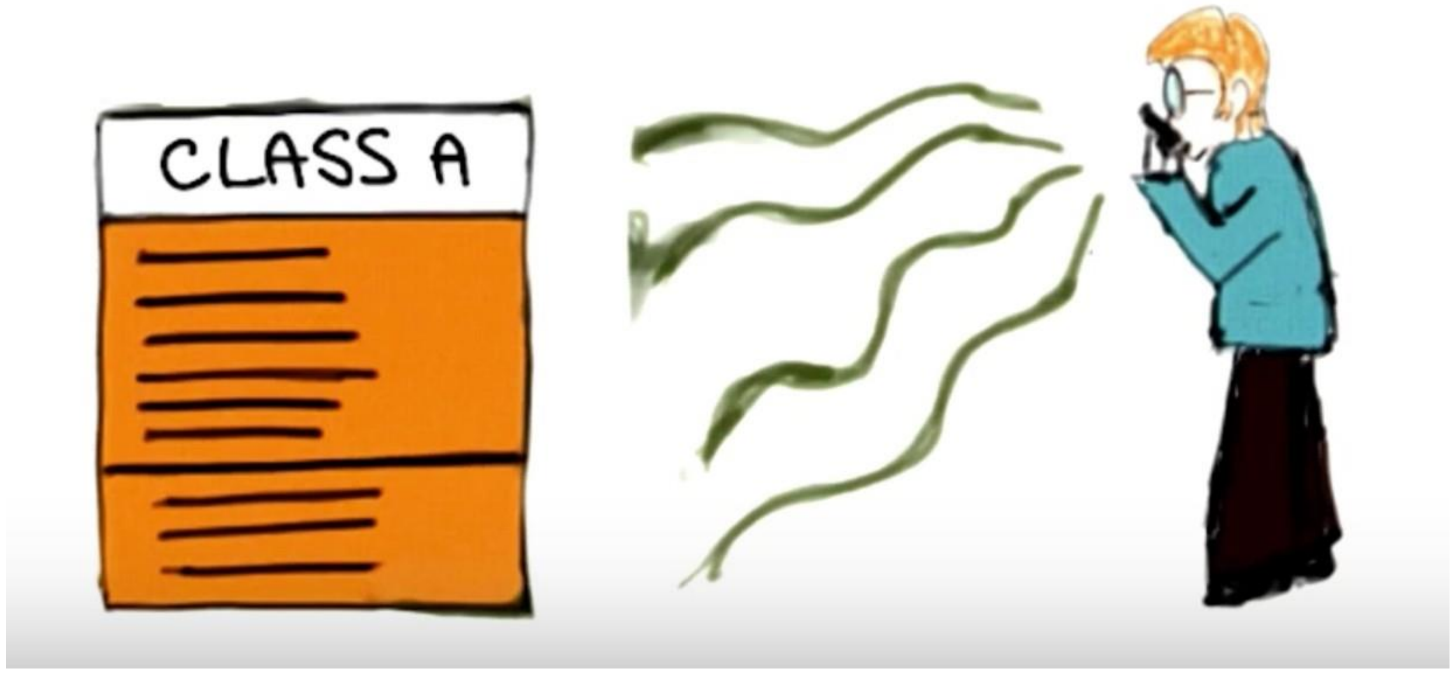
Lecture Outline

- Characteristics of Bad Design
- Bad Code Smells
- Refactoring
- Summary

Lecture Outline

Bad Code Smells

- Duplicate Code
- Long method
- Large class
- Shotgun surgery
- Feature envy



Bad Code Smells

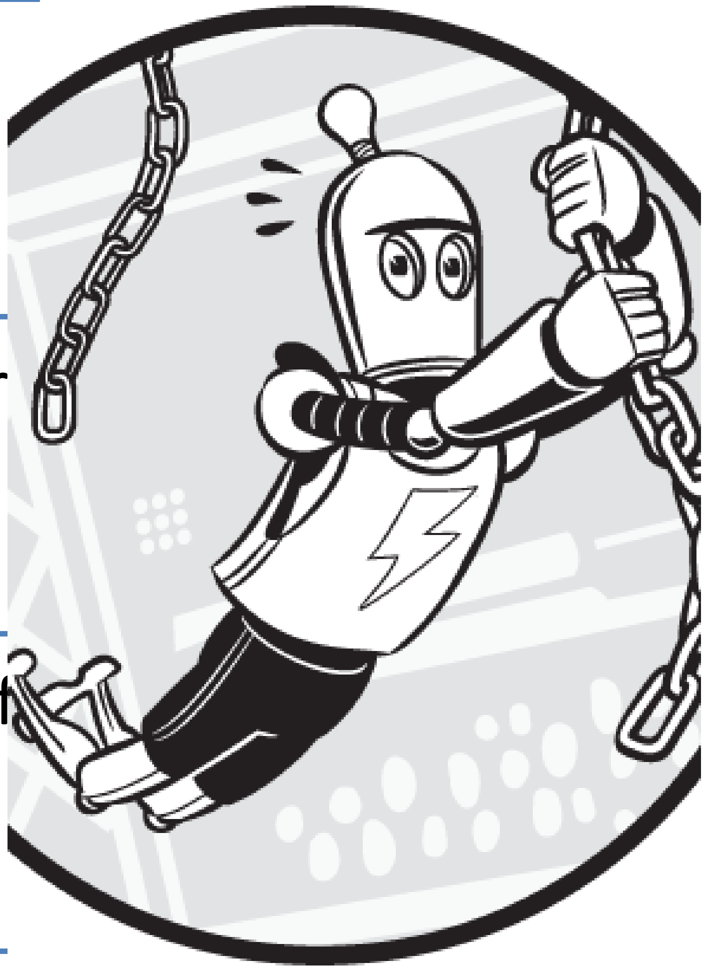
Bad Code smells

Code that causes a program to crash is obviously wrong, but crashes aren't the only indicator of issues in your programs.

Other signs can suggest the presence of more subtle bugs or unreadable code.

Just as the smell of gas can indicate a gas leak or the smell of smoke could indicate a fire, a *code smell* is a source code pattern that signals potential bugs.

A code smell doesn't necessarily mean a problem exists, but it does mean you should investigate your program.



Bad Code Smells

- Duplicate Code
- Long method
- Large class
- Shotgun surgery
- Feature envy

Bad Code Smells

- Duplicate Code
- Long method
- Large class
- Shotgun surgery
- Feature envy

The most common code smell is *duplicate code*. Duplicate code is any source code that you could have created by copying and pasting some other code into your program.

Duplicate Code

Same code structure replicated in more than one place.

↓

Common in copy-paste programming style

↓

Solution:

Extract methods, Deduplicate code, functions, loops

Duplicate Code Examples

Extract Method

Problem

You have a code fragment that can be grouped together.

Solution

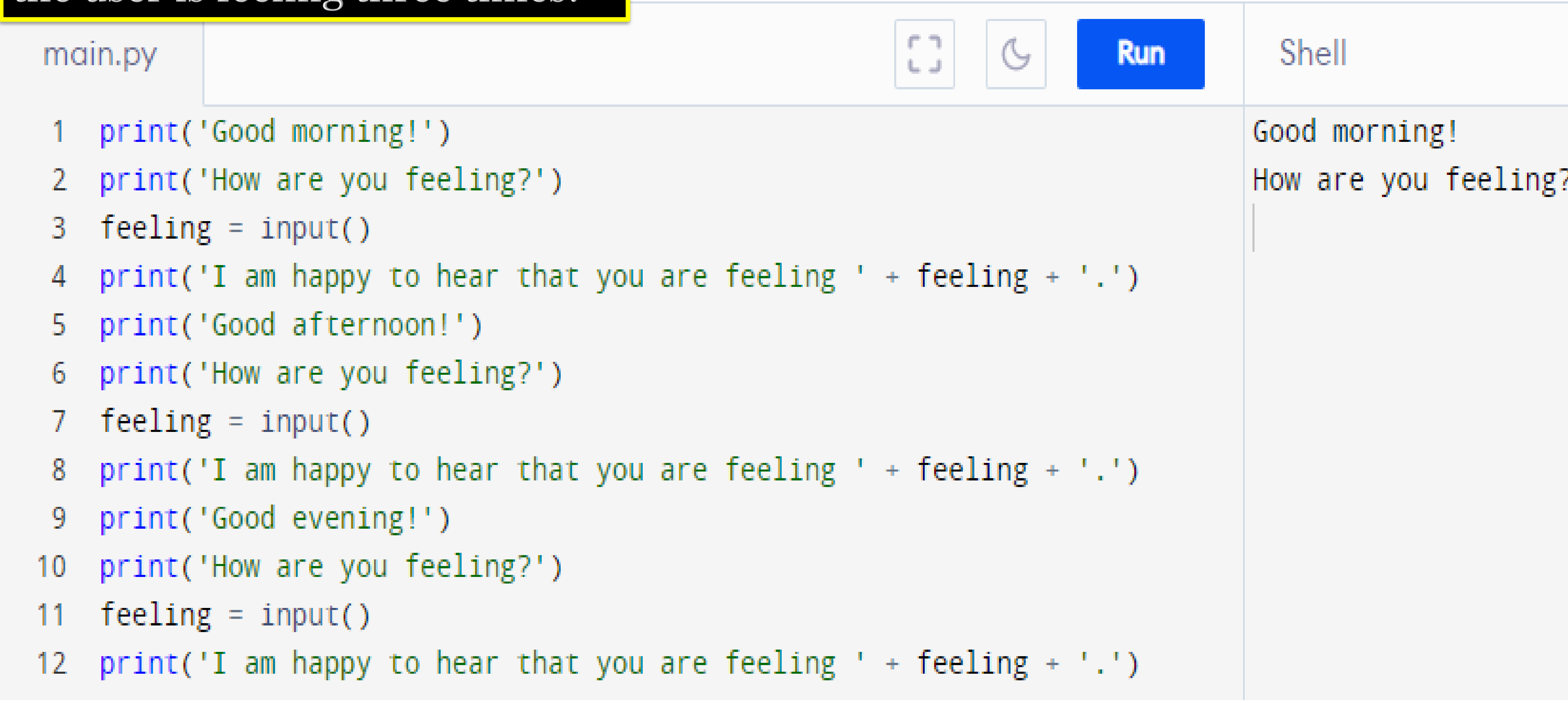
Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
def printOwing(self):
    self.printBanner()

    # print details
    print("name:", self.name)
    print("amount:", self.getOutstanding())
```

```
def printOwing(self):
    self.printBanner()
    self.printDetails(self.getOutstanding())

def printDetails(self, outstanding):
    print("name:", self.name)
    print("amount:", outstanding)
```

For example, this short program contains duplicate code. Notice that it asks how the user is feeling three times:

Problem

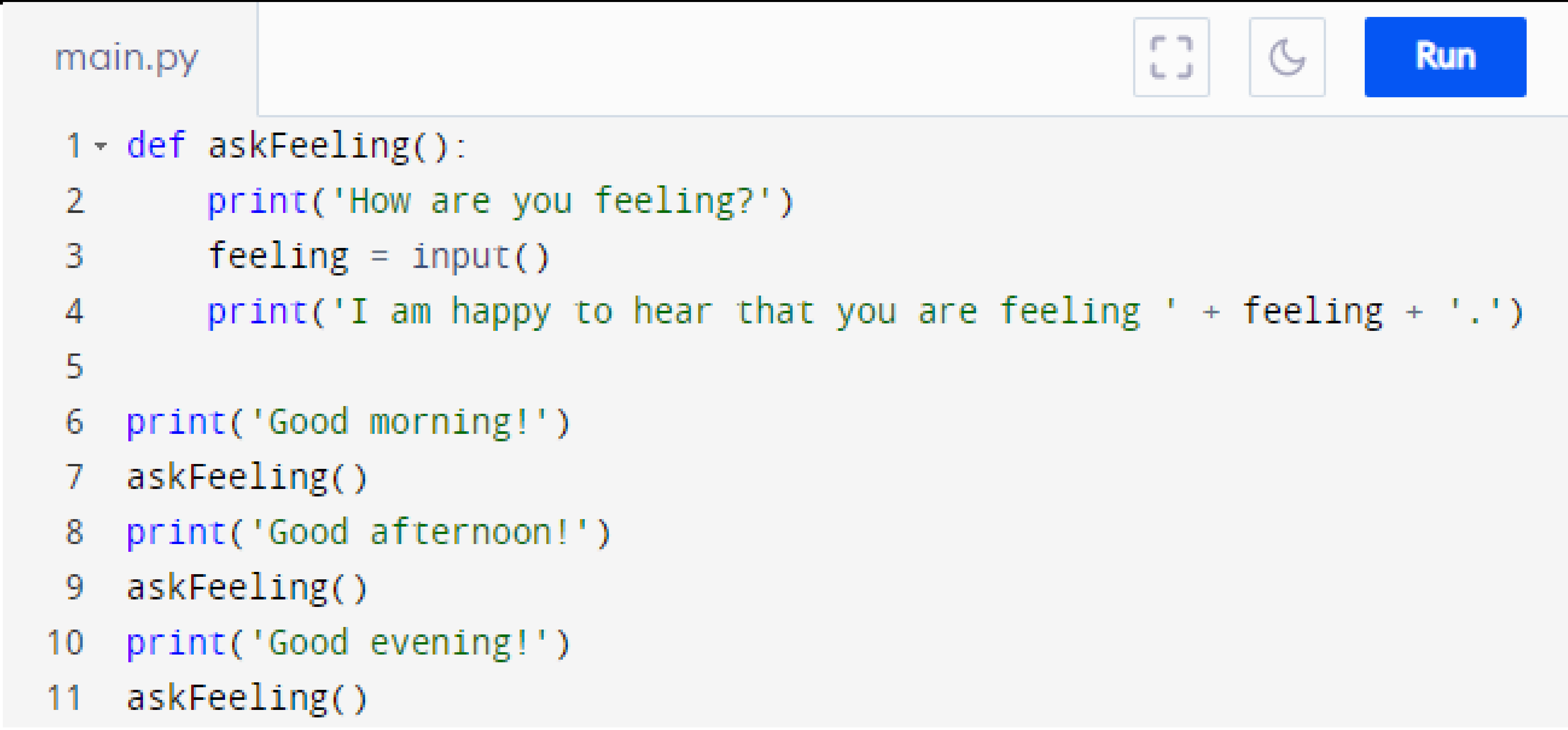


```
print('Good morning!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good afternoon!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good evening!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
```

```
Good morning!
How are you feeling?
```

Moved the duplicate code into a function and then repeatedly called that function

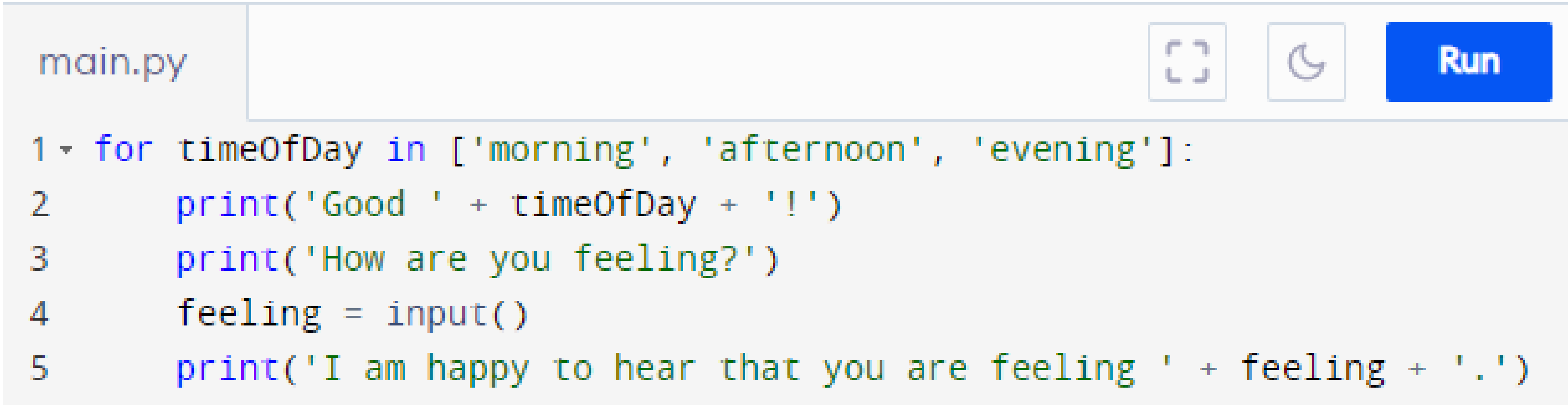
Solution



```python
def askFeeling():
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')

print('Good morning!')
askFeeling()
print('Good afternoon!')
askFeeling()
print('Good evening!')
askFeeling()
```

Moved the duplicate code into a loop:

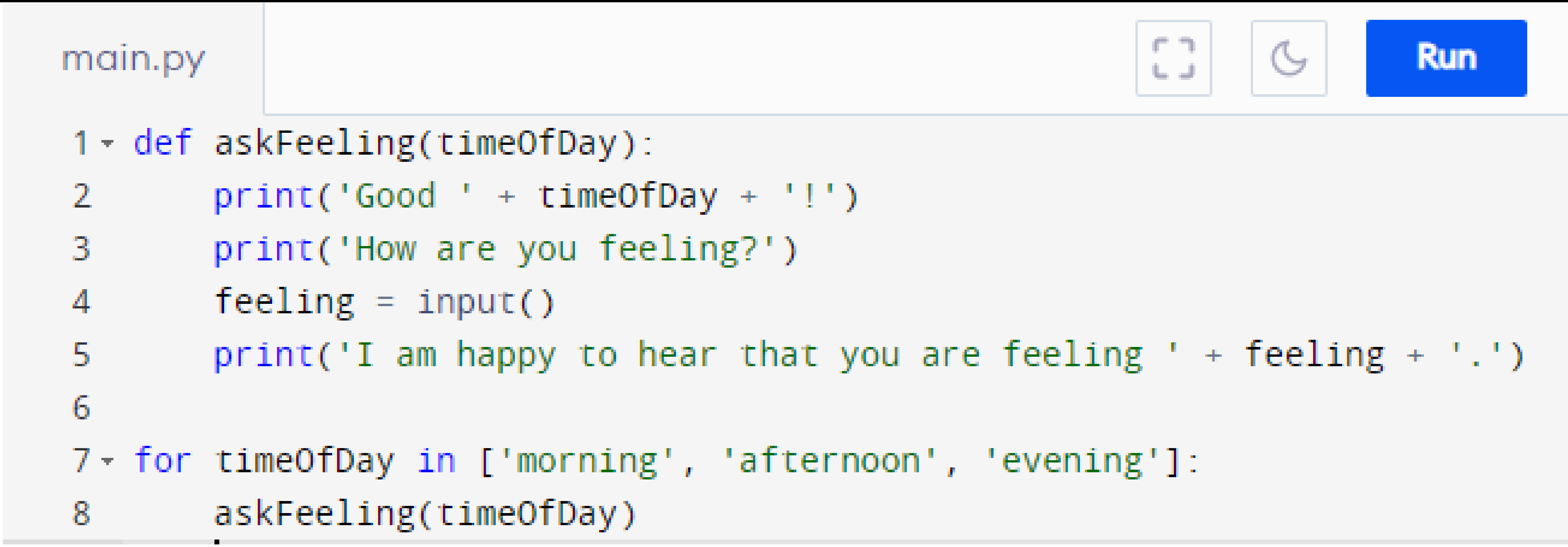
Solution



```python
for timeOfDay in ['morning', 'afternoon', 'evening']:
    print('Good ' + timeOfDay + '!')
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')
```

combined the two techniques and use a function and a loop

Solution



```python
def askFeeling(timeOfDay):
    print('Good ' + timeOfDay + '!')
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')

for timeOfDay in ['morning', 'afternoon', 'evening']:
    askFeeling(timeOfDay)
```

Duplicate Code Example Discussion

Sometimes, code is just not worth the trouble of deduplicating.

Compare the first code example in this section to the most recent one.

Although the duplicate code is longer, it’s simple and straightforward.

The deduplicated example does the same thing but involves a loop, a new timeOfDay loop variable, and a new function with a parameter that is also named timeOfDay.

Duplicate Code Summary

- Duplicate code is a problem because it makes changing the code difficult; a change you make to one copy of the duplicate code must be made to every copy of it in the program.
- If you forget to make a change somewhere, or if you make different changes to different copies, your program will likely end up with bugs.
- The solution to duplicate code is to deduplicate it; that is, make it appear once in your program by placing the code in a function or loop.
- As with all code smells, avoiding duplicate code isn't a hard-and-fast rule you must always follow.
- In general, the longer the duplicate code section or the more duplicate copies that appear in your program, the stronger the case for deduplicating it.
- Generally start to consider deduplicating code when three or four copies exist in my program.

Bad Code Smells

- Duplicate Code
- Long method
- Large class
- Shotgun surgery
- Feature envy

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object. Decompose Conditional

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object. Decompose Conditional

Extract Method

Problem

You have a code fragment that can be grouped together.

Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
def printOwing(self):
    self.printBanner()

    # print details
    print("name:", self.name)
    print("amount:", self.getOutstanding())
```

```
def printOwing(self):
    self.printBanner()
    self.printDetails(self.getOutstanding())

def printDetails(self, outstanding):
    print("name:", self.name)
    print("amount:", outstanding)
```

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object. Decompose Conditional

Replace Temp with Query

Problem

You place the result of an expression in a local variable for later use in your code.

Solution

Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

```
def calculateTotal():
    basePrice = quantity * itemPrice
    if basePrice > 1000:
        return basePrice * 0.95
    else:
        return basePrice * 0.98
```

```
def calculateTotal():
    if basePrice() > 1000:
        return basePrice() * 0.95
    else:
        return basePrice() * 0.98

def basePrice():
    return quantity * itemPrice
```

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

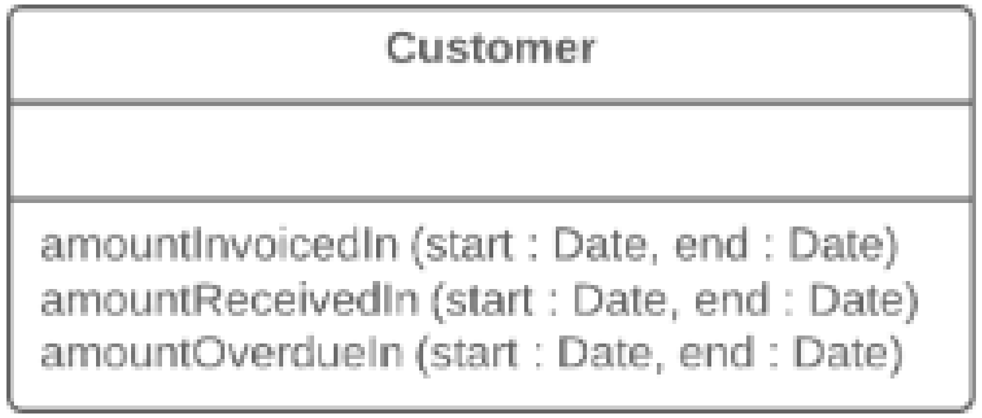
Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object. Decompose Conditional

Introduce Parameter Object

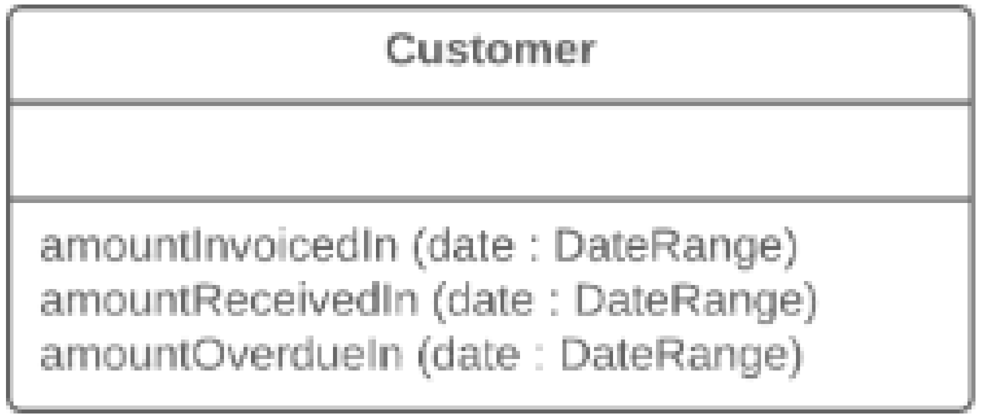
Problem

Your methods contain a repeating group of parameters.


Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)

Solution

Replace these parameters with an object.


Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object. Decompose Conditional

Preserve Whole Object

Problem

You get several values from an object and then pass them as parameters to a method.

```
low = daysTempRange.getLow()
high = daysTempRange.getHigh()
withinPlan = plan.withinRange(low, high)
```

Solution

Instead, try passing the whole object.

```
withinPlan = plan.withinRange(daysTempRange)
```

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object. Replace Method with Method Object, Decompose Conditional

Replace Method with Method Object

Problem

You have a long method in which the local variables are so intertwined that you can't apply *Extract Method*.

```
class Order:
    # ...
    def price(self):
        primaryBasePrice = 0
        secondaryBasePrice = 0
        tertiaryBasePrice = 0
        # Perform long computation.
```

Solution

Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class.

```
class Order:
    # ...
    def price(self):
        return PriceCalculator(self).compute()


class PriceCalculator:
    def __init__(self, order):
        self._primaryBasePrice = 0
        self._secondaryBasePrice = 0
        self._tertiaryBasePrice = 0
        # Copy relevant information from the
        # order object.

    def compute(self):
        # Perform long computation.
```

Long Method

The longer a procedure is, the more difficult is it to understand

Solution:

Identify clumps of code to extract, Break down the method into smaller methods

Extract Method, Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object, Replace Method with Method Object, Decompose Conditional

Decompose Conditional

Problem

You have a complex conditional (`if-then`/ `else` or `switch`).

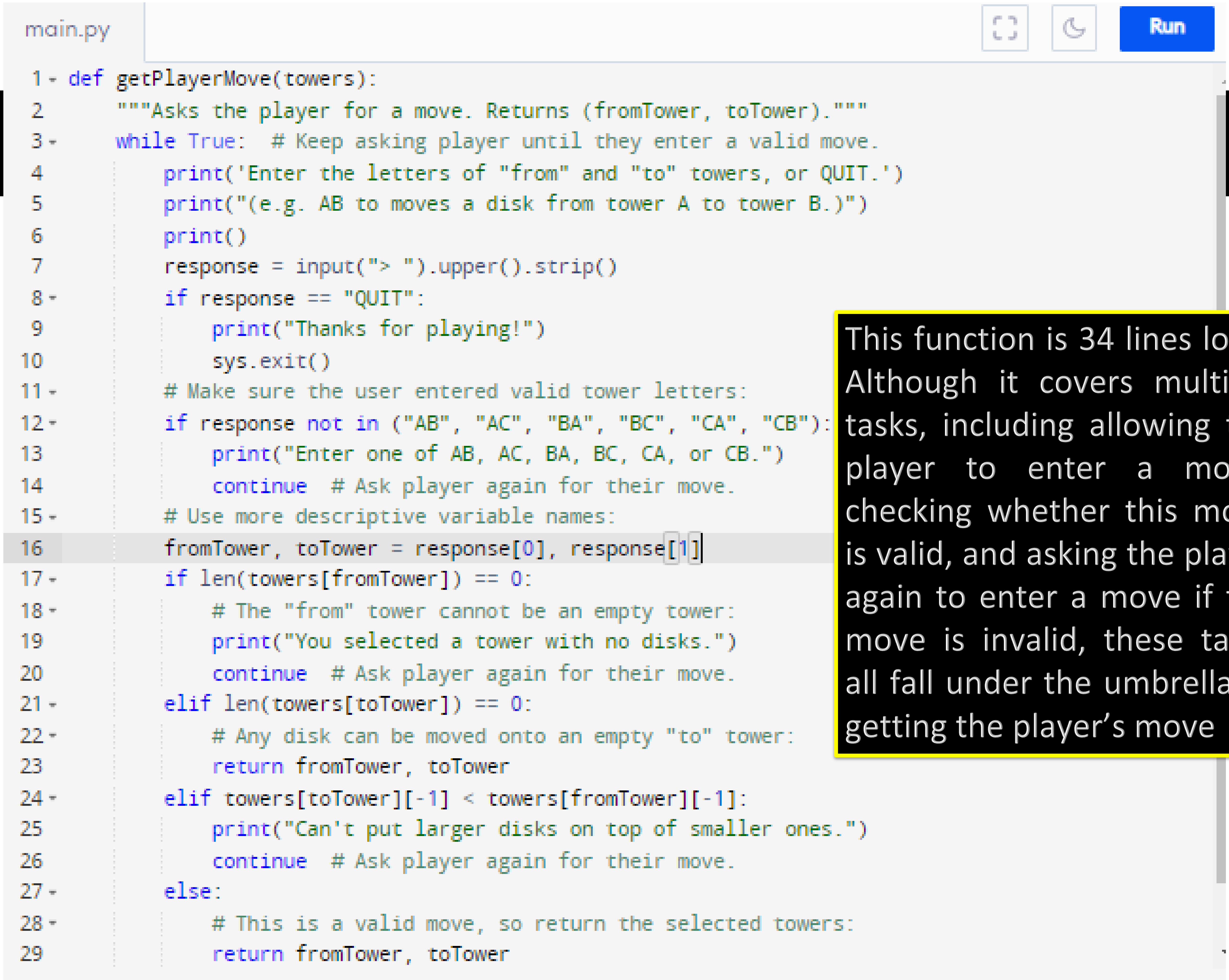
```
if date.before(SUMMER_START) or date.after(SUMMER_END):
    charge = quantity * winterRate + winterServiceCharge
else:
    charge = quantity * summerRate
```

Solution

Decompose the complicated parts of the conditional into separate methods: the condition, `then` and `else`.

```
if isSummer(date):
    charge = summerCharge(quantity)
else:
    charge = winterCharge(quantity)
```

Long Function Examples

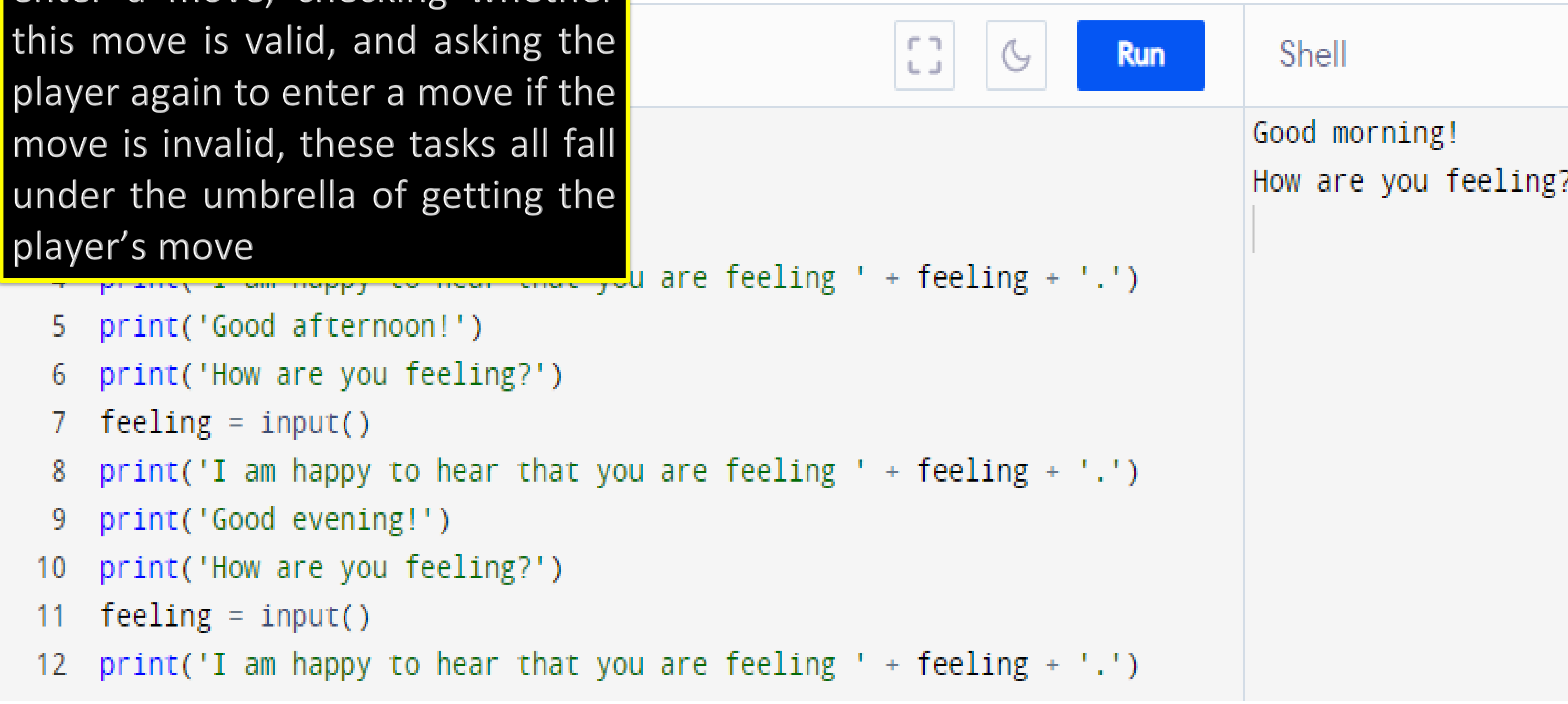


```python
def getPlayerMove(towers):
    """Asks the player for a move. Returns (fromTower, toTower)."""
    while True:  # Keep asking player until they enter a valid move.
        print('Enter the letters of "from" and "to" towers, or QUIT.')
        print("(e.g. AB to moves a disk from tower A to tower B.)")
        print()
        response = input("> ").upper().strip()
        if response == "QUIT":
            print("Thanks for playing!")
            sys.exit()
        # Make sure the user entered valid tower letters:
        if response not in ("AB", "AC", "BA", "BC", "CA", "CB"):
            print("Enter one of AB, AC, BA, BC, CA, or CB.")
            continue  # Ask player again for their move.
        # Use more descriptive variable names:
        fromTower, toTower = response[0], response[1]
        if len(towers[fromTower]) == 0:
            # The "from" tower cannot be an empty tower:
            print("You selected a tower with no disks.")
            continue  # Ask player again for their move.
        elif len(towers[toTower]) == 0:
            # Any disk can be moved onto an empty "to" tower:
            return fromTower, toTower
        elif towers[toTower][-1] < towers[fromTower][-1]:
            print("Can't put larger disks on top of smaller ones.")
            continue  # Ask player again for their move.
        else:
            # This is a valid move, so return the selected towers:
            return fromTower, toTower
```

This function is 34 lines long. Although it covers multiple tasks, including allowing the player to enter a move, checking whether this move is valid, and asking the player again to enter a move if the move is invalid, these tasks all fall under the umbrella of getting the player's move

This function is 34 lines long. Although it covers multiple tasks, including allowing the player to enter a move, checking whether this move is valid, and asking the player again to enter a move if the move is invalid, these tasks all fall under the umbrella of getting the player's move

Problem

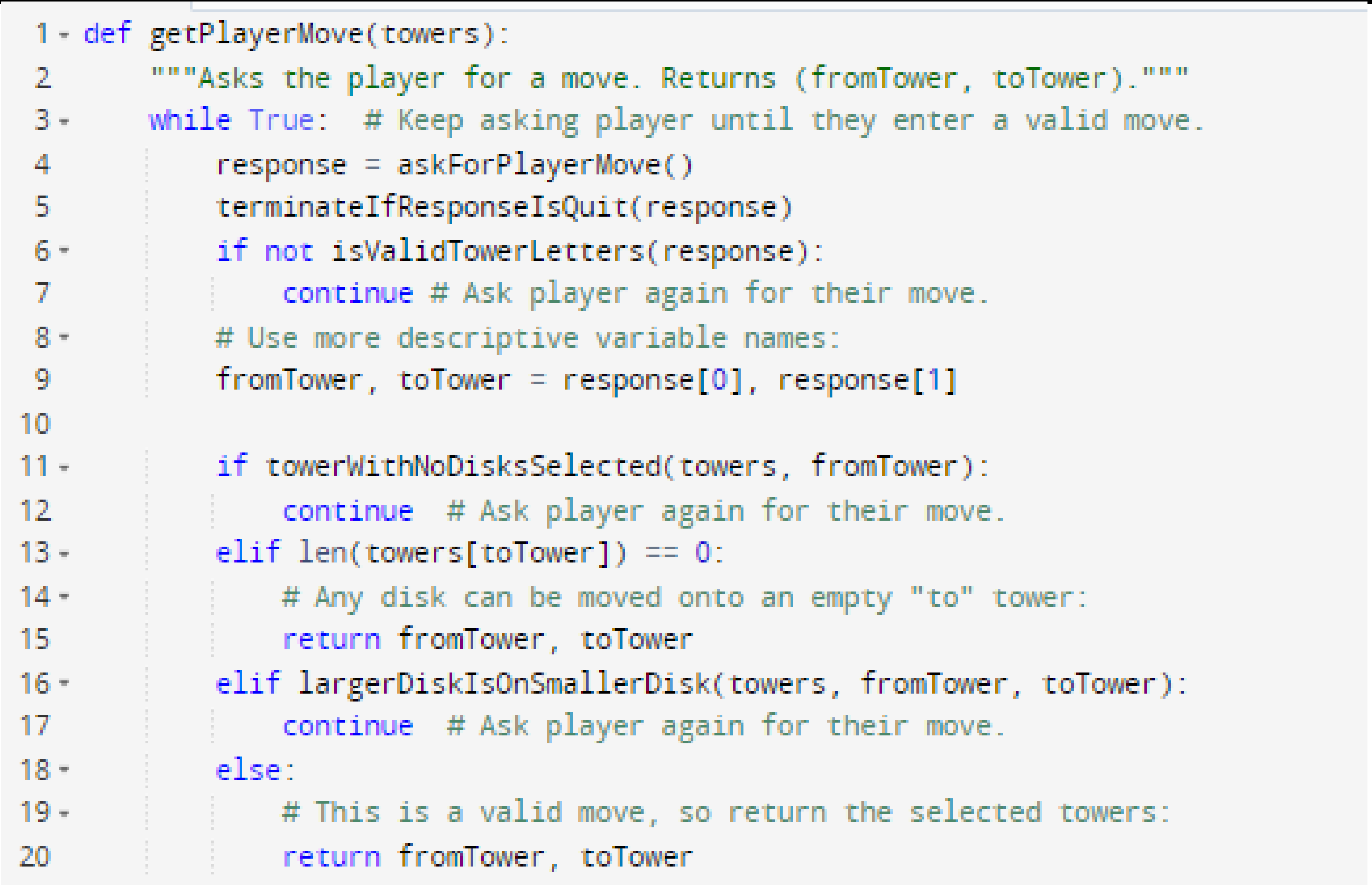


```
        you are feeling ' + feeling + '.')
print('Good afternoon!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good evening!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
```

```
Good morning!
How are you feeling?
```

break the code in getPlayerMove() into smaller functions,

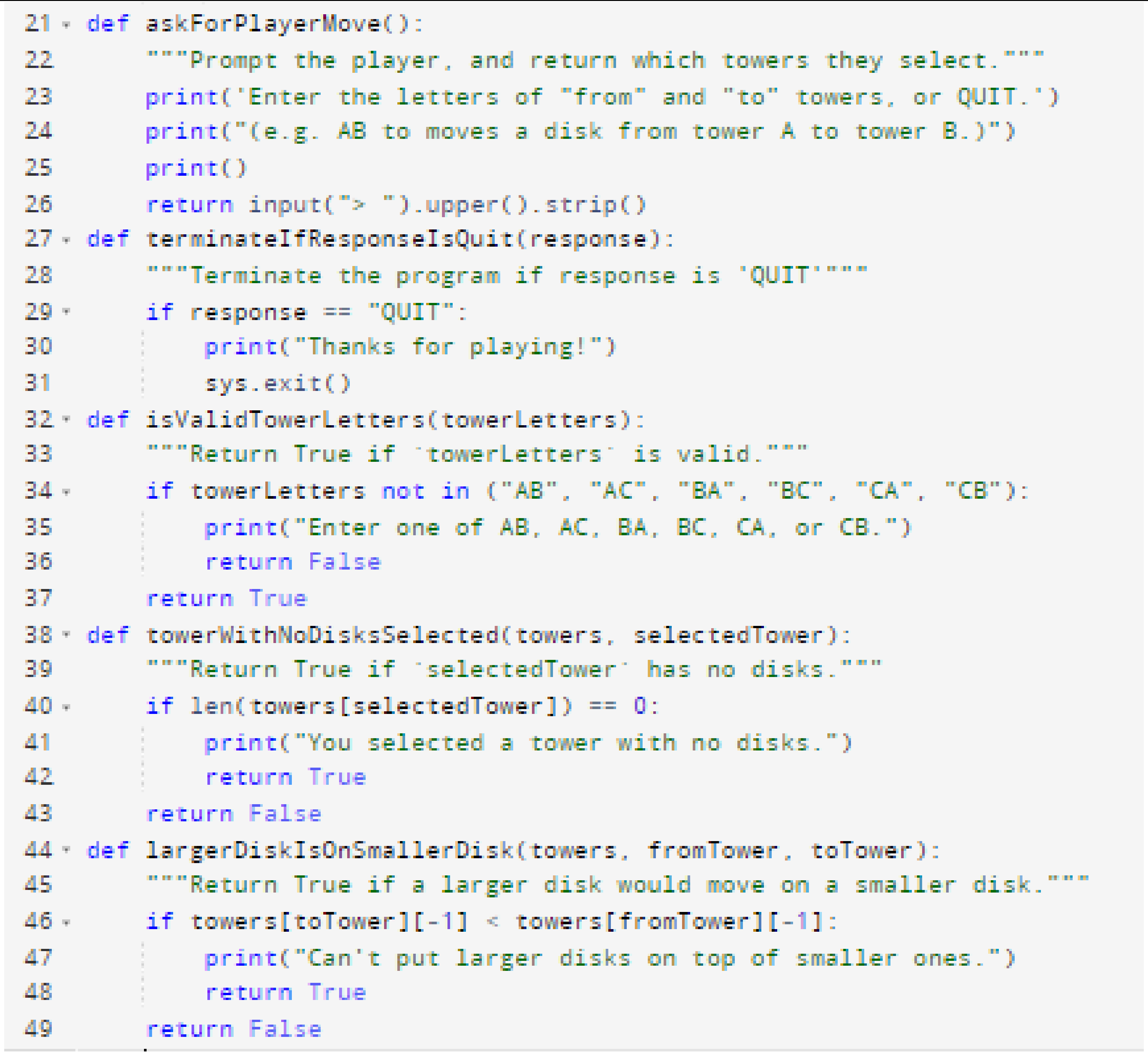
Solution



```python
def getPlayerMove(towers):
    """Asks the player for a move. Returns (fromTower, toTower)."""
    while True:  # Keep asking player until they enter a valid move.
        response = askForPlayerMove()
        terminateIfResponseIsQuit(response)
        if not isValidTowerLetters(response):
            continue # Ask player again for their move.
        # Use more descriptive variable names:
        fromTower, toTower = response[0], response[1]

        if towerWithNoDisksSelected(towers, fromTower):
            continue  # Ask player again for their move.
        elif len(towers[toTower]) == 0:
            # Any disk can be moved onto an empty "to" tower:
            return fromTower, toTower
        elif largerDiskIsOnSmallerDisk(towers, fromTower, toTower):
            continue  # Ask player again for their move.
        else:
            # This is a valid move, so return the selected towers:
            return fromTower, toTower
```

break the code in getPlayerMove() into smaller functions,

Solution



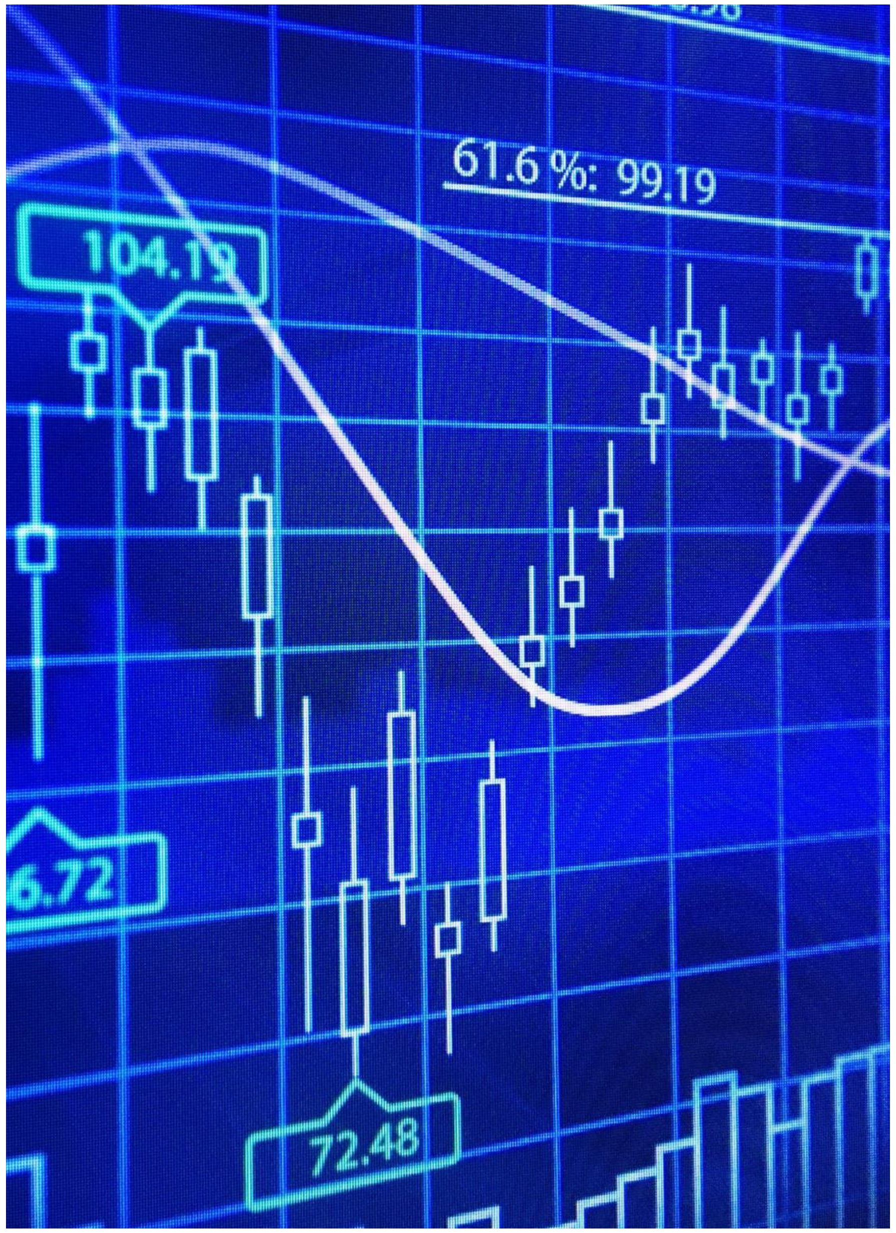
```python
def askForPlayerMove():
    """Prompt the player, and return which towers they select."""
    print('Enter the letters of "from" and "to" towers, or QUIT.')
    print("(e.g. AB to moves a disk from tower A to tower B.)")
    print()
    return input("> ").upper().strip()
def terminateIfResponseIsQuit(response):
    """Terminate the program if response is 'QUIT'"""
    if response == "QUIT":
        print("Thanks for playing!")
        sys.exit()
def isValidTowerLetters(towerLetters):
    """Return True if `towerLetters` is valid."""
    if towerLetters not in ("AB", "AC", "BA", "BC", "CA", "CB"):
        print("Enter one of AB, AC, BA, BC, CA, or CB.")
        return False
    return True
def towerWithNoDisksSelected(towers, selectedTower):
    """Return True if `selectedTower` has no disks."""
    if len(towers[selectedTower]) == 0:
        print("You selected a tower with no disks.")
        return True
    return False
def largerDiskIsOnSmallerDisk(towers, fromTower, toTower):
    """Return True if a larger disk would move on a smaller disk."""
    if towers[toTower][-1] < towers[fromTower][-1]:
        print("Can't put larger disks on top of smaller ones.")
        return True
    return False
```

Function Size Trade-Offs

Some programmers say that functions should be as short as possible and no longer than what can fit on a single screen.

A function that is only a dozen lines long is relatively easy to understand, at least compared to one that is hundreds of lines long.

But making functions shorter by splitting up their code into multiple smaller functions can also have its downsides.



Advantages of Small Functions

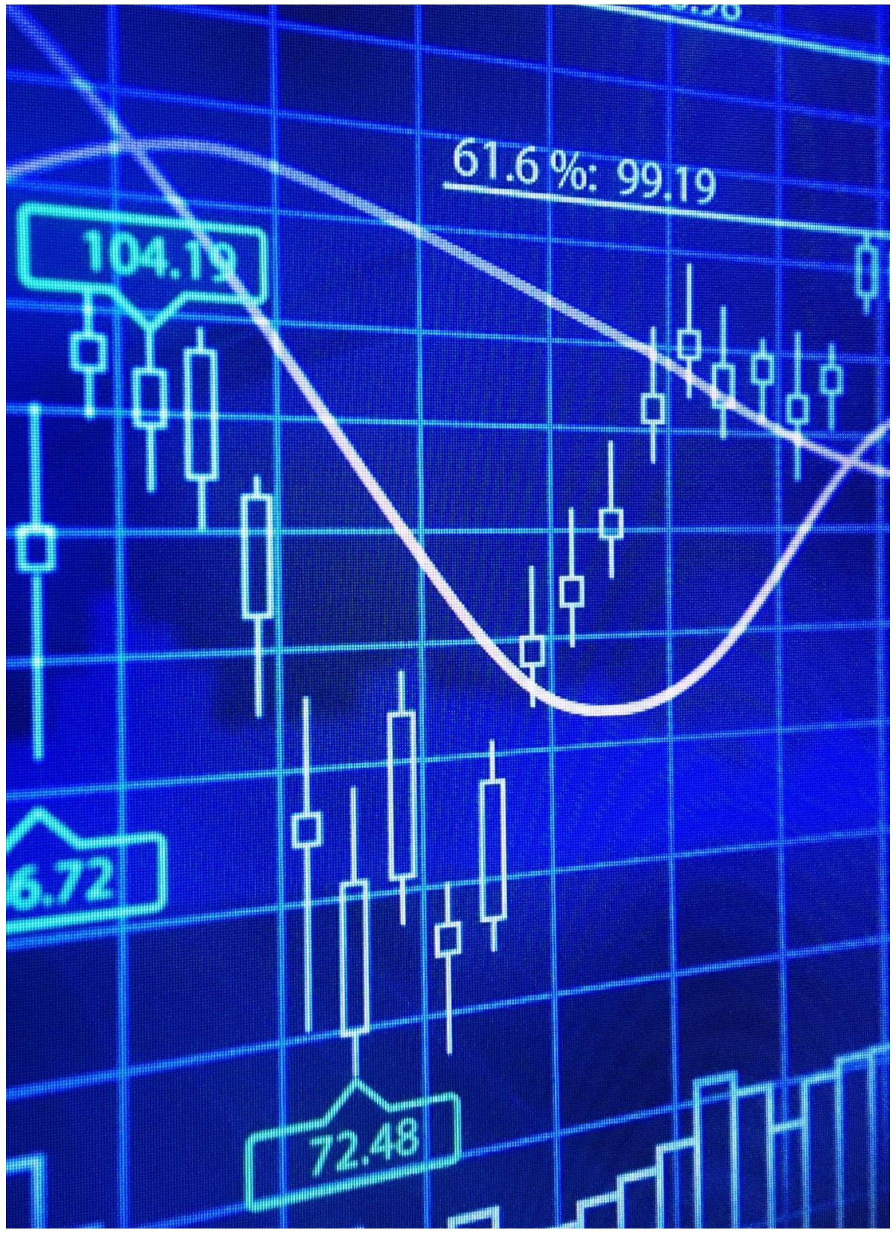
The function's code is easier to understand.

The function likely requires fewer parameters.

The function is less likely to have side effects, as described in "Functional Programming" on page 172.

The function is easier to test and debug.

The function likely raises fewer different kinds of exceptions.



Disadvantages of Small Functions

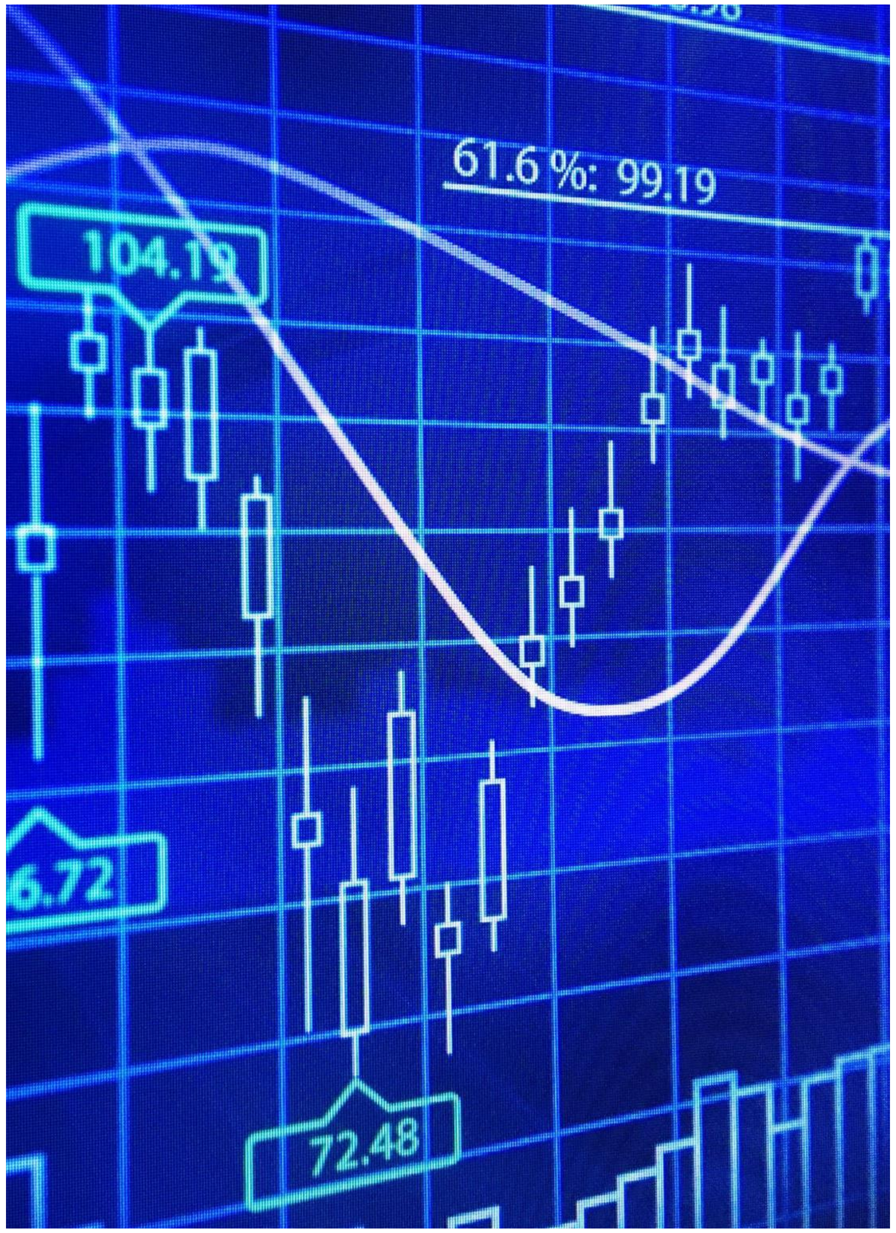
Writing short functions often means a larger number of functions in the program.

Having more functions means the program is more complicated.

Having more functions also means having to come up with additional descriptive, accurate names, which is a difficult task.

Using more functions requires you to write more documentation.

The relationships between functions become more complicated.



Bad Code Smells

- Duplicate Code
- Long method
- Large class
- Shotgun surgery
- Feature envy

Long/Large Class

A class is trying to do too much. A class contains many fields/methods/lines of code.

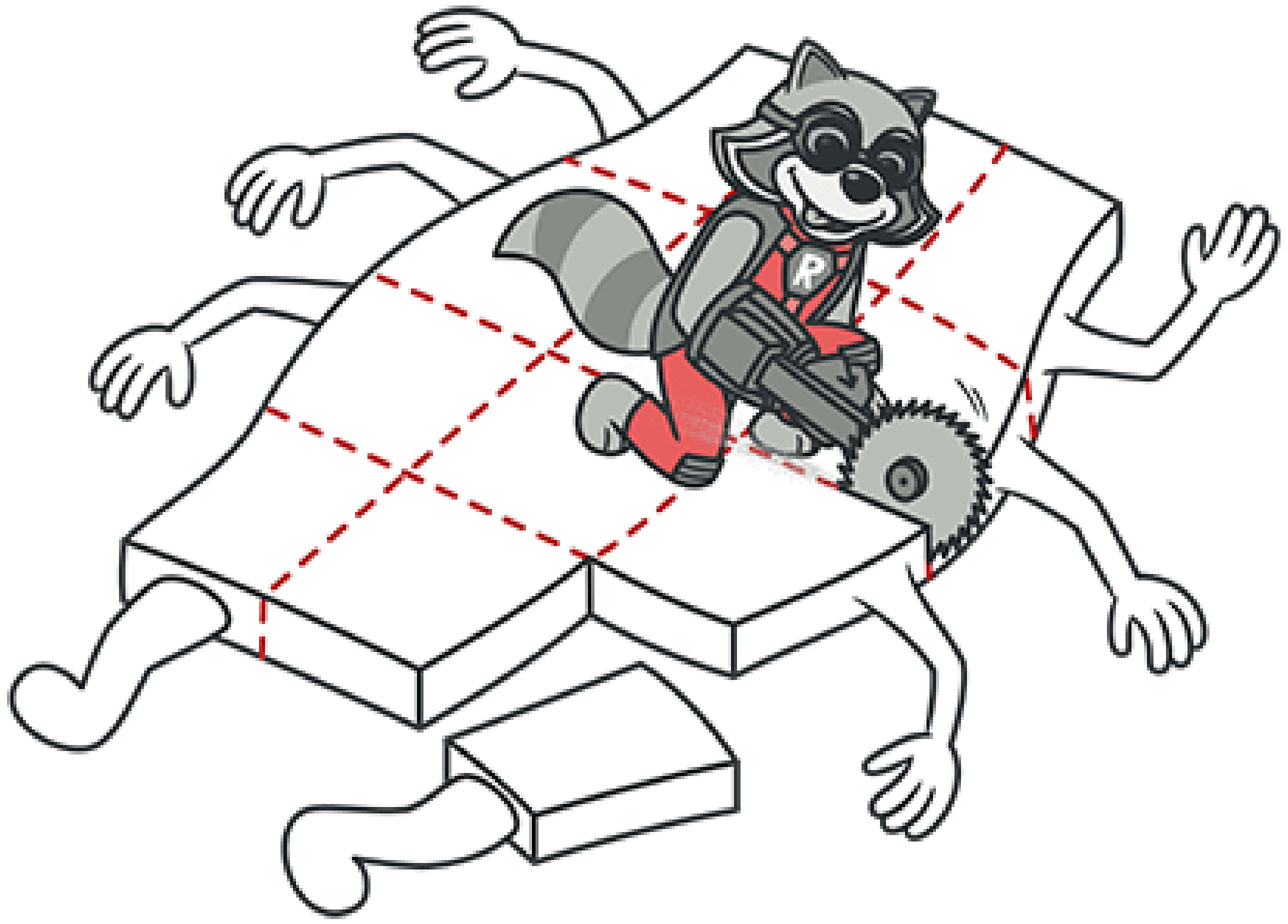
More than a couple dozen methods, or half a dozen variables

Solution:

Extract Class, Extract Subclass, Extract Interface ,Duplicate Observed Data

Reasons for the Problem

Classes usually start small. But over time, they get bloated as the program grows.
As is the case with long methods as well, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.

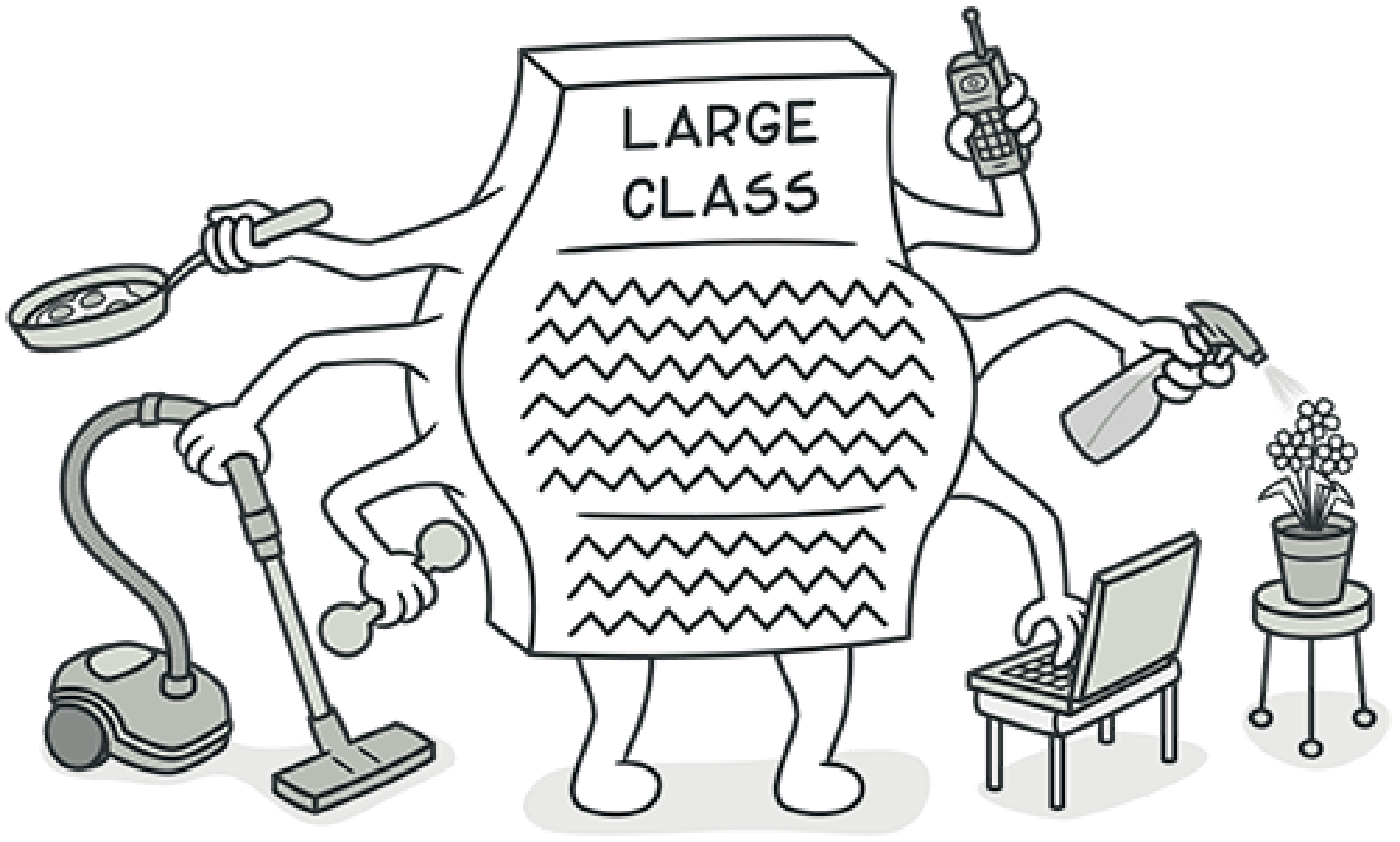


Solution

When a class is wearing too many (functional) hats, think about splitting it up:

- **Extract Class** helps if part of the behavior of the large class can be spun off into a separate component.
- **Extract Subclass** helps if part of the behavior of the large class can be implemented in different ways or is used in rare cases.
- **Extract Interface** helps if it's necessary to have a list of the operations and behaviors that the client can use.
- If a large class is responsible for the graphical interface, you may try to move some of its data and behavior to a separate domain object. In doing so, it may be necessary to store copies of some data in two places and keep the data consistent. **Duplicate Observed Data** offers a way to do this.

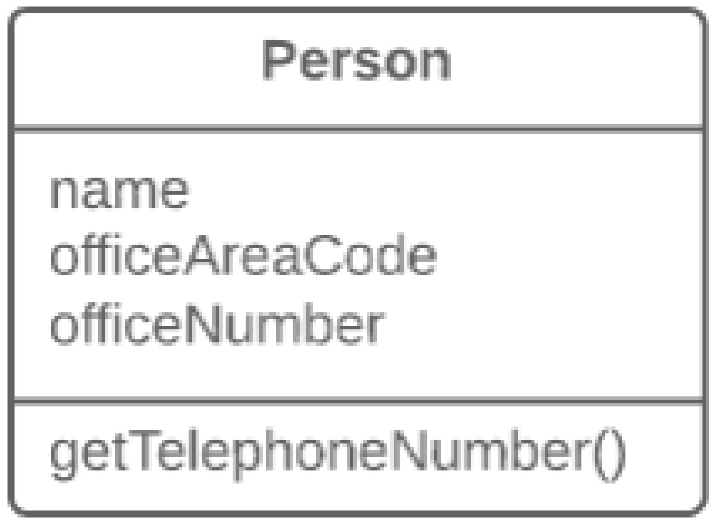
Large class



Extract Class

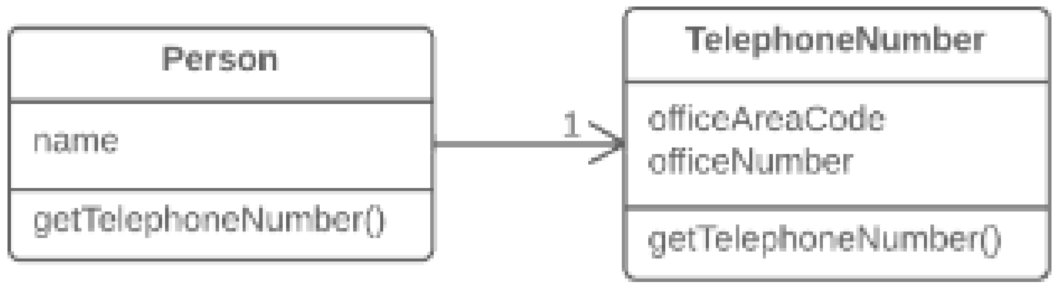
Problem

When one class does the work of two, awkwardness results.



Solution

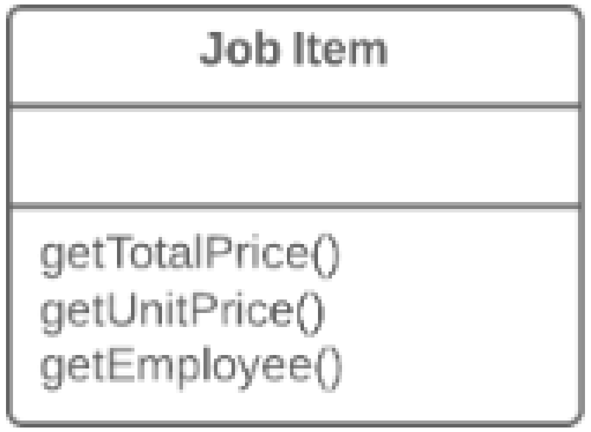
Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



Extract Subclass

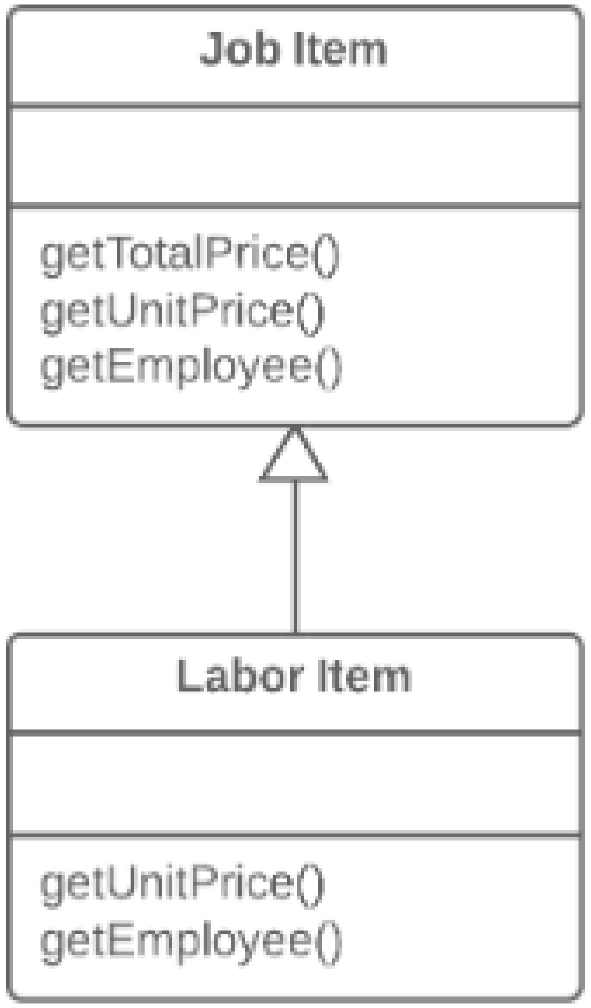
Problem

A class has features that are used only in certain cases.



Solution

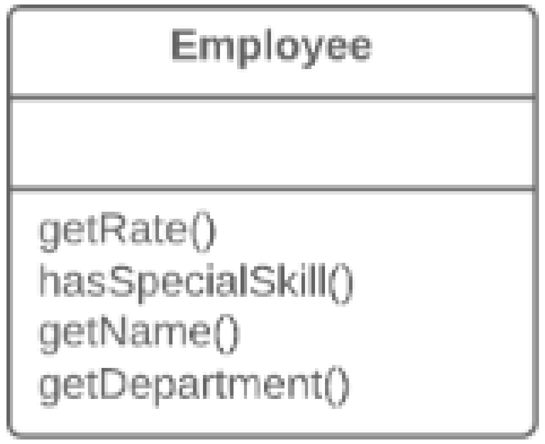
Create a subclass and use it in these cases.



Extract Interface

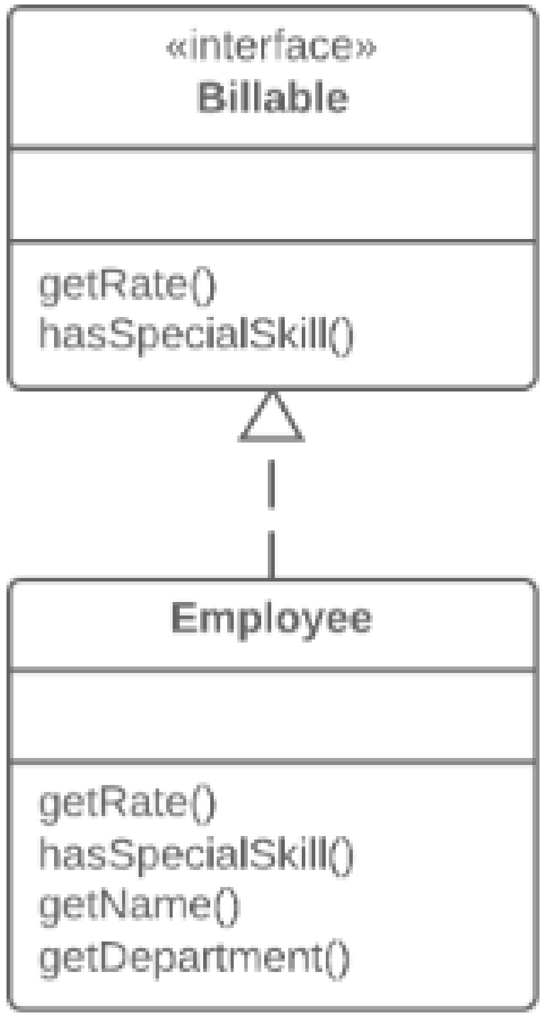
Problem

Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.



Solution

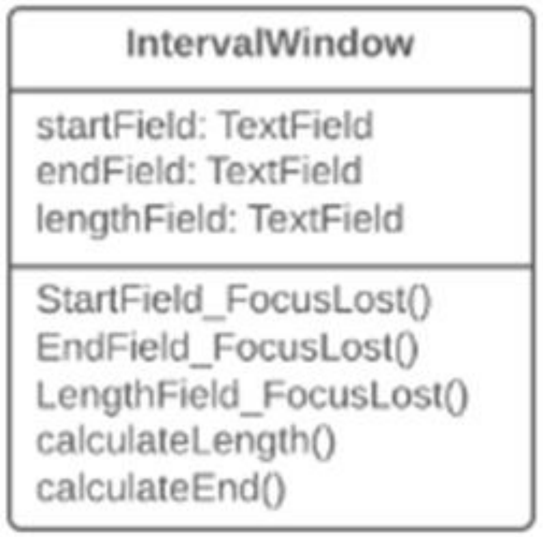
Move this identical portion to its own interface.



Duplicate Observed Data

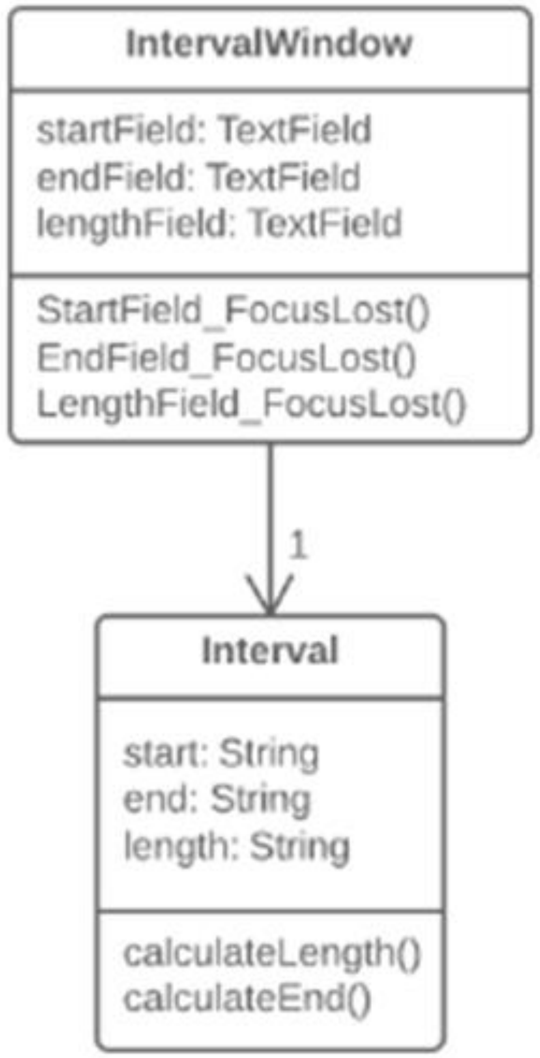
Problem

Is domain data stored in classes responsible for the GUI?



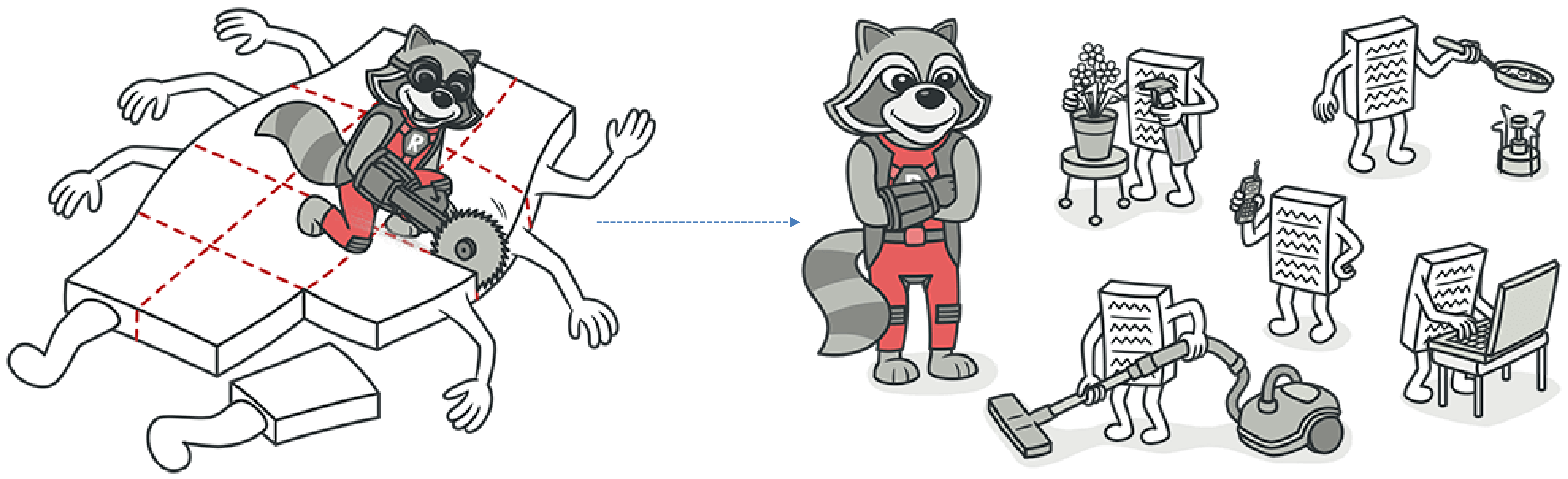
Solution

Then it's a good idea to separate the data into separate classes, ensuring connection and synchronization between the domain class and the GUI.



Payoff

- Refactoring of these classes spares developers from needing to remember a large number of attributes for a class.
- In many cases, splitting large classes into parts avoids duplication of code and functionality.



Short gun surgery

- If every time you make some change to the system, and you have to make many little changes all over the place to many different classes.

- Indicates that functionality is spread among different classes.

- Too much coupling and too little cohesion

- Solution: move method, inline class

Feature Envy

- Refers to a method that seems more interested in a class other than the one it belongs to.
- For example,
  - using lots of fields of another class
  - Is calling a lot of methods of the other class
- Solution:
  - Extract method, move method

How to treat Bad Code Smells

- Duplicate Code
  - Extract method
- Long method
  - Extract method, decompose conditionals
- Large class
  - Extract class
- Shotgun surgery
  - Inline class
- Feature envy
  - Extract method