# Refactoring Without Ropes

Roger Orr
OR/2 Limited

The term 'refactoring' has become popular in recent years; but how do we do it safely in actual practice?

# Refactoring ...

- “Improving the design of existing code” *Martin Fowler*.
- Iterative process for changing code safely

# … without ropes

- When learning to climb, ropes catch you when you fall
- Climbing 'for real' is riskier
  - Someone has to be in front
  - Mistakes become more serious
- The 'three point' rule
  - Only move one hand or foot at a time
  - 'Iterative refactoring'

# What is refactoring?

- Do not alter external behaviour
- Improve internal design
- Be disciplined
- Minimise the chance of introducing bugs

# What is refactoring?

- Refactoring can occur at various levels
  - Inside a single method implementation
  - Inside a class
  - Inside a module
  - Inside an application
- The scope does not affect the principles

# What doesn't change?

- One key defining characteristic of refactoring is that the external behaviour is unaltered
  - Tests unchanged
  - Manuals and user guides unchanged
  - “Bug-compatible” release
- What is *external* for this refactoring?

# So … what changes?

- Internal Implementation
  - Algorithms
  - Methods added/removed/changed
- Class hierarchies changed
- Tools or lower level components

# So … what changes?

- Improve design
- Reduce entropy
- Improve performance (debatable!)
- Prepare for future enhancements

# Be disciplined

- Refactoring is not externally defined
  - Easy to have scope creep
  - Pressure to add 'business benefit'
- Existing code can be
  - Fragile
  - Poorly understood
  - Undocumented

# Introducing bugs?

- All change is dangerous
  - Follow existing patterns
  - Use tools
- No new functionality so testing easier
- What to do with *existing* bugs?

# Ropes for refactoring

- Complete test coverage at the right level
  - Unit tests for small changes
  - Integration/system tests
- Unambiguous existing code
- Safe test environment
- Easy release/backout

# Sample refactoring

- Replace Parameter with Explicit Methods

```
void setValue(String name, int value) {
  if (name.equals("height"))
    _height = value;
  else if (name.equals("width"))
    _width = value;
  Assert.shouldNeverReachHere();
}
```

```
void setHeight(int value) {
    _height = value;
}
void setWidth(int value) {
    _width = value;
}
```

# Is it worth it?

- Pros
  - Avoids conditional code
  - Gain compile time checking
  - Self-documenting
- Cons
  - Harder to change

# Mechanics

- Create new methods
- Call from appropriate leg of old method
- Compile and test
- Replace each call site as appropriate
- Compile and test
- Remove conditional method

# Mechanics

- First move (new method)
  - Check: remove new method on failure
- Second move (change call sites)
  - Check: change call sites back on failure
- Third move (remove old code)
  - Check: put old code back

# Problems with first move

- Bad design
  - Lose sight of overall design by focusing on specifics
  - Code duplication
- Don't completely understand the old code
  - Side effects
  - Unexpected overloads
  - Runtime method discovery

# Problems with first move

- Bugs in new methods
  - All new code may have flaws
- Lack of complete test coverage
  - Non existent
  - Not covering enough cases
  - Not covering failure modes

# Problems with first move

- Bugs in new methods
  - All new code may have flaws
- Lack of complete test coverage
  - Non existent
  - Not covering enough cases
  - Not covering failure modes

The completely unexpected

# Problems with second move

- Fail to correctly modify old code
  - Change similar, but not identical, code
  - Adding wrong arguments
- Don't find all the code to change
  - Parallel version management
  - Use outside your control
- Can't fix the past (examples and memory)

# Problems with second move

- C++ example

```
// returns true or false
bool tryAction();
```

- Refactor to:

```
// returns 0 or error code
int tryAction();
```

- Think about the code you miss...

# Problems with third move

- Never scheduled
  - End result is more complexity
  - Doesn't get easier with time
- Breaks code
  - Know your users
  - Clean migration path
  - Design in deprecation if necessary

# Further complexity

- A method refactoring is 'simplest case'
- Higher level refactoring is more complex
- Keep the principles in mind
  - Move one limb at a time
  - Ensure you are still safe
  - Make sure you can move back

# Further complexity

- Manage complexity by dividing it up
  - Easier to ensure simple changes work
  - Individual steps may use recurring patterns
- Separate what you can
  - Client from server
  - Application from configuration

# Further complexity

- Library code
- Distributed programs
- Configuration
- Database schema
- File formats
- Release cycles

# Library code

- Problems
  - Must decouple from client refactoring
  - Cannot see all the client code
  - Callbacks

# Library code

- Side by side
  - Add new clean interfaces
  - Mark old interfaces as deprecated (...!)
  - Interim is *more* complex
- Big bang
  - Create new library
  - Force client to migrate

# Library code

- Side effects
  - Client code may do things you don't realise
  - Client code may rely on things you don't expect
- Leaky libraries
  - Internal methods may be used
  - Detectable failure is the best outcome

# Library code

- Callbacks
  - Dependency Inversion
  - May be eased by an extra refactor
  - Compile time and/or runtime checks
  - Hard to handle at runtime – who do you tell?

# Callback example

```
public class Server {

  public interface Callback {
    public void method( String arg );
  }

  public void add( Callback callback ) {
    ...
  }

  public void execute() {
    ...
  }
}
```

# Callback example

```
public class Client {
  public static void main( String[] args ) {
    Server server = new Server();
    server.add( new Server.Callback() {
       public void method( String arg ) {
         System.out.println( "Hello " + arg );
       }
    });
    try {
      server.execute();
      System.out.println( "Executed" );
    }
    catch ( Exception ex ) {
      System.out.println( "Execute failed" + ex );
    }
  }
}
```

# Refactor – rename method

- If we rename a method in the Server class old client code won't execute
- If we rename a method in Server.Callback old client code *will* execute but the callback may fail – for example it's in another thread.

# Bullet proofing callback

```
public interface Callback2 {
    public void method( String arg, String arg2 );
 }

 public void add( Callback2 callback ) {
   this.callback = callback;
 }
```

Now if we run old client code against the new server the call to 'add' fails.
We can also support old clients during the refactor by using a shim class:

```
public void add( final Callback callback ) {
  this.callback = new Server.Callback2() {
    public void method( String arg, String arg2 ) {
      callback.method(arg);
    }
  };
}
```

# Further complexity

- Distributed programs
  - Decouple client and server refactoring
  - Callbacks
  - Parallel running

# Refactoring the interface

- First move: new server with old clients
  - Additional interfaces
  - Additional methods
  - Defaulted arguments
- Second move: migrate to new clients
- Third move: remove support for old clients

# Refactoring callbacks example

- First move: new clients with old server
  - All that changes is the callback interface
    - May use same techniques as for library callbacks
  - Ignore new arguments and fields
- Second move: new server
  - Populates new arguments / fields
- Third move: change client again
  - Process the new arguments and fields

# Parallel running

- If you do a lot of refactoring of the interface
  - Do I need a more flexible interface?
  - Extend protocol to supply a version number
  - Add support for multiple simultaneous versions

# Further complexity

- Configuration
  - Rollback
  - Handling of old versions

# Configuration example

- Many applications have complex configuration, so reduce risk
- First move: parse optional new items
- Second move:  add new items to the configuration
- Third move: process new items

# Further complexity

- Database schema
  - Rollback
  - Decouple from application change
  - Named columns
  - Views / stored procedures

# Database changes

- Example of adding new column to a table
  - Move 1 – add the column to the table (DB)
  - Move 2 – write the new column (App)
  - Move 3 – populate missing values (Script)
  - Move 4 – use the column (App)
- Small steps – each with very low risk

# Decoupling interface

- Views and stored procedures
- Support refactoring of database tables
- Can support multiple versions

# Further complexity

- File formats
  - Detecting changes
  - Explicit conversion programs
  - Implicit conversion
    - Reading
    - Writing

# File formats

- Easy to ignore the cost to users of refactoring file formats.
  - MS Word is a good example …
- It's not just the code changes (reading and writing) but the existing files.
- Worst case is not detecting old files
- Critical to read old formats
- Good to have way to convert back to old

# Further complexity

- Release cycles
  - How long is your release cycle?
  - What is the cost of a release?
  - What is the likely number of problems?
  - How easy is it to back out?
    - 'Actual' cost
    - 'Political' cost

# Short cycles

- I like short release cycles
  - Incremental business benefit (Agile methods)
  - Smaller number of changes in each cycle
    - Less to remember
    - Easier to diagnose faults
    - Easier to drop back
  - Mechanism of releasing stays well known

# Short cycles

- I don't like short release cycles
  - Too many releases to remember
    - Need good release tracking
  - Too much testing and paperwork
    - Management/risk issue – may not be fixable
  - Too much manual setup
    - Automate it :-)

# Summary

- Refactoring works by making changes
  - Small
  - Controlled
  - Easily reversible
- Make sure you know
  - What you are changing
  - What you are *not* changing
  - Where you are aiming for