Advanced Software Development: Introduction

CPSC 501: Advanced Programming Techniques
Winter 2025

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Wednesday, January 8, 2025

UNIVERSITY OF CALGARY

Advanced Software Development

- The focus of this topic is 5 modern fundamental software development skills.

1. Git (version control)
2. JUnit (unit testing)
   - Verify code correctness before commit (or on commit attempt with CI/CD tools)
3. Refactoring (improving code iteratively)
   - A skill you've employed but possibly never formally framed as a concept
4. Docker (containerization)
   - A now common tool to managing consistent system images for development and deployment purposes
5. CI/CD (continuous integration/development)
   - Advanced Gitlab/GitHub/etc. tools that all you to compile and deploy committed code automatically to external users

Version Control and Refactoring

Let’s start with a motivating story

Once upon a time …

- **A consultant visited a development project.**
- There was a class hierarchy at the centre of the system.
- **The consultant saw that it was rather messy.**

1. The higher-level classes made certain **assumptions**
2. Super class code **didn't suit all** the subclasses.
3. If superclasses were modified, then **less overriding** would be necessary.
4. In other places the **intention** of the superclass had been **misunderstood**.
5. In yet other places several subclasses did the **same thing**.

I have some ideas!

I have some ideas!

- **This code can be looked at and cleaned up!**
- **BUT the project management didn't seem enthusiastic.**
- The code seemed to work and there were considerable **schedule pressures**.
- The managers said they would get around to it at some **later** point.
- **The programmers initially agreed**.

I have some ideas!

- **The programmers initially agreed**.

- A **second set of eyes** revealed unconsidered issues.

- **It wasn't really their fault.**
- They spent some time to clean it up.
- They **removed** half the code but **the functionality remained.**
- It was now **quicker and easier to make future changes.**

What are you thinking?

What are you thinking?

- The project **management** was **displeased**.
- **Schedules** were **tight** and there was a lot of work to do.
- These two programmers had spent two days doing work that had **done nothing to add** the many **features** the system had to deliver in a few months time.
- The **old code** had worked just **‘fine’**.

What are you thinking?

- The **old code** had worked just **‘fine’**.

- So the design was a bit more **‘pure’** a bit more **‘clean’**.
- The project had to ship code that **worked**, not **please an academic**.

- The consultant **suggested** that this cleaning up be done **elsewhere**.
- Such an activity might **halt the project for a week** or two.
- All this activity was devoted to making the code **‘look’ better**, **not** to making it do **anything new**.

Thoughts?

Thoughts?

1. How do you feel about this story?

2. Do you think the consultant was right to suggest further clean up?

3. Or do you follow that old engineering adage, "**if it works, don't fix it**"?

Thoughts?

1. How do you feel about this story?

2. Do you think the consultant was right to suggest further clean up?

3. Or do you follow that old engineering adage, "if it works, don't fix it"?

- Maybe the issue here wasn’t the concept but the **timing**?

Refactoring

What is it?

Refactoring: Basic Concepts

- Even a well designed software system '**decays**' as it is modified **over time**
  - Loses its design integrity as new features and fixes are added in an ad hoc way
  -
    - i.e. as changes are '**hacked**' in

Refactoring: Basic Concepts

- Refactoring **reverses** this **decay**
  - A bad/chaotic design can be improved with series of **small simple changes**
- Refactoring should be done during software maintenance:
  1. When fixing bugs
  2. When adding new feature
- Waiting until we reach the point in the story is actually bad practice.

Refactoring: Basic Concepts

- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**

Refactoring: Basic Concepts

- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**
  - **Disciplined process**: simple steps, defined start and end points, unambiguous

Refactoring: Basic Concepts

- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**
  - **Disciplined process**: simple steps, defined start and end points, unambiguous
  - **Internal structure**: we change the internal code but don't impact the connection layer to outer components or human user.
    - If you are changing the connection to externalities this is more than 'refactoring' it is a design change.

Refactoring: Basic Concepts

- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**
  - **Disciplined process**: simple steps, defined start and end points, unambiguous
  - **Internal structure**: we change the internal code but don't impact the connection layer to outer components or human user.
    - If you are changing the connection to externalities this is more than 'refactoring' it is a design change.
  - **Understand and maintain**: goal is code clarity for other readers of code, and ease of future changes

Refactoring: Basic Concepts

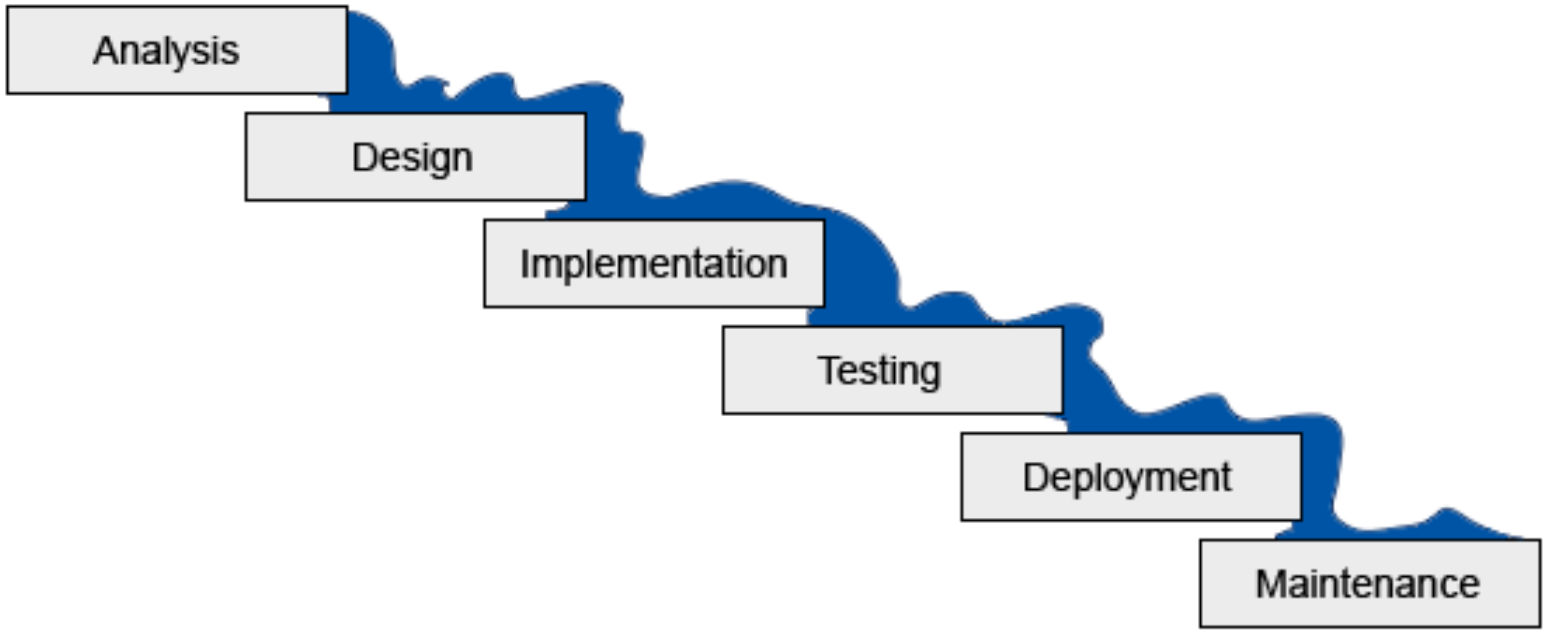
- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**
  - **Disciplined process**: simple steps, defined start and end points, unambiguous
  - **Internal structure**: we change the internal code but don't impact the connection layer to outer components or human user.
    - If you are changing the connection to externalities this is more than 'refactoring' it is a design change.
  - **Understand and maintain**: goal is code clarity for other readers of code, and ease of future changes
  - **External observable behaviour**: refactoring's result should not require anything external to code to be affected.

Refactoring: Basic Concepts

- Definition: **disciplined process** of changing the **internal structure of software** to make it easier to **understand** and **maintain**, without changing its **external observable behaviour**
  - Goal is to improve the design of the code after it is written and already functional.
  - Done in an orderly way to avoid introducing bugs .
  - Changes are made in small steps (**branch and merge**).
  - Every step is tested (usually with **unit tests**).
  - **Version control** allows us to undo a step.

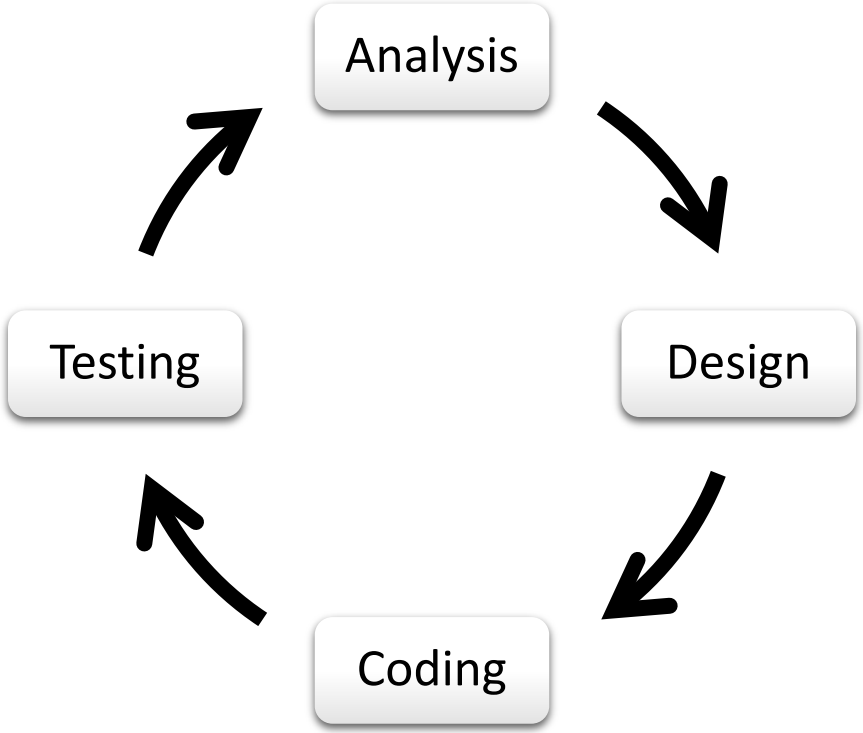
Refactoring: Non-iterative code lifecycles?

- Refactoring doesn't function well with non-iterative development techniques.
- In the traditional "waterfall" lifecycle, design precedes coding, and is never revisited:



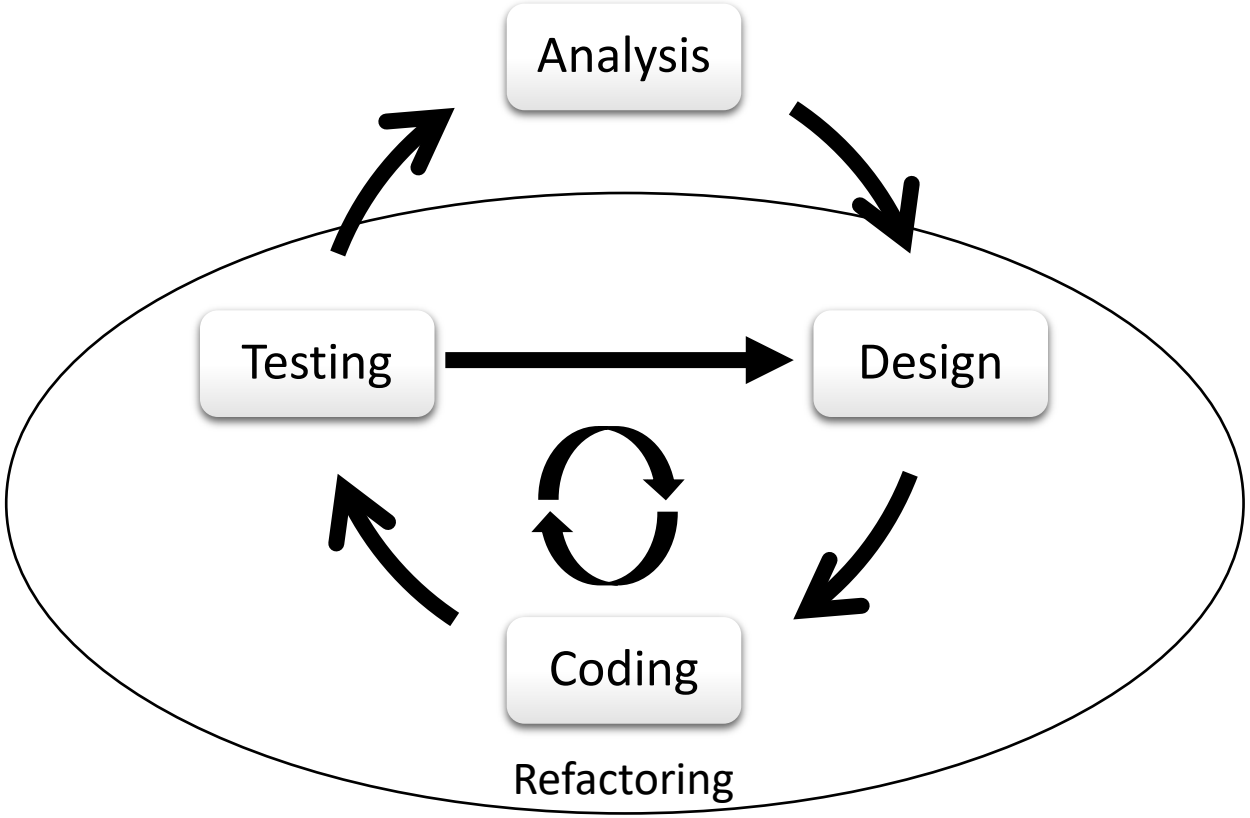
Refactoring: Iterative code lifecycles?

- With iterative development, design occurs continuously:



Refactoring: Iterative code lifecycles?

- Refactoring is a form of redesign that can be superimposed on the iterative lifecycle:



Jumping in

The simplest of examples … you've refactored before!

Refactoring: The simplest of examples

- How could this code be improved?

```
public class Employee {
   private String lastName;

   public void func1(String value) {
      lastName = value;
   }

   public static void main(String[] args) {
      Employee employee = new Employee();
      employee.func1("Smith");
   }
}
```

Refactoring: The simplest of examples

- Rename the method to better describe what it does:

```
public class Employee {
   private String lastName;

   public void func1(String value) {
      lastName = value;
   }

   public static void main(String[] args) {
      Employee employee = new Employee();
      employee.func1("Smith");
   }
}
```

```
public class Employee {
   private String lastName;

   public void setLastName(String value) {
      lastName = value;
   }

   public static void main(String[] args) {
      Employee employee = new Employee();
      employee.setLastName("Smith");
   }
}
```

Refactoring: The simplest of examples

- Rename the method to better describe what it does:
- This is the **Rename Method (Change Function Declaration)** refactoring (Fowler)
  - "The name of the method does not reveal its purpose"
  - Rename the violating method appropriately
  - Most IDEs have a shortcut to do this through-out codebase at once
  - **Danger in choosing a name already in use!**
  - **Compile** and **test** after change

Refactoring: The simplest of examples

Version control process:

1. make branch,
2. make change on branch,
3. test change on branch,
4. make merge request,
5. pass unit testing pipeline,
6. merge request approved into main branch

Jumping in

Structural example … ok maybe a bit more

Refactoring: A structural example

- How could this code be improved?

```
public class Employee {
   protected String lastName;
}

public class Clerk extends Employee {
   public String getLastName() {
      return lastName;
   }
}

public class Cashier extends Employee {
   public String getLastName() {
      return lastName;
   }
}
```

Refactoring: A structural example

- Move functionality from subclass to superclass.

```
public class Employee {
   protected String lastName;
}

public class Clerk extends Employee {
   public String getLastName() {
      return lastName;
   }
}

public class Cashier extends Employee {
   public String getLastName() {
      return lastName;
   }
}
```

```
public class Employee {
   private String lastName;

   public String getLastName() {
      return lastName;
   }
}

public class Clerk extends Employee {
}

public class Cashier extends Employee {
}
```

Refactoring: A structural example

- Move functionality from subclass to superclass.
- This is the **Pull Up Method** refactoring
  - “You have methods with identical results on subclasses”
  - Move them to superclass
  - Eliminates redundant code that is hard to maintain
  - Again test post change and generally done within version control iterative life cycle.

Well that’s easy?


THE
BUTTERFLY EFFECT

Refactoring: Risks

- Refactoring is risky
  - **“The butterfly effect”**
- Changes can create subtle bugs and changes often **cascade**
- Done improperly it can be like digging a hole you can’t escape from
- Refactor ‘**Systematically**’
  - Old style: make a backup, make a change, unit test, iterate
  - **Version control:** make a branch instead of backup, after it passes tests then merge

Onward to …
version control.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY