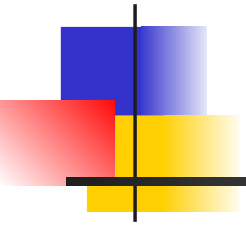


Refactoring

Refactoring

Restructuring an existing body of code without changing its external behavior.

Refactoring as a Discipline:

A series of (usually) small changes that preserve external functionality.

Based on an objective set of goals or criteria.

Refactoring Example

```
class Auction:
    def __init__(self, description, min_increment=1):
        self.name = description
        self.bids = {"no bids": 0}
        self.active = False

    def bid( self, bidder, amount ):
        if self.active and amount >= self.best_bid()+self.min_incr:
            self.bids[bidder] = amount

    def best_bid(self ):
        return max( self.bids.values() )
```

Same Behavior, Different Code

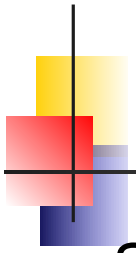
```
class Auction:
    def __init__(self, description, min_increment=1):
        self.name = description
        self.max_bid = 0
        self.active = False
        ...
    def bid( self, bidder, amount ):
        if self.active and amount >= self.max_bid+self.min_incr:
            self.max_bid = amount

    def best_bid( self ):
        return self.max_bid
```

Why Refactor?

Design Changes during Development

Your First Code isn't always Your Best Code



Evolution of Software

Software evolves over time.

Causes:

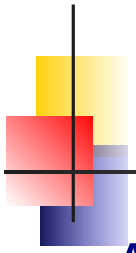
1. *process* - Iterative & Incremental Development

2. *changing requirements* - customer may change or clarify requirements, something encouraged by Agile

3. *discovery* - developers find a better design or better way to implement.

4. *changing technology* - make require software to evolve, e.g. switch from desktop to cloud-based apps.

We need to be able to adapt & improve code.



Continuous Improvement

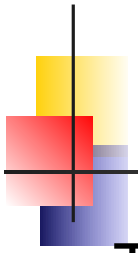
Authors review and rewrite manuscripts.

Composers rewrite songs and musicals many, many times before public performance.

Engineers prototype, analyze, and improve design of new and existing products.

So, what about computer software?

First Example: Pizza



This example contains some common refactorings:

```
https://github.com/ISP19/pizzashop
```

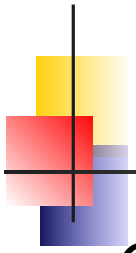
or using ssh:

```
git@github.com:ISP19/pizzashop.git
```

Instructions and explanation are in project **README**.

Refactoring: Signs

- Duplicate code
- Long methods
- One class doing many things
- *Change causes ripple effects*: when you change something in one class or method you have to make several small changes elsewhere
- Violations of good OO design principles, like *SRP*, *Information Expert*, or *separation of concerns*.



Different Levels of Refactoring

Code Complete, Ch 24

1. Data Level Refactoring

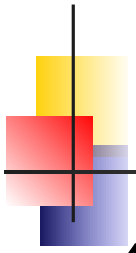
Replace magic number/strings with named constants.

2. Statement Level Refactoring

Introduce explanatory variable

3. Routine Level Refactoring

Extract a method



Different Levels

4. Class Implementation Refactoring

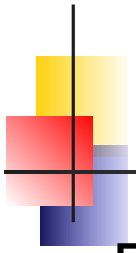
Move common code into a superclass or delegate

5. Class Interface Refactoring

Move a method from one class to another

6. System Level Refactoring

Provide Factory class instead of constructors



Example: Movie Rental

From chapter 1 of *Refactoring* (Fowler).

A store rents movies to customers.

The rental charge depends on how long customer borrows the movie and the type of movie.

New Release: $3 per day

Children's Movie: $1.5 for 3 days + $1.5/day after day 3

Normal Movie: $2 for 2 days + $1.5/day after day 2

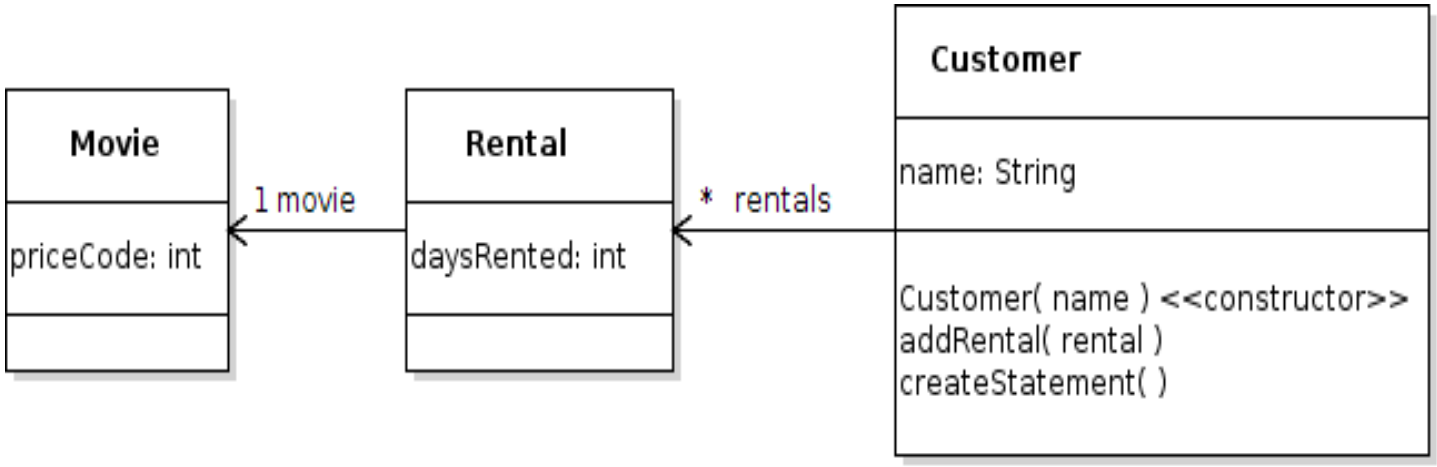
The application should print a statement to standard out.

Frequent Renter Points

Customer earns 1 frequent renter points per rental for ordinary and children's movies.

For new releases, customer gets 1 point for each day of rental.

Movie Rental Domain Model



Get Familiar with the Code

Review the domain code.

`https://github.com/jbrucker/movierental`

What part looks like it could be designed or implemented better?

Before refactoring

You must have working tests.

Verify the tests succeed.

Do they have good code coverage?

First refactoring

Customer.statement() is long and complex.

Create a separate method for the block of code that computes rental charge.

- what are parameters for this method?
- what does it return?

Run the tests.

Guidelines:

- find all variables used in the block
- if variable is not changed, pass it as a parameter
- if variable is changed, is it the result to return?
- avoid too many parameters

Extract Method

Motivation: extract a block of code as a method when
a) method is too long and complex,
b) code or logic is duplicated,
c) want to test the block of code separately.

Mechanics: see Refactoring book, p. 90.

Eclipse does this quite well.

For Eclipse to correctly identify the return value, you need to include assignment of the result as part of block of code you select.

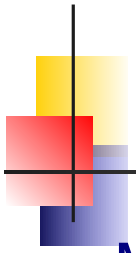
Second Refactoring: Move Method

1. The amountForRental( ) method uses data about the rental, but no data about the customer.
2. Its not *really* the customer's responsibility to know how to compute rental charges.

These are signs that amountForRental() is in the wrong class.

Move it to the Rental class.

Run the tests.



Move Method

Motivation: move a method to another class when
a) method is primarily used by another class, or several other classes,
b) method uses data from another object, not this object,
c) want to simplify a class that is doing too much.

(b) is also known as *Information Expert* principle.

Mechanics: see Refactoring book, p. 116.

Fowler suggests first *copying* the method to another class, editing as needed, change name to suit the new context, and change references to the new method.

Then, when code is working, delete the original method.

3rd Refactor: Query Method

The total charge is being summed in a loop.

Fowler recommends simplifying code by replacing this with a "query method" that computes the total charge.

Catalog: "Replace Temp with Query"

4th: Replace Conditional Logic with Polymorphism

```
switch( getMove().getPriceCode() ) {

case Movie.REGULAR:
    . . .
    break;
case Movie.NEW_RELEASE:
    . . .
    break;
case Movie.CHILDRENS:
    . . .
```

Replace Conditional with Polymorphism

1. Move getRentalCharge() to Movie.
2. Since rental charge depends on Movie type, you might think to create subclasses:

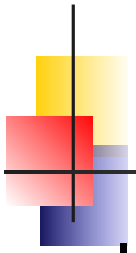
   ChildrensMovie NewReleaseMovie

   but this is a bad idea. (Why?)
3. Instead, apply the State (or Strategy) Pattern.

   a) define an interface for Pricing

   b) write specific instances for each type of movie

   c) delegate to Pricing object



5th: Replace simple type with object

In Movie, what are those NEW_RELEASE, CHILDRENS, and NORMAL constants for?

They are markers, but they don't *do* anything.

How about making them objects from a class or enumeration?

Definition of Refactoring

Refactor (noun) -

a change to the structure of software [*for a purpose*] without modifying the software's observable behavior.

The purpose:

to make software easier to understand or modify.

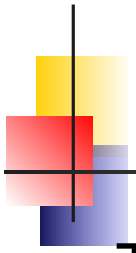
In contrast to:

performance optimization doesn't change observable behavior (except for speed) but isn't refactoring.

Guidelines

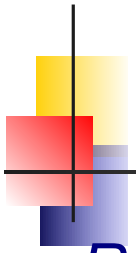
- Write **good tests** first.
- Refactor in **small steps**, run the tests after each change. Test, test, test.
- Don't add functionality while refactoring.
- Don't refactor while adding functionality.
- "Wear two hats": adding-function hat & refactoring hat. Only wear one hat at a time.

Catalog of Refactorings



The Refactoring book contains a catalog of refactorings. They are presented in a format like this:

Name	Extract Method
Summary	create a method from a block of code...
Motivation	...
Mechanics	replace block of code with call to a method. Variables used in the block but not changed may become parameters, . . .
Example	



Resources

Refactoring by Martin Fowler (1999) and (2008).

- first 3 chapters are example and basics
- rest of book is patterns and special situations

http://refactoring.com - patterns from *Refactoring* book.

Refactoring to Patterns by Kerievsky (2004)

- very short catalog

Code Complete 2E, Ch. 24: Refactoring

- good explanation of why and what to look for
- checklist: Reasons to Refactor, Summary of Refactorings

Refactoring

Improve existing code

- *improve design*
- *improve implementation*
- *does not change external functionality*