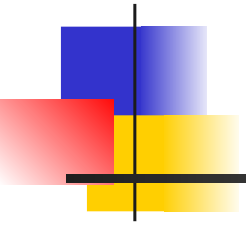


Refactoring Review

Name these refactorings

The *Refactoring Category* is shown at the bottom of each slide.

#1

BEFORE

```
def normalize(text):
  """Reformat some text"""
  text = text.trim()
  text = text.replace('_',
                      ' ')
  return text
```

AFTER

```
def normalize(text):
  """Reformat some text"""
  result = text.trim()
  result =
    result.replace('_',' ')
  return result
```

Refactoring Category: Composing Methods

#2. (two names for this refactoring)

BEFORE

```
def roots(a, b, c):
  """Roots of Quadratic"""
  if b*b - 4*a*c >= 0:
    x1 = (-b +
    sqrt(b*b-4*a*c))/(2*a))
    x2 = (-b -
    sqrt(b*b-4*a*c))/(2*a))
    return (x1, x2)

  return None
```

AFTER

```
def roots(a, b, c):
  """Roots of Quadratic"""
  descrim = b*b - 4*a*c
  if descrim >= 0:
    descrim = sqrt(descrim)
    x1 = (-b + descrim)/(2*a)
    x2 = (-b - descrim)/(2*a)
    return (x1, x2)

  return None
```

#3

BEFORE

```
def find(text: str):
  """Find text in file"""
  found = False
  line = None
  file = open("somefile")
  while not found:
     line = file.readline()
     if text in line:
        found = True
  file.close()
  return line
```

AFTER

```
def find(text: str):
  """Find text in file"""
  with open("somefile")
       as file:
     for line in file:
        if text in line:
           return line

  return None
```

Simplifying Conditional Expressions

(many students wrote code like this in ISP)

#4

BEFORE

```
title = rental.get_movie()\
                .get_title()
```

AFTER

```
title = rental.get_title()


Movie still has get_title()
```

Moving Features Between Objects

#5

BEFORE

```
first = 'Bill'
last  = 'Gates'
email = 'bill@msft.com'

print_person(
    first, last, email)

def print_person(*args):
   print(f"{args[0]}
            {args[1]}
    email <{args[2]}>")
```

AFTER

```
@dataclass
class Person:
    first: str
    last: str
    email: str
p = Person("Bill","Gates",...)
print_person(p)

def print_person(person):
   print(f"{person.first}
            {person.last}
   email <{person.email}>")
```

Simplifying Method Calls

#6

BEFORE

```
def print_rental(title,
        days_rented, price):

   print("{:20s} {:6d} {:f}"
         .format(title,
             days_rented,
             price)


r = Rental("Frozen", 3)
print_rental(r.get_title(),
    r.get_days_rented(),
    r.get_price())
```

AFTER

```
def print_rental(r: Rental):

    print("{:20s} {:6d} {:f}"
       .format(
           r.get_title(),
           r.get_days_rented(),
           r.get_price())


r = Rental("Frozen",3)
print_rental(r)
```

Simplifying Method Calls

#7

BEFORE

```
def vote(question, choice):
  if not question.can_vote():
    messages.error(
        "voting not allowed")
  else:
    if choice not in
      question.choice_set():
      messages.error(
          "invalid choice")
    else:
      Vote.objects.create(
           user=user, question=...)
      return redirect('polls:result')
  # if any error, redirect to detail
  return redirect('polls:detail',...
```

AFTER

```
def vote(question, choice):
  if not question.can_vote():
    messages.error(
        "voting not allowed")
    return redirect('polls:detail',..
  if choice not in
     question.choice_set():
     messages.error(
         "invalid choice")
    return redirect('polls:detail',..

  Vote.objects.create(
       user=user, question=...)
  return redirect('polls:result',...)
```

Simplifying Conditional Expressions

#8 (two possible answers)

BEFORE

```
def greet(firstname):
  if datetime.now().hour<12:

     print("Good morning",
           firstname)
  else:
     print("G'd afternoon",
           firstname)
```

AFTER

```
def greet(firstname):
  if is_morning():
     print("Good morning",
           firstname)
  else:
     print("G'd afternoon",
           firstname)

def is_morning():
  return \
    datetime.now().hour < 12
```

1. Simplifying Conditional Expressions
2. Composing Methods

#9

BEFORE

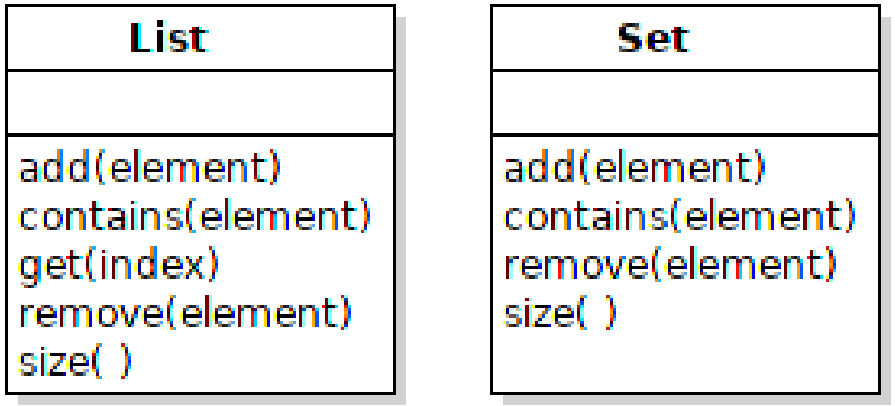
```
game = Game(800, 600)
```

AFTER

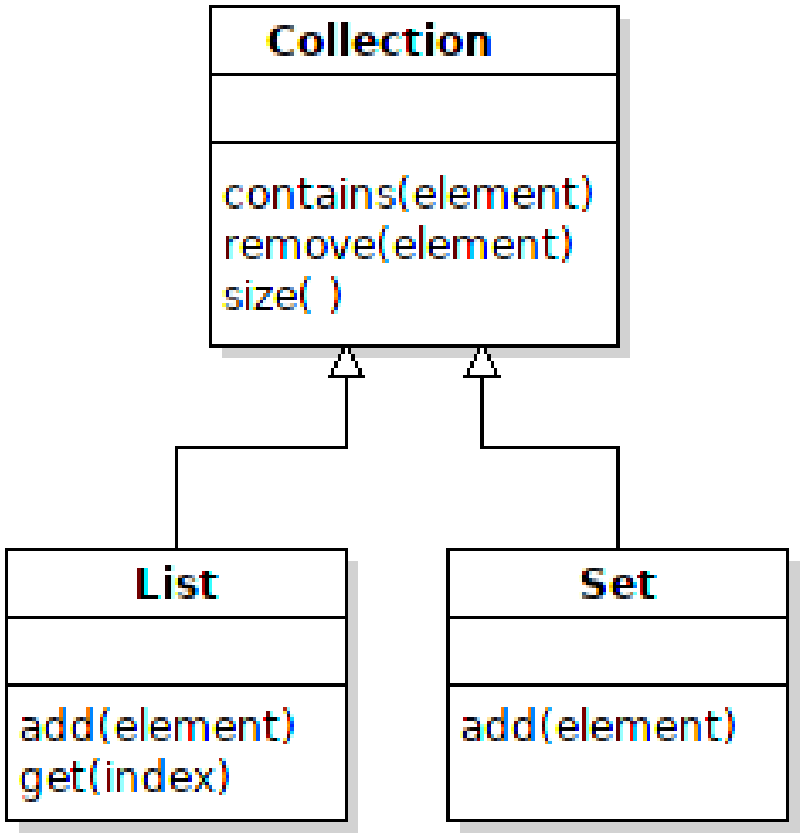
```
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600

game = Game(CANVAS_WIDTH,
            CANVAS_HEIGHT)
```

#10

BEFORE



AFTER

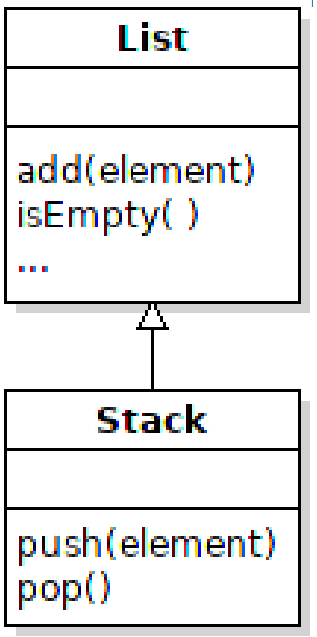


Dealing with Generalization

Why not move `add(element)` to Collection, too?

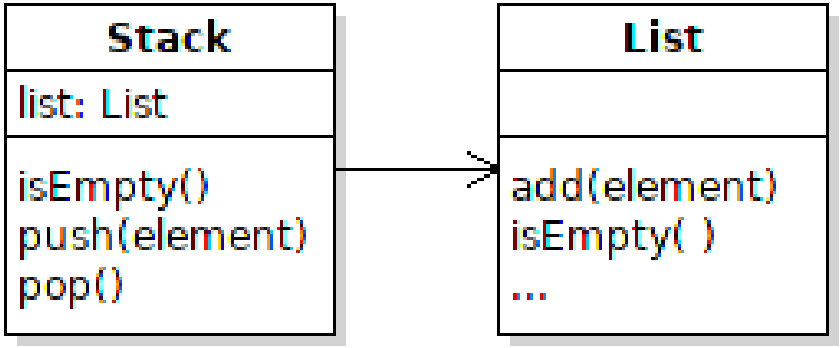
#11

BEFORE




```
class Stack(List):
  def push(self, e):
      super().append(e)
```

AFTER




```
class Stack:
  def push(self, e):
      self.list.append(e)
```

Dealing with Generalization

Why Not Stack extends List?

O-O Basics:

- A Stack *is not* a List. Fails the "*is a*" test.
- Liskov Substitution Principle - *can't substitute Stack for List*

Design Principles:

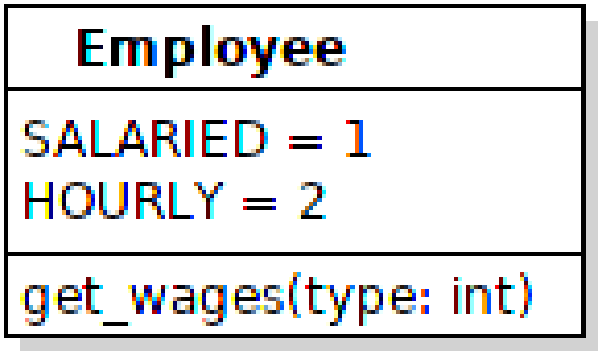
- Prefer *Composition over Inheritance*, also called
- *Prefer Delegation over Inheritance*

Code Symptom:

- *Refused Bequest* - Stack doesn't use most List methods

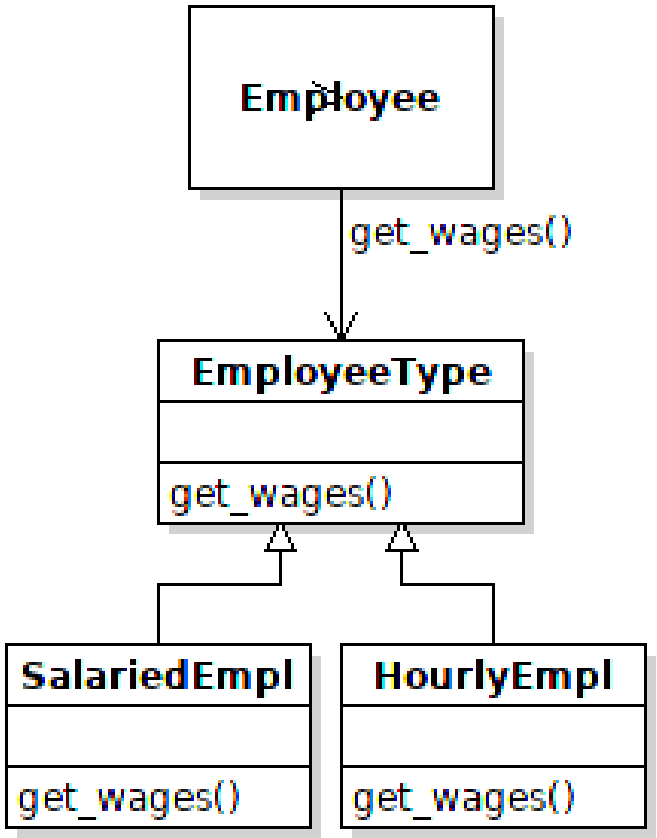
#12 (two names for this refactoring)

BEFORE




```
def get_wages(self, type):
    if type == SALARIED:
        # return salary
    elif type == HOURLY:
        # return wage*hours
```

AFTER



1. Organizing Data

2. Simplifying Conditional Expressions

#13 (Two Refactorings)

BEFORE

```
# Movie rental
def get_price(days: int):
   if type == NEW_RELEASE:
      price = 3*days
   elif type == CHILDREN:
      price = 1.5 +
         1.5*max(0, days-3)
   else:
      price = ...
   return price
```

AFTER

```
class Rental:
   days: int
   price_code: PriceCode

   def get_price(self):
     return self.price_code.\
        get_price(self.days)

class PriceCode(ABC):
   pass
class NewRelease(PriceCode):
   def get_price(self, days):
      return 3*days
```

1. Organizing Data, 2. Simplifying Conditional Expressions

#14

BEFORE

```
SPADES = 1
HEARTS = 2
CLUBS  = 3
DIAMONDS = 4

class Card:
  def __init__(self, value,
               suite: int):
      ...

c = Card(4, HEARTS)
```

AFTER

```
class Suite(Enum):
   SPADES = 1
   HEARTS = 2
   CLUBS  = 3
   DIAMONDS = 4

class Card:
   def __init__(self, value,
                suite: Suite):
       ...
c = Card(4, Suite.HEARTS)
```

Organizing Data, but different from #11.

Many Students Wrote (copied) This

```
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        user = request.user
        choice_id = request.POST['choice']
        selected_choice = question.choice_set.get(pk=choice_id)
    except (KeyError, Choice.DoesNotExist):
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice."} )
    else:
        try:
            vote = Vote.objects.get(user=user, choice__question=question)
            vote.choice = selected_choice
            vote.save()
        except Vote.DoesNotExist:
            Vote.objects.create(user=user, choice=selected_choice).save()
        return redirect('polls:results", args=...)
```

Replace Conditional with Guard Clauses

Consider: Replace Exception with Test

1. Simplify Conditional Expression, 2. Simplify Method Calls

Justify Your Refactorings

For each refactoring, you should be able to explain:

- Benefits
- Be specific

Avoid vague claims like "*easier to ...*".

Instead, state why and how something is "*easier*".

Extract Method

Benefits:

- increase opportunity to reuse code and eliminate duplicate code
- make method *logic* easier to understand, which reduces errors and improves maintainability
- by reducing the amount of work a method is doing, it gets closer to the goal of "1 method does 1 thing", and make make for more descriptive method name