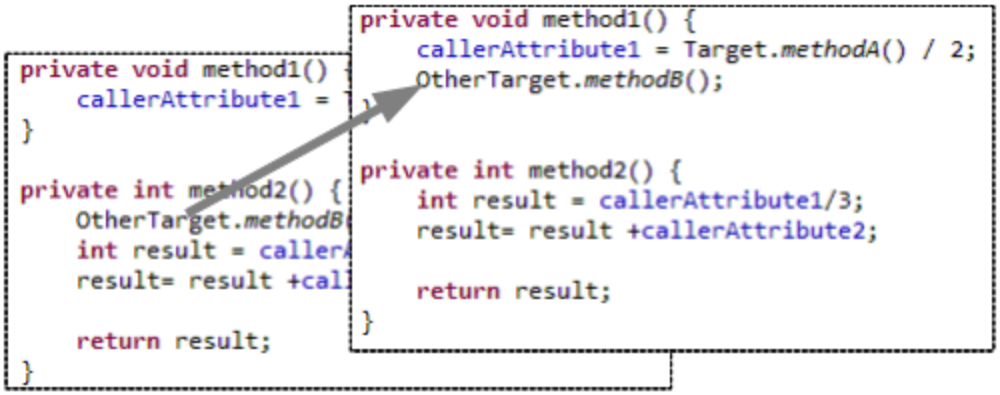
CogCx

This note describes the examples used for CogCx more fully.

Cohesion Levels

Cohesion levels were introduced as part of structured design (Coincidental, Logical, Temporal, Procedural, Communicational, Sequential, Functional). The cohesion element of the CogCx directly measures the Sequential cohesion between two code elements. The example below shows how the metric is consistent with the Communicational level of cohesion.



This example assumes that method1 is always called before method2 so moving the statement that calls methodB as shown doesn't modify the behaviour of the application. In the initial position the statement that calls methodB is a sibling of the other statements in method2 and so the cohesion metric is calculated between them. This results in a high value as methodB and the variables in the other statements of method2 are far apart on the dependency graph. When we move the methodB call, the cohesion metric is now calculated with the statement in method1. In our example, Target.methodA() and OtherTarget.methodB() are within the same package and are closer on the dependency graph due to the dependencies amongst that package. This results in an overall reduction in the cohesion metric.

TABLE I. COMMUNICATIONAL COHESION LEVEL

	element	Cognitive	cohesion	coupling	Total
Before	8	26	1381.22	173.71	1588.93
After	9	10	5.27	175.54	199.81

Aside: There is also a coupling penalty for the calls to methodA and methodB, which could be reduced if they were moved closer (on the structural graph) to the dependency (e.g. by moving method1 to one of the Target classes). Moving closer to Target may have an adverse impact on the cognitive penalty of that class and also on the coupling penalty of the statement that sets the callerAttribute1. As we shall see in the refactoring examples, below, decreasing one part of CogCx often leads to an increase in another.

Extract Method

The below results show the aggregate metric values for a long method (before) that has been split into two (after).

TABLE II. REFACTORING: EXTRACT METHOD

	element	cognitive	cohesion	coupling	Total
Before	9	5041	9.56	11.15	5070.71
After	12	15	2.94	33.45	63.39

The biggest gain in splitting up the before method into two is with the cognitive penalty. Also of note is that cohesion is improved at the expense of a decrement to coupling and element costs.

From

```
	public int before() {
		int f =1;
		int g = f+4;
		int e=f*g;
		int t= 2;
		int u=t+5;
		int s=t*u;
		return e+s;
	}
```

To:

```
	public int after() {
		return afterPart1() + afterPart2();
	}

	public int afterPart1() {
		int g =1;
		int f = g+4;
		return g*f;
	}

	public int afterPart2() {
		int u= 2;
		int t=u+5;
		  u*t;
	}
```

Inline Method

For the inline method example, moving all the behaviour to one method doesn't break the short term memory capacity limit (the cognitive penalty function is a near barrier function). Consequently the improvement in coupling is more than enough to offset degradation of the cohesion cost and cognitive penalty.

TABLE III. REFACTORING: INLINE METHOD

	element	cognitive	cohesion	coupling	Total
Before	10	11	1.92	37.6	60.52
After	6	25	2.22	8.74	41.96

From

```
	public int before() {
		int b =1;
		int a = b*beforePart1(b) ;
		return a ;
	}

	private int beforePart1(int b) {
		int c = b+4;
		return c;
	}
```

To:

```
	public int after() {
		int b =1;
		int c = b+4;
		int a=b*c;
		return a;
	}
```

Design Patterns

The design pattern tests centre around a fictitious requirement that a calling abstraction needs to call behaviour(s) on a target abstraction. The three versions differ based on the complexity of the caller and target behaviours and the detailed relationships between caller and target.

- Simple: A simple, single target behaviour for a number of varieties and there is only one caller of this behaviour.
- Complex 1 to 1: Complex set of target behaviours for a number of varieties. Each of the varieties has a one-to-one dependency with the varieties of the calling abstraction.
- Complex independent: Complex set of behaviours for a number of varieties. Behaviours are called by various unrelated callers.

These versions have been chosen so that the approaches below will be the preferred designs for each version in turn

- Switch: Simple method that utilises switch
- Hierarchy: target behaviours are incorporated into the varieties of the calling code, similar to the Template pattern.
- Strategy: Strategy pattern, behaviours are grouped independent of calling code.

TABLE IV. DESIGN PATTERN

	Switch	Hierarchy	Strategy
Simple	107	171	421
Complex 1 to 1	3386	1374	1513
Complex Independent	10724	3640	1321

More Detailed

Number Of Varieties	Problem	Solution	Element	Cognitive	Cohesion	Coupling	Total
five	complex1to1	hierarchy	216	514	389.29	437.59	1556.88
five	complex1to1	strategy	308	550	331.17	506.07	1695.24
five	complex1to1	switch	247	1386	4923.48	442.36	6998.84
five	complexIndep	hierarchy	695	1215	407.61	1357.08	3674.69
five	complexIndep	strategy	242	302	124.53	659.90	1328.43
five	complexIndep	switch	550	50915	1078.49	1538.19	54081.67
five	simple	hierarchy	57	20	2.51	43.64	123.14
five	simple	strategy	87	42	13.90	93.83	236.73
five	simple	switch	36	22	11.89	37.21	107.10
one	complex1to1	hierarchy	59	117	92.63	109.67	378.30
one	complex1to1	strategy	88	126	83.02	113.43	410.45
one	complex1to1	switch	69	159	85.18	112.93	426.11
one	complexIndep	hierarchy	57	165	53.29	103.61	378.90
one	complexIndep	strategy	61	69	28.60	97.12	255.72
one	complexIndep	switch	52	5099	71.18	111.12	5333.30
one	simple	hierarchy	25	12	2.51	15.13	54.64
one	simple	strategy	41	20	12.09	33.78	106.87
one	simple	switch	23	17	4.07	11.17	55.24

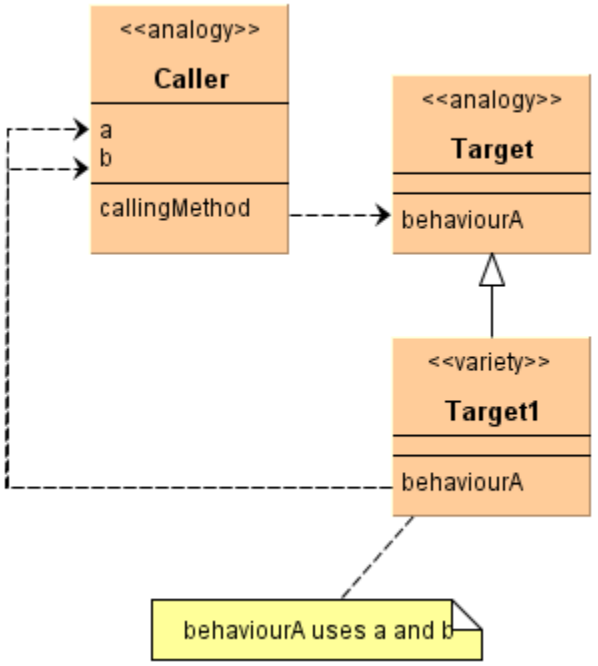
The following sections give UML diagrams for the three requirements. I use the <<analogy>> stereotype to represent the analogy/abstraction of the problem. How these are implemented in code (as classes etc.) will be different for each of the three solutions and these diagrams use the stereotype <<class>>. The three solutions are given for the simple single case.

The Element Costs and path lengths were chosen to reflect the additional cost of the mechanics of traversing the code (specifically on the impact of short term memory). For example, the dependency link between two statements in the same method typically takes a movement of the eye to traverse, but the dependency link between two methods requires a little longer and if you have to research where a class is it may take even longer. As discussed in "Writing Code For Other People" seconds count with short term memory. Details of values used in the metric are as follows:

- Element costs: Methods 2, Type&Package 5 everything else 1
- Structural Path Costs: Package & Type 10, Method 5, statement 2 all else 1. (the actual length is the average of the two ends)

Simple Single

In this case the caller abstractions/analogies is small and simple. The target abstraction/analogy has only one simple behaviour (behaviourA) which is dependent upon attributes from the caller.



Number Of Varieties	Problem	Solution	Element	Cognitive	Cohesion	Coupling	Total
five	Simple	hierarchy	57	20	2.51	43.64	123.14
five	Simple	strategy	87	42	13.90	93.83	236.73
five	Simple	switch	36	22	11.89	37.21	107.10
one	Simple	hierarchy	25	12	2.51	15.13	54.64
one	Simple	strategy	41	20	12.09	33.78	106.87
one	simple	switch	23	17	4.07	11.17	55.24

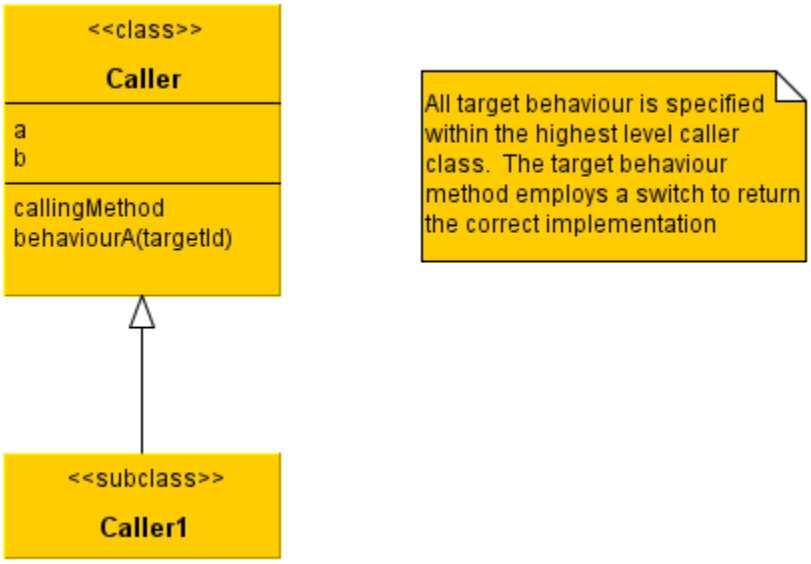
Switch Solution

The switch solution is shown in the UML diagram below. The key part of the switch solution is that the code for the Target analogy is placed in the same class as the caller, and so.

i. Elements that have a dependency link are closer together on the structural graph and so coupling is reduced.

ii. The target behaviour is within the calling class so there are additional cohesion costs (between the target behaviour and the other methods in the calling class).

iii. Adding a method to an existing class can introduce a large cognitive penalty (dependent upon how many methods pre-exist in in the class). For example adding a method to a class with 3 methods introduces an additional penalty of 4!-3! = 18 but if the class already had 4 methods it would introduce an additional penalty of 5!-4! = 96. For our simple solution however the calling class is lightweight and so the additional cost is small.



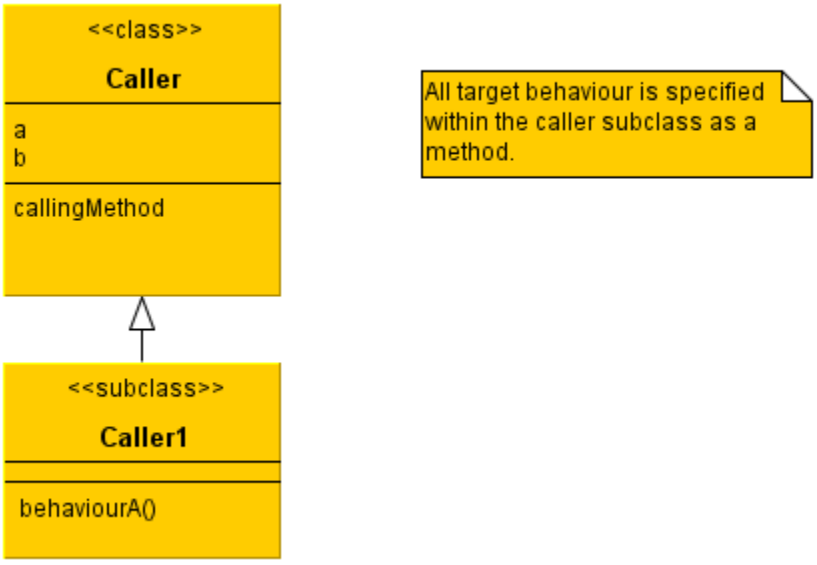
Hierarchy

The UML is below, the key point is that each implementation of behaviourA is within its own class which means that the hierarchy solution will produce more classes than the switch solution. However, the simple nature of the caller and target classes in this example means that those classes will be have a small number of elements (attributes/methods).

i. Element costs are greater (each class has an element cost).

ii. The small number of elements in the classes means there are fewer relationships that contribute to the cohesion cost. However the relationships that exist in the requirements must be satisfied between the classes and so coupling is increased.

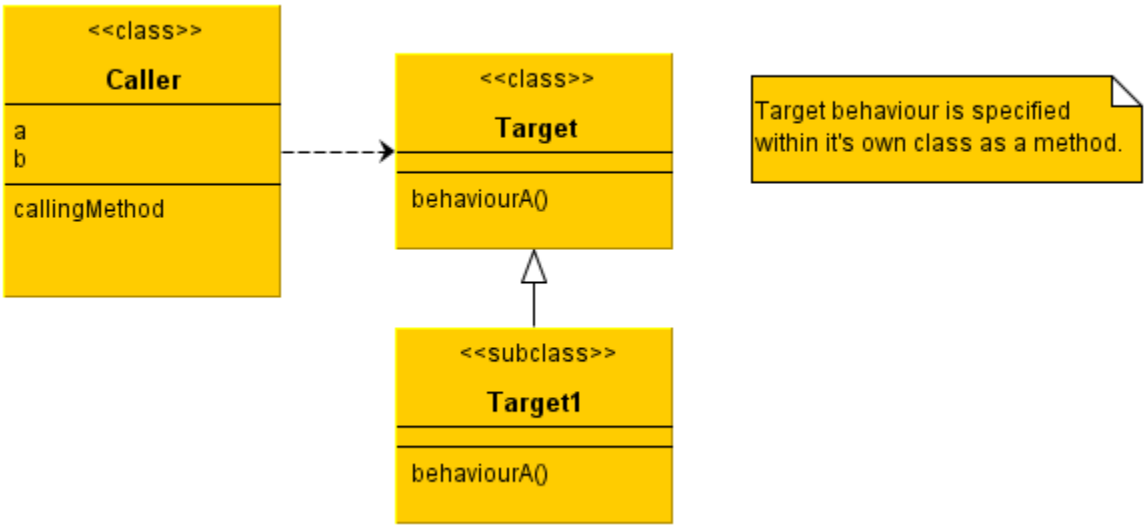
iii. The hierarchy CogCx value for one variety is (marginally) the smallest of the three solutions, but for five varieties, the switch solution is the optimal. As discussed in the extensibility chapter, a good designer will sometimes prefer a slightly non-optimal solution at the start of coding for the promise of it becoming optimal as more varieties are implemented.



Strategy

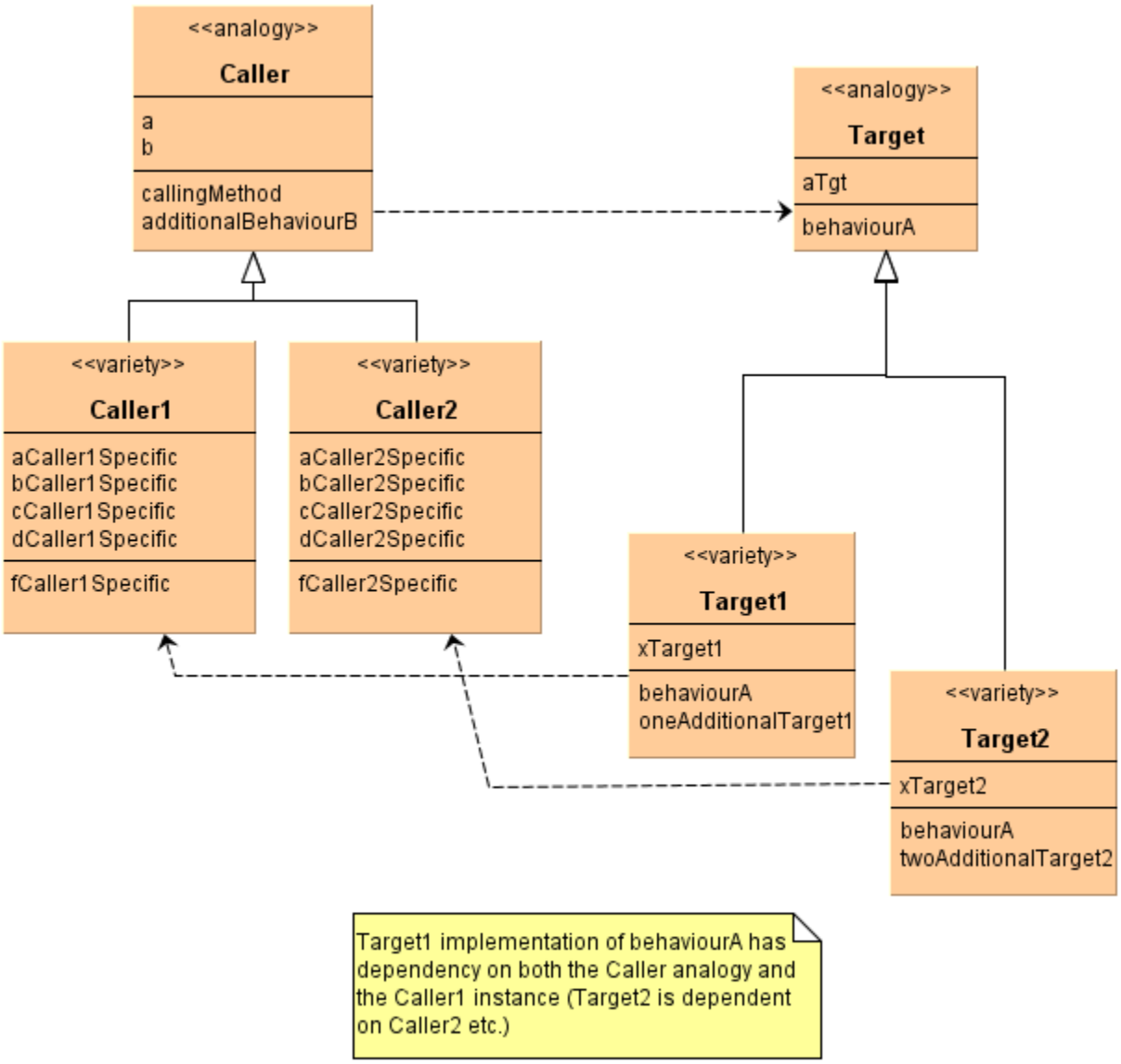
The UML is below, the key point is that the Target analogy is represented in its own class structure. This larger framework/structure will cause increases in all the CogCx costs.

i. A greater number of classes causes a greater element cost.
ii. A larger/deeper structure causes an increase in distances on the structural graph (between dependent elements) and so coupling costs are increased.
iii. Introducing accessor methods for a and b increases distances on the dependency graph and so cohesion costs are increased for elements that access these attributes.
iv. Although individual cognitive costs are small there are more elements to sum over, so the total cognitive cost is increased. Note: as the complexity of Caller, Target and their behaviours becomes more complex (the complex scenarios below) the strategy solution here has more capacity to fit this complexity within its structure without crossing the barrier of the cognitive cost function.



Complex 1 to 1:

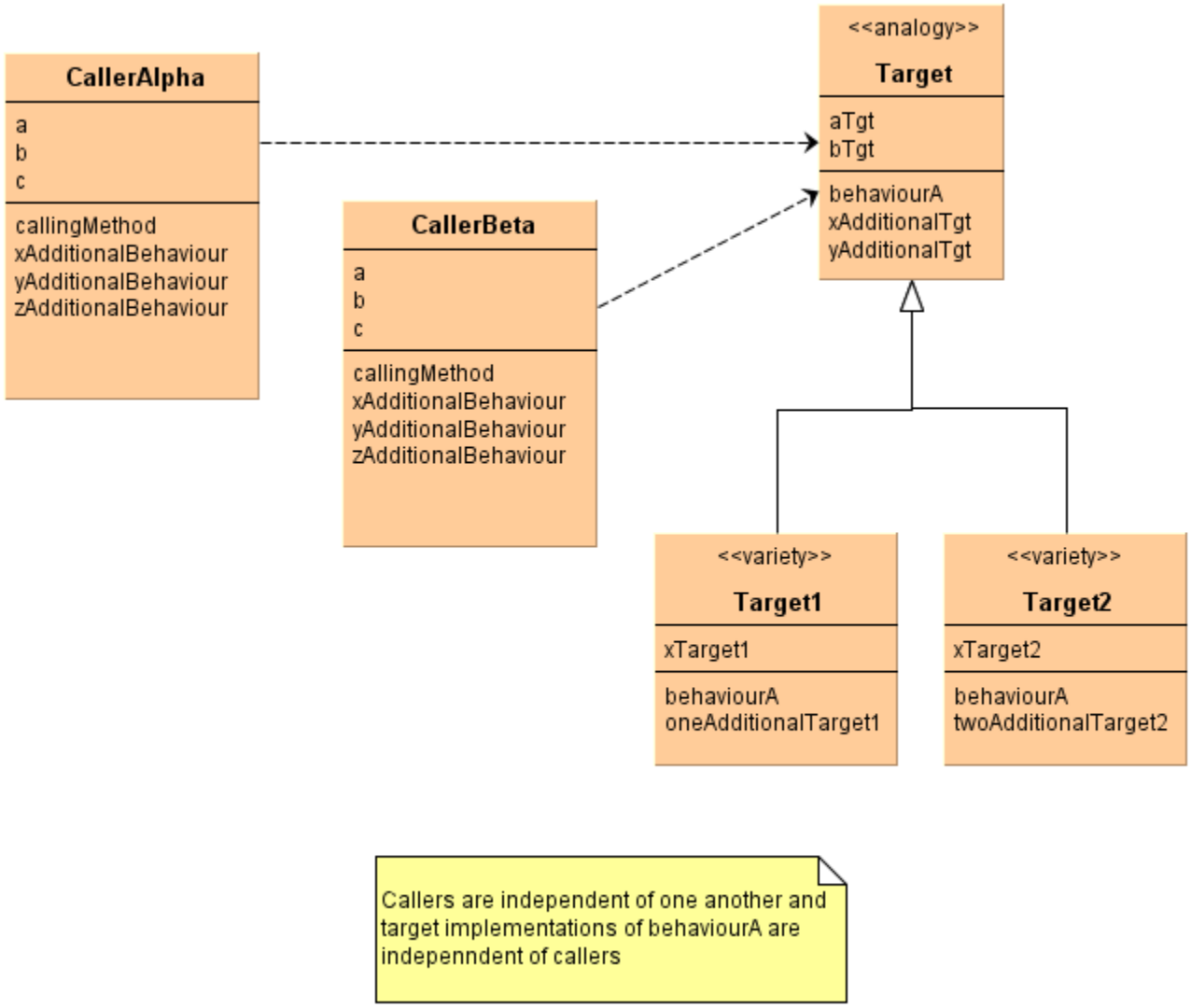
This scenario has a link between the varieties of the Target and Caller and adds complexity to the Caller and Target analogies (the *a*Caller*n*Specific methods; aTgt and xTarget*n* attributes; and *one*AdditionalTarget*1* methods). Please note that these methods and attributes have been introduced to "bulk out" the complexity of the Caller and Target analogies and are meant to represent methods that serve dissimilar behaviour. The difficulty in generating such examples is that they need to provide complexity but shouldn't be so complicated that the example itself becomes difficult to understand. For example, the *a*Caller*1*Specific methods have been written to follow a pattern so that their presence as additional complexity in the example is easily understood. As they stand in the example, a keen eyed developer may suggest that they are candidates for refactoring into an abstraction. However, in the real world cases, these methods would be different behaviours that have very little commonality amongst them and so refactoring wouldn't be realistic.



Number Of Varieties	Problem	Solution	Element	Cognitive	Cohesion	Coupling	Total
five	complex1to1	hierarchy	216	514	389.29	437.59	1556.88
five	complex1to1	strategy	308	550	331.17	506.07	1695.24
five	complex1to1	switch	247	1386	4923.48	442.36	6998.84
one	complex1to1	hierarchy	59	117	92.63	109.67	378.30
one	complex1to1	strategy	88	126	83.02	113.43	410.45
one	complex1to1	switch	69	159	85.18	112.93	426.11

- Switch Solution - The complexity of the target abstraction means that fitting it within the caller class produces a class that is too big and so the cognitive barrier is breached producing a high cognitive cost.
- Hierarchy solution – in comparison with its closest alternative (the strategy solution), the hierarchy solution sacrifices a higher cohesion cost (due to putting the target and caller varieties together) but benefits from a reduced coupling as the dependency between the individual target and caller varieties are closer together on the structural graph.
- Strategy. The element and cognitive penalties are still high due to the independent infrastructure of the solution which (for the complex1to1 scenario at least) isn't providing a large benefit in coupling/cohesion.

Complex Independent



Number Of Varieties	Problem	Solution	Element	Cognitive	Cohesion	Coupling	Total
five	complexIndep	hierarchy	695	1215	407.61	1357.08	3674.69
five	complexIndep	strategy	242	302	124.53	659.90	1328.43
five	complexIndep	switch	550	50915	1078.49	1538.19	54081.67
one	complexIndep	hierarchy	57	165	53.29	103.61	378.90
one	complexIndep	strategy	61	69	28.60	97.12	255.72
one	complexIndep	switch	52	5099	71.18	111.12	5333.30

Here the tactic of incorporating the target and caller into one class (either at the analogy or variety level) that is employed by the switch and the hierarchy solutions, cause a greater cognitive and element cost as much of the code needs to be repeated.

Statement Chunking

The long method below is ripe for decomposition and the proposal is to use blank lines to break it up into chunks of code (so that the chunks are cohesive). There are 9 statements (the "if" statement is the only one uses more than one line). We could position blank lines in any or all of the 8 positions in between the 9 statements. This gives us 2^8 permutations. The metric was calculated for all 256 cases and the lowest scoring solutions are in the table below. The code shown below is the optimum solution, however there are a number of solutions whose measure was similar in value.

```
double calc(PurchaseItem purchase)
{
        double a_lineCost = purchase.lineItemPrice() * purchase.quantity();

        s_monthToDate += a_lineCost;
        double b_monthPercentSaving =0;

        if(s_monthToDate > t_monthThreshold)
                b_monthPercentSaving=u_monthDiscount;
        double c_monthMonetarySaving = a_lineCost * b_monthPercentSaving/100;
        double d_customerMonetarySaving = a_lineCost * v_customerDiscount/100;

        double e_tax = (a_lineCost-d_customerMonetarySaving-c_monthMonetarySaving) * w_taxRate;
        double result = a_lineCost-d_customerMonetarySaving-c_monthMonetarySaving+e_tax;
        return result;
}
```

The "key" is the decimal representation of the choices (i.e. for each blank line between statement n and statement n+1 we add 2^{n-1} to the key)

						Blank Lines Position							
key	cognitive	element	cohesion	coupling	Total	1->2	2->3	3->4	4->5	5->6	6->7	7->8	8->9
37	59	33	479	104	675	Y		Y			Y		
41	59	33	480	104	676	Y			Y		Y		
73	59	33	480	104	676	Y			Y			Y	
18	75	32	479	102	687		Y			Y			
34	75	32	480	102	689		Y				Y		
21	73	33	479	104	689	Y		Y		Y			
69	73	33	480	104	690	Y		Y				Y	
81	73	33	480	104	690	Y				Y		Y	
17	75	32	480	104	691	Y				Y			
19	76	33	479	104	692	Y	Y			Y			
35	76	33	480	104	693	Y	Y				Y		

Code

Caveat:

When trying to understand something we may draw rough sketches, sometimes attaching early names to things. As our understanding grows the initial, draft sketches are no longer an accurate reflection of our current understanding. We may have paid too much attention to what is now known to be a trivial detail and conversely not enough attention to what is now clearly an important part. The initial choices for names may now seem at odds with our deeper understanding. I am typing this caveat in the same environment that much of the code was written, on a busy commuter train early in the morning, traveling to my day job. This is a long way of saying that I know the code in its current state needs a good refactor. The intention was that this code was a prototype, proof of concept that could be thrown away, once enough had been understood about the problem to approach writing an alpha version. Please bear this in mind when you see complexities or confusing names. So why release the code at all? As this version of the code was used to generate the data in the paper, I felt it only appropriate that it be available for confirmation should anyway wish to.

Some Confusions:

- USM – the original chosen name for the CogCx metric (unified software metric)
- Associations/dependencies : used for different purposes throughout the code (sorry ☹)
- Analogical links are added to identify varieties of analogies (need to calculate cognitive penalty). These may just be non-contributing dependencies.

Additionally not all java element structures have been implemented and the performance has not been tested. Consequently, trying this version of the plugin on large codebases will almost certainly be a fruitless endeavour.

Code Description

There are six eclipse projects that need to be loaded into five separate workspaces (in each case create the worksace and then import the project using File→Import, then select "Existing Projects into Workspace" under "General" and select the root directory as defined in the in the table below)

Workspace	Folder in Code Distribution	Project Description
workspace_USMPlugin	USMPlugin/com.tmullen.usm	The metric
	USMTest/com.tmullen.test	classes to call and check the values from the re-factoring examples
workspace_USMRefactoring	USMTestCases/UsmTestCases	the refactoring examples (before and after) code.
workspace_USMPatterns	USMStrategy/Test	Code for the Design Pat-tern example
workspace_USMCohesion	USMCohesion/Test	Code for the cohesion example
workspace_USMParametric	USMParametric/Test	Code for the parametric example

Output and analysis.

The plugin will generate csv files of the data as well as a GrahViz dot file that can be used to generate a graph. These files are saved to a directory specified in the USMCalculator class in the USMPlugin project

```
public static final String directory = "C:\\Users\\Tom\\GraphViz\\" ;
```

Csv and dot files are generated for each Class/Type as well as each package. The UNIVERSE csv file gives all the data in one csv file.

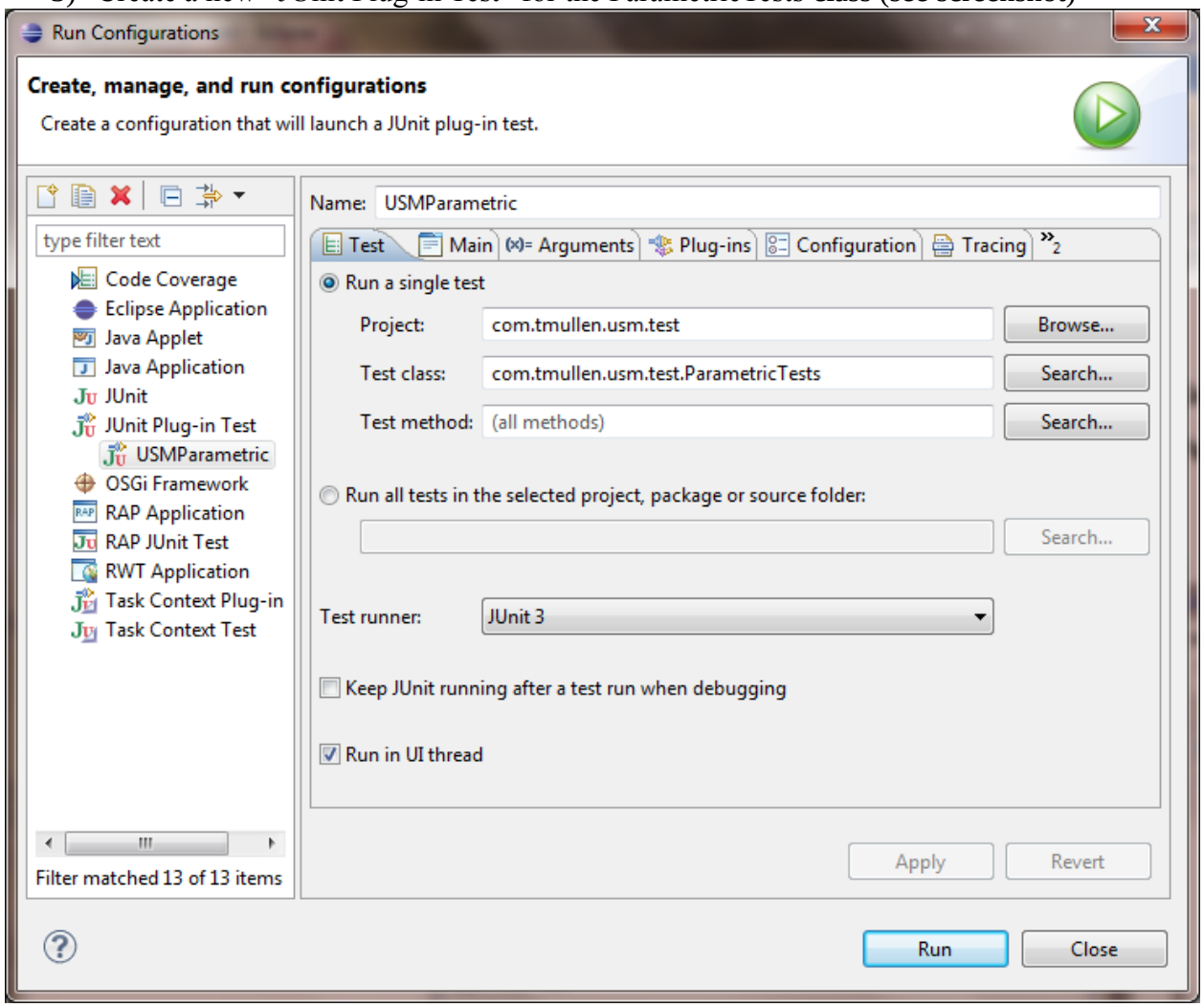
CSV file has the following columns

Column Header	Description
Name	not used (internal name of the node in the dot graph)
Label	code element identifier (the ASTView label naming convention is duplicated here)
parentLabel	identifier of parent in the structural graph (parentRelation-ship=STRUCTURE) or the related element for other types
parentRelationship	relationship type that is being reported (STRUCTURE, ASSOCIATIVE, ASSOC_DIST, STRUC_DIST, ASSOC_LINK)
Type	Type of code element (e.g. VARIABLE, METHOD, PACKAGE,TYPE)
Class	AST class of element
Snippet	short snippet of the text of the code
cogOverload	cognitive penalty for this element
elementCost	element cost for this element
Cohesion	cohesion cost for this element
Coupling	coupling cost for this element
aggCogOverload	aggregate cognitive penalty for this element and all it's descendents on the structural graph
aggElement	aggregate element cost for this element and all it's descendents on the structural graph
aggCohesion	aggregate cohesion cost for this element and all it's descendents on the structural graph
aggCoupling	aggregate coupling cost for this element and all it's descendents on the structural graph

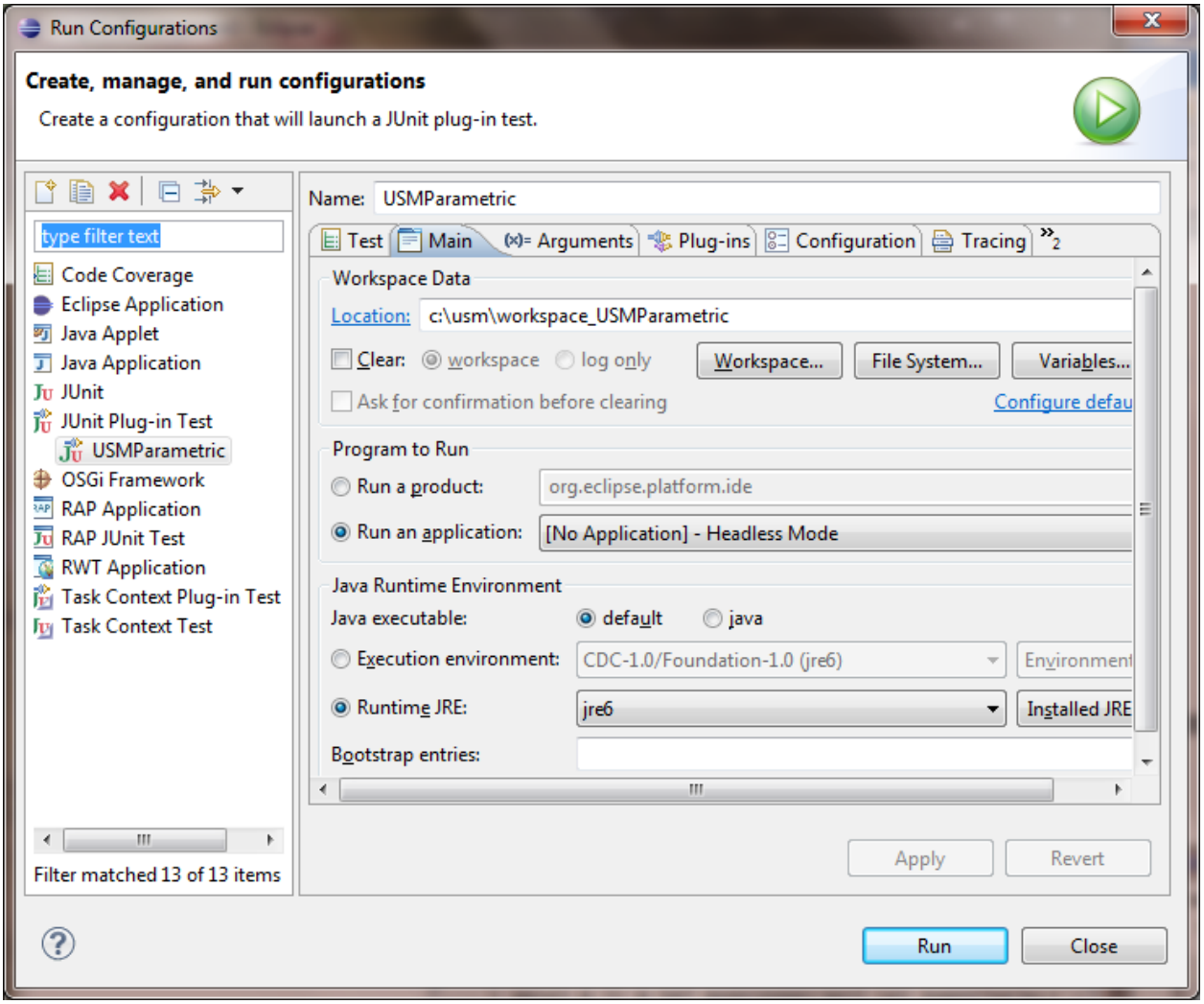
Generating the Data

Parametric

1) Switch to the USMPlugin workspace
2) Go to the Run configurations ("Run"->"Run Configurations…")
3) Create a new "JUnit Plug-in Test" for the ParametricTests class (see screenshot)



4) On the Main tab, point to the USM_Refactoring workspace and make sure the "Clear" tick box is not selected under the "Workspace Data"
5) Select "Run an Application" under "Program To Run" and choose "[No Application] – Headless Mode"



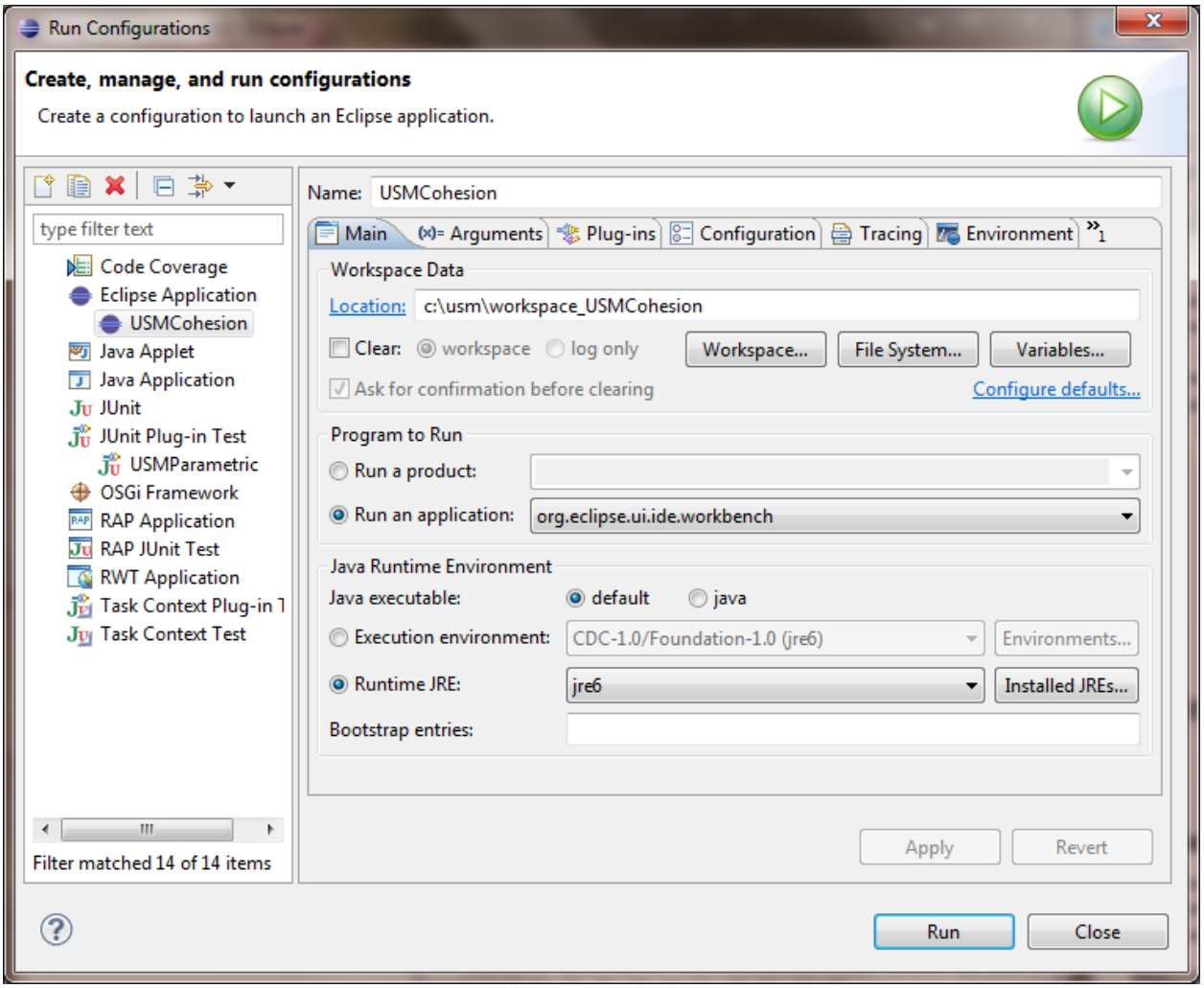
Run this test and upon completion the console log will display a comma separated table of the results (note the lines below have been truncated). Go to the top of the table to get the optimum values (the table is sorted in ascending order by totalCost)

```
---,key,cognitiveOverloadPenalty,elementCost,cohesionCost,couplingCost,chunkCost,totalCost
---,37,59.0,33.0,479.32434642696916,103.84099798701314,616.1653444139823,675.1653444139823…
---,41,59.0,33.0,480.1536501998002,103.84099798701314,616.9946481868134,675.9946481868134…
---,73,59.0,33.0,480.1536501998002,103.84099798701314,616.9946481868134,675.9946481868134…
---,18,75.0,32.0,478.72228643564125,101.59987679913604,612.3221632347772,687.3221632347772…
---,34,75.0,32.0,479.9264064182972,101.59987679913604,613.5262832174333,688.5262832174333…
---,21,73.0,33.0,478.72228643564125,103.84099798701314,615.5632844226544,688.5632844226544…
---,69,73.0,33.0,479.9264064182972,103.84099798701314,616.7674044053103,689.7674044053103…
---,81,73.0,33.0,480.1536501998002,103.84099798701314,616.9946481868134,689.9946481868134…
---,17,75.0,32.0,480.1536501998002,103.84099798701314,615.9946481868134,690.9946481868134…
---,19,76.0,33.0,478.72228643564125,103.84099798701314,615.5632844226544,691.5632844226544…
---,35,76.0,33.0,479.9264064182972,103.84099798701314,616.7674044053103,692.7674044053103…
```

At first glance, this approach may seem useful as a way to automatically suggest design refactoring. However, for a more general case the problem becomes identifying the alternative code structures that are equivalent in behaviour. If the software language were designed with "equivalent" code structures that were easy to transform between, then this approach would become more generally applicable. Alternatively if there was a "usually" equivalent code structures and sufficient automated testing then candidate transformations could be validated.

Cohesion Level

1) Switch to the USMPlugin workspace
2) Go to the Run configurations ("Run"->"Run Configurations…")
3) Create a new "Eclipse Application" that points to the workspace_USMCohesion and for which the program to run is the workbench (see screenshot)



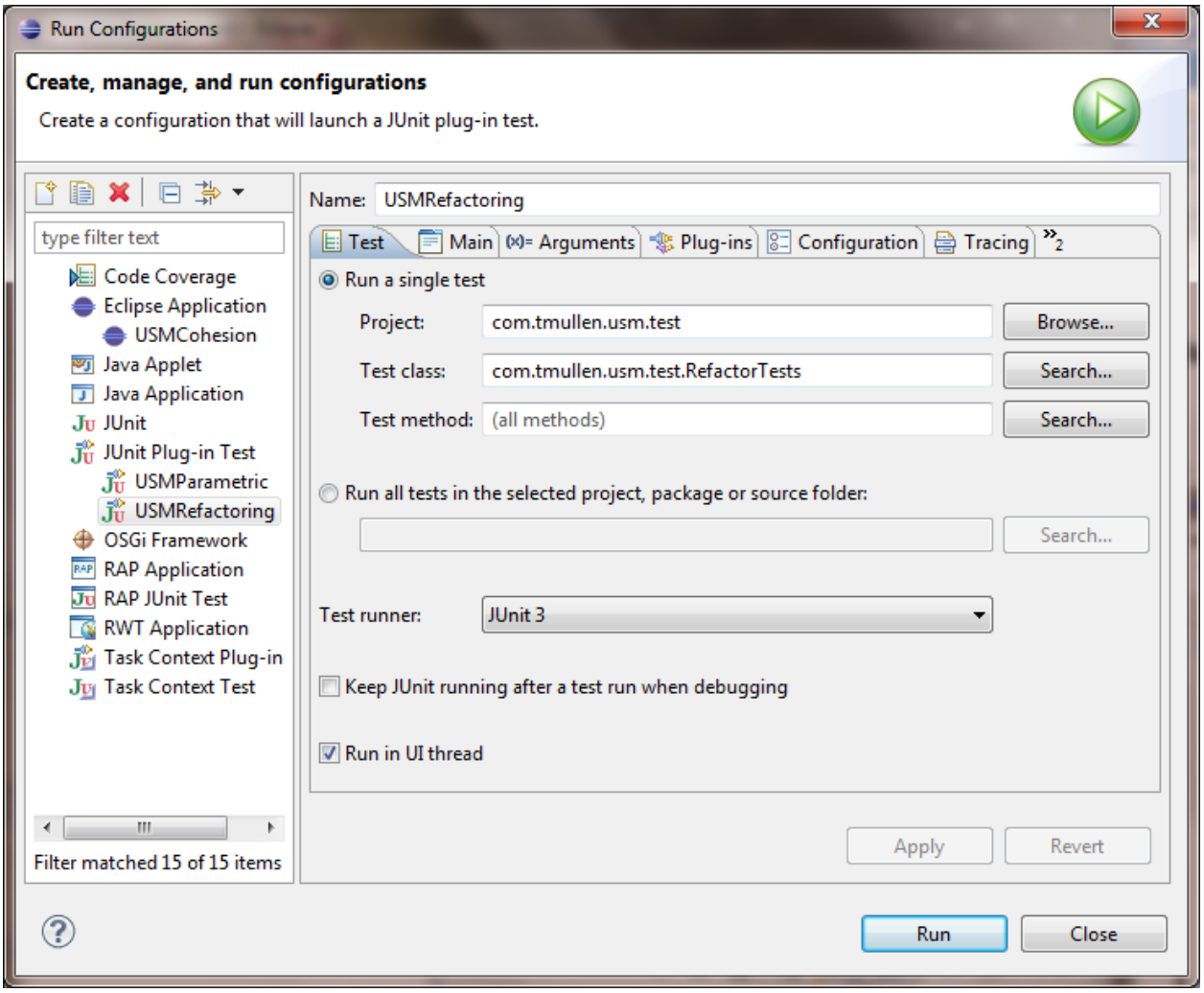
1) Run this "Run configuration"
2) A new workbench with the design patterns test code will be opened. Initiate a complete build ("Project"-> "Clean…" if you have your workspace set to automatic build)
3) Go to the output directory and open up the newly created UNIVERSE.csv file
4) Extract the rows with the following labels (it may be easier to Filter the data to only rows with type = TYPE). The data for each case is in the aggregate columns

   Lcom/tmullen/cohesion/caller/After;

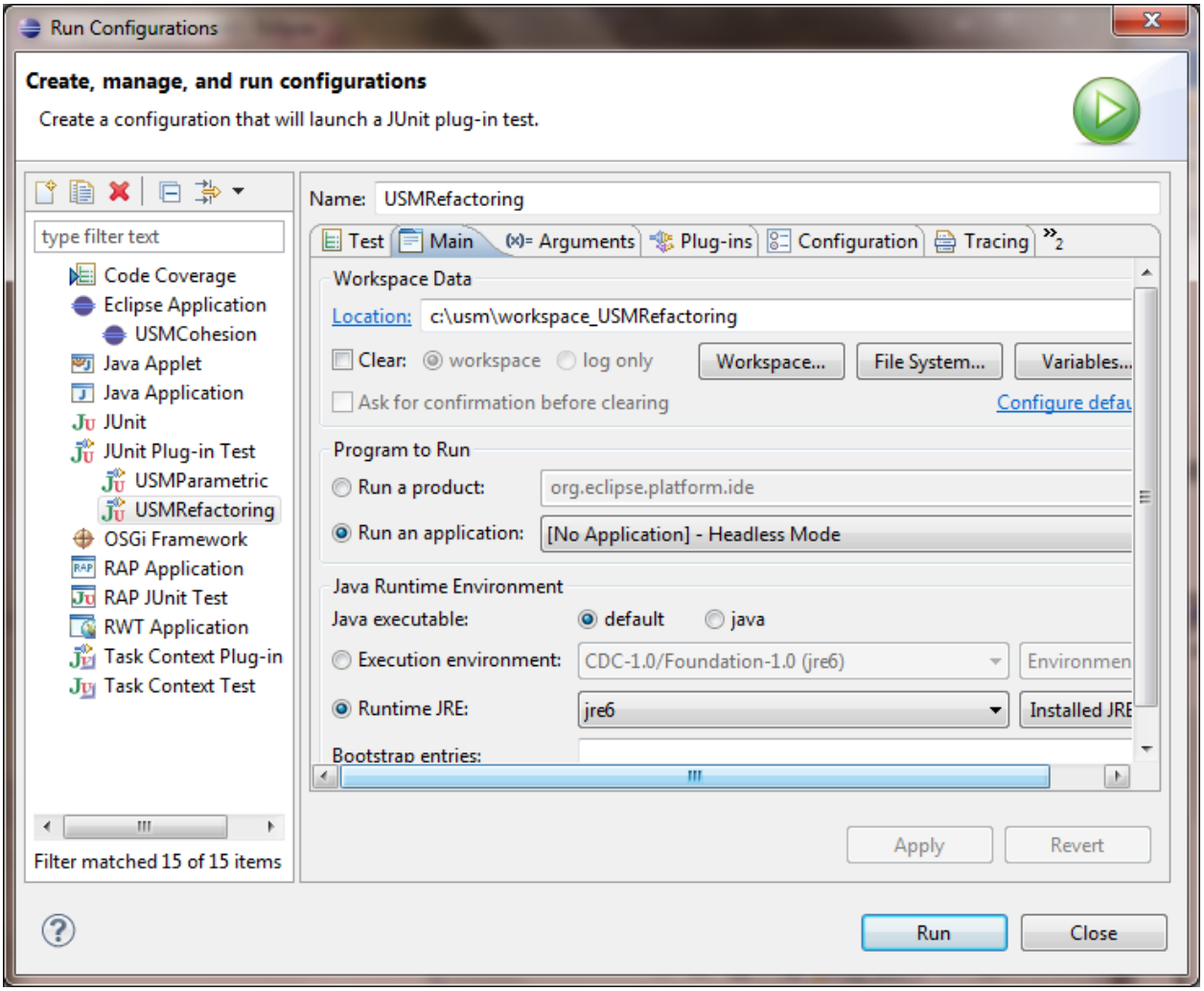
   Lcom/tmullen/cohesion/caller/Before;

Refactoring

1) Switch to the USMPlugin workspace
2) Go to the Run configurations (“Run”->”Run Configurations…”)
3) Create a new “JUnit Plug-in Test” for the RefactorTests class (see screenshot)



4) On the Main tab, point to the USM_Refactoring workspace and make sure the “Clear” tick box is not selected under the “Workspace Data”
5) Select “Run an Application” under “Program To Run” and choose “[No Application] – Headless Mode”



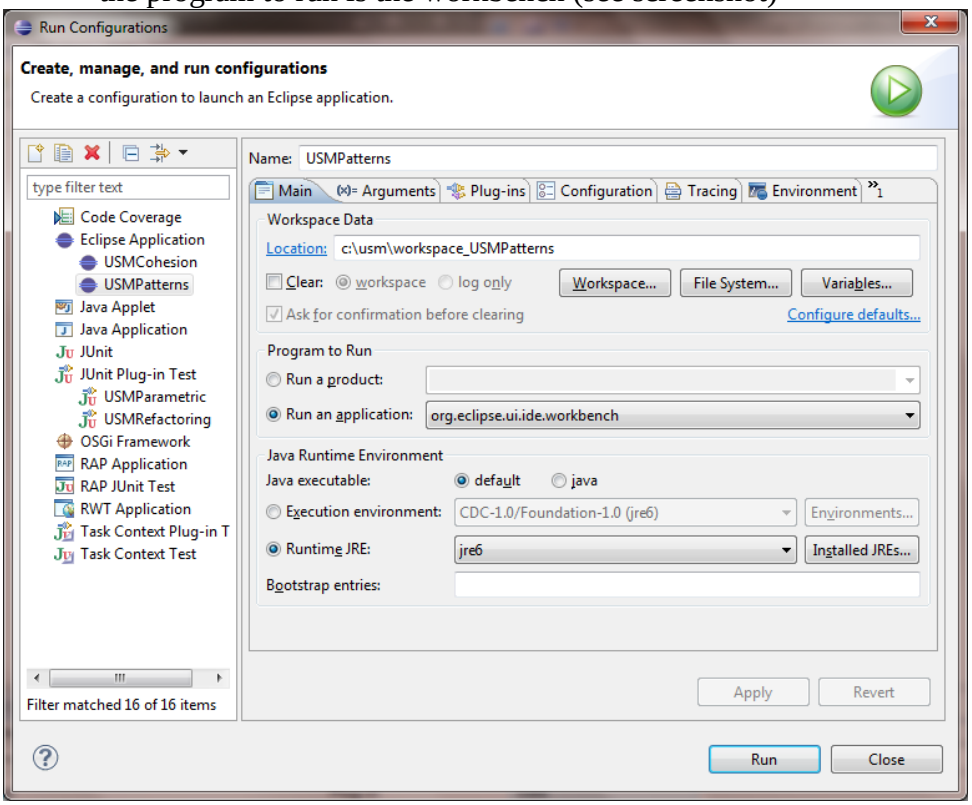
Run this test and upon completion the console log will show the following

```
Testing[StatementGroupTest]..
beforeMetrics  [148.29,   121,   8,     15.98, 3.31,  ]
afterMetrics   [55.97,    27,    9,     18.47, 1.51,  ]
test results   [true,     true,  true,  true,  true,  ]
Overall Test Result[PASS]
Testing[ExtractMethodTest]..
beforeMetrics  [5,088.89, 5,041, 10,    17.78, 20.1,  ]
afterMetrics   [109.8,    18,    18,    68.45, 5.34,  ]
test results   [true,     true,  true,  true,  true,  ]
Overall Test Result[PASS]
Testing[InlineMethodTest]..
beforeMetrics  [65.38,    12,    13,    30.69, 9.69,  ]
afterMetrics   [51.84,    25,    7,     15.38, 4.46,  ]
test results   [true,     true,  true,  true,  true,  ]
Overall Test Result[PASS]
```

The values in the square brackets are [Total, Cognitive Penalty, Element Cost, Coupling, Cohesion] in that order.

Design Patterns Data

1) Switch to the USMPlugin workspace
2) Go to the Run configurations ("Run"->"Run Configurations…")
3) Create a new "Eclipse Application" that points to the workspace_USMPatterns and for which the program to run is the workbench (see screenshot)



4) Run this "Run configuration"
5) A new workbench with the design patterns test code will be opened. Initiate a complete build ("Project"-> "Clean…" if you have your workspace set to automatic build)
6) Go to the output directory and open up the newly created UNIVERSE.csv file
7) Extract the rows with the following labels (it may be easier to Filter the data to only rows with type = PACKAGE). The data for each case is in the aggregate columns

Lcom/tmullen/fivevarieties/onebehaviour/complex1to1/hierarchy
Lcom/tmullen/fivevarieties/onebehaviour/complex1to1/strategy
Lcom/tmullen/fivevarieties/onebehaviour/complex1to1/sw1tch
Lcom/tmullen/fivevarieties/onebehaviour/complexindependent/hierarchy
Lcom/tmullen/fivevarieties/onebehaviour/complexindependent/strategy
Lcom/tmullen/fivevarieties/onebehaviour/complexindependent/sw1tch
Lcom/tmullen/fivevarieties/onebehaviour/simplesingle/hierarchy
Lcom/tmullen/fivevarieties/onebehaviour/simplesingle/strategy
Lcom/tmullen/fivevarieties/onebehaviour/simplesingle/sw1tch
Lcom/tmullen/onebehaviour/complex1to1/hierarchy
Lcom/tmullen/onebehaviour/complex1to1/strategy
Lcom/tmullen/onebehaviour/complex1to1/sw1tch
Lcom/tmullen/onebehaviour/complexindependent/hierarchy
Lcom/tmullen/onebehaviour/complexindependent/strategy
Lcom/tmullen/onebehaviour/complexindependent/sw1tch
Lcom/tmullen/onebehaviour/simplesingle/hierarchy
Lcom/tmullen/onebehaviour/simplesingle/strategy
Lcom/tmullen/onebehaviour/simplesingle/sw1tch