University of
South Australia

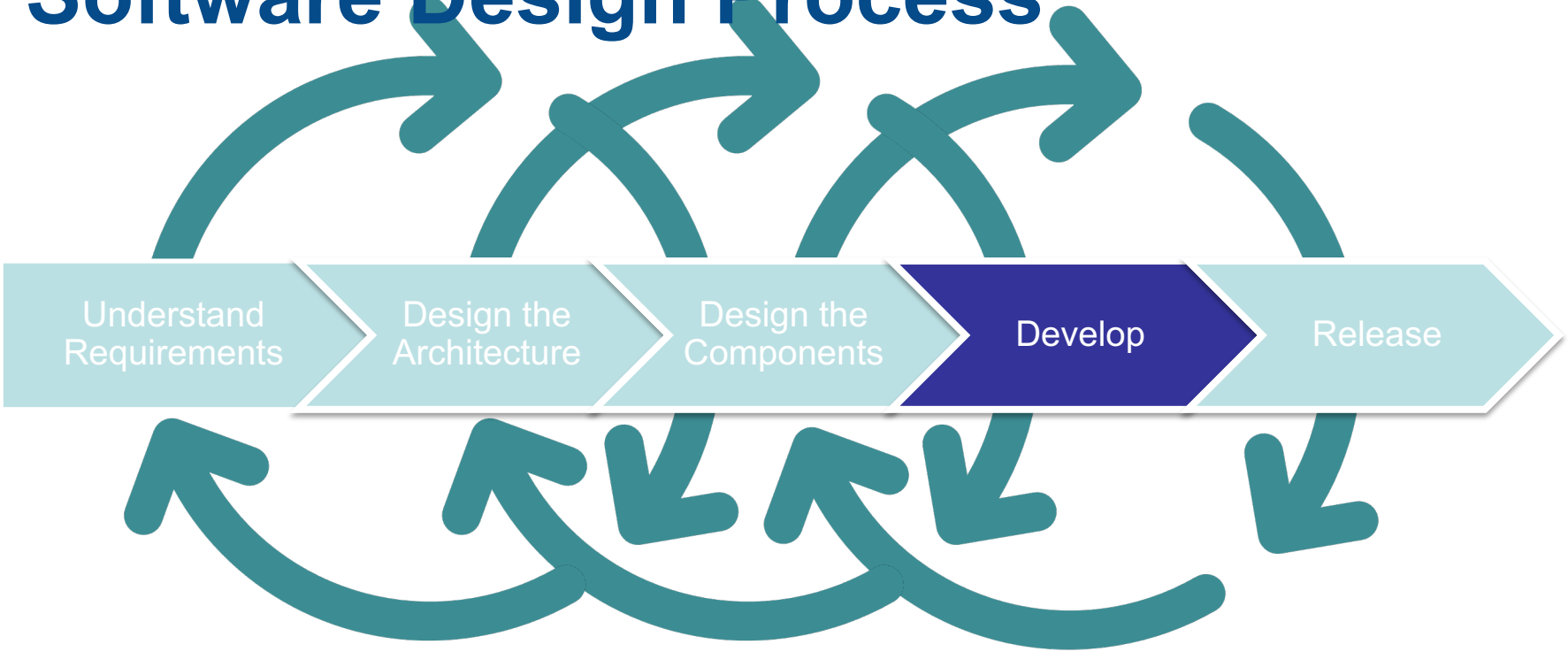
INFS 2044

Week 13
Refactoring

Learning Objectives

- Detect potential design problems in source code (CO4)
- Use automated tools to check adherence to coding standards (CO6)
- Improve code quality through refactoring (CO4)

Software Design Process



Refactoring

- Improving the design of the code after it has been written
- No changes in observable functionality

Refactoring Benefits

- Makes software easier to understand
- Helps find bugs
- Can make your program faster (sometimes)

When to Refactor?

- All the time in little bursts
- Rule of Three
  - The first time you do something, you just do it
  - The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway
  - The third time you do something similar, you refactor.

When to Refactor

- When you add a function
- When you need to fix a bug
- As you do code review

When NOT to Refactor

- The code is in such a mess that it may be faster to start from the beginning
  - Caution: don’t rewrite code just because you would do it differently
- Close to a deadline
  - Productivity gain from refactoring would appear after the deadline

Prerequisites

- Having a solid test suite is essential for refactoring
  - Detect any problems that you may have introduced during refactoring
- Version control
  - So that you can undo your changes easily if they turn out to be unsuccessful.

Code Smells

- Heuristic rules for detecting problems
- Rules of thumb that help detect issues and improve the code
- Use as guidelines when writing code
- Smells trigger refactoring efforts

Code Smells

- Duplicated code
- Long methods
- Large classes
- Long parameter list
- …

Design Smells

- “Anti-Patterns”
  - Structures that reflect common problems in software designs
  - Knowing them help avoid problems early

- Circular dependency
- God object
- Many more
  See e.g. https://en.wikipedia.org/wiki/Anti-pattern

Refactoring: Extract Method (1a)

```
for entry in entries:
  values = [int(value) for value in entries.split(':’)]
  result = sum(values)
  print(f'The sum is {result}')
```

Refactoring: Extract Method (1b)

```
for entry in entries:
  values = parse_entry(entry)
  result = calculate_result(values)
  print(f'The sum is {result}')

def parse_entry(entry):
  values = [int(value) for value in entry.split(':')]
  return values
```

Refactoring: Extract Method (2)

```
if date >= SUMMER_START and date <= SUMMER_END:
  charge = quantity * _summer_rate
else:
  charge = quantity * _winter_rate

if isInSummer(date):
  charge = charge_summer_rate(quantity)
else:
  charge = charge_winter_rate(quantity)
```

Refactoring: Extract Method (3)

```
for entry in entries:
  values = parse_entry(entry)
  result = calculate_result(values)
  print(f'The sum is {result}’)
  print('-----')
```

```
for entry in entries:
values = parse_entry(entry)
result = calculate_result(values)
print_result(result)
```

```
def print_result(value):
  print(f'The sum is {value}’)
  print('-----')
```

Refactoring: Remove Duplication (1)

```
def func1(shape, point):
  vec = shape.center - point
  dsq = vec.x*vec.x + vec.y*vec.y
  if dsq < SELECTION_THRESHOLD:
    delete_shape(shape)

def func2(shape, point, color):
  vec = shape.center - point
  dsq = vec.x*vec.x + vec.y*vec.y
  if dsq < SELECTION_THRESHOLD:
    color_shape(shape, color)
```

```
def dist_sq(point_a, point_b):
  vec = point_a - point_b
  dsq = vec.x*vec.x + vec.y*vec.y
  return dsq

def func1(shape, point):
  dsq = dist_sq(shape.center, point)
  if dsq < SELECTION_THRESHOLD:
    delete_shape(shape)

def func2(shape, point, color):
  dsq = dist_sq(shape.center, point)
  if dsq < SELECTION_THRESHOLD:
    color_shape(shape, color)
```

Refactoring: Remove Duplication (1)

```
def dist_sq(point_a, point_b):
  vec = point_a - point_b
  dsq = vec.x*vec.x + vec.y*vec.y
  return dsq

def func1(shape, point):
  dsq = dist_sq(shape.center, point)
  if dsq < SELECTION_THRESHOLD:
    delete_shape(shape)

def func2(shape, point, color):
  dsq = dist_sq(shape.center, point)
  if dsq < SELECTION_THRESHOLD:
    color_shape(shape, color)
```

```
def isSelected(shape, point):
  dsq = dist_sq(shape.center, point)
  return dsq < SELECTION_THRESHOLD

def func1(shape, point):
  if isSelected(shape,point):
    delete_shape(shape)

def func2(shape, point, color):
  if isSelected(shape,point):
    color_shape(shape, color)
```

Refactoring: Remove Duplication (2)

```
class A:
  def m1(self):
    print("Header")
    print("Content")
    print("Footer")

class B:
  def m2(self):
    print("Header")
    print("Some other content")
    print("Footer")
```

```
class Super:
  def m(self):
    print("Header")
    print_content()
    print("Footer")
  def print_content():
    """template method"""
    pass
class A(Super):
  def print_content():
    print("Content")
class B(Super):
  def print_content():
    print("Some other content")
```

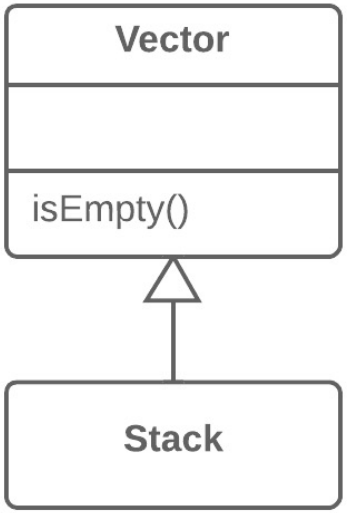
Refactoring: Separate Query from Update

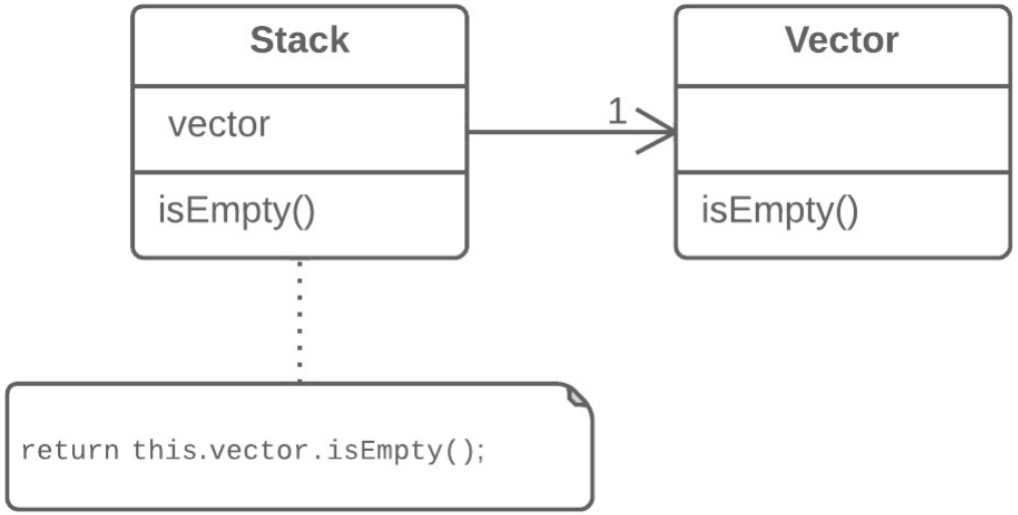
```
def getTotalAndSetReadyForBilling():
  ...
  return total


def getTotal():
  return getTotal
def setReadyForSummary():
  ...
```

Refactoring: Parameter Objects

```
def shipTo(appt, num, street, suburb, postcode, country):
  ...


class Address:
  def __init__(appt, num, ..., country)
    ...

  def shipTo(address):
    ...
```

Refactoring: Delegation





https://refactoring.guru/replace-inheritance-with-delegation

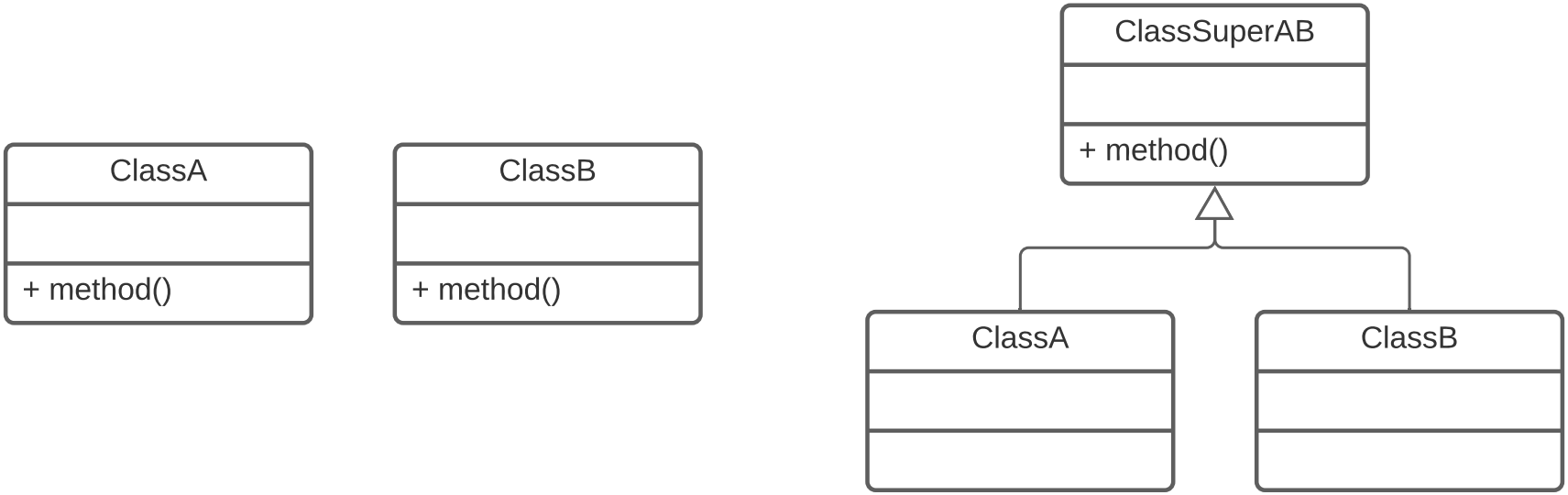
Refactoring: Hide Delegate

```
address = client.getPerson().getEmail()
address.send_message("Hello")


client.send_message("Hello")
```

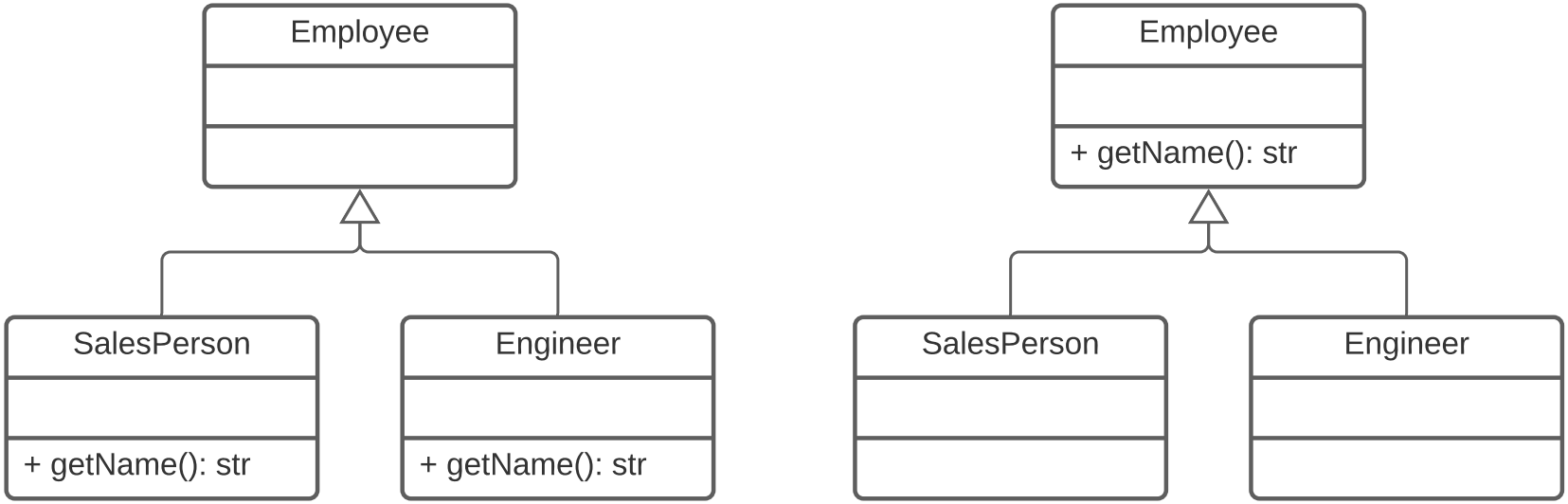
Refactoring: Encapsulate Members

```
class Student:
  def get_courses(self):
    return self.courses

class Student:
  def get_courses(self):
    return frozenset(self.courses)
  def add_course(self, course):
    self.courses.add(course)
  def remove_course(self, course):
    del self.courses.remove(course)
```

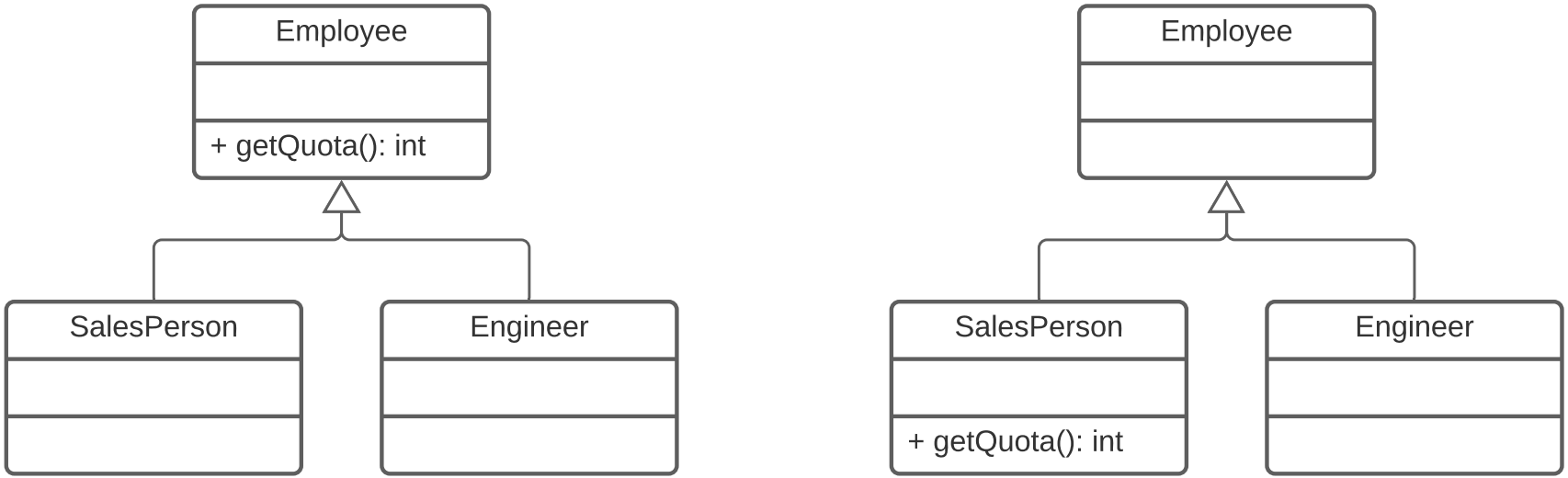
Refactoring: Extract Superclass



Refactoring: Pull Up Field/Method



Refactoring: Push Down Field/Method



Coding Standards

- Naming conventions
- Formatting (spaces, indent, line breaks, line length)
- Parenthesis (brace) placement
- Comment formatting

Code Checking

- Automated tools can help detect poor style & patterns
  - Pylint
  - Some IDEs have built-in checks

Pylint Example

```
def my_sum(val1,val2,val3,val4):
  return val1+val2+val3+val4

def testFunction(VALUE1, value2, value3, value4, value5):
  total = my_sum(VALUE1, value2, value3, value4, value5)
  if total == 5:
    print('High five!’)
  else:
    print('Better luck next time!')
```

Pylint Example

```
% pylint ex.py

************* Module ex
ex.py:1:0: C0114: Missing module docstring (missing-module-docstring)
ex.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)
ex.py:4:0: C0103: Function name "testFunction" doesn't conform to snake_case naming style (invalid-name)
ex.py:4:0: C0103: Argument name "VALUE1" doesn't conform to snake_case naming style (invalid-name)
ex.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)
ex.py:5:12: E1121: Too many positional arguments for function call (too-many-function-args)


-----------------------------------
Your code has been rated at -4.29/10
```

Summary

- Refactoring makes code easier to understand and easier to modify
- Preparation for subsequent extension of the code
- Requires comprehensive test suite to validate correctness
- Coding standards help humans and tools process source code more easily

Activities this Week

- Read the required readings
- Participate in Practical 4
- Complete Assignment 2

University of
South Australia