

# Refactoring and Patterns

Peter Dolog  
dolog [at] cs [dot] aau [dot] dk  
5.2.47  
Information Systems  
March 9, 2009

# Goal

Refactoring

Tutorial on MDD and UP (D403b – relieved from duties and  
s601e)

Design Patterns

# Learning Goal

To identify where refactoring could help

To reflect on MDD and UP

To apply design patterns in future work

# Goal

## Refactoring

- What is refactoring
- Arguments for refactoring
- Bad Smells
- Obstacles

Tutorial on MDD and UP (D403b and s601e)

Design Patterns

# Refactoring

Martin Fowler (and Kent Beck, John Brant, William Opdyke, Don Roberts),  
*Refactoring- Improving the Design of Existing Code*, Addison Wesley, 1999.

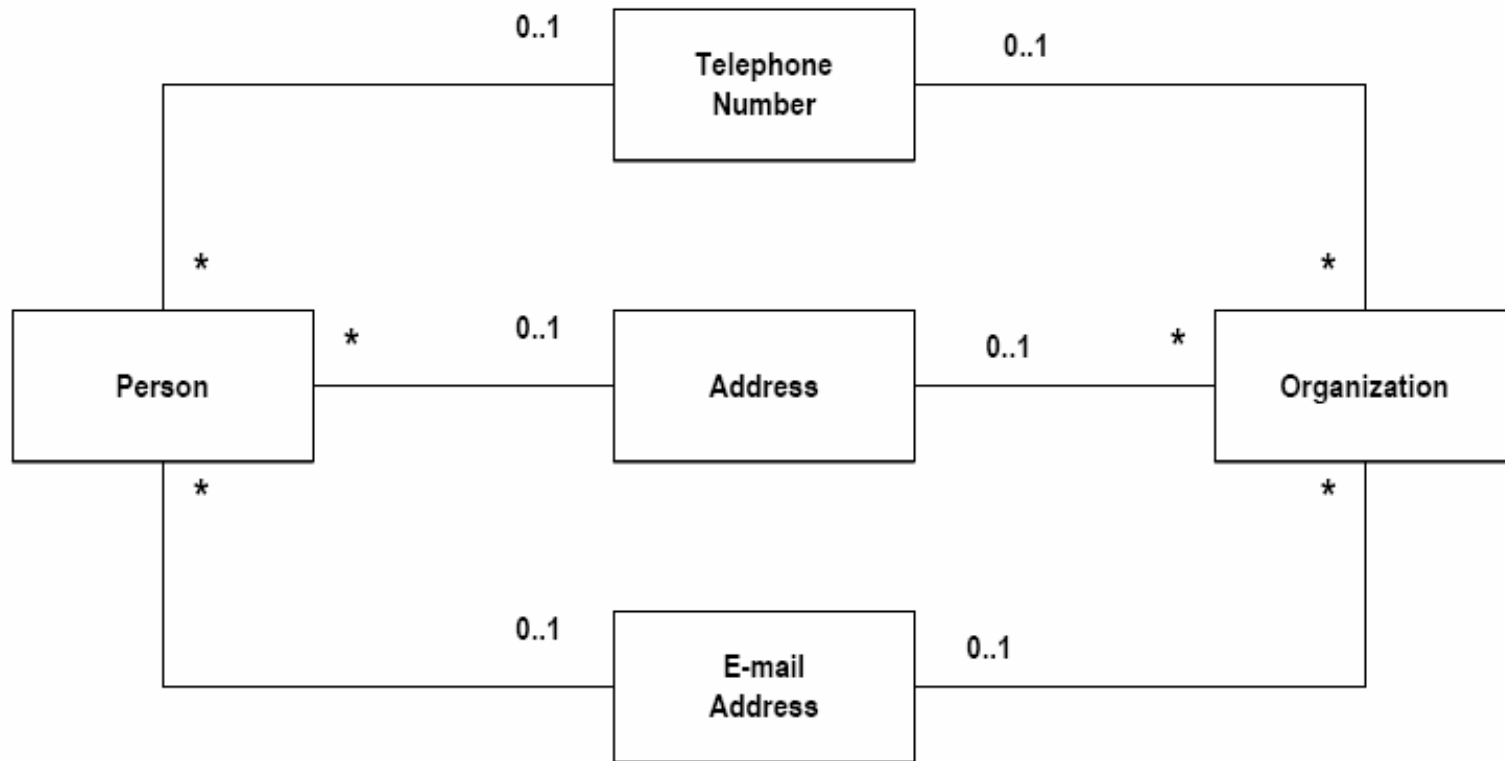
Refactoring (noun):

a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactor (verb):

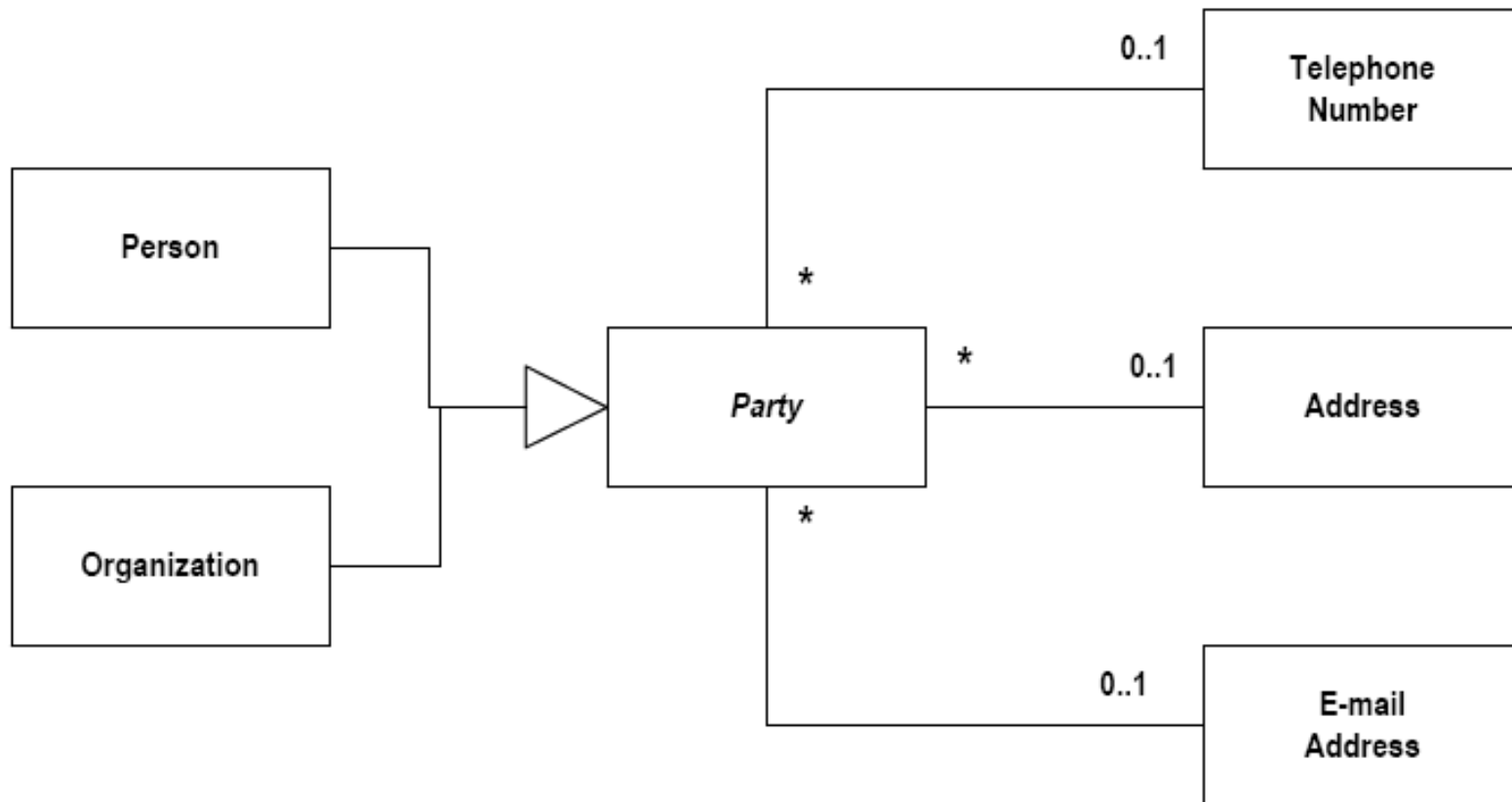
to restructure software by applying a series of refactorings.

# Address Book



**Analysis Patterns book: Martin Fowler**

# Party



**Analysis Patterns book: Martin Fowler**

# Refactoring: duplicated code

case 0:

```
activePiece = RightHook.getRightHook();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 1:

```
activePiece = LeftHook.getLeftHook();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 2:

```
activePiece = RightRise.getRightRise();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 3:

```
activePiece = LeftRise.getLeftRise();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break; //more
```

Source: Daniel  
H Steinberg



# Refactoring: duplicated code

```
case 0:
    activePiece = RightHook.getRightHook();    break;
case 1:
    activePiece = LeftHook.getLeftHook();      break;
case 2:
    activePiece = RightRise.getRightRise();    break;
case 3:
    activePiece = LeftRise.getLeftRise();      break;
case 4:
    activePiece = Hill.getHill();              break;
case 5:
    activePiece = StraightPiece.getStraightPiece(); break;
case 6:
    activePiece = Square.getSquare();          break;
}
ml = new MoveListener(activePiece);
gameBoard.addKeyListener(ml);
```

Source: Daniel H Steinberg

# Why should refactoring be done?

## Argument I

Refactoring **Improves the Design** of Software.

Without refactoring, the design of the program will decay. As people change code - changes to realize short-term goals or changes made without a full comprehension of the design of the code - the code loses its structure.

# When should refactoring be done?

## Argument 2

Refactoring makes software **easier to understand**.

There are users of your code. The computer, the writer, and the updater. The most important is the updater. Who cares if the compiler takes a few more cycles to compile your code. If it takes someone 3 weeks to update your code that is a problem.

# Who should be doing refactoring?

## Argument 3

Refactoring helps you **find bugs**.

Part of refactoring code is understanding the code and putting that understanding back into the code. In that process a clarification takes place. In that clarification bugs will be found.

# Who should be doing refactoring?

## Argument 4

Refactoring Helps you **Program Faster**.

Without a good design, you can progress quickly for a while, but soon poor design start to slow you down. You spend time finding and fixing bugs and understanding the system instead of adding new function. New features need more coding as you patch over a patches...

# Bad smells

Knowing how to refactor something does not tell you when to refactor and how much to refactor.

Kent Beck and Martin Fowler coined a phrase 'Bad Smells' to describe the hint of when to refactor.

This phrase was meant to reflect the ability and experience gained over time by a programmer that is needed to recognize bad coding structure.

# Bad smells: Duplicated code

“The #1 bad smell”

Same expression in two methods in the same class?

Make it a private ancillary routine and parameterize it - gather duplicated code  
(Extract method)

Same code in two related classes?

- Push commonalities into closest mutual ancestor and parameterize
- Use template method DP for variation in subtasks - gather similar parts, leaving holes (Form template method)

## Bad smells: Duplicated code

Same code in two unrelated classes?

- Ought they be related?
  - Introduce abstract parent (Extract class, Pull up method)
- Does the code really belongs to just one class?
  - Make the other class into a client (Extract method)



# Bad smells: Long method

Often a sign of:

Trying to do too many things

Poorly thought out abstractions and boundaries

Best to think carefully about the major tasks and how they inter-relate.

Break up into smaller private methods within the class (Extract method)

Delegate subtasks to subobjects that “know best” (i.e., template method DP)  
(Extract class/method, Replace data value with object)

# Bad smells in code: Long method

Fowler's heuristic:

- When you see a comment, make a method.
- Often, a comment indicates:
  - The next major step
  - Something non-obvious whose details detract from the clarity of the routine as a whole.
- In either case, this is a good spot to “break it up”.

# Bad smells: Feature envy

A method seems more interested in another class than the one it's defined in e.g., a method

A::m() calls lots of get/set methods of class B

Solution:

Move m() (or part of it) into B!

(Move method/field, extract method)

Exceptions:

Visitor/iterator/strategy DP where the whole point is to decouple the data from the algorithm

Feature envy is more of an issue when both A and B have interesting data

## Bad smells: Data clumps

You see a set of variables that seem to “hang out” together

e.g., passed as parameters, changed/accessed at the same time

Usually, this means that there’s a coherent subobject just waiting to be recognized and encapsulated

```
void Scene::setTitle (string titleText,  
                    int titleX, int titleY,  
                    Colour titleColour){...}  
  
void Scene::getTitle (string& titleText,  
                    int& titleX, int& titleY,  
                    Colour& titleColour){...}
```

## Bad smells: Data clumps

In the example, a Title class is dying to be born

If a client knows how to change a title's x, y, text, and colour, then it knows enough to be able to “roll its own” Title objects.

However, this does mean that the client now has to talk to another class.

This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.

Moving the data may create feature envy initially

May have to iterate on the design until it feels right.

(Preserve whole object, extract class, introduce parameter object)

# Bad smells: Primitive obsession

All subparts of an object are instances of primitive types

- (int, string, bool, double, etc.)  
e.g., dates, currency, SIN, tel.#, ISBN, special string values

Often, these small objects have interesting and non-trivial constraints that can be modelled

- e.g., fixed number of digits/chars, check digits, special values

Solution:

- Create some “small classes” that can validate and enforce the constraints. This makes your system more strongly typed.

# Bad Smell Examples

Duplicate Code

Long Methods

Large Classes

Long Parameter Lists

Feature Envy

Data Clumps

Primitive Obsession

# Why not refactor?

Conventional wisdom would discourage  
modifying a design

- You might break something in the code
- You have to update the documentation
- Both expensive

But, there are longer term concerns: sticking with an inappropriate design

- Makes the code harder to change
- Makes the code harder to understand and maintain
- Very expensive in the long run



# Refactoring Philosophy

Make all changes small and methodical

- Follow design patterns

Retest the system after each change

- By rerunning all of your unit tests
- If something breaks, it's easy to see what caused the failure

# Principles of Refactoring

In general, each refactoring aims to

- decompose large objects into smaller ones
- distribute responsibility

Like design patterns

- Adds composition and delegation
- In some sense, refactorings are ways of applying design patterns to existing code

# Obstacles to Refactoring

## **Complexity**

Changing design is hard, understanding code is hard

## **Possibility to introduce errors**

Mitigated by testing

Clean first Then add new functionality

## **Cultural Issues**

Producing negative lines of code, what an idea!

*We pay you to add new features, not to improve the code!*

## **If it ain t broke, don't fix it**

*We do not have a problem, this is our software!*

# Refactoring: duplicated code

case 0:

```
activePiece = RightHook.getRightHook();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 1:

```
activePiece = LeftHook.getLeftHook();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 2:

```
activePiece = RightRise.getRightRise();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 3:

```
activePiece = LeftRise.getLeftRise();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break; //more
```

Source: Daniel  
H Steinberg

# Refactoring: duplicated code

case 4:

```
activePiece = Hill.getHill();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 5:

```
activePiece = StraightPiece.getStraightPiece();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break;
```

case 6:

```
activePiece = Square.getSquare();  
ml = new MoveListener(activePiece);  
gameBoard.addKeyListener(ml);  
break; //...
```

Source: Daniel H Steinberg

# Refactoring: duplicated code

```
case 0:
    activePiece = RightHook.getRightHook();    break;
case 1:
    activePiece = LeftHook.getLeftHook();       break;
case 2:
    activePiece = RightRise.getRightRise();     break;
case 3:
    activePiece = LeftRise.getLeftRise();       break;
case 4:
    activePiece = Hill.getHill();               break;
case 5:
    activePiece = StraightPiece.getStraightPiece(); break;
case 6:
    activePiece = Square.getSquare();          break;
}
ml = new MoveListener(activePiece);
gameBoard.addKeyListener(ml);
```

Source: Daniel H Steinberg

## Could create an array

case 0:

**activePiece = RightHook.getRightHook(); break;**

case 1:

**activePiece** **Piece [] pieceList = { RightHook.getRightHook(),**

case 2:

**activePiece** **LeftHook.GetLeftHook(), RightRise.getRightRise(),**

case 3:

**activePiece** **LeftRise.getLeftRise(), Hill.getHill(),**

case 4:

**StraightPiece.getStraightPiece(), Square.getSquare() };**

**activePiece = ~~mm.getmm()~~, break,**

case 5:

**activePiece = StraightPiece.getStraightPiece(); break;**

case 6:

**activePiece = Square.getSquare(); break;**

**}**

**ml = new MoveListener(activePiece);**

**gameBoard.addKeyListener(ml);**

Source: Daniel H Steinberg

# Now your switch statement becomes

case 0:

```
activePiece = RightHook.getRightHook();    break;
```

case 1:

```
activePiece = LeftHook.getLeftHook();      break;
```

case 2:

```
activePiece = R
```

case 3:

```
activePiece = L
```

case 4:

```
activePiece = H
```

case 5:

```
activePiece = S
```

case 6:

```
activePiece = S
```

```
public void letsRoll(){
```

```
    activePiece =  
    pieceList[(int)(Math.random()*7)]
```

```
    ml = new MoveListener(activePiece);
```

```
    gameBoard.addKeyListener(ml);
```

```
}
```

```
ml = new MoveListener(activePiece);
```

```
gameBoard.addKeyListener(ml);
```



# Introduce a new method

```
public void letsRoll(){  
    activePiece = pieceList[(int)(Math.random()*7)]  
    ml = new MoveListener(activePiece);  
    gameBoard.addKeyListener(ml);  
}
```

Source: Daniel H Steinberg

```
public void letsRoll(){  
    activePiece = selectNextActivePiece();  
    ml = new MoveListener(activePiece);  
    gameBoard.addKeyListener(ml);  
}  
  
private Piece selectNextActivePiece(){  
    return pieceList[(int) Math.random()*7];  
}
```

# Introduce a new method

We don't need to create a new Move Listener...(and what's ml?)

```
public void letsRoll(){  
    activePiece = selectNextActivePiece();  
    ml = new MoveListener(activePiece);  
    gameBoard.addKeyListener(ml);  
}  
  
private Piece selectNextActivePiece(){  
    return pieceList[(int) Math.random()*7];  
}
```

Source: Daniel H Steinberg

## Introduce a new method

We don't need to create a new Move Listener...(and what's ml?)

```
public void letsRoll(){  
    activePiece = selectNextActivePiece();  
    ml = new MoveListener(activePiece);  
    gameBoard.addKeyListener(ml);  
}  
  
private Piece selectNextActivePie  
return pieceList[(int) Math.rand  
}
```

```
public void letsRoll(){  
    activePiece = selectNextActivePiece();  
    moveListener.setActivePiece(activePiece);  
}  
  
private Piece selectNextActivePiece(){  
    return pieceList[(int) Math.random()*7];  
}
```

# Eliminate the variable activePiece

```
public void letsRoll(){  
    activePiece = selectNextActivePiece();  
    moveListener.setActivePiece(activePiece);  
}
```

```
private Piece selectNextActivePiece()  
{  
    return pieceList[(int) Math.random()*7];  
}
```

```
public void letsRoll(){  
    moveListener.setActivePiece(  
        selectNextActivePiece() );  
}  
  
private Piece selectNextActivePiece(){  
    return pieceList[(int) Math.random()*7];  
}
```

# Goal

## Refactoring

- What is refactoring
- Arguments for refactoring
- Bad Smells
- Obstacles

## Tutorial on MDD and UP (D403b and s601e)

## Design Patterns