CS247

Refactoring

Adapted from Martin Fowler’s text and Mike Godfrey’s slides.
Today’s topics

• What is refactoring and why do we care?
• The rule of three.
• A simple case study that applies some refactorings.
• A few "bad smells".
• Support for refactoring.
Design matters

http://xkcd.com/1513/
What is refactoring?

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

• Basic metaphor:
  • Start with an existing code base and improve the design.
  • Change the internal structure (in-the-small to in-the-medium) while preserving the overall semantics
    \textit{i.e.}, rearrange the "factors" but end up with the same final "product”.

• The idea is that you should improve the design of the code in some significant way. For example:
  • Consolidate near-duplicate code into a single place.
  • Improve cohesion, lessen coupling.
  • Improve parameterization, understandability, maintainability, flexibility, abstraction, efficiency, etc.
    ... 
  • Typically, we do this by moving, renaming, merging, splitting, pro/demoting, etc. methods.

• Re-run your unit test suite after every change to catch errors.
What does refactoring affect?

• In principle, **the system-level tests should not have to change** after performing refactoring
  • ... because you didn't change the externally-visible behaviour.
  • However, the unit (i.e., class-level) tests will likely change a lot, as you will be moving the contents around and changing how they interact. Also, you will likely be creating new classes and eliminating old ones.

• Bug fixing, adding features, adding new platform support etc. should **not** be performed during a refactoring effort
  • Treat refactoring as a special maintenance task of its own.
Refactoring and “technical debt”

• Failure to refactor leads to crufty designs that are increasingly hard to evolve over time.
  • Rather than "do the right thing", developers are forced to work around bad designs that they don't fully understand.
  • The risk of changing an existing (but poor) design that "works" usually means that bad design choices will fossilize over time; it never makes business sense in the short term to refactor.

• Doing the “quick thing” instead of the “clean design” can help you meet your Friday deadline, but you are effectively borrowing against the integrity of the system's design.
  • This is sometimes called technical debt; the metaphor implies that this is something you ought to "repay" by cleaning up/refactoring the design on Monday.
References

• Reference:
  • *Refactoring: Improving the Design of Existing Code*, by Martin Fowler (*et al.*), 1999, Addison-Wesley
  • See also Fowler's website: 

• Fowler and Beck are rock stars in the OO world
  • Smalltalk experience
  • OO design patterns
  • XP (extreme programming)
When to refactor?

“When should I refactor? How often? How much time should I dedicate to it?”

• It's not something you should dedicate two weeks for every six months; rather, you should do it as you develop!
  • Refactor when you recognize a warning sign (a "bad smell") and know what to do.
  • ... when you add a function
    • Likely it's not an island unto itself.
  • ... when you fix a bug
    • Is the bug symptomatic of a design flaw?
  • ... when you do a code review
    • A good excuse to re-evaluate your designs, share opinions.
When **not** to refactor?

- Invasive changes breaks the mental model other developers have of the code.
  - They will have to spend time learning the new design.

- It's expensive, time-consuming, and requires a lot of re-testing to make sure you didn't break anything.
  - But if you are religious about designing and maintaining your unit tests, the risk is much less.

- Generally, refactoring earlier rather than later is less risky and less expensive.
When not to refactor?

- Fowler says don't refactor when:
  - It would be easier to just rewrite the code from scratch.
  - The code is really buggy and not worth saving; rewrite instead.
  - You are close to a deadline and don't want to incur the risk... at the moment.

- The philosophy of agile development is to "embrace change".
  - Don't fear it, get good at it.
  - Expect to change your mind often, especially early in development as you learn more about the domain, the schedule, and which features turn out to be problematic.
The rule of three (XP)

• The first time you code a given task, *just do it*.

• The second time you code the *same* idea, *wince* and code it up again.

• The third time you code the same idea, it's time to *refactor*!
  • Any programming construct can be made more abstract... but that's not necessarily a good thing.
  • Generality (flexibility) costs too.

• Don't over-engineer; practise *Just-in-Time abstraction*.
  • Don't spin you wheels designing and coding the most abstract system you can imagine.
  • *Expect* that you will be re-arranging your code constantly. Don't worry about it. *Embrace change.*
Fowler’s refactoring catalog

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Duplicate Observed Data
- Dynamic Method Definition
- Eagerly Initialized Attribute
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Module
- Extract Surrounding Method
- Extract Subclass
- Extract Superclass
Fowler’s refactoring catalog

- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier
- Inline Class
- Inline Method
- Inline Module
- Inline Temp
- Introduce Assertion
- Introduce Expression Builder
- Introduce Foreign Method
- Introduce Gateway
- Introduce Local Extension
- Introduce Named Parameter
- Introduce Null Object
- Introduce Parameter Object
- Isolate Dynamic Receptor
- Lazily Initialized Attribute
- Move Eval from Runtime to Parse Time
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
Fowler’s refactoring catalog

- Replace Exception with Test
- Replace Hash with Object
- Replace Inheritance with Delegation
- Replace Loop with Collection Closure Method
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Temp with Chain

- Replace Temp with Query
  ★ Replace Type Code with Class
  - Replace Type Code with Module Extension
  - Replace Type Code With Polymorphism
  - Replace Type Code with State/Strategy
  - Replace Type Code with Subclasses
  - Self Encapsulate Field
  ★ Separate Query from Modifier
  - Split Temporary Variable
  - Substitute Algorithm
Fowler’s refactoring catalog

- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Recompose Conditional
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Named Parameter
- Remove Parameter

- Remove Setting Method
- Remove Unused Default Parameter
- Rename Method
- Replace Abstract Superclass with Module
- Replace Array with Object
- Replace Conditional with Polymorphism

- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Dynamic Receptor with Dynamic Method Definition
- Replace Error Code with Exception
Case study

- Fowler uses a simple case study to motivate the refactoring process.
- Consider a simple movie rental system as a starting point, with three types of movies.
- The statement printed for the customer is initially just a string, but we'd eventually like to be able to create a web version using HTML codes.
Initial customer statement code

```cpp
template::statement() const {
    double totalAmount = 0.0;
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";

    for ( auto each : rentals_ ) {
        double thisAmount = 0.0;

        switch( each.getMovie().getPriceCode() ) {
            case Movie::MovieType::REGULAR:
                thisAmount += 2;
                if ( each.getDaysRented() > 2 ) {
                    thisAmount += ( each.getDaysRented() - 2 ) * 1.5;
                } // if
                break;
        }
    }

    result += "Total amount: 
    return result;
}
```
case Movie::MovieType::NEW_RELEASE:
    thisAmount += each.getDaysRented() * 3;
    break;

case Movie::MovieType::CHILDRENS:
    thisAmount += 1.5;
    if ( each.getDaysRented() > 3 ) {
        thisAmount += ( each.getDaysRented() - 3 ) * 1.5;
    } // if
    break;
} // switch

// add frequent renter points
frequentRenterPoints++;

// add bonus for a two day new release rental
if ( each.getMovie().getPriceCode() == Movie::MovieType::NEW_RELEASE && each.getDaysRented() > 1 ) frequentRenterPoints++;
Initial customer statement code

    // show figures for this rental
    string cost = convert( thisAmount );
    result += "\t\t" + each.getMovie().getTitle() + "\t\t\$" + cost + "\n";
    totalAmount += thisAmount;
} // for

    // add footer lines
    result += "Amount owed is $" + convert(totalAmount) + "\n" +
            "You earned " + convert( frequentRenterPoints ) +
            " frequent renter points";
    return result;
} // Customer::statement
Apply Extract Method (110)

double Customer::amountFor( Rental each ) {
    double thisAmount = 0.0;
    switch( each.getMovie().getPriceCode() ) {
        case Movie::MovieType::REGULAR:
            thisAmount += 2;
            if ( each.getDaysRented() > 2 )
                thisAmount += ( each.getDaysRented() - 2 ) * 1.5;
            break;
        case Movie::MovieType::NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie::MovieType::CHILDRENS:
            thisAmount += 1.5;
            if ( each.getDaysRented() > 3 )
                thisAmount += ( each.getDaysRented() - 3 ) * 1.5;
            break;
    } // switch
    return thisAmount;
} // Customer::amountFor
double Customer::amountFor( Rental aRental ) {
    double result = 0.0;
    switch( aRental.getMovie().getPriceCode() ) {
        case Movie::MovieType::REGULAR:
            result += 2;
            if ( aRental.getDaysRented() > 2 )
                result += ( aRental.getDaysRented() - 2 ) * 1.5;
            break;
        case Movie::MovieType::NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie::MovieType::CHILDRENS:
            result += 1.5;
            if ( aRental.getDaysRented() > 3 )
                result += ( aRental.getDaysRented() - 3 ) * 1.5;
            break;
    } // switch
    return result;
} // Customer::amountFor
Apply Move Method (142)

double Rental::getCharge() {
    double result = 0.0;
    switch( getMovie().getPriceCode() ) {
        case Movie::MovieType::REGULAR:
            result += 2;
            if ( getDaysRented() > 2 )
                result += (getDaysRented() - 2) * 1.5;
            break;
        case Movie::MovieType::NEW_RELEASE:
            result += getDaysRented() * 3;
            break;
        case Movie::MovieType::CHILDRENS:
            result += 1.5;
            if ( getDaysRented() > 3 )
                result += (getDaysRented() - 3) * 1.5;
            break;
    } // switch
    return result;
} // Rental::getCharge

double Customer::amountFor( Rental aRental ) {
    return aRental.getCharge();
}
string Customer::statement() const {
    double totalAmount = 0.0;
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";
    for ( Rental each : rentals_ ) {
        double thisAmount = amountFor(each);
        frequentRenterPoints++; // add frequent renter points
        if ( each.getMovie().getPriceCode() ==
            Movie::MovieType::NEW_RELEASE && each.getDaysRented() > 1 )
            frequentRenterPoints++;
        string cost = convert(thisAmount);
        result += "\t" + each.getMovie().getTitle() + "\t" + cost + "\n";
        totalAmount += thisAmount;
    } // for
    result += "Amount owed is $" + convert(totalAmount) + "\n" +
        "You earned " + convert( frequentRenterPoints ) +
        " frequent renter points";
    return result;
}
Revised code, take 3

```cpp
string Customer::statement() const {
    double totalAmount = 0.0;
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";
    for (Rental each : rentals_) {
        double thisAmount = each.getCharge();
        frequentRenterPoints++;
        // add frequent renter points
        if (each.getMovie().getPriceCode() ==
            Movie::MovieType::NEW_RELEASE &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;
        string cost = convert(thisAmount);
        result += "\t" + each.getMovie().getTitle() + "\t" + cost + "\n";
        totalAmount += thisAmount;
    } // for
    result += "Amount owed is $" + convert(totalAmount) + "\n" +
        "You earned " + convert(frequentRenterPoints) + " frequent renter points";
    return result;
}
```
string Customer::statement() const {
    double totalAmount = 0.0;
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";
    for ( auto each : rentals_ ) {
        frequentRenterPoints++; // add frequent renter points
        if ( each.getMovie().getPriceCode() ==
            Movie::MovieType::NEW_RELEASE && each.getDaysRented() > 1 )
            frequentRenterPoints++;
        string cost = convert(each.getCharge());
        result += "\t" + each.getMovie().getTitle() + "\t$" + cost + "\n";
        totalAmount += each.getCharge();
    } // for
    result += "Amount owed is $" + convert(totalAmount) + "\n" +
            "You earned " + convert( frequentRenterPoints ) +
            " frequent renter points";
    return result;
}
Apply Extract Method + Move Method

```cpp
string Customer::statement() const {
    double totalAmount = 0.0;
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";

    for ( auto each : rentals_ ) {
        frequentRenterPoints = each.getFrequentRenterPoints();
        string cost = convert(each.getCharge());
        result += "\t" + each.getMovie().getTitle() + "\t$" + cost + "\n";
        totalAmount += each.getCharge();
    } // for

    result += "Amount owed is $" + convert(totalAmount) + "\n" + "You earned " + convert( frequentRenterPoints ) + " frequent renter points";
    return result;
}
```
double Customer::getTotalCharge() const {
    double result = 0.0;
    for (auto each : rentals_ ) result += each.getCharge();
    return result;
} // Customer::getTotalCharge

string Customer::statement() const {
    int frequentRenterPoints = 0;
    string result = "Rental record for " + nameIs() + "\n";
    for (auto each : rentals_ ) {
        frequentRenterPoints = each.getFrequentRenterPoints();
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  convert(each.getCharge()) + "\n";
    } // for
    result += "Amount owed is $" + convert(getTotalCharge()) + "\n" +
              "You earned " + convert( frequentRenterPoints ) +
              " frequent renter points";
    return result;
}
int Customer::getTotalFrequentRenterPoints() const {
    int result = 0.0;
    for ( auto each : rentals_ ) {
        result += each.getFrequentRenterPoints();
    } // for
    return result;
} // Customer::getTotalFrequentRenterPoints

string Customer::statement() const {
    string result = "Rental record for " + nameIs() + "\n";
    for ( auto each : rentals_ ) {
        result += "\t" + each.getMovie().getTitle() + "\t" +
                   convert(each.getCharge())+ "\n";
    } // for
    result += "Amount owed is $" + convert(getTotalCharge()) + "\n" +
              "You earned " + convert( getTotalFrequentRenterPoints() ) + " frequent renter points";
    return result;
}
Add htmlStatement

```cpp
string Customer::htmlStatement() const {
    string result = "<H1>Rentals for <EM>" + getName() + "</EM></H1><P>
    for ( auto each : rentals_ ) {
        // show figures for this rental
        string cost = convert( each.getCharge() );
        result += each.getMovie().getTitle() + " : $" + cost + "<BR>\n";
    } // for

    // add footer lines
    result += "</P><P>You owe <EM>$" + convert( getTotalCharge() ) + 
    "</EM></P><P>\" + "On this rental you earned <EM>" + 
    convert( getTotalFrequentRenterPoints() ) + 
    "</EM> frequent renter points</P>";
    return result;
} // Customer::htmlStatement
```
Case study after initial revisions

```java
// Customer
- name : string
+ Customer( name : string )
+ getName() : string {readonly}
+ addRental( Rental r ) : void
+ statement() : string {readonly}
- getTotalCharge() : double {readonly}
- getTotalFrequentRenterPoints() : int {readonly}
+ htmlStatement() : string {readonly}

// Rental
- daysRented : int
+ Rental( Movie m, int daysRented)
+ getDaysRented() : void {readonly}
+ getMovie() : Movie {readonly}
+ getCharge() : double {readonly}
+ getFrequentRenterPoints() : int {readonly}

// Movie
- title : string
- priceCode : MovieType
+ Movie( title : string, priceCode: MovieType )
+ getTitle() : string {readonly}
+ getPriceCode() : MovieType {readonly}
+ setPriceCode( p : MovieType )
```

MovieType = {REGULAR, NEW_RELEASE, CHILDRENS}
Move getCharge calculation to Movie

double Movie::getCharge( int numDaysRented ) const {
    double result = 0.0;
    switch( getPriceCode() ) {
        case MovieType::REGULAR:
            result += 2;
            if ( numDaysRented > 2 ) result += ( numDaysRented - 2 ) * 1.5;
            break;
        case MovieType::NEW_RELEASE:
            result += numDaysRented * 3;
            break;
        case MovieType::CHILDRENS:
            result += 1.5;
            if ( numDaysRented > 3 ) result += ( numDaysRented - 3 ) * 1.5;
            break;
    } // switch
    return result;
} // Movie::getCharge

double Rental::getCharge() const {
    return getMovie().getCharge( getDaysRented() );
} // Rental::getCharge
Move `getFrequentRenterPoints` calculation to Movie

```cpp
int Movie::getFrequentRenterPoints( int numDaysRented ) const {
    if ( getPriceCode() == MovieType::NEW_RELEASE && numDaysRented > 1 )
        return 2;
    return 1;
} // Movie::getFrequentRenterPoints

int Rental::getFrequentRenterPoints() const {
    return getMovie().getFrequentRenterPoints( getDaysRented() );
}
```
Self Encapsulate Field

Movie::Movie( string title, Movie::MovieType priceCode ) :
  title_{title} { setPriceCode( priceCode ); }

void Movie::setPriceCode( Movie::MovieType arg ) {
  priceCode_ = arg;
} // Movie::setPriceCode
Use the new Price class hierarchy

```cpp
Movie::~Movie() { delete price_; }

void Movie::setPriceCode( Movie::MovieType arg ) {
    delete price_;  
    switch( arg ) {
        case MovieType::REGULAR:
            price_ = new RegularPrice();
            break;
        case MovieType::NEW_RELEASE:
            price_ = new NewReleasePrice();
            break;
        case MovieType::CHILDRENS:
            price_ = new ChildrensPrice();
            break;
    } // switch
} // Movie::setPriceCode
```
Replace Conditional with Polymorphism (225)

double Price::getCharge( int numDaysRented ) const = 0;

double NewReleasePrice::getCharge( int numDaysRented ) const {
    return numDaysRented * 3;
} // NewReleasePrice::getCharge

double RegularPrice::getCharge( int numDaysRented ) const {
    double result = 0.0;
    result += 2;
    if ( numDaysRented > 2 ) result += ( numDaysRented - 2 ) * 1.5;
    return result;
} // RegularPrice::getCharge

double ChildrensPrice::getCharge( int numDaysRented ) const {
    double result = 0.0;
    result += 1.5;
    if ( numDaysRented > 3 ) result += ( numDaysRented - 3 ) * 1.5;
    return result;
} // ChildrensPrice::getCharge
Replace Conditional with Polymorphism (225)

```cpp
double Price::getFrequentRenterPoints( int numDaysRented ) const {
    return 1;
} // Price::getFrequentRenterPoints

int NewReleasePrice::getFrequentRenterPoints( int numDaysRented ) const {
    if ( numDaysRented > 1 ) return 2;
    return 1;
} // NewReleasePrice::getFrequentRenterPoints
```
Case study after further revisions
Bad smells & what to do about them

• Beck & Fowler's list 22 creatively-named "bad smells"; for each one:
  • describes what it looks like,
  • why it can be problematic,
  • why it often emerges in code, and
  • what refactorings might be applicable to make it less odourous.

• You can find out about more design anti-patterns here:
  • http://en.wikipedia.org/wiki/Antipattern

• For each refactoring, they list:
  • a description, the motivation, mechanics and an example.
Bad smells catalog

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies

- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class
- Refused bequest
- Comments
Bad smell: **Duplicated code**

- Same expressions in two methods of the same class.
  - Apply *Extract Method (110)* and invoke from both places.

- Same expression in two sibling subclasses.
  - Apply *Extract Method (110)* and then *Pull Up Method (322)*.

- Similar expression in two sibling subclasses.
  - Apply *Extract Method (110)* to separate similar bits from different bits and then *Form Template Method (345)*.

- If the methods do the same thing with a different algorithm,
  - apply *Substitute Algorithm (139)*.

- Duplicate code in two unrelated classes.
  - May want to use *Extract Class (149)* in one class and then use the new component in the other.
  - Or maybe use the Strategy design pattern i.e. *Replace Method with Method Object (135)*
Not all code duplication is bad

1. Forking
   - Hardware variation eg., Linux SCSI drivers
   - Platform variation
   - Experimental variation

2. Templating
   - Boilerplating
   - API / library protocols
   - Generalized programming idioms
   - Parameterized code

3. Customizing
   - Bug workarounds
   - Replicate + specialize

“`Cloning Considered Harmful' Considred Harmful: Patterns of Cloning in Software”,
Bad smell: **Long method**

- Best to think carefully about the major tasks and how they inter-relate. Be aggressive!
  - Break up into smaller private methods within the class (*Extract method*)
  - Delegate subtasks to sub-objects that "know best" i.e., Template Method design pattern. (*Extract Class/Method, Replace Data Value with Object*)

- Fowler's heuristic: if you see a comment, this is a good place to break it up.

- Try to get rid of temporary variables and long parameter lists.
  - *Replace Temp with Query (120)*
  - *Replace Parameter with Method (292)/Introduce Parameter Objects (295)/Preserve Whole Object (288)*

- Extract conditionals and loops.
  - *Decompose Conditional (238)*
  - Extract loop and associated code into its own method.
Bad smell: **Large class**

- If a class has too many instance variables, this may be a sign that the class is trying to do too much.
  - *Extract Class (149)*/ *Extract Subclass (330)*

- If a class has too much code, try to eliminate redundancy.
  - *Extract Class (149)*/ *Extract Subclass (330)*/ *Extract Interface (341)*

**Counter example:**

- Library classes often have large, fat interfaces (many methods, many parameters, lots of overloading, ...).

- If the many methods exist for the purpose of flexibility, that's OK in a library class.
Bad smell: **Long parameter list**

- Long parameter lists make methods difficult for clients to understand.
- This is often a symptom of:
  - Trying to do too much, too far from home, with too many disparate subparts.
- Solution:
  - Trying to do too much?
    - Break up into sub-tasks. *(Extract Method)*
  - ... too far from home?
    - Localize passing of parameters; don't blithely pass down several layers of calls.
    - *(Preserve Whole Object, Introduce Parameter Object)*
  - ... with too many disparate subparts?
    - Gather up parameters into aggregate subparts.
    - *(Preserve Whole Object, Introduce Parameter Object)*
Bad smell: **Divergent change**

- One class is commonly changed in different ways for different reasons.
  - The class is trying to do too much + contains many unrelated subparts
- Over time, some classes develop a "God complex".
  - They acquire details/ownership of subparts that belong elsewhere.
  - Unrelated elements in the same containers are a sign of poor cohesion.

**Solution:**
- Break it up, reshuffle, reconsider relationships and responsibilities.
- *(Extract Class)*
Bad smell: **Shotgun surgery**

- The opposite of divergent change.
  - Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways.
- Also a classic sign of poor cohesion.
  - Related elements are not in the same container!

**Solution:**
- Look to do some gathering, either in a new or existing class.
  - *(Move Method/Field)*
Bad smell: Feature envy

• A method seems more interested in another class than the one it's defined in.
  • eg., 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 design patterns 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 smell: **Primitive obsession**

- All subparts of an object are instances of primitive types
  - (int, string, bool, double, etc.)
  - eg., dates, currency, SIN, telephone number, ISBN, special string values
- Often, these small objects have interesting and non-trivial constraints that can be modelled
  - eg., 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 mode strongly typed
  - *(Replace Data Value with Object, Extract Class, Introduce Parameter Object)*
Bad smell: **Lazy class**

- A class doesn't do much that's different from other classes
  - eg., if there are several sibling classes that don't exhibit polymorphic behavioural differences, then consider just collapsing them back into the parent and add some parameters
- Often, lazy classes are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour
  - *(Collapse Hierarchy, Inline Class)*
Automated support for refactoring

• There are some really good refactoring tools in the Java world:
  • especially, the (open source, free) Eclipse IDE works well, and
  • also, IntelliJ (not free, but not expensive).

• Until recently, there were no really good tools for C++.
  • Mostly, this reflects the complexity of the C++ language and the fact that it requires multiple
    phases to "understand" the code.
  • MS Visual Studio (not free) has some support for refactoring C++ via a (not free) third-party
    add-on tool, Visual Assist.
  • Eclipse CDT has some support too for C/C++.
  • Google CLANG.
  • As of 2015, some people also recommended Klocwork, ReSharper Ultimate and CLion in a
    thread on Stack Overflow (http://stackoverflow.com/questions/1388469/is-there-a-working-
c-refactoring-tool)
Automated testing

• For every new feature you add, add the relevant tests and re-run all of your previous tests.
  • If done automatically, easy to do. Tests should check their own results so you don't have to!
  • Set up test suite to let you know which tests failed; may want to introduce an error just to verify your test code.
  • Since it's the code you've been working on most recently, easier to find mistakes.

• Unit testing versus functional testing.

• Test exceptions.
Summary

• Fowler et al.'s "Refactoring" is a well-written book that summarizes a lot of "best practices" of OOD/OOP with nice examples
  • Most of the advice in this book is aimed at low-level OO programming.
  • i.e., loops, variables, method calls, and class definitions.
  • Next obvious step up in abstraction/scale is to object-oriented design patterns.
  • i.e., collaborating classes

• It's an excellent book for the intermediate-level OO programmer.
• Experienced OO programmers will have discovered a lot of the techniques already on their own.
What to get out of this?

• Recognition
  • What is a “bad smell”?
  • What is refactoring?

• Comprehension
  • Considerations of whether to refactor (e.g. effort, XP rule of three).
  • Argue different design alternatives:
    • extract/merge methods or classes
    • primitive data members vs. object members
    • multiple parameters vs. parameter objects
    • conditional code in methods vs. polymorphism
    • inheritance vs. composition/delegation