Design Patterns

Produced by

Eamonn de Leastar
edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning
support unit

Solver Example

Idiomatic, Template Method, Strategy

Case Study

- Examine a simple problem/solution from three perspectives
  - Template Method Pattern
  - Strategy Pattern

Solver - Find the minima of a line

- Build test class first:

```
public class MinimaSolverTest
{

  private double[] line = { 1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0 };
  private MinimaSolver solver;

  @Test
  public void minima()
  {
    solver = new MinimaSolver();
    double[] result = solver.minima(line);
    assertTrue(result[0] == 1.1);
    assertTrue(result[1] == 2.2);
  }

}
```

Solver - the MinimaSolver class

- Some preprocessing and post processing is carried out (commented out in the code).
- Note that there are in fact two algorithms for finding the minima - our code just uses the leastSquaresAlgorithm.
- The algorithm implementations are replaced with stubbs to keep the code focussed on the patterns

```
public class MinimaSolver
{
  public MinimaSolver()
  {
  }

  public double[] minima(double[] line)
  {
    // do some pre-processing
    double[] result = leastSquaresAlgorithm(line);
    // do some post-processing
    return result;
  }

  public double[] leastSquaresAlgorithm(double[] line)
  {
    return new double[] { 1.1, 2.2 };
  }

  public double[] newtonsMethodAlgorithm(double[] line)
  {
    return new double[] { 3.3, 4.4 };
  }
}
```

Enums

- Further enhance readability with enums.

```
public class MinimaSolver
{
  public enum AlgorithmTypes
  {
    LeastSquares, NewtonsMethod
  }

  private AlgorithmTypes algorithm;

  public MinimaSolver(AlgorithmTypes algorithm)
  {
    this.algorithm = algorithm;
  }

  public double[] minima(double[] line)
  {
    // do some pre-processing
       double[] result = null;
    if (algorithm == AlgorithmTypes.LeastSquares)
    {
      return leastSquaresAlgorithm(line);
    }
    else if (algorithm == AlgorithmTypes.NewtonsMethod)
    {
      return newtonsMethodAlgorithm(line);
    }
    // do some post-processing
    return result;
  }
```

```
@Test
public void leastSquaresAlgorithm()
{
  solver = new MinimaSolver(AlgorithmTypes.Le
  double[] result = solver.minima(line);
  assertTrue(result[0] == 1.1);
  assertTrue(result[1] == 2.2);
}

@Test
public void newtonsMethodAlgorithm()
{
  solver = new MinimaSolver(AlgorithmTypes.NewtonsMethod);
  double[] result = solver.minima(line);
  assertTrue(result[0] == 3.3);
  assertTrue(result[1] == 4.4);
}
```

Introducing a new Algorithm

- Write the test first.
- It will fail to compile (as Bisection is not a valid enum)
- Once Bisection is introduced, the test will compile but fail to pass until we implement the method.

```
@Test
public void bisection()
{
  solver = new MinimaSolver(AlgorithmTypes.Bisection);

  double[] result = solver.minima(line);
  assertTrue(result[0] == 5.5);
  assertTrue(result[1] == 6.6);
}
```

New Algorithm - Bisection

- new enum for bisection.

```
public enum AlgorithmTypes
{
  LeastSquares, NewtonsMethod, Bisection
}
```

- new method to implement the algorithm.

```
public double[] bisectionAlgorithm(double[] line)
{
  return new double[] { 5.5, 6.6 };
}
```

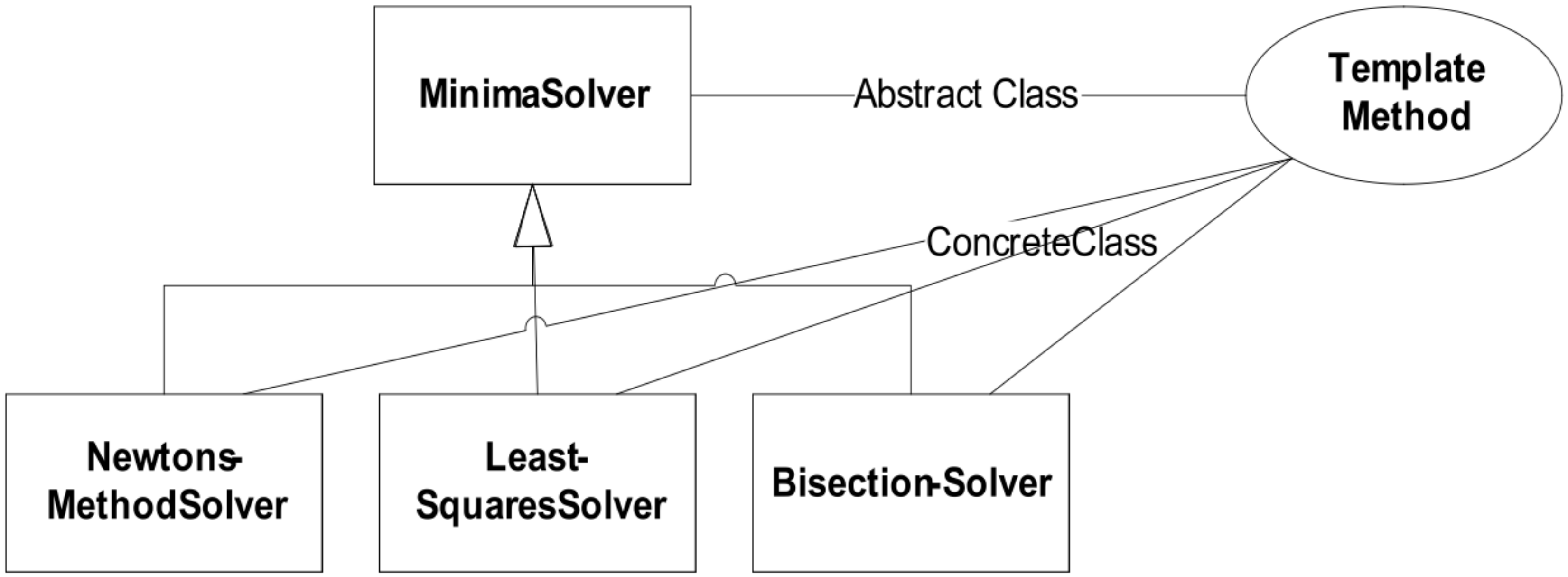
- new clause in the if statement to dispatch to bisection if selected.

```
public double[] minima(double[] line)
  {
    // do some pre-processing
       double[] result = null;
    if (algorithm == AlgorithmTypes.LeastSquares)
    {
      return leastSquaresAlgorithm(line);
    }
    else if (algorithm == AlgorithmTypes.NewtonsMethod)
    {
      return newtonsMethodAlgorithm(line);
    }
    else if (algorithm == AlgorithmTypes.Bisection)
    {
      return bisectionAlgorithm(line);
    }
    // do some post-processing
    return result;
  }
```

```
public class MinimaSolver
{
  public enum AlgorithmTypes
  {
    LeastSquares, NewtonsMethod, Bisection
  }

  private AlgorithmTypes algorithm;

  public MinimaSolver(AlgorithmTypes algorithm)
  {
    this.algorithm = algorithm;
  }

  public double[] minima(double[] line)
  {
    // do some pre-processing
        double[] result = null;
    if (algorithm == AlgorithmTypes.LeastSquares)
    {
      return leastSquaresAlgorithm(line);
    }
    else if (algorithm == AlgorithmTypes.NewtonsMethod)
    {
      return newtonsMethodAlgorithm(line);
    }
    else if (algorithm == AlgorithmTypes.Bisection)
    {
      return bisectionAlgorithm(line);
    }
    // do some post-processing
    return result;
  }

  public double[] leastSquaresAlgorithm(double[] line)
  {
    return new double[] { 1.1, 2.2 };
  }

  public double[] newtonsMethodAlgorithm(double[] line)
  {
    return new double[] { 3.3, 4.4 };
  }

  public double[] bisectionAlgorithm(double[] line)
  {
    return new double[] { 5.5, 6.6 };
  }
```

Template Method Pattern

- Improve the maintainability of the solution.
- Enable new algorithms to be introduced without modifying the MinimaSolver class.
- Continue to improve readability



Template Method - Abstract Class

- Remove all of the algorithm implementations.
- Replace with an abstract method algorithm.
- Adjust minima method to invoke this algorithm.
- Mark class as abstract.

```
public abstract class MinimaSolver
{

  public MinimaSolver()
  {
  }

  double[] minima(double[] line)
  {

    // do some pre-processing
    double[] result = null;

    result = algorithm(line);

    // do some post-processing
    return result;
  }

  public abstract double[] algorithm(double[] line);
}
```

Template Method - Concrete Classes

- Algorithms extend MinimaSolver and implement algorithm method.

```
public class LeastSquaresSolver extends MinimaSolver
{
  public double[] algorithm(double[] line)
  {
    return new double[]{1.1, 2.2};
  }
}
```

```
public class NewtonsMethodSolver extends MinimaSolver
{
  public double[] algorithm(double[] line)
  {
    return new double[]{3.3, 4.4};
  }
}
```

Template Method Test

- Tests now instantiate appropriate MinimaSolver subclass - and test as before.

```
@Test
public void leastSquaresAlgorithm()
{
  solver = new LeastSquaresSolver();
  double[] result = solver.minima(line);
  assertTrue(result[0] == 1.1);
  assertTrue(result[1] == 2.2);
}

@Test
public void newtonsMethodAlgorithm()
{
  solver = new NewtonsMethodSolver();
  double[] result = solver.minima(line);
  assertTrue(result[0] == 3.3);
  assertTrue(result[1] == 4.4);
}
```

Template Method - Introducing new Algorithm

- Just define new class (no need to modify MinimaSolver base class)

- And test by instantiating an object of this class

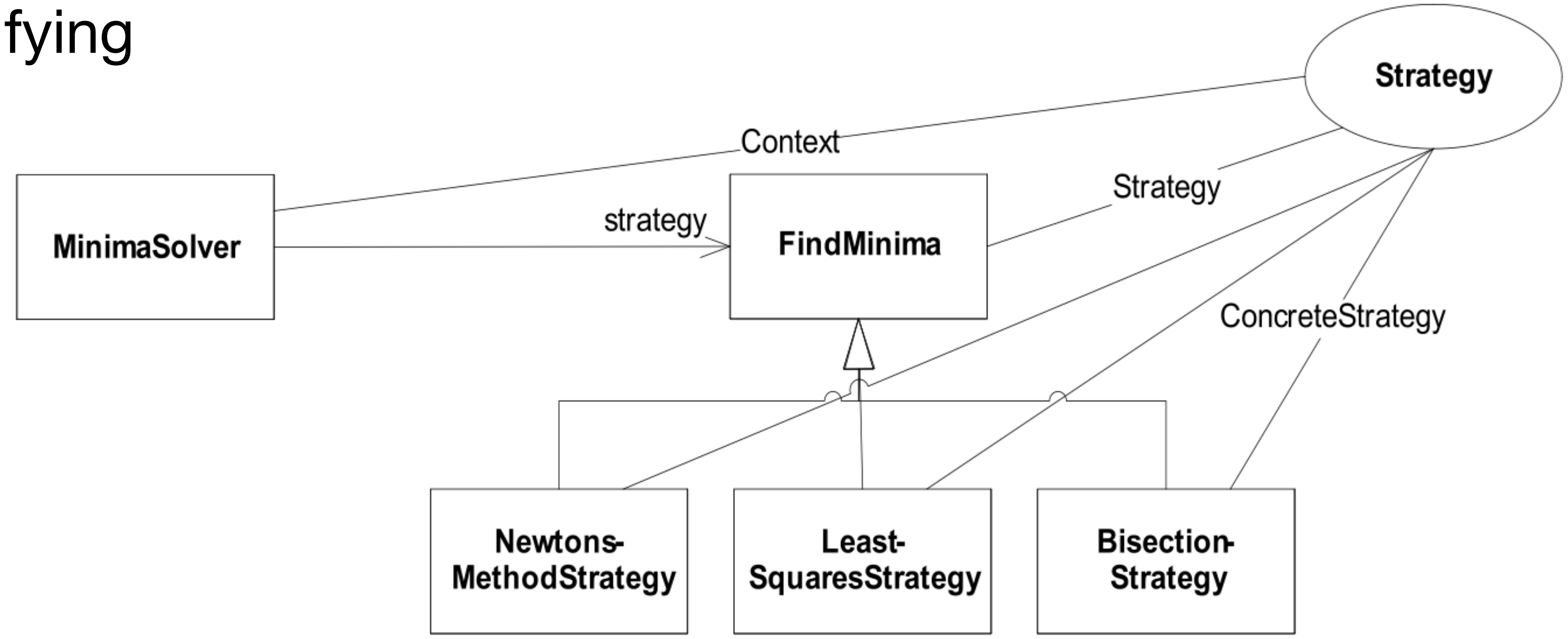
- Contrast with previous mechanism for introducing new algorithm

```
public class BisectionSolver extends MinimaSolver
{
  public double[] algorithm(double[] line)
  {
    return new double[]{5.5, 6.6};
  }
}
```

```
  @Test
  public void bisection()
  {
    solver = new BisectionSolver();

    double[] result = solver.minima(line);
    assertTrue(result[0] == 5.5);
    assertTrue(result[1] == 6.6);
  }
```

Strategy Pattern

- Improve the variability of the solution.
- Enable new algorithms to be introduced without modifying the MinimaSolver class.
- Continue to improve readability
- Enable algorithms to be changed at run time.



Strategy Interface and Concrete Implementations

- Encapsulate algorithm in an interface.

- Realise algorithms as standalone classes implementing this interface

```
public interface FindMinima
{
  double[] algorithm(double[] line);
}
```

```
public class LeastSquaresStrategy implements FindMinima
{
  public double[] algorithm(double[] line)
  {
    return new double[]{1.1, 2.2};
  }
}
```

```
public class NewtonsMethodStrategy implements FindMinima
{
  public double[] algorithm(double[] line)
  {
    return new double[]{3.3, 4.4};
  }
}
```

Strategy Context

- MinimaSolver will now be initialised with the appropriate strategy object.
    - by the constructor...
    - or by a changeStrategy() method...
- minima() delegates algorithm to strategy object.

```
public class MinimaSolver
{
  private FindMinima strategy;

  public MinimaSolver(FindMinima strategy)
  {
    this.strategy = strategy;
  }

  double[] minima(double[] line)
  {

    // do some pre-processing
    double[] result = null;

    result = strategy.algorithm(line);

    // do some post-processing
    return result;
  }

  public void changeStrategy(FindMinima newStrategy)
  {
    strategy = newStrategy;
  }
}
```

Strategy Test

- MinimaSolver + the appropriate Strategy object need to be created.

```
@Test
public void leastSquaresAlgorithm()
{
  solver = new MinimaSolver(new LeastSquaresStrategy());
  double[] result = solver.minima(line);
  assertTrue(result[0] == 1.1);
  assertTrue(result[1] == 2.2);
}

@Test
public void newtonsMethodAlgorithm()
{
  solver = new MinimaSolver(new NewtonsMethodStrategy());
  double[] result = solver.minima(line);
  assertTrue(result[0] == 3.3);
  assertTrue(result[1] == 4.4);
}
```

Defining a new Algorithm

- Just provide a new implementation of FindMinima...

```
public class BisectionStrategy implements FindMinima
{
  public double[] algorithm(double[] line)
  {
    return new double[]{5.5, 6.6};
  }
}
```

- ... and pass an implementation of this to the solver.

```
  @Test
  public void bisection()
  {
    solver = new MinimaSolver(new BisectionStrategy());
    double[] result = solver.minima(line);
    assertTrue(result[0] == 5.5);
    assertTrue(result[1] == 6.6);
  }
```

Changing the Strategy at Runtime

- Just call changeStrategy() with a new implementation.
- The same solver object is used in both tests here.
- This is not possible with Template Method.

```
@Test
public void testChangeAlgorithm()
{
  solver = new MinimaSolver(new LeastSquaresStrategy());

  double[] result = solver.minima(line);
  assertTrue(result[0] == 1.1);
  assertTrue(result[1] == 2.2);
  solver.changeStrategy(new BisectionStrategy());

  result = solver.minima(line);
  assertTrue(result[0] == 5.5);
  assertTrue(result[1] == 6.6);
}
```