University of
South Australia

INFS 2044

Week 10
Maintainable Code

Learning Objectives

- Understand how the remaining weeks will be organized
- Explain the need for maintainable code (CO4)
- Understand the principles of maintainable code (CO4)

Learning Arrangements

- Lecture (1-2 hours per week)
- Practical (2 hours)
- Assignment 2
- Continuous assessment (4 quizzes)

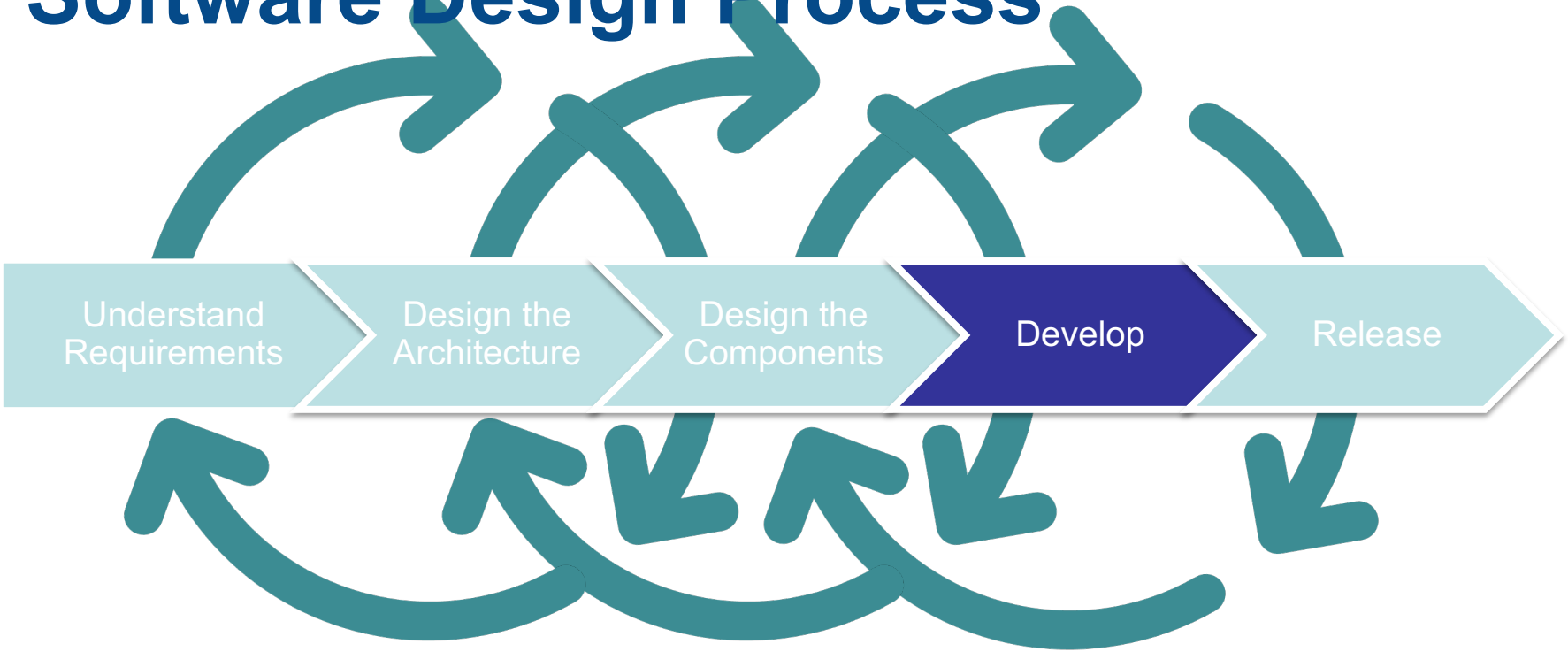
Teaching Team Update

- Mr Mateusz Kowalski
- Ms Siaw (Mei) Sim
- Mr Dhenesh Subramanian
- Ms Amali Weerasinghe
- Billy Bizilis

Assignment 2

- Individual Assignment, due Monday 14 June 2021
- 40% of course total marks
- Python programming
  - Design an implementation & document it
  - Write code
  - Test
  - Reflect

Software Design Process



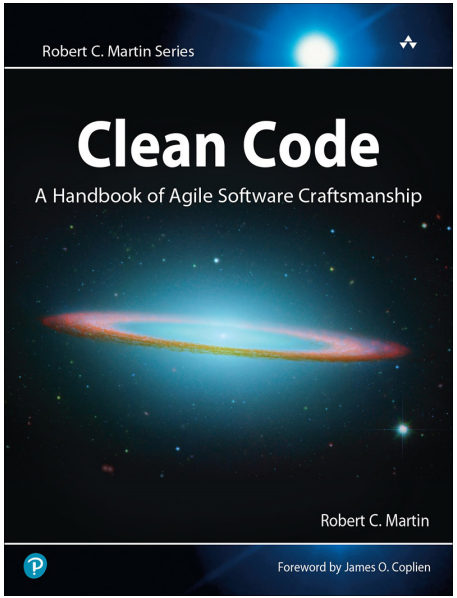
Software Design Recap

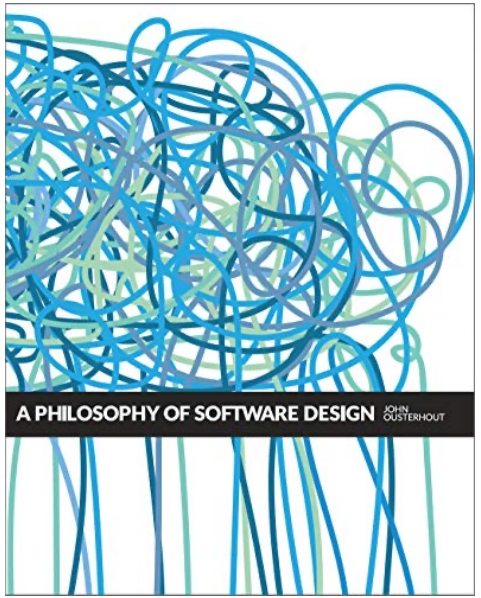
- Decomposition
- Interface Design
- Interaction & Implementation Design
- Design Patterns

Topics for Next Weeks

- Principles of good code (today)
- Testing
- Naming and Commenting
- Refactoring and code smells

Resources


Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Foreword by James O. Coplien


A PHILOSOPHY OF SOFTWARE DESIGN
JOHN OUSTERHOUT

Why Good Code?

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

Martin Fowler, 2008.

Good Code Characteristics

- Human readable
- Easy to change and maintain
  - Solid, not rigid
  - Flexible, not fragile
  - Reusable, not ephemeral

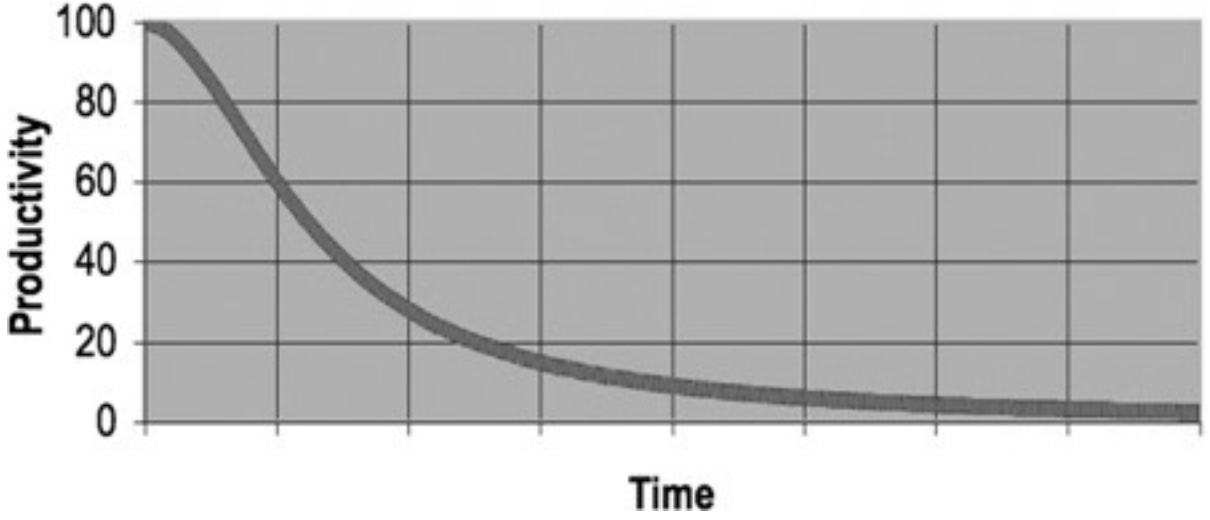
Signs of Good Code

- Self-documenting
- Simple
- Readable
- Expected
- Maintainable
- Minimal dependencies
- Testable and tested
- Efficient
- Expressive
- Elegant

Signs of Bad Code

- Inappropriate dependencies, coupling
- Incorrect behaviour and boundaries
- Duplication
- Mixes levels of abstraction
- Comments: obsolete, redundant, or poorly written
- Dead code, commented out code

Impact of Bad Code



Why Care?

- Coding is hard and expensive
- Productivity decreases over time
- Bad code quickly becomes a legacy mess
- Rewriting all code some time in the future is not a solution

What Can be Done?

- Software design
  - Patterns, reuse, embrace design principles (S.O.L.I.D.), test
- Coding conventions
  - Naming things consistently
- Good habits
  - Continually improve code, don’t be clever

Clean Code Principles

- The Boy Scout Rule
- Meaningful naming
- Constants instead of hard-coded strings/numbers
- Small Functions

R.C. Martin, Clean Code

Boyscout Rule

- Always leave the campground cleaner than you found it.

- Always leave the code you are editing a little better than you found it. (Robert C. Martin)

Meaningful Names

- Descriptive, intention-revealing
- Words, not abbreviations
  - Better long than ambiguous
- Consistent
- Searchable

Simple Implementation Design

1. Runs all the tests
2. Contains no duplication
3. Expresses the intent of the programmer
4. Minimizes the number of classes and methods

(Kent Beck)

Good Software Design Principles

Coupling is Dangerous

- Coupling is a measure of how closely related code is linked
- Tight coupling hampers understanding, independent reuse, and containment of changes
- It is best to have code that is loosely coupled

SOLID Principles

General Software Design Principles

- Single Responsibility
- Open Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Single Responsibility Principle

- A class should have only one reason to change.
- Separation of Concerns
- Break up “large” functions/classes, use design patterns

Open-Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Use interfaces and polymorphism to add functionality rather than rely on repeatedly changing the code

Liskov Substitution Principle

- Objects of a supertype shall be replaceable with objects of its subtype without breaking the program

- Design the interface so that subtypes can guarantee the promised pre- and post-conditions
- Avoid depending on implementation details

Interface Segregation Principle

- Clients should not be forced to depend on methods that they do not use.

- Define interfaces/classes that are cohesive
- Split large interfaces/classes

Dependency Inversion Principle

- High-level modules should not depend on low-level modules.
  Both should depend on abstractions.
- Abstractions should not depend on details.
  Details should depend on abstractions.

- Code to interfaces, not concrete implementations
- Objects should not create their collaborators

Design Principles for Functions & Methods

Small Functions

- Each function should **do one thing** completely and do it well
- One level of abstraction
- Easier to understand, modify, reuse, avoid duplication, test

Good Functions

- Small
- Do one thing
- Do it well
- Do it only (no side-effects)
- One level of abstraction
- Reads top-down like a story
- Few arguments are better
  - Use data classes or instance variables to pass data

Do One Thing Only

- public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
  includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
  }

- “TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.”

Do One Thing Only (cont.)

- If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing.
- Functions decompose a larger concept (the name of the function) into a set of steps at the next level of abstraction.

Top-Down Narratives

- Code should read like a top-down narrative
  - “TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns
  - TO include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.
  - TO include the setups, we include the suite setup if this is a suite, then we include the regular setup.
  - To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an include statement with the path of that page.
  - To search the parent…

One Level of Abstraction

- All statements in a function should be at the same level of abstraction
  - getHTML() and path parsing, rendering elements, adding line breaks, etc are at different levels of abstraction
  - Do not mix the details with the higher level abstractions in a function

Naming Functions

- Describe WHAT the function does, not how it does it
- Long descriptive name is better than a short ambiguous name and a long explanatory comment
  - renderPageWithSetupsAndTeardowns() vs. testableHTML()
- Use a naming convention
- Be consistent with naming (verbs, nouns, phrases) at all levels (function, classes, modules)

Command-Query Separation

- Functions should either **do something** *or* **answer something**, but *not both*.

```
def set(attribute, value):
    …

if set(”name”, ”bob”): …
```

```
if attributeExists(”username”):
    setAttribute(”name”, ”bob”)
```

No Side-Effects

- Your function promises to do one thing, but it also does other *hidden* things (the “side-effect”)
  - Change instance variables, argument objects, global variables
  - checkPassword(userName, password) also initializes the session. This creates a temporal coupling, as the function can only be called when it is safe to initialize the session.
  - Could rename to checkPasswordAndInitializeSession() but that violates the “do one thing” principle.

Error Handling

- Prefer structured exception handling to returning error codes.
  - Need to check error codes is error prone and leads to repeated error handling code.
  - If error handling is complex, create a separate function for handling error. Removes complexity from the successful path and avoids repetition.

Error Handling Example

```
if doA() == SUCCEESS:
    if doB() == SUCCESS:
        if doC() == SUCCESS:
            ...
        else:
            handleError()
    else:
        handleError()
else:
    handleError()
```

```
try:
    doA()
    doB()
    doC()
    ...
except MyError as err:
    handleError(er)
```

Don’t Repeat Yourself (DRY)

- Duplicated code adds complexity
  - We must detect and change all occurrences as the code evolves
  - Duplication bloats the code
  - Cognitive effort expended on understanding the code (again)

Duplication Example

```
class List:
    def append(self, item):
        self.items.append(item)
        self.emptyFlag = False
        self.itemCount += 1

    def isEmpty(self):
        return self.emptyFlag

    def getSize(self):
        return self.itemCount
```

```
class List:
    def append(self, item):
        self.items.append(item)
        self.itemCount += 1

    def isEmpty(self):
        return self.size() == 0

    def getSize(self):
        return self.itemCount
```

Removing Duplication

- Create a function or class containing the common code
  - Abstract base class
  - Template methods
  - Function parameters
  - Change the algorithm

Design Principles for Classes

Objects vs Data Structures

- Objects hide their data behind abstractions and expose functions that operate on that data.
  - Easy to add new classes
  - Difficult to add functions (all classes must change)
- Data structures expose their data and have no meaningful functions.
  - Easy to add new functions
  - Hard to add new data structures (all functions must change)

Information Hiding

- Objects should hide their implementation details
- All manipulation of state should be achieved through methods
  - Instance variables are private
  - Careful not to leak information through getters and setters
  - Methods ensure integrity of the object by enforcing invariants and business rules

Law of Demeter (LoD)

- An object should assume as little as possible about the structure or properties of anything else (including its subcomponents)
  - principle of "information hiding”

Law of Demeter – More Detail

- A method m of a class C should only call the methods of these:
  - Class C
  - An object created by m
  - An object passed as an argument to m
  - An object held in an instance variable of C
- m should *not* invoke methods on objects that are returned by any of the allowed methods. “talk to friends, not to strangers.”

Violation of LoD

- outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath()

- better?:

  opts = ctxt.getOptions()
  scratchDir = opts.getScratchDir()
  outputDir = scratchDir.getAbsolutePath()

Violation of LoD (cont.)

- outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath()
- better?:

  ctxt.getAbsolutePathOfScratchDirectoryOption()

  ctx.getScratchDirectoryOption().getAbsolutePath()

LoD: Objects Do Something

- outputStream = ctxt.createScratchFileStream(classFileName)

- Hide the internals of ctxt
- Avoids caller having to navigate the internals of ctxt
- We can now change the implementation of ctxt without affecting its clients

Boundaries

- sensors = dict() # map sensorID -> Sensor object
  …
  sensor = sensors[id]

- If the map is exposed to callers, it will be difficult to change, and callers may be able to access functions we don’t want/need (e.g. modify the dict)

Boundaries (cont.)

- Hide the map as it is considered an implementation detail

```
class Sensors:
    def __init__(self):
        self.sensors = dict()
    def getById(self, id):
        return self.sensors[id]
```

- sensor = sensors.getById(id)

Class Should be “Small” Enough

- Classes should be “small”
  - Single responsibility principle
  - Cohesion
- Split classes to reduce responsibilities and improve cohesion
- Purpose of classes is to organize complexity
  - Helps find and understand relevant parts of the code
  - May help re-use

Designing for Change

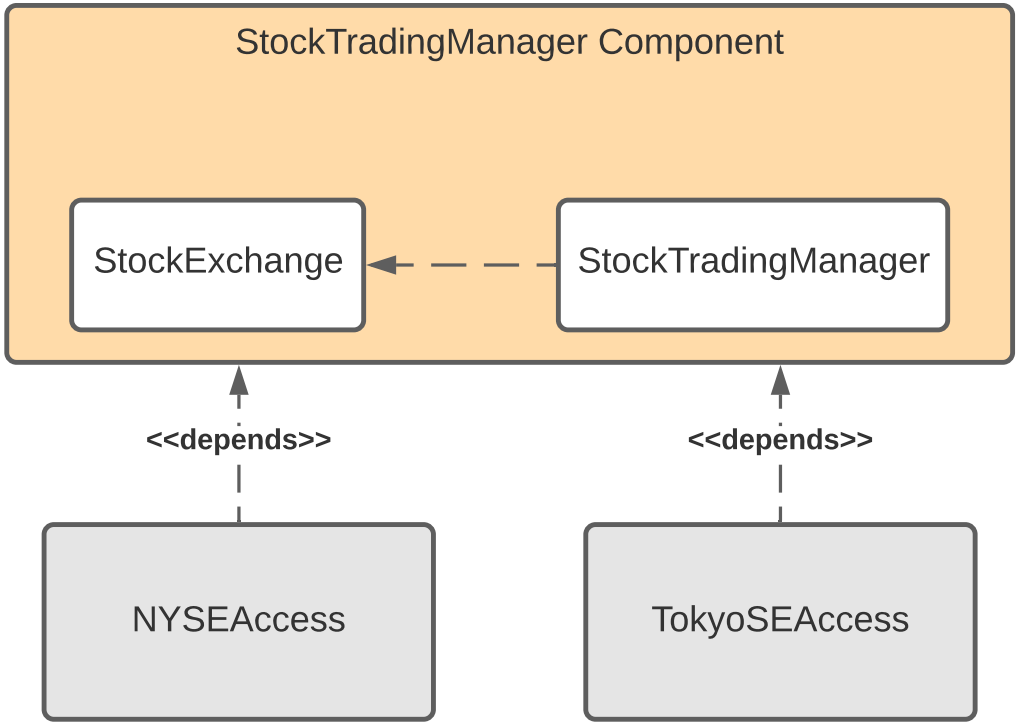
- Reduce risk induced by change
  - What changes will likely be required?
  - Avoid having to “open” the class to make modifications (OCP)
  - Avoid depending on concrete implementations (DIP)
- Find the stable & common abstractions, then
- Define interfaces, and
- Build classes around them

Code to Interface Example

```
class StockTradingManager:

    def __init__(self, exchange):
        self.exchange = exchange

    def analyseStockTrend(self, stockSymbol):
        prices = self.exchange.getPrices(stockSymbol, ...)
        ...
```

```
class StockExchange(ABC):
    @abstractmethod
    def getPrices(self, symbol):
        pass

class NYStockExchange(StockExchange):
    def getPrices(self, symbol):
        ...

class TokyoStockExchange(StockExchange):
    def getPrices(self, symbol):
        ...
```

Code to Interface Example



Separate Construction from Use

- Separation of Concerns
  - the startup process, where the application objects are constructed and the dependencies are “wired” together, from
  - the runtime logic that takes over after startup.
- Objects should not construct their parts/collaborators

Counterexample: Lazy Initialization

- def getService(self):
    if self.service is None:
        self.service = **MyServiceImpl**(…)
    return self.service

- Introduces a dependency on the concrete implementation
- The method has two responsibilities
  - creating the object, and
  - knowing the current service object

Dependency Injection

```
class Foo:
  def __init__(self, service):
      self.service = service

  def getService(self):
      return self.service
```

```
def main():
    myService = MyServiceImpl(…)
    foo = Foo(myService)
    …
```

Summary

- Maintainable code is human-readable and easy to change
- Code should be designed and organised to minimise complexity and cognitive effort
- Design principles and heuristics help identify “code smells” and improve the code
- Continuous effort must be invested in maintaining code quality

Activities this Week

- Read the required readings
- Participate in Practical 1
- Complete Quiz 6

University of
South Australia