# Refactoring

Incrementally Improving Quality



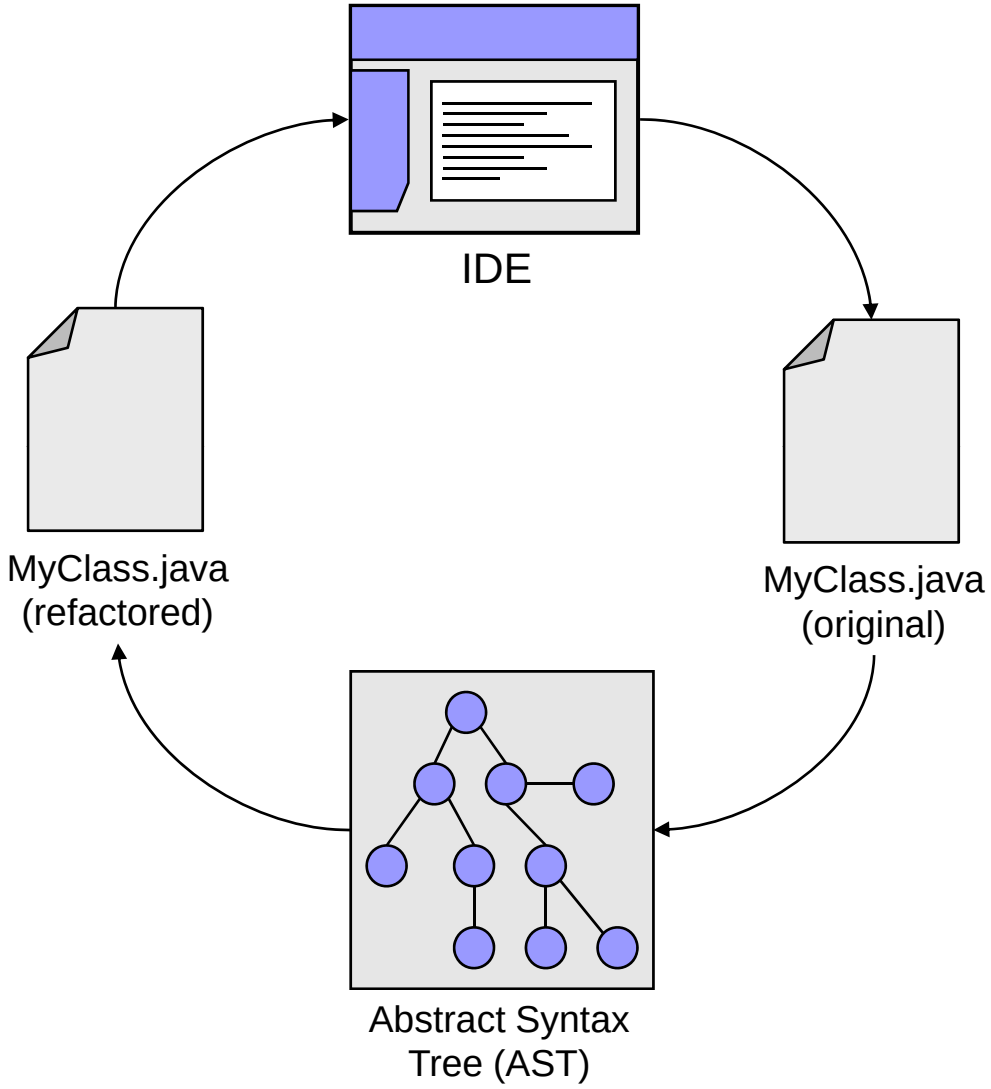
garth@ggilmour.com

# Introducing Refactoring

- Refactoring is:
  - A separate activity from development
    - Refactoring does not add new functionality
    - It makes a successful implementation intelligible
  - A separate activity from optimisation
    - Refactoring is for people not compilers
  - All about increasing code quality
    - Cleaner, simpler and easier to understand
  - A key activity within Agile processes
    - Where the goal is to ‘keep the code alive’

# Automated Refactoring

- Most refactorings are performed automatically
  - Java IDE's like IntelliJ and Eclipse have lead the way in making refactoring a 'point and click' operation
- A refactoring is performed at the compiler level
  - Your source code is transformed into an Abstract Syntax Tree (AST) so that all symbols can be fully disambiguated
    - E.g the local variable 'myvar' is distinguished from the field 'myvar'
  - The syntax tree is modified and the source code regenerated
- Refactoring API's are slowly being published
  - Allowing power-users to write their own refectoring plug-ins
  - Eventually (hopefully) a unified API will emerge



# Bad Smells In Code

- Refactoring removes ‘bad smells’
  - Detrimental code that accumulates over time
  - The most common examples are:
    - Duplicated code
    - Feature envy
    - Inappropriate intimacy
    - Overuse of comments
    - Long methods or parameter lists
    - Overuse of conditionals
    - Speculative generality

# Bad Smells In Code

- Duplicated Code creates update errors
  - Especially when it occurs inside class hierarchies
  - Refactoring common code increases maintainability
    - Well named helper methods reduce the need for comments
- Feature Envy violates encapsulation
  - A class should do one thing and do it well
  - Methods should not be overly interested in other classes
- Inappropriate Intimacy also violates encapsulation
  - Special relationships between classes are a bad idea

# Bad Smells In Code

- Long Comments usually signal poor code
  - If the comment is trying to make up for code that is too convoluted or obfuscated to be understood
    - Comments should not be an apology for poor code
    - Complex code that is introduced in the name of efficiency will often have its benefits removed by those that have to maintain it
  - Well written code only needs terse comments
    - API comments like JavaDoc are a special case
- Long Methods or Parameter Lists are not readable
  - First attempts at a method need to be decomposed
  - Multiple parameters need to be encapsulated

# Bad Smells In Code

- Long Conditionals are very fragile
  - Every time something is added new branches are required
  - There are many ways to remove conditionals
    - One method is to introduce the Command Design Pattern
- Speculative Generality delays development
  - Developers are trained to anticipate future needs
    - This can result in adding functionality customers ‘should like’
  - In many cases the extra functionality is not required
    - The extra time and effort is wasted
    - The design is complicated unnecessarily
    - A clean and simple design allows scope for change

# Refactoring

- Refactorings are arranged in catalogs
  - Martin Fowlers book is the ‘refactoring bible’
  - Lists of possible refactorings are always growing
- The most common refactorings are the simplest
  - You probably use them already
    - Great improvements can be made just by renaming
    - Such as from ‘ctxMx’ to ‘maximumConnectionTime’
  - Others are specific to particular smells
- Devoting time to refactoring is never wasted
  - It prepares the way for new functionality
  - The time should be built into the project plan
  - Refactoring never adds functionality

# Common Refactorings

- *Extract Method*
  - Used constantly to simplify larger methods
  - Any method >15 lines is a potential target
- *Rename Method/Variable*
  - Short names and abbreviations are a bad idea
  - Verbose code reduces the need for commenting
- *Move Method*
  - Relocate a service to avoid or minimize feature envy
- *Separate Query from Modifier*
  - A method should query an objects state or change it
  - Separating the two simplifies the client code

# Common Refactorings

- *Extract Interface*
  - Lift an interface from public methods
- *Replace Magic Number with Constant*
  - Magic numbers are always a bad idea
- *Replace Error Code with Exception*
  - Exceptions are designed to separate out error handling
- *Replace Conditional with Polymorphism*
  - Do not switch on the type of an object or constant
  - Create a class hierarchy and use polymorphism
- *Replace Parameter with Explicit Method*
  - In general the simpler the method the better

# Common Refactorings

- *Introduce Parameter Object*
  - Avoid passing basic types in method calls
  - Encapsulate them in a separate class
- *Preserve Whole Object*
  - Pass the whole object in a method call
- *Replace Parameter with Method*
  - Don’t pass a return value as a function parameter if the receiving method could call the function itself
- *Pull Up / Push Down Method*
  - In order to keep a class hierarchy balanced
  - Hierarchies tend to become ‘bottom heavy’ with use

# Specific Refactorings

- *Introduce Null Object*
  - A Null Object is a special class in your hierarchy
  - Its methods are all overridden to do nothing
    - Except log errors or produce debug messages
    - Null Objects are usually Singletons
  - The Null Object is returned instead of ‘null’
    - This avoids having to write ‘if(retval != null)’
- *Replace Method with Method Object*
  - Complex methods can be turned into objects
  - Local variables become fields of the class
  - The method call can be decomposed into helpers

# Refactoring To Patterns

- An intriguing idea is using refactorings to incrementally convert existing code into a pattern-friendly form
  - This is set out in ‘Refactoring to Patterns’ by Joshua Kerievsky
- For example when faced with complex conditional logic you could introduce the Strategy Pattern:
  - Create a base class with a method to represent the operation
    - Move all the conditional logic into this method
  - Create a derived class for each branch in the conditional
    - Move the associated logic into an overridden version of the method
  - Make the base class abstract and provide a means for the parent class to hold and (optionally) change the appropriate strategy
    - Usually via a constructor parameter or a setter method

# Future Directions in Refactorings

- Refactoring support is continually improving
  - New refactorings are likely to be increasingly powerful and tailored to the specific needs of Java developers
- The new features in Java 5 and 6 offer lots of scope:
  - Refactor loop from normal for to Java 5 for-each
  - Refactor collection class to use type parameters (generics)
  - Convert collection usage from thread-safe to thread-enabled
  - Replace constant sets of values with ‘enum’ classes
  - Add ‘@Override’ annotation to overridden methods
  - Eliminate code made redundant by boxing and varargs
  - Convert unit test from JUnit 3 to JUnit 4