CE/CS/SE 3354 Software Engineering

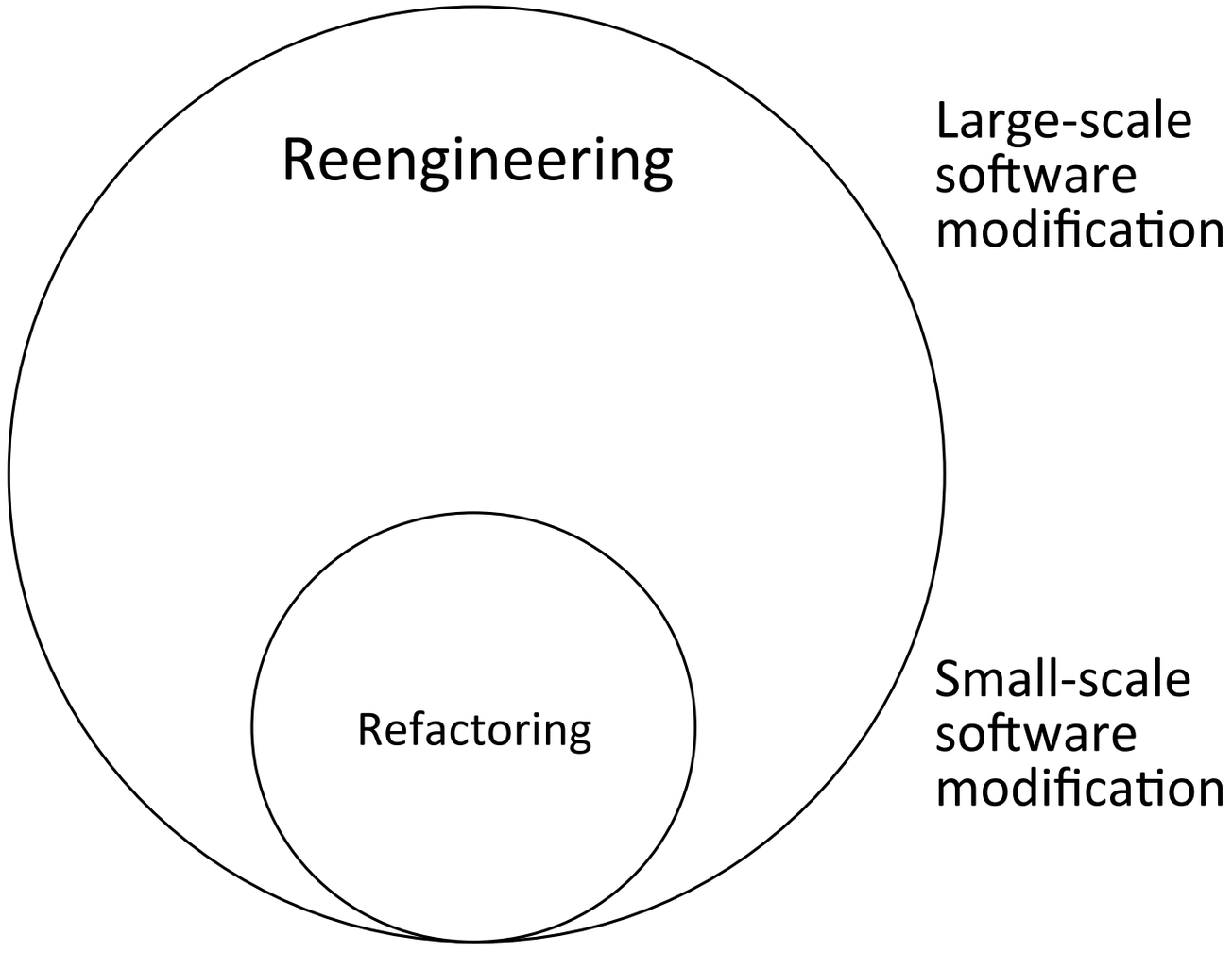
Software Refactoring

Refactoring definitions

- A program restructuring operation to support the design, evolution, and reuse of object oriented frameworks that preserve the behavioral aspects of the program [1]
- A process of changing the internal structure of software, not its observable behavior, in order to improve its internal quality [2]

[1] Opdyke, William F. ***Refactoring object-oriented frameworks***. Diss. University of Illinois at Urbana-Champaign, 1992.
[2] Fowler, M., **Refactoring: Improving the Design of Existing Code,** Addison-Wesley, 1999

Refactoring vs restructuring



Refactoring specifics

- Refactoring is a source to source transformation
- The language remains the same, e.g., Java to Java, C++ to C++
- It is originally designed for object-oriented languages, but can also be applied to non-object oriented language features (i.e., functions, procedures, databases, etc.)

Refactoring specifics

- Not the same as "cleaning up code" (which may cause changes to the behavior of the program)
- Strong testing support is required to ensure behavioral preservation
- As refactoring is small it is easier to control
  - Small refactorings can be composed into a big refactoring

Reasons to refactor

- Refactoring is intended to improve software quality and design
  - Code complexity is reduced
  - the code is more maintainable (easy to change, reusable, etc.)

    Duplication is removed
  - Reduced smells and anti-patterns -> the code is "cleaner"
  - It reduces the effects of software aging and code decay

Reasons to refactor

- Makes software easier to understand
  - Improved code structure and thus the code is easier to read
  - It better communicates its purpose
  - Helps to understand unfamiliar code
  - Helps developers to see things about the design that they could not see before

Reasons to refactor

- Helps to find bugs
  - Promotes a thorough understanding of the code and its structure -> helps in spotting bugs
  - Promotes continuous testing
- Software development is faster
  - Less time on understanding and changing your code

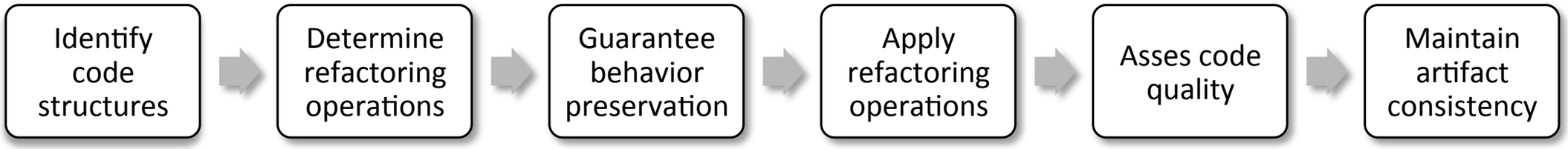
    Less time finding and fixing bugs

    Uncoupled code leads to fewer classes to be changed
  - Promotes good design (e.g., reusability)
  - Refactoring is a key activity in eXtreme Programming

When should you refactor?

- Before doing any change
  - To minimize the impact of the change
- After doing any change
  - To eliminate any introduced bad smell or anti-pattern
- What type of changes?
  - New features
  - Fixing bugs

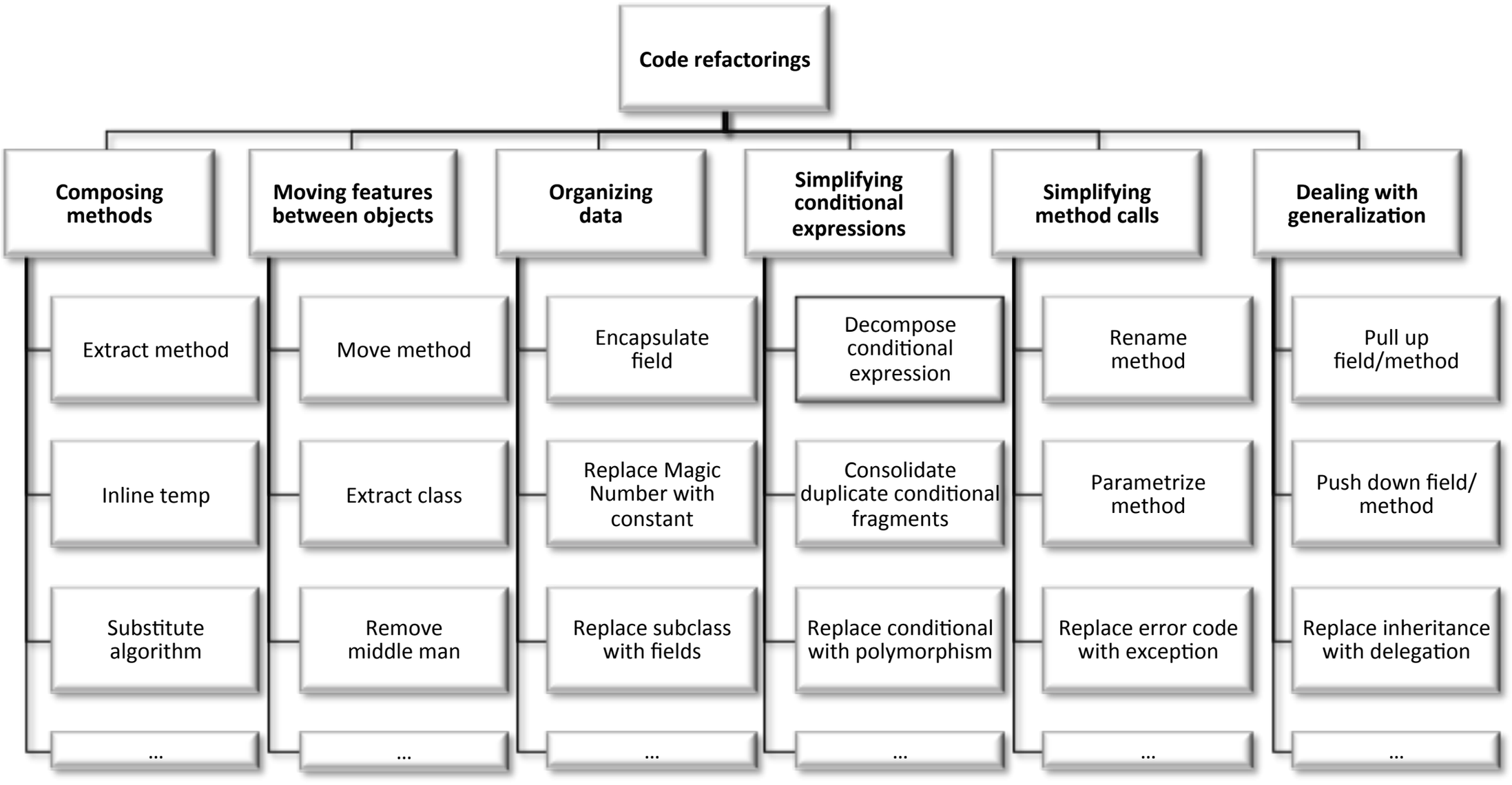
Refactoring process [1]



- Refactoring process
  - Identify where the code should be refactored
  - Determine the refactoring operations that should be applied
  - Check and guarantee behavior preservation by the refactoring
  - Apply refactoring operations
  - Assess the internal quality of source code and process
  - Maintain consistency among artifacts (e.g., docs., designs, etc.)

[1] Mens, T. and Tourwé, T. **"A survey of software refactoring."** *IEEE Transactions on Software Engineering,* 30.2 (2004): 126-139.

A refactorings catalog [1]



[1] Fowler, M., **Refactoring: Improving the Design of Existing Code,** Addison-Wesley, 1999. refactoring.com

Notes on Fowler's catalog

- The refactorings contained in the catalog are based on Fowler's experience
- The catalog describes the main circumstances in which the refactorings are applied
- Like any recipe, there is the need to adapt them to your own context
- The refactorings are described with single-process software in mind (as opposed to concurrent and distributed programs)

Notes on Fowler's catalog

- Some refactorings are about the introduction of GoF design patterns into the code
  - Not every pattern has a refactoring in the catalog
- The refactorings in the catalog are small and can be composed to define complex and bigger refactorings
- There are many more refactorings than the ones described in Fowler's and other catalogs

Category: Composing methods

- These refactorings are devoted to correctly compose methods
  - The idea is to organize behavior contained in the methods
- Usually the main problem is long methods
  - Contain lots of information and complex logic
  - Hard to understand and change
- Refactoring techniques in this group
  - Streamline methods
  - Remove code duplication

Technique: Extract method

- You have a code fragment that can be grouped together

- Turn the fragment into a method with a name that explains its purpose

Extract method - Motivation

- A method is too long or the code needs comments to understand its purpose
  - Long in the sense of the semantic distance between the method name and its body
- Short and well-named methods are preferred because
  - It eases changes and increases reuse
  - Overriding is easier

Extract method
- Example: before refactoring

```
void printOwing() {

   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   // print banner
   System.out.println ("**************************");
   System.out.println ("***** Customer Owes ******");
   System.out.println ("**************************");

   // calculate outstanding
   while (e.hasMoreElements()) {
       Order each = (Order) e.nextElement();
       outstanding += each.getAmount();
   }

   //print details
   System.out.println ("name:" + _name);
   System.out.println ("amount" + outstanding);
}
```

Extract method
- Example: after refactoring

```
void printOwing() {

        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        printDetails(outstanding);
}

void printBanner() {
        // print banner
        System.out.println ("**************************");
        System.out.println ("***** Customer Owes ******");
        System.out.println ("**************************");
}

void printDetails (double outstanding) {
        System.out.println ("name:" + _name);
        System.out.println ("amount" + outstanding);
}
```

Extract method - Other considerations

- Opposite refactoring:
  - Inline method
- Requirements:
  - Be sure the new method's name describes the method's purpose
- Effects:
  - Less code duplication and more reuse
  - Isolates independent parts of code
  - Errors are less likely
  - Readability and communication improvement

Category: Moving features between objects

- Fundamental decision in OO design:
  - Where to put responsibilities!
- It is not easy and requires incremental code evolution (refactoring)
- Common operations:
  - Move functionality between classes
  - Create new classes
  - Hide implementation details from public access

Extract class

- You have a class doing the work that should be done by two classes

- Create a new class and move the relevant fields and methods from the old class into the new class

Extract class - Motivation

- Classes tend to grow, implementing many responsibilities
- Classes become too complicated
- Hard to understand

Extract class
- Example: before refactoring

```
class Person {

    private String _name;
    private String _officeAreaCode;
    private String _officeNumber;

    public String getName() {
        return _name;
    }

    public String getTelephoneNumber() {
        return ("(" + _officeAreaCode + ") " + _officeNumber);
    }

    String getOfficeAreaCode() {
        return _officeAreaCode;
    }

    void setOfficeAreaCode(String arg) {
        _officeAreaCode = arg;
    }

    String getOfficeNumber() {
        return _officeNumber;
    }

    void setOfficeNumber(String arg) {
        _officeNumber = arg;
    }

}
```

Extract class - Example: after refactoring

```
class Person {

    private String _name;
    private TelephoneNumber _officeTelephone =
                                new TelephoneNumber();

    public String getName() {
        return _name;
    }

    public String getTelephoneNumber() {
        return _officeTelephone.getTelephoneNumber();
    }

    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }
}
```

```
class TelephoneNumber {

    private String _number;
    private String _areaCode;

    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }

    String getAreaCode() {
        return _areaCode;
    }

    void setAreaCode(String arg) {
        _areaCode = arg;
    }

    String getNumber() {
        return _number;
    }

    void setNumber(String arg) {
        _number = arg;
    }
}
```

Extract class - Other considerations

- Opposite refactoring: Inline class
- Related refactorings: Move field/method
- Effects:
  - The classes would adhere to the *Single Responsibility Principle*
  - The classes will be more coherent, understandable, tolerant to changes, less fault-prone
- Drawbacks:
  - Excessive use of this refactoring could lead to classes doing almost nothing -> Perform Inline class

Category: Organizing data

- These refactorings make working with data easier
  - They help with data handling, replacing primitives with rich class functionality
  - They help untangling class associations, which makes classes more portable and reusable

Replace magic number with symbolic constant

- You have a literal number with particular meaning

- Create a constant, name it accordingly (expressing the meaning), and replace the number with it

Replace magic number - Motivation

- Numbers with special values are not obvious
  - You have a hard time figuring out their meaning and their role in the code
- Magic numbers are nasty when you need to reference the same logical number in more than one place
  - If the numbers eventually have to change, making the change is really hard

Replace magic number - Example: before refactoring

```
double calculateCircumference(double diameter) {
    return 3.14 * diameter;
}
```

Replace magic number - Example: after refactoring

```
private static final double PI_CONSTANT = 3.14;

double calculateCircumference(double diameter) {
    return PI_CONSTANT * diameter;
}
```

Replace magic number - Other considerations

- Not all numbers are magical (e.g., array indexes and counts, etc.)
  - If the purpose of the value is obvious there is not need to replace it
- Effects:
  - Symbolic constants can serve as documentation about the meaning of the values
  - It is easier to change constant values than to search for the values throughout the entire code
  - Reduce duplication
  - No cost in performance and great improvement in readability

Category: Simplifying conditional expressions

- Conditionals tend to get more and more complicated in their logic over time
- Refactoring is the way to make conditionals simpler

Introduce null object

- You have repeated checks for a null value
- Replace the null value with a null object that exhibits the default behavior

Introduce null object - Motivation

- Dozens of checks for null make your code longer and uglier
- Of course, there is code duplication
  - Those checks for null are spread all over the place
- This refactoring simplifies the code and considers the default data/logic/computation in case of null values

Introduce null object - Example: before refactoring

```
class OtherClass {

    Site site = new Site()
    //other fields...

    void method() {

        // billing plan
        Customer customer = site.getCustomer();
        BillingPlan plan;
        if (customer == null)
            plan = BillingPlan.basic();
        else
            plan = customer.getPlan();

        // customer name
        String customerName;
        if (customer == null)
            customerName = "occupant";
        else
            customerName = customer.getName();

        // ...
        int weeksDelinquent;
        if (customer == null)
            weeksDelinquent = 0;
        else
            weeksDelinquent = customer.getHistory()
                    .getWeeksDelinquentInLastYear();
    }
}
```

```
class Site{
    Customer _customer;

    Customer getCustomer() {
        return _customer;
    }
}
```

Introduce null object - Example: after refactoring

```
class Customer {

    public boolean isNull() {
        return false;
    }

    //factory for the null customer
    static Customer newNull() {
        return new NullCustomer();
    }

}

class Site{
    Customer _customer;

    Customer getCustomer() {
        return (_customer == null) ?
            Customer.newNull():
            _customer;
    }
}
```

```
class NullCustomer extends Customer {

    @Override
    public boolean isNull() {
        return true;
    }

    @Override
    public String getName(){
        return "occupant";
    }

    @Override
    public BillingPlan getPlan(){
        return BillingPlan.basic();
    }

    @Override
    public PaymentHistory getHistory(){
        return PaymentHistory.newNull();
    }
}
```

```
class OtherClass {

    Site site = new Site()

    void method() {

        // billing plan
        Customer customer = site.getCustomer();
        BillingPlan plan = customer.getPlan();

        // customer name
        String customerName = customer.getName();

        // ...
        int weeksDelinquent = customer.getHistory()
                    .getWeeksDelinquentInLastYear();
    }
}
```

Introduce null object - Other considerations

- Drawbacks:
  - The price of getting rid of conditionals is creating yet another new class

Category: Simplifying method calls

- Interfaces should be easy to understand and use
- These refactorings make interfaces more straightforward
  - They make method calls simpler

Parameterize method

- Several methods perform similar actions that are different only in their internal values
- Combine these methods by using parameters for the different values

Parametrize method - Motivation

- If different methods are similar, then probably they are clones
- If a new method is needed, which is another version of the existing ones, there will be more clones
- Instead, you could run one existing method with a different parameter value
  - The trick is to spot code that is repetitive on the basis of a few values that can be passed in as parameters

Parametrize method - Example: before refactoring

```
class Class1 {
    protected Dollars baseCharge() {
        double result = Math.min(lastUsage(), 100) * 0.03;
        if (lastUsage() > 100) {
            result += (Math.min(lastUsage(), 200) - 100) * 0.05;
        }
        if (lastUsage() > 200) {
            result += (lastUsage() - 200) * 0.07;
        }
        return new Dollars(result);
    }
}
```

Parametrize method - Example: after refactoring

```
class Class1 {
    protected Dollars baseCharge() {
        double result = usageInRange(0, 100) * 0.03;
        result += usageInRange(100, 200) * 0.05;
        result += usageInRange(200, Integer.MAX_VALUE) * 0.07;
        return new Dollars(result);
    }

    protected int usageInRange(int start, int end) {
        if (lastUsage() > start)
            return Math.min(lastUsage(), end) - start;
        else
            return 0;
    }
}
```

Parametrize method - Other considerations

- Effects:
  - Removes code duplication
  - Increases code's flexibility
- Drawbacks:
  - Sometimes this refactoring could result in a long and complicated common method instead of multiple simpler ones

Replace error code with exception

- A method returns a special value to indicate an error
- Throw an exception instead

Replace error code with exception - Motivation

- Return error values comes from procedural programming
  - This is obsolete and make the code messy
- The advantage of exceptions is that they are named, thus can have meaning, and the error handling is simplified
  - Error-handling code is ignored in normal conditions but activated when an error occurs

Replace error code with exception - Example: before refactoring

```
class Account {
    private int _balance;

    int withdraw(int amount) {
        if (amount > _balance)
            return -1;
        else {
            _balance -= amount;
            return 0;
        }
    }

}

class Bank {

    void doTransaction(Account account, int amount) {
        if (account.withdraw(amount) == -1)
            handleOverdrawn();
        else
            doTheUsualThing();
    }
}
```

Replace error code with exception - Example: after refactoring

```
class BalanceException extends Exception {
}

class Account {
    private int _balance;

    void withdraw(int amount) throws BalanceException {
        if (amount > _balance)
            throw new BalanceException();
        _balance -= amount;
    }

}

class Bank {

    void doTransaction(Account account, int amount) {
        try {
            account.withdraw(amount);
            doTheUsualThing();
        } catch (BalanceException e) {
            handleOverdrawn();
        }
    }
}
```

Replace error code with exception - Other considerations

- Effects:
  - Simplifies the code from a large number of conditionals for checking various error codes
  - Exception handlers as more succinct to differentiate normal and abnormal execution paths
  - Exception classes can implement their own methods (e.g., send error messages)
- Drawbacks:
  - Exceptions can be used to control the flow of the program

    This is an anti-pattern. They should only used to inform about errors

Category: Dealing with generalization

- These refactoring deal mostly with
  - Moving fields and methods around class hierarchies,
  - Creating new classes and interfaces, and
  - Replacing inheritance with delegation and vice versa

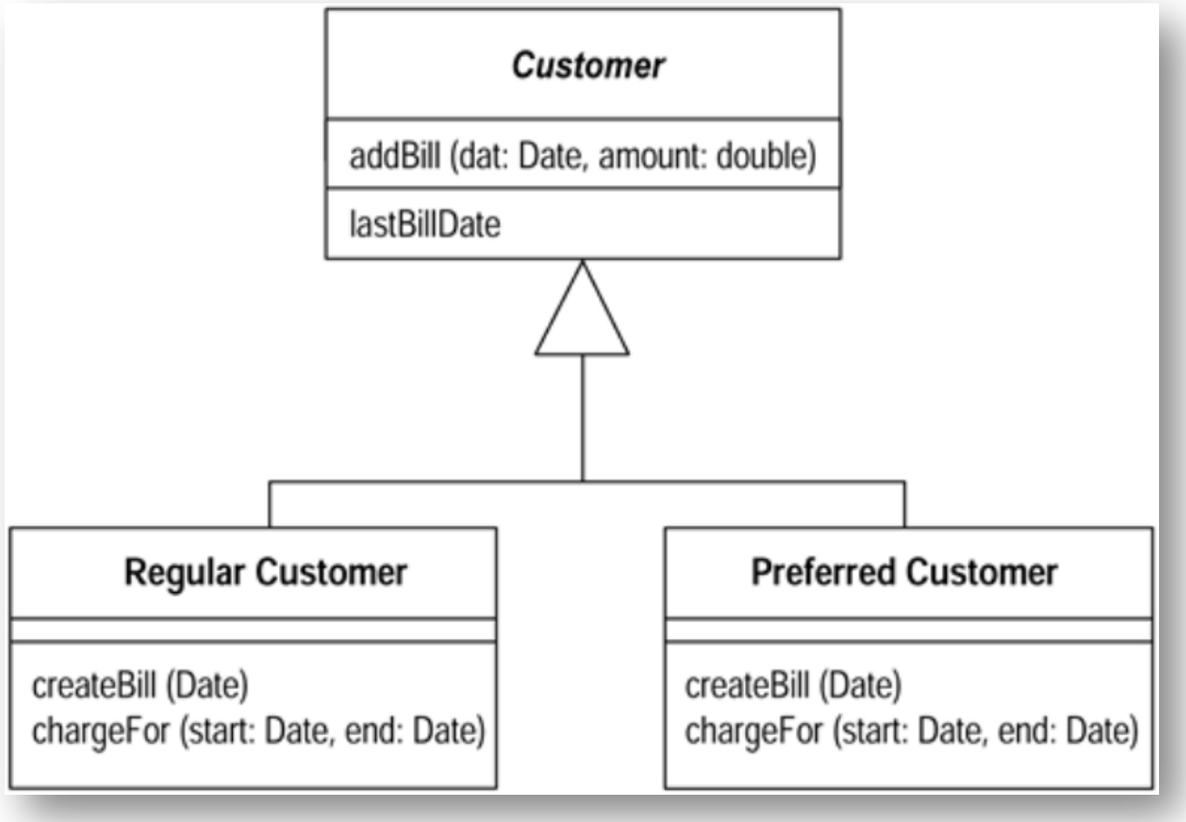
Pull up method

- The methods of a particular class perform the same tasks
- Move the methods to the superclass

Pull up method - Motivation

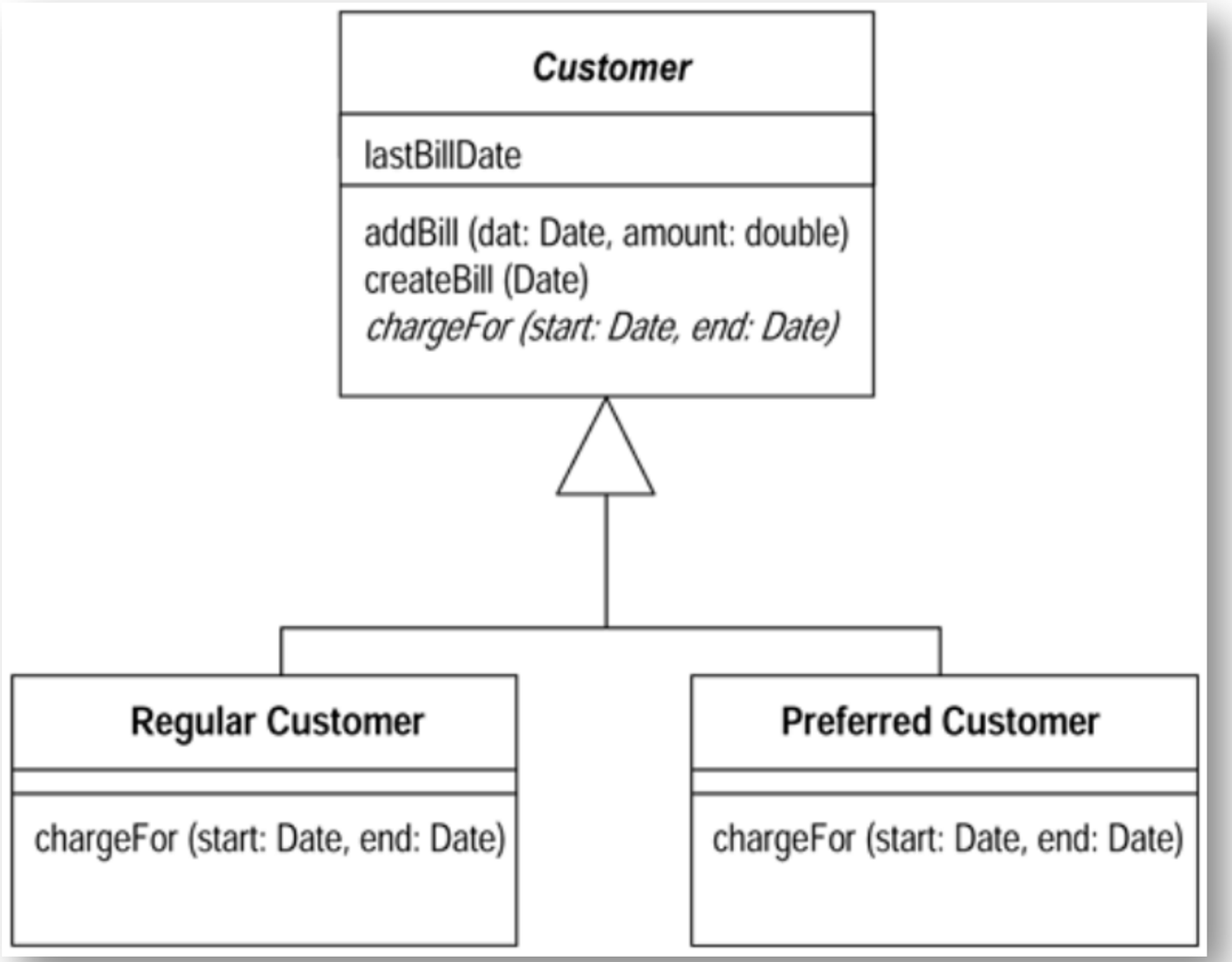
- Subclasses grow and are developed independently of one another
  - This causes duplication (particularly in methods)
- In some cases, the best way of removing duplication is to pull up the common code across subclasses to the super class

Pull up method
- Example: before refactoring




```
void createBill (date Date) {
    double chargeAmount = chargeFor(lastBillDate, date);
    addBill (date, charge);
}
```

Pull up method
- Example: after refactoring



Pull up method - Other considerations

- Opposite refactoring: Push down method
- Related refactorings:
  - Pull up field
  - Form template method
- Requirements:
  - The methods in the subclasses must be identical, if they are not (but do the same thing) then the developer should make them identical
- Effects:
  - Removes duplicate code