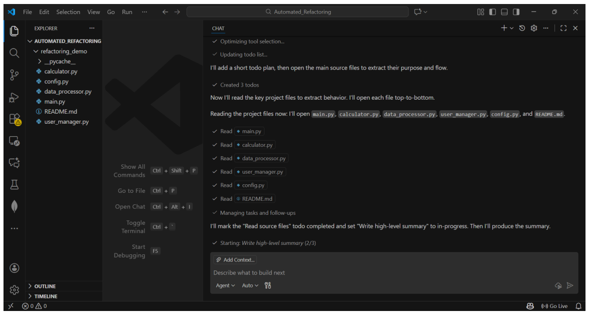
- Select the relevant code in Visual Studio Code
- Open inline chat and type: **/explain** and press **Enter**



AI responses may be inaccurate.

Generate Agent Instructions to onboard AI onto your codebase.

@workspace /explain Explain how the code in your active editor works



✓ Optimizing tool selection...

✓ Updating todo list...

I'll add a short todo plan, then open the main source files to extract their purpose and flow.

✓ Created 3 todos

Now I'll read the key project files to extract behavior. I'll open each file top-to-bottom.

Reading the project files now: I'll open `main.py`, `calculator.py`, `data_processor.py`, `user_manager.py`, `config.py`, and `README.md`.

✓ Read main.py

✓ Read calculator.py

✓ Read data_processor.py

✓ Read user_manager.py

✓ Read config.py

✓ Read README.md

✓ Managing tasks and follow-ups

I'll mark the "Read source files" todo completed and set "Write high-level summary" to in-progress. Then I'll produce the summary.

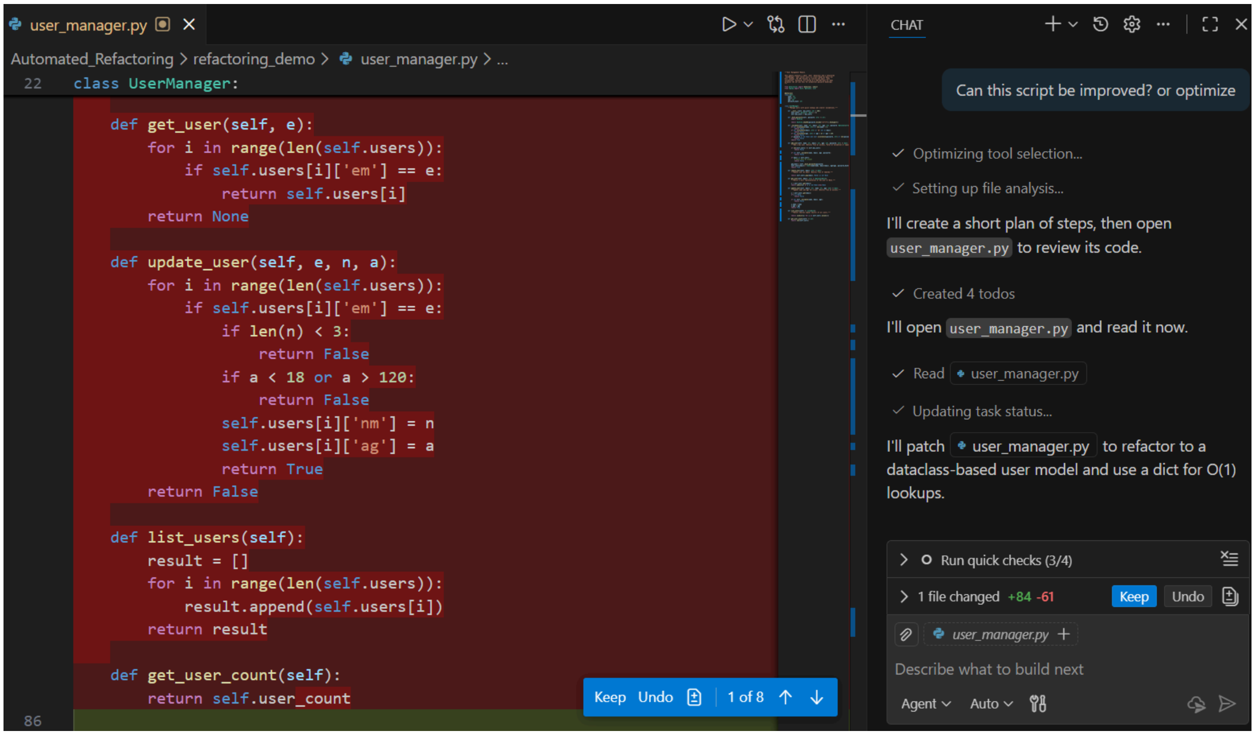
✓ Starting: *Write high-level summary* (2/3)

Describe what to build next

- ### Optimizing inefficient code:

- **Files Optimized:** data_processor.py, user_manager.py, calculator.py

- **Code Selection:** Select whole file
- **Type:** Can this script be improved? or optimize


user_manager.py
Automated_Refactoring > refactoring_demo > user_manager.py > ...
CHAT
Keep
Undo
1 of 8
Keep
Undo
user_manager.py
Describe what to build next
Agent
Auto

```
class UserManager:
    def get_user(self, e):
        for i in range(len(self.users)):
            if self.users[i]['em'] == e:
                return self.users[i]
        return None

    def update_user(self, e, n, a):
        for i in range(len(self.users)):
            if self.users[i]['em'] == e:
                if len(n) < 3:
                    return False
                if a < 18 or a > 120:
                    return False
                self.users[i]['nm'] = n
                self.users[i]['ag'] = a
                return True
        return False

    def list_users(self):
        result = []
        for i in range(len(self.users)):
            result.append(self.users[i])
        return result

    def get_user_count(self):
        return self.user_count
```

Can this script be improved? or optimize

✓ Optimizing tool selection...

✓ Setting up file analysis...

I'll create a short plan of steps, then open `user_manager.py` to review its code.

✓ Created 4 todos

I'll open `user_manager.py` and read it now.

✓ Read user_manager.py

✓ Updating task status...

I'll patch user_manager.py to refactor to a dataclass-based user model and use a dict for O(1) lookups.

○ Run quick checks (3/4)

1 file changed +84 -61

- **Problems Solved:**

1. **Inefficient search in user_manager.py**

Before Refactoring: get_user() used index loop with manual search:

```
for i in range(len(self.users)):
    if self.users[i]['em'] == e:
        return self.users[i]
```

After Refactoring: Replaced with generator expression: next((u for u in self.users if u["email"] == email), None)

2. **Memory inefficiency in data_processor.py**

Before Refactoring: calc_stats() converted numbers to list inline every time
After Refactoring: Converted once, then computed all stats

3. **Redundant state in user_manager.py**

Before Refactoring: Maintained user_count separately from len(self.users)
After Refactoring: Removed user_count, used len(self.users) directly

4. **Shipping cost calculation inefficiency in calculator.py**

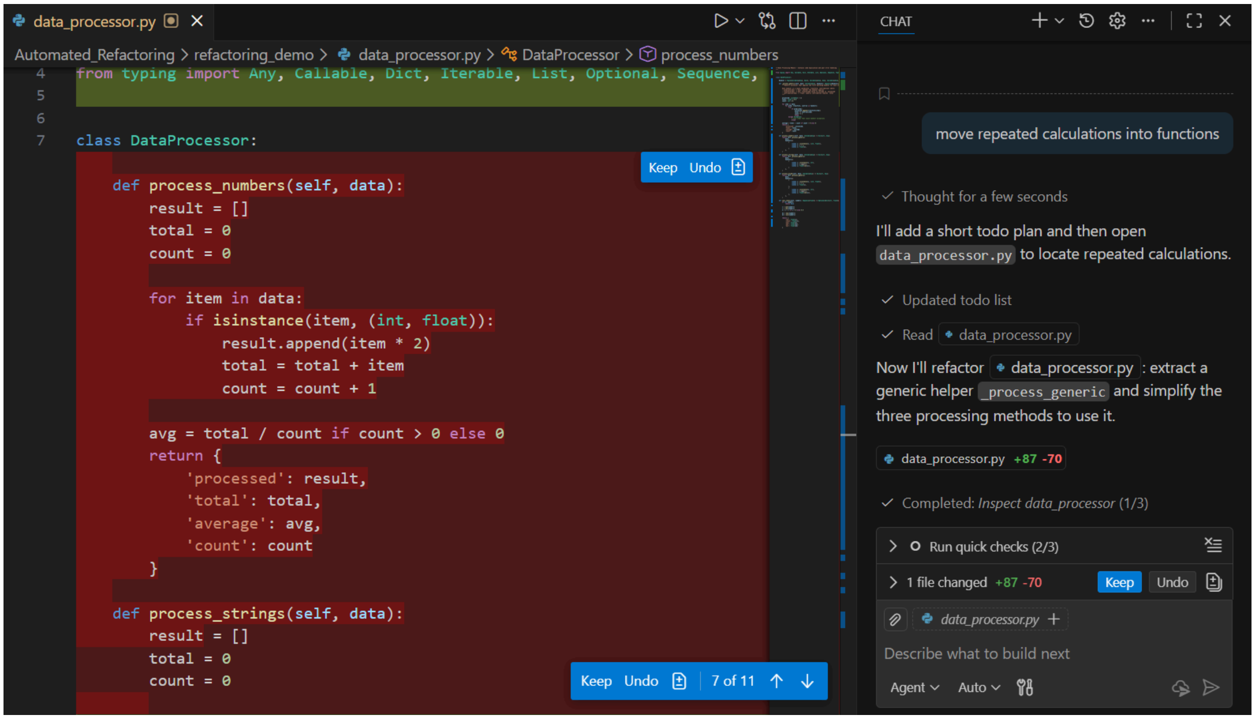
Before Refactoring: Nested if/elif chain for weight brackets (checked all conditions each time)
After Refactoring: Created SHIPPING_BRACKETS list and performed linear search for match

- **Performance Improvements:**

✓ Reduced function calls: 3 identical aggregations to 1 generic method
✓ Fewer iterations: One loop instead of multiple
✓ Memory reduction: Removed redundant user_count tracking
✓ Optimized search: Better use of Python's built-in functions

- **Cleaning up repeated code:**

- **Files Refactored:** data_processor.py, user_manager.py, calculator.py
- **Code Selection:** Select whole file
- **Type:** move repeated calculations into functions


data_processor.py
Automated_Refactoring > refactoring_demo > data_processor.py > DataProcessor > process_numbers
Keep Undo
Keep Undo 7 of 11
CHAT

```
from typing import Any, Callable, Dict, Iterable, List, Optional, Sequence,


class DataProcessor:

    def process_numbers(self, data):
        result = []
        total = 0
        count = 0

        for item in data:
            if isinstance(item, (int, float)):
                result.append(item * 2)
                total = total + item
                count = count + 1

        avg = total / count if count > 0 else 0
        return {
            'processed': result,
            'total': total,
            'average': avg,
            'count': count
        }

    def process_strings(self, data):
        result = []
        total = 0
        count = 0
```

move repeated calculations into functions

✓ Thought for a few seconds

I'll add a short todo plan and then open `data_processor.py` to locate repeated calculations.

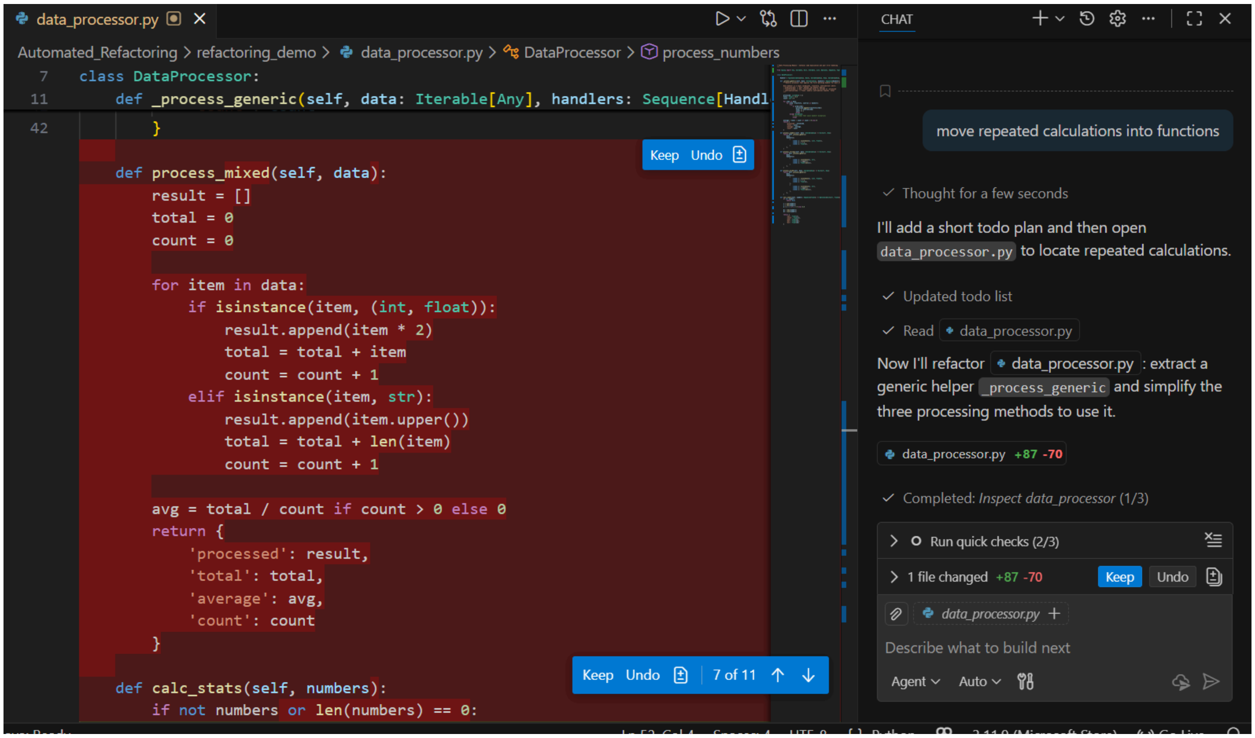
✓ Updated todo list

✓ Read data_processor.py

Now I'll refactor data_processor.py : extract a generic helper `_process_generic` and simplify the three processing methods to use it.

data_processor.py +87 -70

✓ Completed: *Inspect data_processor* (1/3)



data_processor.py
Automated_Refactoring > refactoring_demo > data_processor.py > DataProcessor > process_numbers
Keep
Undo
Keep
Undo
7 of 11

```
class DataProcessor:
    def _process_generic(self, data: Iterable[Any], handlers: Sequence[Handl
        }

    def process_mixed(self, data):
        result = []
        total = 0
        count = 0

        for item in data:
            if isinstance(item, (int, float)):
                result.append(item * 2)
                total = total + item
                count = count + 1
            elif isinstance(item, str):
                result.append(item.upper())
                total = total + len(item)
                count = count + 1

        avg = total / count if count > 0 else 0
        return {
            'processed': result,
            'total': total,
            'average': avg,
            'count': count
        }

    def calc_stats(self, numbers):
        if not numbers or len(numbers) == 0:
```

move repeated calculations into functions

✓ Thought for a few seconds

I'll add a short todo plan and then open `data_processor.py` to locate repeated calculations.

✓ Updated todo list

✓ Read data_processor.py

Now I'll refactor data_processor.py : extract a generic helper `_process_generic` and simplify the three processing methods to use it.

data_processor.py +87 -70

✓ Completed: *Inspect data_processor* (1/3)

○ Run quick checks (2/3)

1 file changed +87 -70

- Problems Solved:

1. Aggregation logic duplication (data_processor.py)

Before Refactoring: process_numbers, process_strings, process_mixed all had identical aggregation:

- Initialize result[], total=0, count=0
- Loop through items
- Append to result, add to total, increment count
- Calculate avg = total / count
- Return dict with processed, total, average, count

After Refactoring: Extracted _process_with_rules() helper method

2. Validation logic duplication (user_manager.py)

Before Refactoring: add_user() and update_user() both had identical validation:

- Check name length >= 3
- Check email has '@'
- Check age 18–120

- Check password length >= 6

After Refactoring: Extracted UserValidator class with validate_* methods

3. Discount calculation duplication (calculator.py)

Before Refactoring: calc_discount() had if/elif chain, process_order() had identical logic duplicated:

- if dt == 'A': discount = subtotal * 0.1
- elif dt == 'B': discount = subtotal * 0.15

After Refactoring: process_order() now calls calc_discount() instead of duplicating

4. Tax calculation duplication (calculator.py)

Before Refactoring: calc_tax() method exists but process_order() recalculated tax inline
After Refactoring: process_order() now calls calc_tax()

5. Shipping calculation duplication (calculator.py)

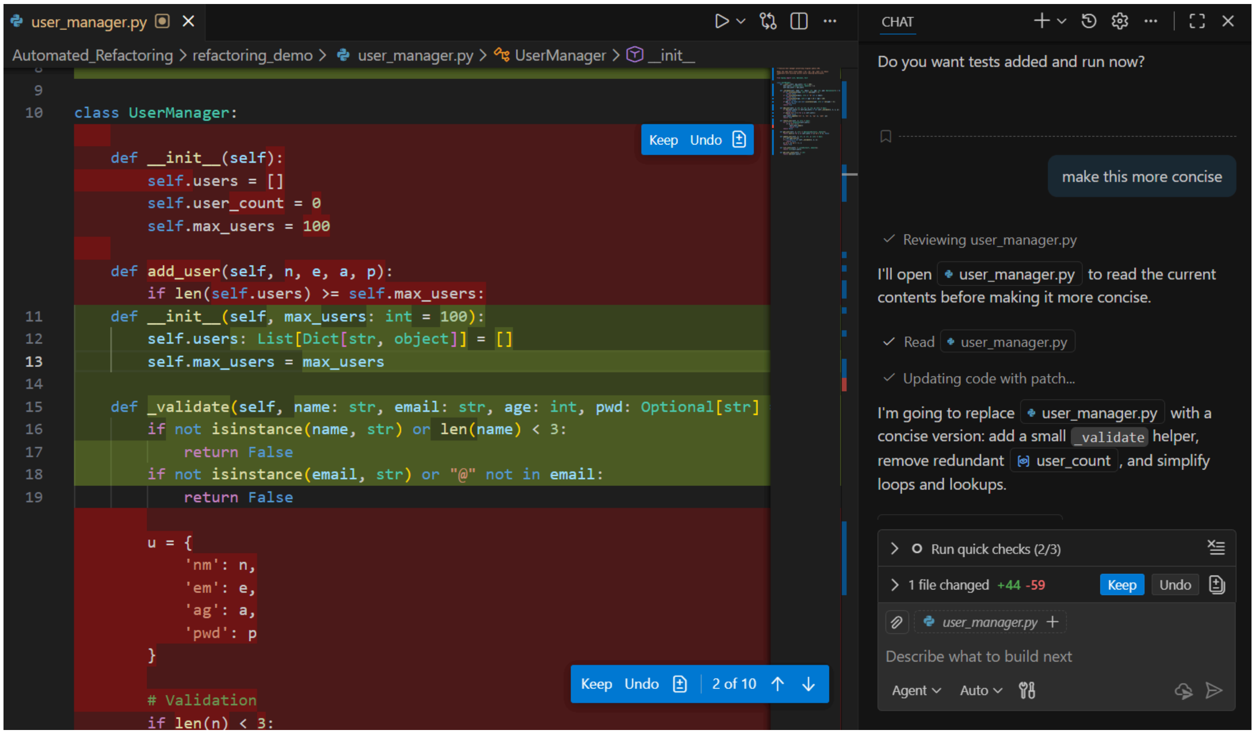
Before Refactoring: calc_shipping() method exists but process_order() had duplicate shipping logic
After Refactoring: process_order() now calls calc_shipping()

- **Code Reduced:**

✓ data_processor.py: ~90 lines → ~45 lines (50% reduction)
✓ user_manager.py: ~70 lines → ~50 lines (30% reduction)
✓ calculator.py: Eliminated ~20 lines of duplicate logic

Making code more concise:

- **Files Refactored:** user_manager.py, main.py
- **Code Selection:** Select whole file
- **Type:** make this more concise


user_manager.py
Automated_Refactoring > refactoring_demo > user_manager.py > UserManager > __init__
Keep
Undo
2 of 10
CHAT

```
class UserManager:

    def __init__(self):
        self.users = []
        self.user_count = 0
        self.max_users = 100

    def add_user(self, n, e, a, p):
        if len(self.users) >= self.max_users:
    def __init__(self, max_users: int = 100):
        self.users: List[Dict[str, object]] = []
        self.max_users = max_users

    def _validate(self, name: str, email: str, age: int, pwd: Optional[str]
        if not isinstance(name, str) or len(name) < 3:
            return False
        if not isinstance(email, str) or "@" not in email:
            return False

        u = {
            'nm': n,
            'em': e,
            'ag': a,
            'pwd': p
        }

        # Validation
        if len(n) < 3:
```

Do you want tests added and run now?

make this more concise

✓ Reviewing user_manager.py

I'll open user_manager.py to read the current contents before making it more concise.

✓ Read user_manager.py

✓ Updating code with patch...

I'm going to replace user_manager.py with a concise version: add a small `_validate` helper, remove redundant user_count, and simplify loops and lookups.

> ○ Run quick checks (2/3)

> 1 file changed +44 -59 Keep Undo

user_manager.py +

Describe what to build next

Agent ⌄ Auto ⌄

- **Problems Solved:**

1. **Verbose list creation in user_manager.py**

Before Refactoring: list_users() used manual loop:

```
result = []
for i in range(len(self.users)):
    result.append(self.users[i])
return result
```

After Refactoring: return self.users.copy()

2. **Verbose removal in user_manager.py**

Before Refactoring: remove_user() used index loop:

```
for i in range(len(self.users)):
    if self.users[i]['em'] == e:
        self.users.pop(i)
```

After Refactoring: Find user, then use list.remove()

3. **Manual comparison in user_manager.py**

Before Refactoring: user_count maintenance: self.user_count = self.user_count + 1
After Refactoring: Removed, use len(self.users) instead

4. Verbose output in main.py

Before Refactoring: main() mixed business logic with print statements
After Refactoring: Extracted to demo_* functions with shared print_header()

5. Redundant string concatenation

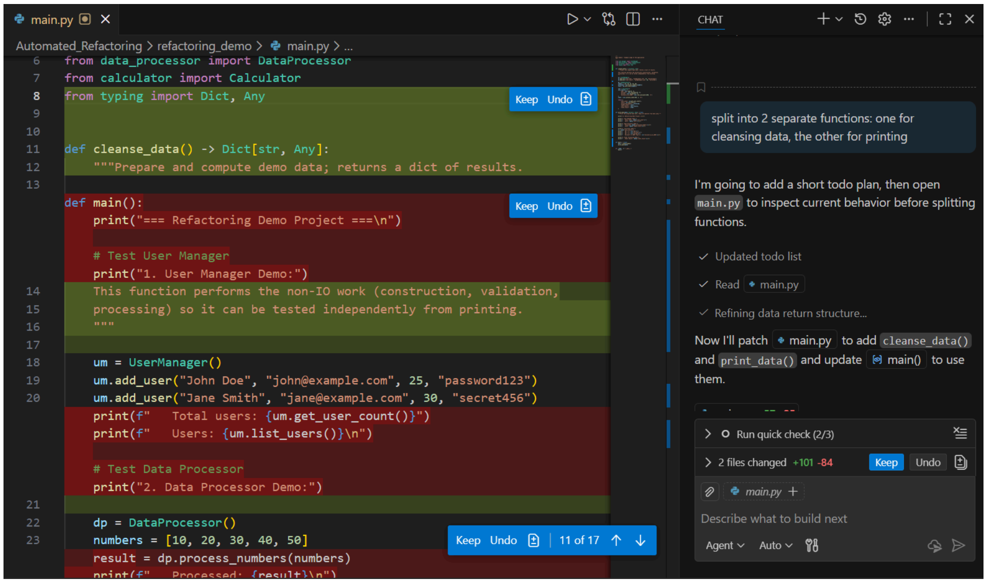
Before Refactoring: Multiple string building operations
After Refactoring: Used f-strings consistently

- Overall Conciseness Improvements:

✓ Total lines reduced: ~280 lines → ~200 lines (30% reduction)
✓ Code more Pythonic: Uses generators, dict.get(), next(), list comprehensions
✓ Easier to read: Intent is clearer with less boilerplate

• Split complex functions

- **Files Refactored:** main.py, calculator.py
- **Code Selection:** Place cursor on function name
- **Type:** split into 2 separate functions: one for cleansing data, the other for printing



```
from data_processor import DataProcessor
from calculator import Calculator
from typing import Dict, Any


def cleanse_data() -> Dict[str, Any]:
    """Prepare and compute demo data; returns a dict of results.

def main():
    print("=== Refactoring Demo Project ===\n")

    # Test User Manager
    print("1. User Manager Demo:")
    This function performs the non-IO work (construction, validation,
    processing) so it can be tested independently from printing.
    """

    um = UserManager()
    um.add_user("John Doe", "john@example.com", 25, "password123")
    um.add_user("Jane Smith", "jane@example.com", 30, "secret456")
    print(f"   Total users: {um.get_user_count()}")
    print(f"   Users: {um.list_users()}\n")

    # Test Data Processor
    print("2. Data Processor Demo:")

    dp = DataProcessor()
    numbers = [10, 20, 30, 40, 50]
    result = dp.process_numbers(numbers)
    print(f"   Processed: {result}\n")
```

split into 2 separate functions: one for cleansing data, the other for printing

I'm going to add a short todo plan, then open `main.py` to inspect current behavior before splitting functions.

✓ Updated todo list
✓ Read main.py
✓ Refining data return structure...

Now I'll patch main.py to add `cleanse_data()` and `print_data()` and update main() to use them.

- **Problems Solved:**

1. Monolithic main() function

Before Refactoring: main() had 30 lines doing multiple things:

- Create instances (initialization)
- Run UserManager demo (demo logic)
- Run DataProcessor demo (demo logic)
- Run Calculator demo (demo logic)
- Run Order Processing demo (demo logic)
- Mixed output formatting with business logic

After Refactoring: Split into:

- main(): Orchestrates high-level flow
- demo_user_manager(um): Focused on UserManager demo
- demo_data_processor(dp): Focused on DataProcessor demo
- demo_calculator(calc): Focused on Calculator demo
- demo_order_processing(calc): Focused on Order demo
- print_header(title): Reusable output formatting

2. Monolithic process_order() function

Before Refactoring: process_order() had 40 lines doing multiple things:

- Validate inputs
- Calculate subtotal
- Calculate discount (duplicated from calc_discount!)
- Calculate tax (duplicated from calc_tax!)
- Calculate weight (custom logic mixed in)
- Calculate shipping (duplicated from calc_shipping!)
- Build result dict

After Refactoring: Split into:

- process_order(): High-level orchestration only
- calc_order_weight(): Separate concern (calculate weight)
- Reuse calc_discount(), calc_tax(), calc_shipping()

3. Mixed concerns in main()

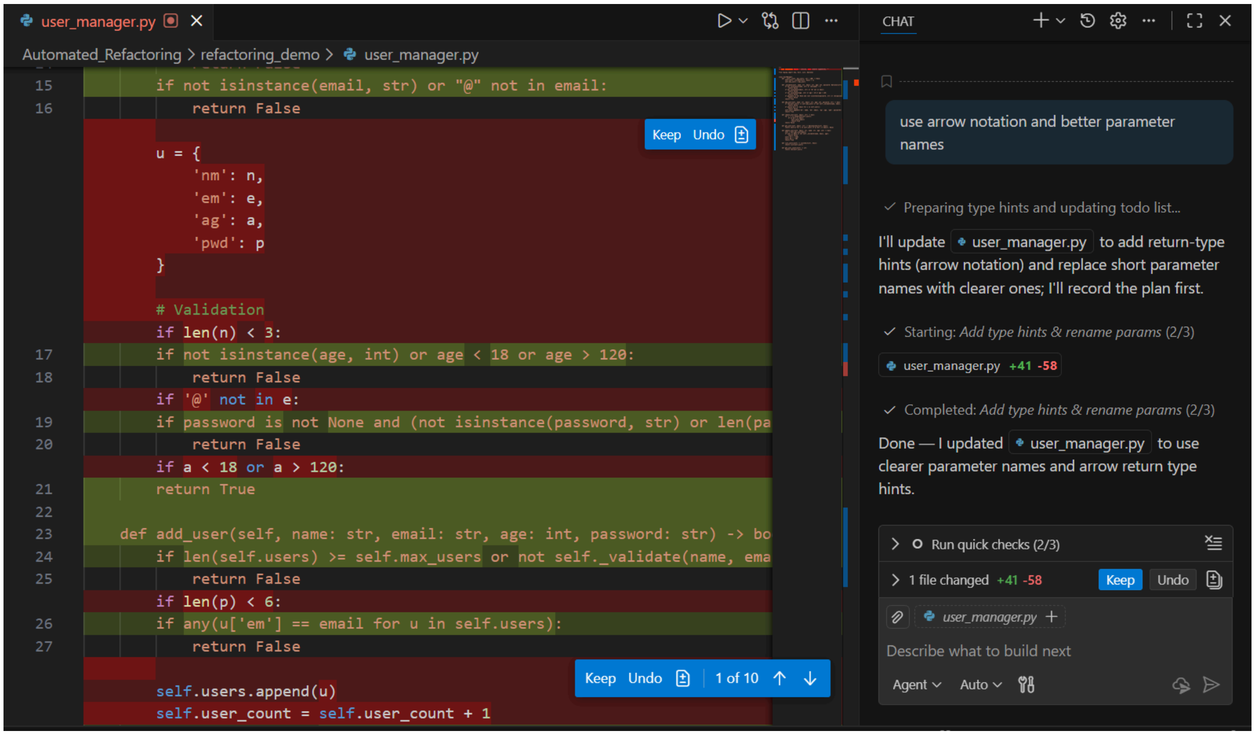
Before Refactoring: Business logic mixed with presentation (print statements)
After Refactoring: Separate demo functions handle business logic + their own output

- **Benefits of Function Splitting:**

✓ Single Responsibility Principle: Each function does one thing
✓ Testability: Smaller functions easier to unit test
✓ Reusability: demo_* functions can be called independently

Improving the name of a symbol

- **Files Refactored:** user_manager.py, calculator.py, data_processor.py, main.py
- **Code Selection**: Place cursor on function or variable
- **Type:** use arrow notation and better parameter names



```
        if not isinstance(email, str) or "@" not in email:
            return False

        u = {
            'nm': n,
            'em': e,
            'ag': a,
            'pwd': p
        }

        # Validation
        if len(n) < 3:
        if not isinstance(age, int) or age < 18 or age > 120:
            return False
        if '@' not in e:
        if password is not None and (not isinstance(password, str) or len(pa
            return False
        if a < 18 or a > 120:
        return True

    def add_user(self, name: str, email: str, age: int, password: str) -> bo
        if len(self.users) >= self.max_users or not self._validate(name, ema
            return False
        if len(p) < 6:
        if any(u['em'] == email for u in self.users):
            return False

        self.users.append(u)
        self.user_count = self.user_count + 1
```

use arrow notation and better parameter names

✓ Preparing type hints and updating todo list...

I'll update user_manager.py to add return-type hints (arrow notation) and replace short parameter names with clearer ones; I'll record the plan first.

✓ Starting: *Add type hints & rename params* (2/3)

user_manager.py +41 -58

✓ Completed: *Add type hints & rename params* (2/3)

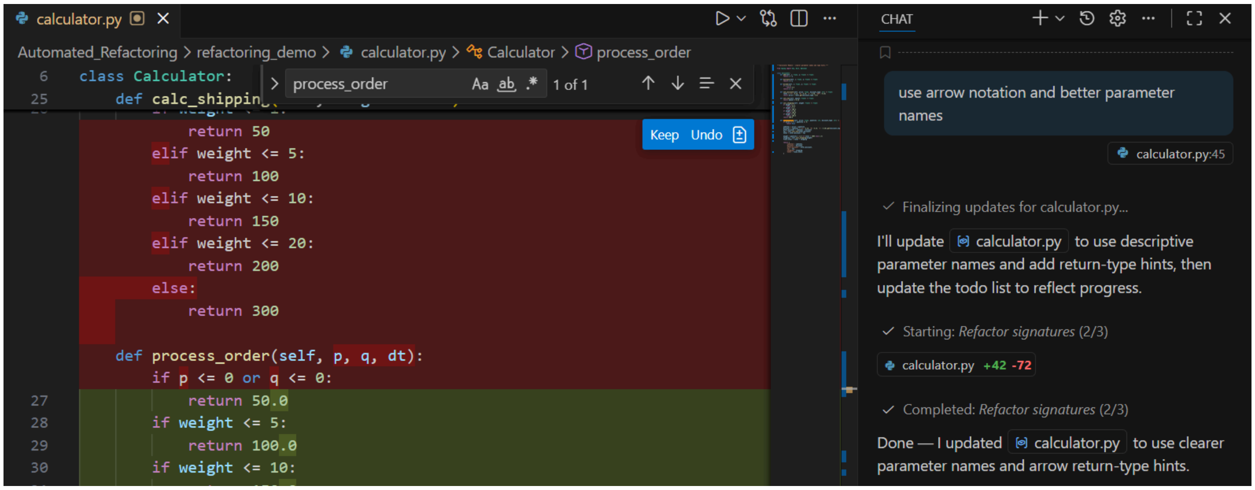
Done — I updated user_manager.py to use clearer parameter names and arrow return type hints.

○ Run quick checks (2/3)

1 file changed +41 -58

user_manager.py +

Describe what to build next



```
class Calculator:
    def calc_shippin
            return 50
        elif weight <= 5:
            return 100
        elif weight <= 10:
            return 150
        elif weight <= 20:
            return 200
        else:
            return 300

    def process_order(self, p, q, dt):
        if p <= 0 or q <= 0:
            return 50.0
        if weight <= 5:
            return 100.0
        if weight <= 10:
```

use arrow notation and better parameter names

✓ Finalizing updates for calculator.py...

I'll update calculator.py to use descriptive parameter names and add return-type hints, then update the todo list to reflect progress.

✓ Starting: *Refactor signatures* (2/3)

✓ Completed: *Refactor signatures* (2/3)

Done — I updated calculator.py to use clearer parameter names and arrow return-type hints.

- Problems Solved:

1. Cryptic parameter names in add_user()

Before Refactoring: add_user(self, n, e, a, p)

- n: unclear (name? number? net?)
- e: unclear (email? error? exception?)
- a: unclear (age? amount? account?)
- p: unclear (password? price? payload?)

After Refactoring: add_user(self, name: str, email: str, age: int, password: str)

2. Cryptic dictionary keys in user_manager.py

Before Refactoring: User dict with abbreviated keys:

{'nm': 'John', 'em': 'john@example.com', 'ag': 25, 'pwd': 'password'}

Hard to understand at a glance, error-prone

After Refactoring: {'name': 'John', 'email': 'john@example.com', 'age': 25, 'password': 'password'}

3. Cryptic parameter names in process_order()

Before Refactoring: process_order(self, p, q, dt)

- p: price or payment or product?
- q: quantity or query or question?
- dt: discount_type? data_type? document_type?

After Refactoring: process_order(self, price: Number, quantity: Number, discount_type: str)

4. Cryptic variable names in calc_stats()

Before Refactoring: s, c, a, mx, mn (abbreviated variable names)
return {'sum': s, 'count': c, 'avg': a, 'max': mx, 'min': mn}

After Refactoring: Renamed to clear Python variable names using sum(), count, avg, max, min

5. Lambda functions for simple transformations

Before Refactoring: Needed to pass transformations to _process_with_rules()

After Refactoring: Used arrow/lambda notation:
lambda x: x * 2 (for doubling numbers)
lambda s: s.upper() (for upper-casing strings)
lambda s: len(s) (for measuring length)

6. Missing documentation for abstract rules

Before Refactoring: _process_with_rules() had complex generic logic

After Refactoring: Added comprehensive docstring explaining rules tuple format:
"""(type_check, transform, measure) tuples:
- type_check: Type or tuple of types to match with isinstance
- transform: Function to apply to matched items
- measure: Function returning numeric value for aggregation"""

7. Generic function naming

Before Refactoring: _process_with_rules() was internal/private with generic name

After Refactoring: Kept as private (_) but added clear documentation
Also kept public methods with specific names:
- process_numbers()
- process_strings()
- process_mixed()

8. Missing type hints

Before Refactoring: No type hints on any methods (hard to understand expected types)

After Refactoring: Added type hints throughout:

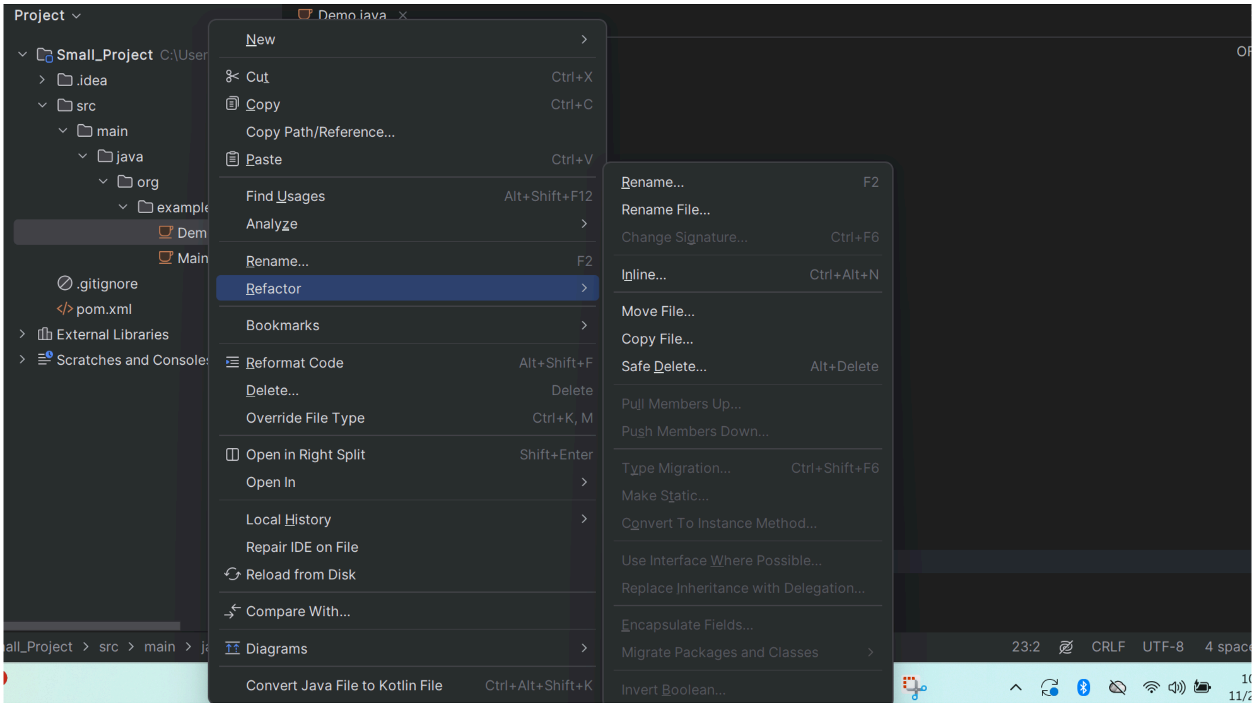
```
def add_user(self, name: str, email: str, age: int, password: str) -> bool:
def process_numbers(self, data: Iterable[Any]) -> Dict[str, Any]:
def calc_discount(self, price: Number, discount_type: str) -> Number:
```

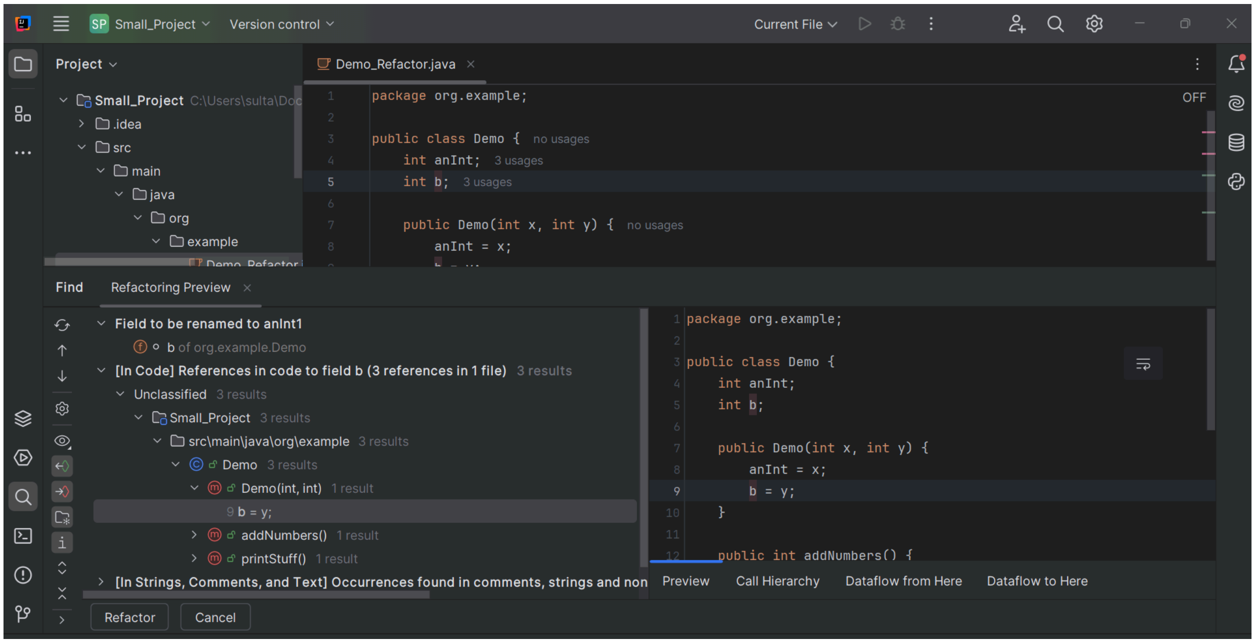
- **Benefits:**

✓ Code is self-documenting
✓ No need to look up variable meanings
✓ Easier to debug (meaningful names in stack traces)
✓ New developers understand code faster
✓ Fewer naming-related bugs

Demonstration (IntelliJ IDEA)

I also performed refactoring on a small Java project in IntelliJ following the given tutorial. Here are the screenshots:





```
package org.example;

public class Demo {  no usages
    int anInt;  3 usages
    int b;  3 usages

    public Demo(int x, int y) {  no usages
        anInt = x;
```

Field to be renamed to anInt1
- b of org.example.Demo

[In Code] References in code to field b (3 references in 1 file) 3 results
- Unclassified 3 results
  - Small_Project 3 results
    - src\main\java\org\example 3 results
      - Demo 3 results
        - Demo(int, int) 1 result
          - 9 b = y;
        - addNumbers() 1 result
        - printStuff() 1 result

[In Strings, Comments, and Text] Occurrences found in comments, strings and non

```
package org.example;

public class Demo {
    int anInt;
    int b;

    public Demo(int x, int y) {
        anInt = x;
        b = y;
    }

    public int addNumbers() {
```