Software Evolution and Maintenance

Software Engineering
Prof. Maged Elaasar

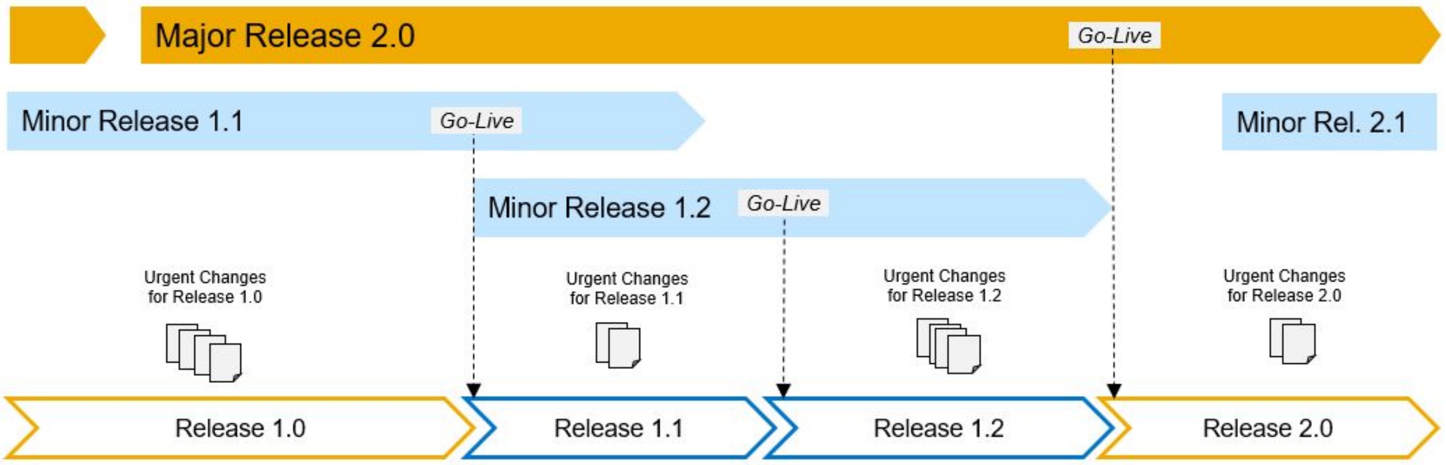
Learning objectives

- Software evolution and maintenance
- Software refactoring and its operations
- Software anti-patterns or code smells

Software evolution & maintenance

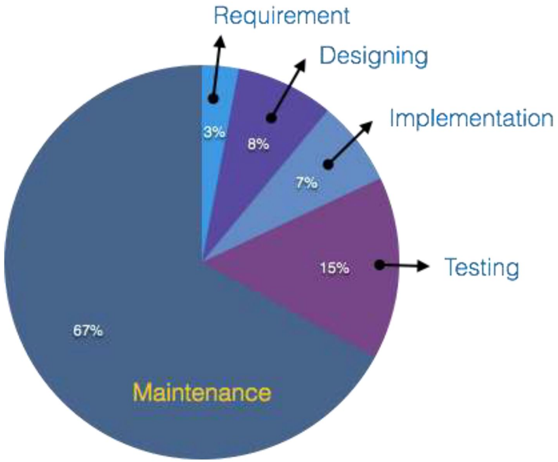
Software evolution

- Software evolution includes all development activities that intend to take the software from a basic to a more advanced state



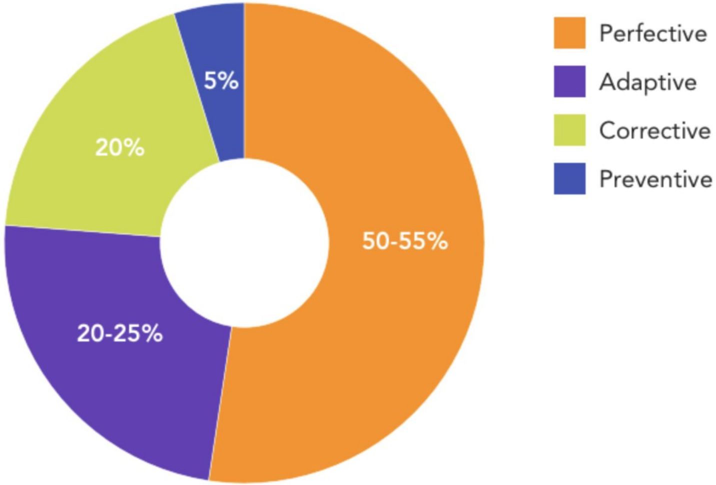
Software maintenance

- Software maintenance includes activities required to keep the software operational and responsive after deployment
- Involves making **small changes** to a deployed system but no major change to its architecture or requirements



Types of software maintenance

- Corrective change
  - Triggered to address bugs
- Adaptive change
  - Triggered by changes to the OS, hardware, software dependencies, or business rules and policies.
- Perfective change
  - Triggered by end users requesting small changes in requirements and features in an existing system.
- Preventative change
  - Triggered to increase the **maintainability** of software in the long run.



When \ Why	Correction	Enhancement
Proactive	Preventative	Perfective
Reactive	Corrective	Adaptive

Change Impact Analysis

- Software maintenance and evolution involve change

- Change impact analysis is the task of estimating the parts of the software that could be affected by a proposed change.

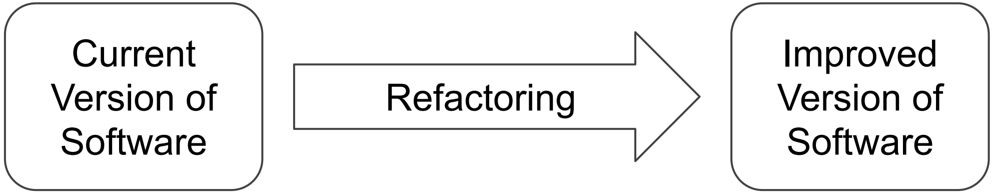
- Techniques for performing impact analysis:
  - **Traceability analysis**:
    - Uses explicit traceability relations established between different development artifacts like requirements, design, code, and test cases, to infer possibility of change impact.
  - **Dependency analysis**
    - Uses syntactic and semantic dependencies between the development artifacts to infer the possibility of change impact
  - **Change history analysis**
    - Uses the commit history in a source control system to infer implicit correlations between different development artifacts

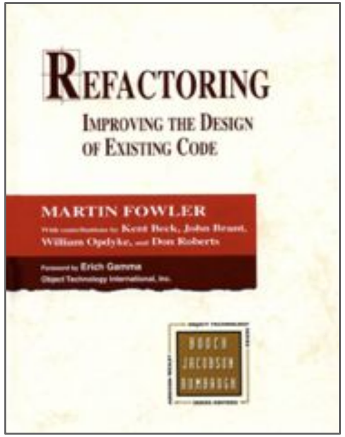
Software refactoring

Software refactoring

- Software refactoring is the process of making a change to the internal structure of software to make it easier to understand and cheaper to maintain without changing its observable behavior.

- It is a way to pay back the technical debt, which is necessary work that gets delayed during the development of a software project in order to hit a particular deliverable or deadline.




REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

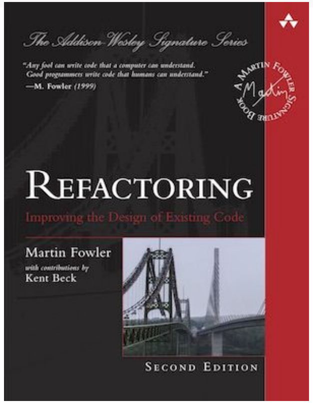
MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

Fowler’s refactoring operations

- **Method refactorings**
  - Extract Method | Move Method
  - Form Template Method
- **Parameter refactorings**
  - Parameterize Method | Introduce Parameter Object
  - Preserve Whole Object | Replace Parameter with Method Call
- **Field refactorings**
  - Encapsulate Field | Encapsulate Collection
  - Replace Primitive Value with Object | Change Bidirectional Association to Unidirectional
- **Class refactorings**
  - Extract Subclass | Extract Superclass
  - Collapse Hierarchy | Extract Class
  - Replace Enumeration with Inheritance | Replace Enumeration with Composition
  - Replace Delegation with Inheritance | Replace Inheritance with Delegation


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Method refactorings 1/2

Extract Method: when you have a code fragment that can be reused, move this code to a separate new method and replace the old code with a call to the method.

```
void finishPayment() {
  Invoice invoice = getInvoice();
  System.out.println("name: "+invoice.name);
  System.out.println("date: "+invoice.date);
  System.out.println("amount: "+invoice.outstanding);
}
```

```
void finishPayment() {
  Invoice invoice = getInvoice();
  printl(invoice);
}
void print(Invoice invoice) {
  System.out.println("name: " + invoice.name);
  System.out.println("date: "+invoice.date);
  System.out.println("amount: " + invoice.amount);
}
```

Move Method: when a method is used more in another class than in its own class, move the method to the class that uses it the most. Turn the code of the original method into a call to the new method.

```
class A {
   public void foo() {...}
}
class B {
   private A a;
   public void bar() {...
      a.foo()
   ...}
}
```

```
class A {
}
class B {
   public void foo() {...}
   public void bar() {...
      foo()
   ...}
}
```

Method refactorings 2/2

Form Template Method: when subclasses have methods that contain similar statements in the same order, move the steps to a new method on a common superclass, and override their details in the subclasses.

```
abstract class Site {}

class ASite extends Site {
  float getAmount() {
    float base = unit * rate;
    float tax = base * taxRate;
    return base +  tax;
  }
}

class BSite extends Site {
  float getAmount() {
    float base = unit * rate * 0.5;
    float tax = base * taxRate * 0.2;
    return base +  tax;
  }
}
```

```
abstract class Site {
  float getAmount() {
    return getBase() +  getTax();
  }
  float getBase();
  float getTax();
}

class ASite extends Site {
  float getBase() {return unit * rate;}
  float getTax() {return base * taxRate;}
}

class BSite extends Site {
  float getBase() {return unit * rate * 0.5;}
  float getTax() {return base * taxRate * 0.2;}
}
```

Parameter refactorings 1/2

Parameterize Method: when multiple methods perform similar actions that are different only in some aspects, combine these methods by using a parameter that will pass the necessary special aspects.

```
class Employee {
  void raiseFivePercent();
  void raiseTenPercent();
}
```

```
class Employee {
  void raise(int percentage );
}
```

Replace Parameter with Method Call: when a method makes a query call and sends the result to another method as a parameter, try placing the query call inside the method that uses it directly.

```
void calculate() {
   int basePrice = quantity * itemPrice;
   double fees = this.getFees();
   double finalPrice = discountedPrice(basePrice, fees);
}
double discountedPrice(int base, double fees) {
   return base * 2 + fees;
}
```

```
void calculate() {
   int basePrice = quantity * itemPrice;
   double finalPrice = discountedPrice(basePrice);
}
double discountedPrice(int base) {
   double fees = this.getFees();
   return base * 2 + fees;
}
```

Parameter refactorings 2/2

Preserve Whole Object: when you get several values from an object and then pass them as parameters to a method. Instead, try passing the whole object.

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```
boolean withinPlan = plan.withinRange(daysTempRange);
```

Introduce Parameter Object: when your methods contain a repeating group of parameters, replace these parameters with an object.

```
class Customer {
  void amountInvoiced(Date start, Date end);
  void amountReceived(Date start, Date end);
  void amountOverview(Date start, Date end);
}
```

```
class Customer {
  void amountInvoiced(DateRange date);
  void amountReceived(DateRange date);
  void amountOverview(DateRange date);
}
class DateRange {
   public Date start, end;
}
```

Field refactorings 1/2

Encapsulate Field: when you have a public field, make the field private and create access methods for it.

```
class Person {
  public String name;
}
```

```
class Person {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String arg) {
    name = arg;
  }
}
```

Encapsulate Collection: when a class contains a collection field and a simple getter and setter for working with the collection, make the getter return read-only collection and add methods for adding/deleting elements of the collection.

```
class Person {
  private List<Course> courses;
  public List<Course> getCourses() {
     return courses;
  }
  public void setCourses(List<Course> courses) {
    this.courses = courses;
  }
}
```

```
class Person {
  private List<Course> courses;
  public List<Course> getCourses() {
     return Collections.unmodifiableList(courses);
  }
  public void addCourses(Course c) { courses.add(c); }
  public void removeCourse(Course c) { courses.remove(c); }
}
```

Field refactorings 2/2

Replace Primitive Value with Object: when a class contains a primitive field with its own behaviors, create a new class, move the old field and its behavior to it, and replace the field with one typed by the new class.

```
class Order {
  private String customer;
  private String getCustomerName() { … }
  public void foo() {
    String name = getCustomerName();
  }
}
```

```
class Order {
  private Customer customer;
  public void foo() {
    String name = customer.getName();
  }
}
class Customer {
  private String getName() { … }
}
```

Chang Bidirectional Association to Unidirectional: when you have a bidirectional association between classes, but one of the classes doesn’t use the other’s features, remove the unused association.

```
class Customer {
  List<Order> orders;
  public purchase(Order order) {
    order.customer = this;
   orders.add(order); }
}
class Order {
  Customer customer;
}
```

```
class Customer {
  List<Order> orders;
  public purchase(Order order) {
    orders.add(order);
  }
}
class Order {
}
```

Class refactorings 1/4

Extract Subclass: when a class has features used only in certain cases, create a subclass and use it in these cases.

```
class Animal {
  void eat();
  void drink();
  void fly();
}
```

```
class Animal {
  void eat();
  void drink();
}
class FlyingAnimal extends Animal {
  void fly();
}
```

Extract Superclass: when you have two classes with common fields and methods, create a shared superclass for them and move all the identical fields and methods to it.

```
class Department {
  String getName() {..}
  int getHeadCount() {..}
  int getTotalAnnualCost() {..}
}
class Employee {
  String getName() {..}
  String getId() {..}
  int getTotalAnnualCost() {..}
}
```

```
class Party {
  String getName() {..}
  int getTotalAnnualCost();
}
class Department extends Party {
  int getHeadCount() {..}
  int getTotalAnnualCost() {..}
}
class Employee extends Party {
  String getId() {..}
  int getTotalAnnualCost() {..}
}
```

Class refactorings 2/4

Collapse Hierarchy: when you have a class hierarchy in which a subclass is practically the same as its superclass, merge the subclass and superclass.

```
class Employee {
  String getId() {..}
}
class Worker extends Employee {
  String getName() {..}
}
```

```
class Employee {
  String getId() {..}
  String getName() {..}
}
```

Extract Class: when one class does the work of two, awkwardness results. Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.

```
class Person {
   String name;
   String officeAreaCode;
   String officeNumber
   String getOfficeTelephoneNumber() {..}
}
```

```
class Person {
   String name;
   TelephoneNumber officeNumber;
   String getOfficeTelephoneNumber() {..}
}
class TelephoneNumber {
   String officeAreaCode;
   String officeNumber
}
```

Class refactorings 3/4

Replace Enumeration with Inheritance: when you have behavior that is affected by an enumeration, create an inheritance hierarchy instead, allocate the behaviors to it, and call those behaviors polymorphically instead.

```
class Employee {
  public final in ENGINEER = 1;
  public final in SCIENTIST = 2;
  int type;
  public void foo() {
    switch(workerType) {
       case ENGINEER: System.out.println(“I am an engineer”); break;
       case SCIENTIST: System.out.println(“I am a scientist”); break;
}}}
```

```
class Employee {
  void print();
  void foo() { print(); }}

class Engineer extends Employee {
   void print() { System.out.println(“I am an engineer”); }}

class Scientist extends Employee {
   void print() { System.out.println(“I am a scientist”); }}
```

Replace Enumeration with Composition: when you have a behavior that is affected by an enumeration, and you can’t use inheritance to mitigate this, replace the enumeration with a composition instead.

```
class Employee {
  public final in ENGINEER = 1;
  public final in SCIENTIST = 2;
  int type;
  public void foo() {
    switch(type) {
       case ENGINEER: System.out.println(“I am an engineer”); break;
       case SCIENTIST: System.out.println(“I am a scientist”); break;
}}}
```

```
class Employee {
  EmployeeBehavior behavior;
  void foo() { type.print(); }}

class EmployeeBehavior { void print(); }
class EngineerBehavior extends EmployeeBehavior {
   void print() { System.out.println(“I am an engineer”); }}
class ScientistBrhavior extends EmployeeBehavior {
   void print() { System.out.println(“I am a scientist”); }}
```

Class refactorings 4/4

Replace Delegation With Inheritance: when a class contains many simple methods that delegate to all methods of another class, make the class a subclass instead, which makes the delegating methods unnecessary.

```
class Person {
  String getName() {..}
}
class Employee {
  Person person;
  String getName() {
    return person.getName();
  }
}
```

```
class Person {
  String getName() {..}
}
class Employee extends Person {
}
```

Replace Inheritance By Delegation: when you have a subclass that uses only a portion of the methods of its superclass, create a field and put a superclass object in it, delegate methods to the superclass, and get rid of inheritance.

```
class Vector {
  boolean isEmpty() {..}
}
class Stack extends Vector {
}
```

```
class Vector {
  boolean isEmpty() {..}
}
class Stack {
  Vector vector;
  boolean isEmpty() {
    return vector.isEmpty();
  }
}
```

Software anti-patterns (code smells)

Some of Fowler’s code smells

- **Bloaters**
  - Primitive obsession
  - Long parameter list
  - Data clumps
- **Change preventers**
  - Divergent Change
  - Shotgun Surgery
  - Parallel Inheritance Hierarchies
- **Object-oriented abusers**
  - Alternative Classes with Different Interfaces
  - Refused Bequest
  - Switch Statements
- **Dispensables**
  - Duplicate Code
  - Speculative Generality
  - Lazy Class
  - Data Class
- **Couplers**
  - Feature Envy
  - Inappropriate Intimacy

Primitive Obsession

- When primitive types are used instead of real data types:
  - int price = 5;
  - String telephone = “818-1188112”;
  - String password = “xxxx”;
  - String color = “blue”;
  - float temperature = 32;

- Problem:
  - Invalid values could be accidentally set
  - Values of different types could be used interchangeably
  - Validation would have to be done in multiple places

- Refactoring:
  - **Replace Primitive Value With Object**

Long Parameter List

- When a long list of parameters are used for a method

```
public void setAddress(String country, String state, String city, String street, int house, ...)

public float calculatePayment(float price , int quantity, int discount, int fees, ...)
```

- Refactoring:
    - **Introduce Parameter Object** (if parameters are always coupled)
    - **Replace Parameter with Method Call** (if a parameters can be queried within method)

Data Clumps

- When different parts of the code contain identical groups of variables

```
String databaseURL = “xxxx”;
String username = “yyyyy”;
String password = “zzzzz”;
```

- Refactoring:
  - **Extract Class** (if variables are members of a class)
  - **Preserve Whole Object** (if they are derived from an existing object)

Divergent Change

- When one class is changed in different ways for different reasons (a sign that the Single Responsibility principle is not followed)

  e.g., class **Customer** is changed when a new transaction type is supported, or when the price of a product is changed.

- Refactoring
  - **Extract Class**
  - **Extract Superclass**
  - **Extract Subclass**

Shotgun Surgery

- When a small change is implemented by changing multiple classes (a sign that the Separation of Concerns principle is not followed)

  e.g., to create a new **‘SupperAdmin’** user role, you found yourself editing some methods in **Profile**, **Product** and **Employee** classes.

- Refactoring
  - **Move Method, Move Field** (to an existing common class)
  - **Extract Class** (and move related fields and methods to it)

Parallel Inheritance Hierarchy

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class (a special case of Shotgun Surgery).

  e.g., you have a class hierarchy for **Department**, and a class hierarchy for the **Privilege** for each department.

- Refactoring
  - **Move Method and Move Field** (to a common class)

Alternative Class with Different Interfaces

- When two classes perform identical functions but have different method signatures (usually due to lack of communication)

  e.g., class **QuickSorter** has a method void **sort**() and class **BubbleSorter** has method void **getSorted**(int[] data).

- Refactoring
  - **Rename Method** (to a common name)
  - **Move Method, Add Parameter, Parameterize Method** (to unify the interface)
  - **Extract Superclass** (to unify part of the interface and/or implementation)

Refused Bequest

- When a subclass uses only some of the methods and properties inherited from its parents (a sign they are not proper subclass/superclass)

  e.g., class **Person** inherits from class **Building** to reuse its attributes **name and address**.

  e.g., class **Order** inherits from class **List** to reuse its operations **add** and **remove**.

- Refactoring
  - **Extract Superclass** (and move the reused parts to it and make both classes inherit it)
  - **Replace Inheritance By Delegation** (to still achieve reuse)

Switch Statements

- When you have big switch statements or big sequences of if statements scattered in many places in the code

```
e.g., int area = switch (type) {
   case “square”:  calcSquareArea(4); break;
   case “circle”:  calcCircleArea(5); break;
   ...
}
```

```
e.g., switch (state) {
   case “state1”:  handleState1Event(e); break;
   case “state1”:  handleState2Event(e); break;
   ...
}
```

- Refactoring
  - **Replace Enumeration with Inheritance**
  - **Replace Enumeration with Composition**

Duplicate Code

- When there are two code fragments that look almost identical.

```
void calculateRectangleArea (Rectangle r) {
  int width = r.width;
  int height = r.height;
  return width * height;
}
void getAreaOfRectangle (Rectangle r) {
  return r.width * r.height;
}
```

- Refactoring
  - **Extract Method**, **Move Method, Inline Method** (to unify the code)
  - **Form Template Method** (when the code is similar but not identical)

Speculative Generality

- When there is an unused class, method, field, or parameter.

  e.g., class **BestUtils** has no attributes or methods (could be a feature that did not pan out)

- Refactoring
  - **Collapse Hierarchy** (for abstract classes)
  - **Inline Class, Inline Method, Remove Parameter** (to move used class, method or param)

Lazy Class

- When a class does not do enough to justify its existence

  e.g., class **ControllerUtils** only has a single method void static **synchronize**(View view, Model model).

- Refactoring
  - **Inline Class** (for near-useless classes)
  - **Collapse Hierarchy** (if the useless class is a subclass)

Data Class

- When a class contains only fields and crude methods for accessing them (getters and setters) and not real behavior.

  e.g., class **Rectangle** with only int **width** and int **height** attributes, while a method **calculateArea**(Rectangle) is on another class.

  e.g., class **Account** with a public attribute ArrayList<Order> **orders**, while the method **add**(Account a, Order o) is on another class.

- Refactoring
  - **Encapsulate Field, Encapsulate Collection** (to hide the implementation details)
  - **Move Method, Extract Method** (to move relevant behavior to the data class)

Feature Envy

- When a method access the data of another object more than its own (this could be after the data has moved to a data class).

  e.g., method **2DUtils.calculateArea** accesses attribute int **width** and int **height** on class **Rectangle**.

- Refactoring
  - **Move Method, Extract Method** (to move the method to the class with the data)

Inappropriate Intimacy

- When one class uses the internal fields and methods of another class.

  e.g., class **Account** has a method **onChange**() that loops on its List<AccountObserver> **observers** calling not only the **notify()** method from the inherited Observer interface, but also method **dispatchEmail**() on class **AccountObserver**.

- Refactoring
  - **Move Method, Move Field** (to the using class if they do not belong to the used class)
  - **Replace Delegation With Inheritance** (if the classes are subclass / superclass)

Anti-patterns Quiz

References

- Fowler, M.: “Refactoring: Improving the Design of Existing Code, 2nd edition”, Addison-Wesley, 1999.
- Refactoring Guru: https://refactoring.guru/refactoring/catalog
- Tripathy P., Naik, K.: "Software Evolution and Maintenance: A Practitioner's Approach," John Wiley & Sons, 2014.