Mercutio, do you have a minute?
© 2004 David Morgan-Mar

Just a second, Will. I'm refactoring some of my code.

What does that mean?

It means I'm rewriting it the way it should have been written in the first place, but it sounds cooler.

Refactoring or How to Improve the Design of Existing Code!

- An Introduction -

Peter Steinbach

Institute for Nuclear and Particle Physics, TU Dresden

September 30th, 2011

TECHNISCHE UNIVERSITÄT DRESDEN

Outline

Motivation

What is Refactoring?

What is Refactoring and where is it used?

Unit Testing And Test-Driven Development

Code Smells

Refactorings

Summary

References

Motivation

Use Design-First ?

- **Yes**, before one codes a package, one should sit down with pen and paper
- **No** pen-and-paper design ended up 1:1 in production releases

Motivation

Use Design-First ?

- **Yes**, before one codes a package, one should sit down with pen and paper
- **No** pen-and-paper design ended up 1:1 in production releases

Change is everywhere

- requirements change
  `''Can your package read XML files as well?''`
- environments change
  `''We are moving to tbb::concurrent_vector! Please provide a check-in until tomorrow!''`
- experience grows
  `''Why in the world did I ever write such crap?''`
- upgrade legacy code
  `''Ahh, and here is the code of your predecessor. You can use it as a starting point!''`

What is Refactoring?

Definition

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.[8]

What is REFACTORING?

Definition

REFACTORING is a disciplined technique for restructuring an existing body of code, altering its **internal structure** without changing its **external behavior**.[8]

What is REFACTORING?

Definition

REFACTORING is a disciplined technique for restructuring an existing body of code, altering its **internal structure** without changing its **external behavior**.[8]

History

- 1990 PhD thesis of John Opdyke (student of GoF member Ralph E. Johnson)
- 2000 Martin Fowler's book "REFACTORING" published (standard reference, [8])
- 2004 Micheal Feather's book "Working Effectively with Legacy Code" ([7])

What is REFACTORING and where is it used?

What is it?

- formal method of describing of what most people do anyhow (useful standard)
- setting up check-points to fix external behavior (with unit tests)
- rewrite/reorder internal structure of program

What is REFACTORING and where is it used?

What is it?

- formal method of describing of what most people do anyhow (useful standard)
- setting up check-points to fix external behavior (with unit tests)
- rewrite/reorder internal structure of program

Test-Driven Development

- agile development technique
- coding follows 3 rules
  1. write a unit test that fails
  2. implementation code that makes the test succeed
  3. REFACTORING

What is REFACTORING and where is it used?

What is it?

- formal method of describing of what most people do anyhow (useful standard)
- setting up check-points to fix external behavior (with unit tests)
- rewrite/reorder internal structure of program

Test-Driven Development

- agile development technique
- coding follows 3 rules
  1. write a unit test that fails
  2. implementation code that makes the test succeed
  3. REFACTORING

Legacy Code

- definition vague in literature
- code that needs to be changed to make it ...

What is REFACTORING and where is it used?

What is it?

- formal method of describing of what most people do anyhow (useful standard)
- setting up check-points to fix external behavior (with unit tests)
- rewrite/reorder internal structure of program

Test-Driven Development

- agile development technique
- coding follows 3 rules
  1. write a unit test that fails
  2. implementation code that makes the test succeed
  3. REFACTORING

Legacy Code

- definition vague in literature
- code that needs to be changed to make it ...
  - easier to understand
  - easier to modify

What is REFACTORING and where is it used?

What is it?

- formal method of describing of what most people do anyhow (useful standard)
- setting up check-points to fix external behavior (with unit tests)
- rewrite/reorder internal structure of program

Test-Driven Development

- agile development technique
- coding follows 3 rules
  1. write a unit test that fails
  2. implementation code that makes the test succeed
  3. REFACTORING

Legacy Code

- definition vague in literature
- code that needs to be changed to make it ...
  - easier to understand
  - easier to modify
- dedicated REFACTORING needed to break dependencies etc. (see [7])

Unit Testing

Definition

A method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. (from [2])

Unit Testing

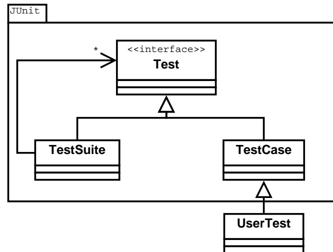
Definition

A method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. (from [2])

Where did it come from?

- JUnit first unit test framework in java (written by Kent Beck and Erich Gamma)
- since then, many ports to other languages written (see [5] and [1])
- provides unified interface and clean environment
- for C++, I prefer boost's unified testing framework ([4])

How does it work?



Unit Testing Demonstrated

Let's have a look at the MagVector class tests

Unit Testing Demonstrated

Let's have a look at the MagVector class tests

- what to test?
  1. contract of a class (its requirements)
  2. how exceptions/errors are handled
- start from a clean instance (see Code Examples, page 3)

Unit Testing Demonstrated

Let's have a look at the MagVector class tests

- what to test?
  1. contract of a class (its requirements)
  2. how exceptions/errors are handled
- start from a clean instance (see Code Examples, page 3)

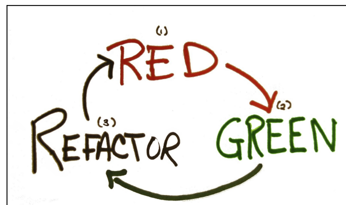
Essentials about unit testing

- important that tests run/compile quick and immediately
- minimize time to find bugs (if occuring)
- provide constant feedback to developer
  (your new class in action before clients use it)
- can serve as documentation
- good IDEs have plugins that make testing very easy
- good investment[a]

---

[a] if used effectively

Test-Driven Development



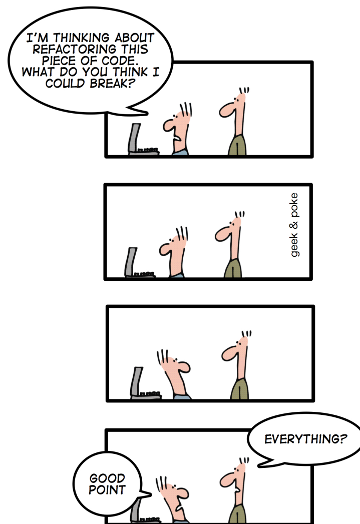
Background

- agile development technique
- formulated by Kent Beck in 2002 ([6])
- enforces simple design and testability

A Demonstration Might Save a 1000 Words!

Adding different norms to `MagVector`!

Unit Testing And Test-Driven Development: Summary



Returning to REFACTORING!

What code needs to be refactored?

What code needs to be refactored?

If it stinks, change it!

What code needs to be refactored?

If it stinks, change it!

Code Smells

- simple heuristics to identify unmaintainable code
- in-class smells (in methods, not touching class interaction)
- inter-class smells (on the structure of classes themselves)
- can't cover all (more extensive list at [3] or in the literature)

in-class smells: The obvious ones

Comments

- clarify "why" not "what"
- can become visual noise
- code should be readable from the first place

```
...
//releasing memory from ptr
delete ptr;
...
```

in-class smells: The obvious ones

Comments

- clarify "why" not "what"
- can become visual noise
- code should be readable from the first place

```
...
//releasing memory from ptr
delete ptr;
...
```

Doublicate Code

- **DRY** principle
- copy-and-pasting hides atomic changes

Dead Code

Delete It![a]

[a] You have version control to go back.

in-class smells: Naming

Type Embedded Name

- renaming is a double take here

```
double getDoubleValue() return myDouble;
```

in-class smells: Naming

Type Embedded Name

- renaming is a double take here

```
double getDoubleValue() return myDouble;
```

Uncommunicative Name

Choose:

```
virtual Double_t GetUxmax() const
```

```
virtual Double_t GetMaximumXInUserCoords() const
```

in-class smells: From typing addicts

Long Method

- the shorter, the easier to read
- fit on laptop screen (640x480)

in-class smells: From typing addicts

Long Method

- the shorter, the easier to read
- fit on laptop screen (640x480)

Large Class

- too many member variables smell like doublicate code

in-class smells: From typing addicts

Long Method

- the shorter, the easier to read
- fit on laptop screen (640x480)

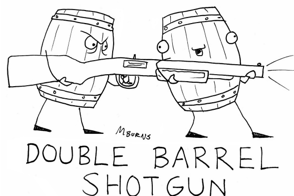
Large Class

- too many member variables smell like doublicate code

Conditional Complexity

- cluttered if/else/elif statements
- switch/case statements

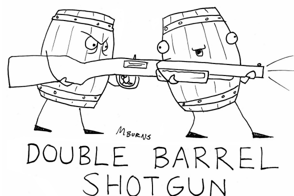
inter-class smells: Changes



Shotgun Surgery

- A change results in the need to make a lot of little changes in several classes
- too tight coupling

inter-class smells: Changes



Shotgun Surgery

- A change results in the need to make a lot of little changes in several classes
- too tight coupling

Divergent Changes

- repeated variations of the system commonly result in changing one class repeatedly
- e.g. adding a new sorting algorithm requires rewriting sorter class every time

inter-class smells: Messaging

Feature Envy

- Often a method that seems more interested in a class other than the one it's actually in

```
void A::alterState(const B& input)
```

inter-class smells: Messaging

Feature Envy

- Often a method that seems more interested in a class other than the one it's actually in

```
void A::alterState(const B& input)
```

Message Chains

```
double result = a.b().c().d();
```

- client has to use one object to get another etc.
- any change to the intermediate relationships causes the client to have to change

inter-class smells: Messaging

Feature Envy

- Often a method that seems more interested in a class other than the one it's actually in

```
void A::alterState(const B& input)
```

Message Chains

```
double result = a.b().c().d();
```

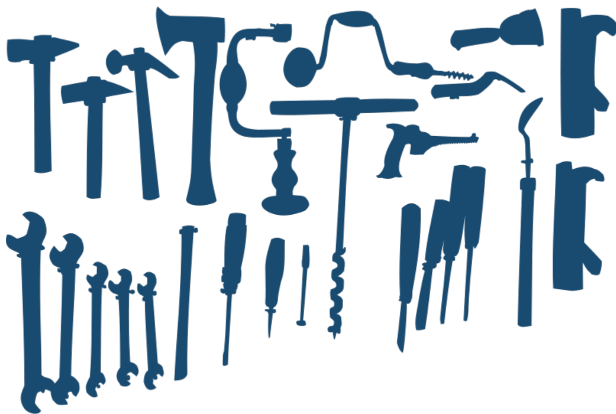
- client has to use one object to get another etc.
- any change to the intermediate relationships causes the client to have to change

Middle Man

```
double result = a.b().c().d();
```

- if a class is delegating almost everything to another class
- why have the middle man at all?

Tools



Tools to do something about it!

Refactorings: Trivial Ones

As we have already heard

- `Delete`
- `Rename Method` (IDEs can do that globally by one click)
- `Add Parameter`, `Remove Parameter`

Refactorings: Trivial Ones

As we have already heard

- `Delete`
- `Rename Method` (IDEs can do that globally by one click)
- `Add Parameter`, `Remove Parameter`

`Extract Field`

before

```
class Course {
 public:
  List Students;
}
```

after

```
class Course {

 private:
  List Students;

 public:
  List  getStudents() const;
  void  setStudents(const List&);

}
```

→ no magic, just good practice (with a name)

Refactorings: Extract Method

- often methods simply do too much
- extract functionality to a separate function

before

```
class Course {
 public:
  double getAverageGrade(){
  ...
  //Compute Score
  result = a*b + c;
  result *= fudgeFactor;
  ...
  };
}
```

after

```
class Course {
 public:
  double computeScore(int a,float b, float c, float fuc
    return (a*b + c)*fudge;
  };
  double getAverageGrade(){
  ...
  result = computeScore(a, b, c, fudge);
  ...
  };
}
```

Refactorings: Extract Class

- break one class into two

before

```
class Course {
 private:
  string name;
  string locationRoom;
  string locationBuilding;
  ...
 }
```

after

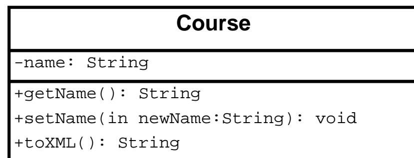
```
class Course {
 private:
  string name;
  GeoLocation location;
  ...
}

class GeoLocation {
 private:
  string Room;
  string Building;
  ...
}
```

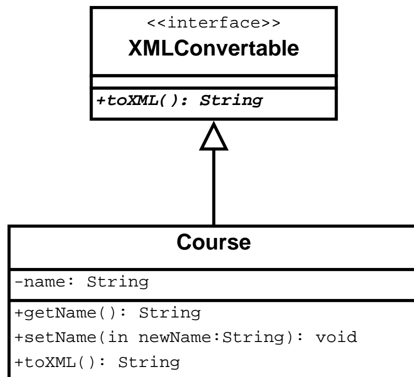
REFACTORINGS: Extract Interface

- functionality of one class might be useful for others as well

before



after



Refactorings: Move Method

- a class method uses more features of another class than the one hosting it

before

```
class Student {
 public:
  bool isTaking(const Course& aCourse){
    return aCourse.getStudents().contains(this);
  };
 }

class Course {
 private:
  List allStudents;

 public:
  List getStudents(){
    return allStudents;
  }
}
```

after

```
class Student {
}

class Course {
 private:
  List allStudents;

 public:
  List getStudents(){
    return allStudents;
  }
  bool isTaking(const Student& aStudent){
    return allStudents.contains(this);
  };
}
```

Refactorings: Replace Error Code by Exception

- a method returns a special code to indicate failure success

before

```
class BlackBoard {
 public:
  int getContainer(const String& Containername,
                   Container& ContainerToLoad){
    ...
    if(notAvailable)
      return -1;
    else
      return 0;
  };
 }
```

after

```
class BlackBoard {
 public:
  void getContainer(const String& Containername,
                    Container& ContainerToLoad){
    ...
    if(notAvailable)
      throw Exceptions::ContainerUnavailable();

  };
}
```

Coming back to our Test-Driven Design Safari

Let's finish our MagVector!

Refactorings: Replace Conditional with Polymorphism

- in order to replace switch/case statements introduce polymorphism

before

```
class Expert {
 enum Animal { lion = 1, tiger = 2, flee =3, ... };
 public:
  float getEstimatedWeight(const Animal& animal){
    switch(animal){
      case 1:
        return 80.;
      case 2:
        return 85.;
      ...
    }
  };
 }
```

after

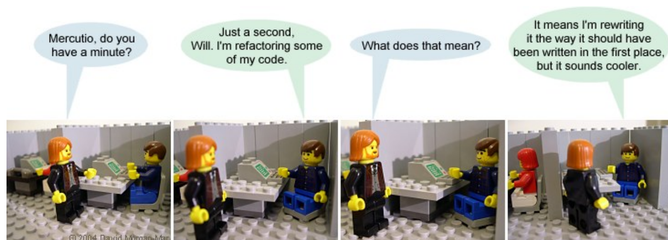
```
class Expert {

 public:
  float getEstimatedWeight(Animal* animal){
    return animal.getWeight();
    }
  };
 }

class Animal {
 public:
  virtual float getWeight() = 0;
}

class Lion: public Animal {
 public:
  virtual float getWeight(){return 80.;};
}
```

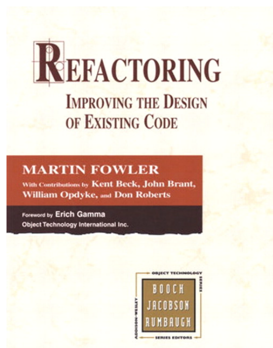
Summary



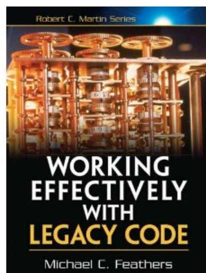
Refactoring

- method catalogue how to change code effectively
  $\rightarrow$ see refactoring.com
- most REFACTORINGs are trivial and atomic
  $\rightarrow$ exactly where bugs come from
- Unit Tests ensure that external behavior remains untouched
- is essential for Test-Driven Development
- is one reason why we have IDEs

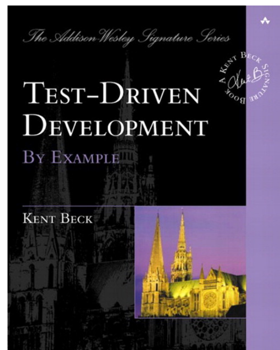
Literature



[8]



[7]



[6]

References

[1] http://en.wikipedia.org/wiki/list_of_unit_testing_frameworks.

[2] http://en.wikipedia.org/wiki/unit_tests.

[3] http://wiki.java.net/bin/view/people/smellstorefactorings.

[4] http://www.boost.org/doc/libs/1_47_0/libs/test/doc/html/utf.html.

[5] http://www.xprogramming.com/software.htm.

[6] Kent Beck.
Test-Driven Development by Example.
Number ISBN 0321146530. Addison-Wesley Longman, 2002.

[7] Michael Feathers.
Working Effectively with Legacy Code.
Robert C. Martin Series. Prentice Hall, 2004.

[8] Martin Fowler et al.
Refactoring - Improving the Design of Existing Code.
Addison Wesley, 2000.