Refactoring

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- What is refactoring?
- Code smells
- Refactoring techniques

What is refactoring?

What is refactoring?

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***

Refactoring (verb):

to restructure *software by applying a series of refactorings without changing its observable behavior*

(Martin Fowler)

When do we refactor?

- We need to add a feature to a program
  - but it is inconvenient: the code is not well structured
- Well structured program:
  - it is convenient to add a new feature
  - easy to maintain
- If the feature is inconvenient to add:
  - 1. Refactor the program, to make it easier to add the feature
  - 2. Add the feature

When to refactor?

- Rule of thumb: Three Strikes and You Refactor
  - *The first time you do something, you just do it.*
  - *The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway.*
  - *The third time you do something similar, you refactor.*
- When adding a function
  - maintains code structure and good design
- When needing to fix a bug
  - clearer structure helps understanding
- When doing code review
  - makes code more clear to review

Refactoring vs. adding a new feature

- Refactoring:
  - is about changing existing code
  - no new functionality
  - only makes the code better
- Adding a feature:
  - existing code is not changed
  - usually just extending the program with new modules, classes or methods

Rules of refactoring

- Must have a solid set of tests
    - functional requirements must be kept
    - refactoring changes code:
        - bugs can happen
        - human error
    - automatic tests are best
        - helps repeating them without effort
- Take small steps
    - bugs are easier to find
    - check often: unit testing
- Have no fear of changing names
    - code must be understood by humans
    - good IDE helps with changes

Steps of refactoring

- 1. Have a solid set of unit tests
  - use existing ones
  - create new ones
- 2. Make sure the tests pass on the old code
- 3. Make a small change
  - easier to test
  - the changed code must be (more) readable
- 4. Run tests on the changed code
  - do not start other changes until the new code passes the tests
- 5. Repeat for other changes from step 1.

Advantages of refactoring

- Improves design
  - design decays: with each modification the code gets worse
  - refactoring helps to keep the structure
- Makes the code easier to understand
  - code is more read than written
  - people will have to maintain it
- Helps finding bugs
  - for the code to be refactored it must be understood
  - during rewriting bugs can emerge
- Helps faster programming
  - sounds counterintuitive
  - without good design no fast change can be made

Problems of refactoring

- Databases
  - tables are rigid
  - code might rely on them
  - object-relational mapping layer might be needed
- Interfaces
  - be careful with changing public interface
  - retain and support the old interface for a while
    - mark the old one deprecated
  - don't publish interfaces prematurely

Code smells

Code smells

- How to find code needing refactoring?
  - No clear criteria
- Code smells are close
  - bad designs that catch attention
  - identification is half victory: solution is usually easy or trivial
- *A code smell is a surface indication that usually corresponds to a deeper problem in the system (Martin Fowler)*
- *Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality*
- Code smells are not bugs:
  - they are not technically incorrect
  - they do not prevent the program from functioning
- But:
  - they just indicate weaknesses in design
  - they may slow down development
  - they may increase the risk of bugs and failures in the future

S1. Duplicated code

- Description:
  - identical or very similar code exists in more than one location
- Problem:
  - violation of DRY
  - usually a violation of TDA
  - modification is error-prone
- Refactor:
  - Extract method
  - Extract class

S2. Long method

- Description:
  - method is too long
  - too many branches or loops
- Problem:
  - difficult to understand
  - difficult to modify
- Refactor:
  - Split into multiple methods
    - e.g. along comments: they indicate semantic distance between blocks
  - Introduce method object
    - create a class with multiple methods from the long method
  - Decompose conditionals, loops, blocks into methods

S3. Long parameter lists

- Description:
  - method has too many (>3) parameters
- Problem:
  - difficult to pass parameters from the client code
  - difficult to understand
- Refactor:
  - Split into multiple methods
  - Change long parameter lists to parameter objects

S4. Large class

- Description:
  - class has too many methods
- Problem:
  - class has too much responsibility -> violation of SRP
  - probably it is a god-class
  - clients probably don’t use all the methods -> violation of ISP
- Refactor:
  - Split up the class into smaller classes
  - Consider creating an inheritance hierarchy
  - Distribute the responsibilities among other classes
  - Use ISP

S5. Divergent change

- Description:
  - class is changed from variation to variation
    - e.g. different UI technologies, different DB drivers, etc.
- Problem:
  - might cause unnecessary changes in other parts of the code
- Refactor:
  - Separate variety-specific part into a class: instable classes
  - Non-changing parts into different class: stable classes
    - let the rest of the code depend only from the stable classes

S6. Shotgun surgery

- Description:
  - a change results in many small alterations in other classes
    - e.g. changing units from imperial to metric, changing literal values, etc.
  - excessive use of literals
- Problem:
  - changes are error-prone
  - violation of DRY
- Refactor:
  - Put all changes into single class
    - e.g. create a constant for literal values
  - Create new class if necessary
    - e.g. create a utility class
  - Literals should be coded as named constants, to improve readability and to avoid programming errors
  - Literals can and should be externalized into resource files

S7. Feature envy

- Description:
  - a method is too interested in an other class
    - usually interest concerns data
- Problem:
  - responsibility at the wrong place
  - high coupling between the method and the class
- Refactor:
  - Move the method to the other class: higher cohesion
  - Extract the relevant part of the method and put into the other class: lower the coupling
  - Put things together that change together: high cohesion

S8. Data clumps

- Description:
  - data items group naturally
    - e.g. people's name, age, etc.
  - same group of parameters across multiple method calls
- Problem:
  - procedural design, not OO
  - leads to data classes + god classes
- Refactor:
  - Encapsulate data into classes
  - Create parameter object
  - Look for methods in other classes (feature envy)

S9. Primitive obsession

- Description:
  - data is stored in primitive types instead of classes
- Problem:
  - not OO, can not be extended easily
  - behavior cannot be attached to primitive types
- Refactor:
  - Replace groups of primitive data with class(es)
  - Replace type code with class (inheritance)
  - Replace type code with state/strategy
  - Replace array of different items with object

S10. Switch statements

- Description:
  - switch statement in code
  - null-checking
- Problem:
  - usually leads to code duplication
  - misplaced responsibility
- Refactor:
  - Replace type code with class (inheritance)
  - Replace type code with state/strategy
  - Introduce NullObject

S11. Parallel inheritance hierarchies

- Description:
  - every time you create a new subclass, you also have to make a subclass of another
  - usually the classes of an inheritance hierarchy have the same prefixes/suffixes
  - subcase of shotgun surgery
- Problem:
  - leads to dependent modifications, duplications
- Refactor:
  - Move methods and fields from the referring hierarchy to the other: the referring hierarchy disappears
- Caution:
  - parallel hierarchies may be a deliberate design decision (e.g. simulating multiple inheritance)

S12. Lazy class

- Description:
  - class is not doing enough
  - could have been downsized by refactoring
  - could have been added because of changes that were planned but not made
- Problem:
  - overkill to maintain
- Refactor:
  - Eliminate it
  - Inline the class
  - If it is a subclass, collapse the hierarchy

S13. Speculative generality

- Description:
  - "we might need this ability someday“
  - heavy extension machinery which is not used
  - code only used by tests
  - forced usage of overcomplicated design patterns where simpler design would suffice
- Problem:
  - too much unnecessary code to maintain
  - violation of YAGNI
- Refactor:
  - Get rid of the unused heavy machinery
  - Collapse hierarchy
  - Unused parameters should be removed

S14. Temporary field

- Description:
  - an attribute is set only in certain circumstances
    - e.g. object scope "global" helper variables
    - e.g. variables used only in some of the methods running a complex algorithm, but not in others
- Problem:
  - difficult to understand and maintain
  - an object does not use all of its variables
    - low cohesion
- Refactor:
  - Extract such attributes to new class
    - with relevant methods as well
  - Introduce NullObject to eliminate conditional code

S15. Message chains

- Description:
  - too long method chains
    - e.g. getA().getB().getC()....
- Problem:
  - client is coupled to the structure of the navigation
  - change to the intermediate relationships causes the client to have to change
  - violation of LoD
- Refactor:
  - Hide delegation
  - Move methods between classes

S16. Middle man

- Description:
  - too much delegation a class to another
- Problem:
  - delegation overhead
- Refactor:
  - Remove the middle man (talk to the target class directly)
  - Inline methods
  - Replace delegation with inheritance

S17. Inappropriate intimacy

- Description:
  - classes accessing each other’s private members directly
- Problem:
  - responsibilities at the wrong place
  - too much coupling between the classes
- Refactor:
  - Move methods and fields between the classes to reduce coupling
  - Change bidirectional association to unidirectional
  - Let another class act as go-between
  - Replace delegation with inheritance

S18. Alternative classes with different interfaces

- Description:
  - classes for the same task having different interfaces
- Problem:
  - classes are not interchangeable
- Refactor:
  - Rename methods
  - Move methods into other classes if necessary
  - Extract superclass if possible
  - Goal: reach a common interface

S19. Incomplete library class

- Description:
  - library class (server) can not be modified
- Problem:
  - usual refactoring does not work on it, we need to make an adaptor
- Refactor:
  - Introduce new method into client, server is parameter
  - Create new subclass of server with new functionality

S20. Data class

- Description:
  - class with only setter and getter methods
- Problem:
  - not OO, encapsulation is violated
  - class has no responsibility
- Refactor:
  - Remove setting method on read only attributes
  - Move behavior into the data class from clients
    - both whole and partial methods might work
  - Goal: the class has to gain real responsibility

S21. Refused bequest

- Description:
  - subclass doesn't need the superclass functionality
- Problem:
  - not strong smell, but can cause confusion
  - possibly the inheritance order is wrong
  - superclass has unnecessary responsibility
- Refactor:
  - Reorder the inheritance hierarchy
  - Push down method or field into relevant subclass
  - If parent interface is refused, replace inheritance with delegation

S22. Comments

- Description:
    - too much explanation comment in the code
- Problem
    - the code is overcomplicated
- Solution
    - Extract methods and simplify the code
    - Rename method if necessary
    - If comment is needed to clarify what the code is doing, try to refactor
        - comments should say why you did something

S23. Downcasting

- Description:
  - use of type cast
  - use of instanceof
- Problem:
  - a type cast breaks the abstraction model
  - violation of OCP, LSP
- Refactor:
  - The abstraction may have to be refactored or eliminated
  - Move the behavior into the class to which you cast

Refactoring techniques

Composing methods

- F1. Extract method
  - take a piece of code and turn it into a method
- F2. Inline method
  - take a method call and replace it with the body of the method
- F3. Inline temporary
  - if the temporary variable is used only once, get rid of it
- F4. Replace temporary with query
  - extract temporary variable as a method
- F5. Introduce explaining variable
  - replace a complex expression with a temporary variable
- F6. Split temporary variable
  - use separate temporary variables for unrelated assignments
- F7. Remove assignments to parameters
  - use a temporary variable instead of assigning to a parameter
- F8. Replace method with method object
  - extract method with local variables
- F9. Substitute algorithm
  - replace algorithm with a clearer one

Moving features between objects

- F10. Move method
  - move responsibility from one class to another
- F11. Move field
  - move field from one class to another
- F12. Extract class
  - select some fields and methods and create a new class for them
- F13. Inline class
  - eliminate a class by moving its fields and methods into another class
- F14. Hide delegate
  - create a method to prevent call chaining
- F15. Remove middle man
  - get the client to call the delegate directly
- F16. Introduce foreign method
  - put a new method in the client with the server as parameter
- F17. Introduce local extension
  - create a new subclass of the server with the new methods

Organizing data I.

- F18. Self encapsulate field
  - create getter/setter methods for the field
- F19. Replace data value with object
  - turn data item into an object
- F20. Change value to reference
  - turn many equal instances to references (e.g. flyweight)
- F21. Change reference to value
  - turn immutable reference objects to separate instances
- F22. Replace array with object
  - replace the array with an object that has a field for each element
- F23. Duplicate observed data
  - e.g. database – model – GUI layers
- F24. Change unidirectional association to bidirectional
  - two-way administration
- F25. Change bidirectional association to unidirectional
  - drop the unneeded end

Organizing data II.

- F26. Replace magic numbers with symbolic constant
  - name literals as constants
- F27. Encapsulate field
  - make a non-private attribute private and provide accessors
- F28. Encapsulate collection
  - provide add/remove methods, provide read-only view
- F29. Replace record with data class
  - interface with a traditional programming environment
- F30. Replace type code with class
  - multiple classes instead of a type code
- F31. Replace type code with subclasses
  - inheritance and polymorphism instead of a type code
- F32. Replace type code with state/strategy
  - if inheritance cannot be used replace type code with a state/strategy object
- F33. Replace subclass with fields
  - subclasses have no added behavior, move methods that return constant data to the superclass as fields

Simplifying conditional expressions

- F34. Decompose conditional
  - extract methods from the if, then, else parts
- F35. Consolidate conditional expression
  - combine sequence of conditional tests with the same result into a single conditional expression
- F36. Consolidate duplicate conditional fragments
  - move same fragments of code in all branches of an if outside
- F37. Remove control flag
  - use a break or return instead of a control flag variable
- F38. Replace nested conditional with guard clauses
  - replace nested conditionals with a series of if-else constructs
- F39. Replace conditional with polymorphism
  - replace conditional depending on the type of an object with polymorphism
- F40. Introduce null object
  - replace the null value with a null object to avoid null-checks
- F41. Introduce assertion
  - make assumptions explicit with assertions

Making method calls simpler I.

- F42. Rename method
  - change name of the method to reveal its purpose
- F43. Add parameter
  - add a parameter to pass more information from caller
- F44. Remove parameter
  - remove parameter if it is not used anymore
- F45. Separate query from modifier
  - create two methods, one for the querying and one for the modification
- F46. Parameterize method
  - combine similar methods into one method with additional parameters
- F47. Replace parameter with explicit methods
  - split a method with multiple cases to multiple methods with fewer parameters
- F48. Preserve whole object
  - instead of passing parts of an object as multiple parameters, pass the whole object

Making method calls simpler II.

- F49. Replace parameter with method
  - instead of passing the result of one method to another, let the second method call the first method
- F50. Introduce parameter object
  - replace a group of parameters that naturally go together with an object
- F51. Remove setting method
  - set attributes in the constructor, do not provide setter methods
- F52. Hide method
  - if other classes do not use a method, make it private
- F53. Replace constructor with factory method
  - if more than a simple construction is needed, use a factory method
- F54. Encapsulate downcast
  - if the result of a method needs to be downcasted by the clients, move the downcast into the method
- F55. Replace error code with exception
  - throw an exception instead of returning special values
- F56. Replace exception with test
  - instead of catching exceptions check the parameters before passing them to the server

Dealing with generalization I.

- F57. Pull up field
  - move a common field in two subclasses to a superclass
- F58. Pull up method
  - move a common method in two subclasses to a superclass
- F59. Pull up constructor body
  - move the identical parts of the constructors of two subclasses to a superclass
- F60. Push down method
  - if a method is relevant only for a subset of the subclasses, move it to those subclasses
- F61. Push down field
  - if a field is relevant only for a subset of the subclasses, move it to those subclasses
- F62. Extract subclass
  - if a subset of features is relevant only for a subset of the subclasses, create a subclass for those features

Dealing with generalization II.

- F63. Extract superclass
  - if two subclasses have similar features, create a superclass from these features
- F64. Extract interface
  - if several classes have a common interface, extract this subset into an interface
- F65. Collapse hierarchy
  - if the subclass add little or no additional behavior, merge it with its superclass
- F66. Form template method
  - if two subclasses have similar methods performing similar steps in the same order, make a template method from them in the superclass
- F67. Replace inheritance with delegation
  - if the subclass uses only a subset of the superclass’s interface, use delegation instead of inheritance
- F68. Replace delegation with inheritance
  - if the class does mostly delegation, make the delegating class a subclass of the delegate

High level refactoring

- F69. Tease apart inheritance
  - if an inheritance hierarchy is doing two jobs at once, split them into two hierarchies
- F70. Convert procedural design to objects
  - move behavior to data classes
- F71. Separate domain from presentation
  - move domain logic from the GUI to separate domain classes
- F72. Extract hierarchy
  - if a class does too much work, create a hierarchy of classes in which each subclass represents a special case