|  |  |  |  |
| --- | --- | --- | --- |
| **Conditional Statements with Static Configuration Class (OPTXOR)** | | | |
| **Summary** | | | |
| A refactoring to implement OPTXOR variability based on conditional statements with configuration in a static Java class. | | | |
| **Configuration Mechanism** | | | |
| The configuration is realized using String constants in Java class to be configured before compilation. | | | |
| **Motivation** | | | |
| The resulting code implements variants at one place; hence it improves identifying the executed code. | | | |
| **Supported Characteristics** | | **Supported Elements** | |
| **Binding Time** | Compile | **JaMoPP Java Model** | |
| **Variability Type** | OPTXOR | |  |  |  | | --- | --- | --- | | * CompilationUnit * Import * Class * Interface | * Enumeration * Field * Method * Constructor | * Block * Statement | | |
| **Extensible** | No |
| **Quality Goal** | | Conciseness | |
| **Limitations** | | | |
| Following software elements are not supported because they cannot co-exist:   * Variable Class- and Interface- Signatures (extends, implements) * Methods with equal names but varying return types * Fields with equal names but different types * Local variables with equal names but different types and referencing elements out- side the containing VP. | | | |
| **Alternatives** | | | |
|  | | | |
| **Example** | | | |
| Two implementations of a statement in a method being combined by introducing variant-specific conditions which evaluate a configuration from a configuration class. | | | |
| **Leading** | | | **Integration** |
| **public** **void** doSth(){  print("Leading");  } | | | **public** **void** doSth(){  print("Integration");  } |
| **Refactored SPL** | | | |
| **public** **void** doSth(){ // Line breaks reduced for the sake of brevity  **if**(Config.CONF1.equals("Leading")) { print("Leading"); }  **if**(Config.CONF1.equals("Integration")) { print("Integration"); }  }  **class** Config {  **public** **static final** String *CONF1* = "Integration"; // Either “Leading” or ” Integration”  } | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction: CompilationUnit** | | | | |
| **Summary** | | | | |
| A SPL must integrate the compilation units from all Variants. Therefore, this refactoring copies the compilation units from the integration copies to the leading copy, if they do not yet exist. | | | | |
| **Preconditions** | | | | |
| **Location** | | | | |
| Element | | CompilationUnit | | |
| Exclusion | |  | | |
| **Implementing Elements** | | | | |
| Element | | CompilationUnit | | |
| Exclusion | |  | | |
| **Example** | | | | |
| Merges the missing ClassB.java (contained in org.example.somepackage) into the leading Variant. | | | | |
| **Leading** | | | **Integration** | |
| **org.example.somepackage:**  **• ClassA.java**  **• ClassC.java** | | | **org.example.somepackage:**  **• ClassA.java**  **• ClassB.java**  **• ClassC.java** | |
| **Refactored SPL** | | | | |
| **org.example.somepackage:**  **• ClassA.java**  **• ClassB.java**  **• ClassC.java** | | | | |
| **Additional Parameters** | | | | |
| String: leadingSrcPath: The path to the source folder of the leading project to add the compilation units into. | | | | |
| **Mechanics** | | | | |
| Iterate over all integration Variants and their compilation units. Build the URI representing the path for each compilation units, create a new model resource for this URI, and place the compilation units in it. | | | | |
| **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading then  continue;  **endif**    **foreach** CompilationUnit:cu ∈ variant.implementingElements **do**  **foreach** String:segment ∈ cu.nameSpaces **do**  leadingSrcPath ← leadingSrcPath.concatenate(leadingSrcPath, segment);  leadingSrcPath ← leadingSrcPath.concatenate(leadingSrcPath, getFileSeparator());  **endforeach**    compilationUnitName ← getFileName(cu);  leadingSrcPath ← leadingSrcPath.concatenate(leadingSrcPath, compilationUnitName);  Resource:resource ← rs.createResource(URI( leadingSrcPath));  resource.contents.add(cu);  **endforeach**  **endforeach** | | | | |
| **Instruction: Import** | | | | |
| **Summary** | | | | |
| To allow for a complete single code base, all dependencies must be reflected. The refactoring carries over all imports. | | | | |
| **Preconditions** | | | | |
| **Location** | | | | |
| Element | CompilationUnit | | | |
| Exclusion |  | | | |
| **Implementing Elements** | | | | |
| Element | Import | | | |
| Exclusion |  | | | |
| **Example** | | | | |
| Merges the import from the integration copy into the leading copy. | | | | |
| **Leading** | | | | **Integration** |
| **import** com.example.SimpleClass; | | | | **import** com.example.ExtendedClass; |
| **Refactored SPL** | | | | |
| **import** com.example.SimpleClass;  **import** com.example.ExtendedClass; | | | | |
| **Additional Parameters** | | | | |
|  | | | | |
| **Mechanics** | | | | |
| Iterates over the Variants and their imports. It adds the import to the VP’s location, if it does not contain an equal import. | | | | |
| CompilationUnit: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Import:import ∈ variant.implementingElements **do**  **if** !vpLocation.contains(import) **then**  vpLocation.add(import);  **endif**  **endforeach**  **endforeach** | | | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Class in a MemberContainer** | | |
| **Summary** | | |
| The SPL must contain all classes from the leading and integration copies. This refactoring integrates the classes contained in a member container from the integration copies into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Class | |
| Exclusion | * VP with a location that already has an interface or enumeration with an equal name. | |
| **Example** | | |
| Merges class B into the leading Variant’s member container (a class in this case). | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **private class** A {…};  } | | **public** **class** SomeClass {  **private class** B {…};  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  **private class** A {…};  **private class** B {…};  } | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their classes. Adds the class to the VP’s location if it does not contain a class, interface or enumeration with the same name. | | |
| MemberContainer: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Class:class ∈ variant.implementingElements **do**  vpLocation.add(class);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Class in a CompilationUnit** | | |
| **Summary** | | |
| The SPL must contain all classes from the leading and integration copies. This refactoring integrates the classes contained in a compilation unit from the integration copies into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | CompilationUnit | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Class | |
| Exclusion | * VP with a location that already has an interface or enumeration with an equal name. | |
| **Example** | | |
| Merges class B into the leading Variant’s compilation unit. | | |
| **Leading** | | **Integration** |
| **SomeClass.java:**  **private class** A {…}; | | **SomeClass.java:**  **private class** B {…}; |
| **Refactored SPL** | | |
| **SomeClass.java:**  **private class** A {…};  **private class** B {…}; | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their classes. Adds the class to the VP’s location if it does not contain a class, interface or enumeration with the same name. | | |
| CompilationUnit: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Class:class ∈ variant.implementingElements **do**  vpLocation.add(class);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Interface in a MemberContainer** | | |
| **Summary** | | |
| The SPL must contain all interfaces from the leading and integration copies. This refactoring integrates the interfaces contained in a member container from the integration copies into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Interface | |
| Exclusion | * VP with a location that already has a class or enumeration with an equal name. | |
| **Example** | | |
| Merges interface B into the leading Variant’s member container (a class in this case). | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **private interface** A {…};  …  } | | **public** **class** SomeClass {  **private interface** B {…};  …  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  **private interface** A {…};  **private interface** B {…};  …  } | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their interfaces. Adds the interface to the VP’s location if it does not contain a class, interface or enumeration with the same name. | | |
| MemberContainer: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Interface:interface ∈ variant.implementingElements **do**  vpLocation.add(interface);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Interface in a CompilationUnit** | | |
| **Summary** | | |
| The SPL must contain all interfaces from the leading and integration copies. This refactoring integrates the interfaces contained in a compilation unit from the integration copies into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | CompilationUnit | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Interface | |
| Exclusion | * VP with a location that already has a class or enumeration with an equal name. | |
| **Example** | | |
| Merges interface B into the leading Variant’s compilation unit. | | |
| **Leading** | | **Integration** |
| **SomeClass.java:**  **private interface** A {…}; | | **SomeClass.java:**  **private interface** B {…}; |
| **Refactored SPL** | | |
| **SomeClass.java:**  **private interface** A {…};  **private interface** B {…}; | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their interfaces. Adds the interface to the VP’s location if it does not contain a class, interface or enumeration with the same name. | | |
| CompilationUnit: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Interface:interface ∈ variant.implementingElements **do**  vpLocation.add(interface);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Enumeration in a MemberContainer** | | |
| **Summary** | | |
| The SPL must integrate the enumerations from any Variant, including their constants. This refactoring integrates enumerations contained in a member container from the integration copies into the leading copies. It also merges the constants of enumerations with equal names. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Enumeration | |
| Exclusion | * VP with a location that already has a class or interface with an equal name. | |
| **Example** | | |
| Missing enumeration AnotherEnum gets integrated into the leading member container. Also shows the integration of enumeration constant B from the integration implementation of SomeEnum into the leading implementation. | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **public** **enum** SomeEnum { ***A***; }  } | | **public** **class** SomeClass {  **public** **enum** SomeEnum { ***B***; }  **public** **enum** AnotherEnum { ***X***; }  } |
| **Refactored SPL** | | |
| **public** **class** A {  **public** **enum** SomeEnum { ***A, B***; }  **public** **enum** AnotherEnum { ***X***; }  } | | |
| **Additional Parameters** | | |
|  | | |

|  |
| --- |
| **Mechanics** |
| This algorithm has two key steps: In the first step (first foreach), it collects all available enumeration names from the Variants. It stores their enumeration objects (leading or, if not available, first integration), the constants of that enumeration in all Variants and whether or not it already has a leading implementation.  In the next step (second foreach), it uses the maps to get the enumeration object, adds the missing constants to that object and adds the enumeration to the VP’s location if it has no leading implementation. |
| MemberContainer: vpLocation ← vp.location;  Map<String, Enumeration>: enumerationsToName;  Map<String, Set<String>>: constantsToEnumName;  Map<String, Boolean>: leadingToEnumName;  **foreach** Variant: variant ∈ vp.variants **do**  **foreach** Enumeration:enumeration ∈ variant.implementingElements **do**  **if** enumerationsToName.containsKey(enumeration.name) || variant.isLeading) **then**  enumerationsToName.put(enumeration.name, enumeration);  **endif**  leadingToEnumName.put(enumeration.name, variant.isLeading);  **foreach** EnumConstant: enumConst ∈ enumeration.constants **do**  constantsToEnumName.gets(enumeration.name).add(enumConst.name);  **endforeach**  **endforeach**  **endforeach**  **foreach** String: enumName ∈ enumerationsToName.keys **do**  enumeration ← enumerationsToName.get(enumName);  **foreach** String: constName ∈ constantsToEnumName.get(enumName) **do**  **if** !hasConstantWithSameName(enumeration, constName) **then**  enumConst.name ← constName;  enumeration.constants.add(enumConst);  **endif**  **endforeach**    **if** !leadingToEnumName.get(enumName) **then**  vpLocation.add(enumeration);  **endif**  **endforeach** |

|  |  |  |
| --- | --- | --- |
| **Instruction: Enumeration in a CompilationUnit** | | |
| **Summary** | | |
| The SPL must integrate the enumerations from any Variant, including their constants. This refactoring integrates enumerations contained in a compilation unit from the integration copies into the leading copies. It also merges the constants of enumerations with equal names. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | CompilationUnit | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Enumeration | |
| Exclusion | * VP with a location that already has a class or interface with an equal name. | |
| **Example** | | |
| Missing enumeration AnotherEnum gets integrated into the leading compilation unit. Also shows the integration of enumeration constant B from the integration implementation of SomeEnum into the leading implementation. | | |
| **Leading** | | **Integration** |
| **SomeClass.java:**  **public** **enum** SomeEnum { ***A***; } | | **SomeClass.java:**  **public** **enum** SomeEnum { ***B***; }  **public** **enum** AnotherEnum { ***X***; } |
| **Refactored SPL** | | |
| **SomeClass.java:**  **public** **enum** SomeEnum { ***A***, ***B***; }  **public** **enum** AnotherEnum { ***X***; } | | |
| **Additional Parameters** | | |
|  | | |

|  |
| --- |
| **Mechanics** |
| This algorithm has two key steps: In the first step (first foreach), it collects all available enumeration names from the Variants. It stores their enumeration objects (leading or, if not available, first integration), the constants of that enumeration in all Variants and whether or not it already has a leading implementation.  In the next step (second foreach), it uses the maps to get the enumeration object, adds the missing constants to that object and adds the enumeration to the VP’s location if it has no leading implementation. |
| CompilationUnit: vpLocation ← vp.location;  Map<String, Enumeration>: enumerationsToName;  Map<String, Set<String>>: constantsToEnumName;  Map<String, Boolean>: leadingToEnumName;  **foreach** Variant: variant ∈ vp.variants **do**  **foreach** Enumeration:enumeration ∈ variant.implementingElements **do**  **if** enumerationsToName.containsKey(enumeration.name) || variant.isLeading) **then**  enumerationsToName.gets(enumeration.name) ← enumeration;  **endif**  leadingToEnumName.gets(enumeration.name) ← variant.isLeading;  **foreach** EnumConstant: enumConst ∈ enumeration.constants **do**  constantsToEnumName.gets(enumeration.name).add(enumConst.name);  **endforeach**  **endforeach**  **endforeach**  **foreach** String: enumName ∈ enumerationsToName.keys **do**  enumeration ← enumerationsToName.gets(enumName);  **foreach** String: constName ∈ constantsToEnumName.gets(enumName) **do**  **if** !hasConstantWithSameName(enumeration, constName) **then**  enumConst.name ← constName;  enumeration.constants.add(enumConst);  **endif**  **endforeach**    **if** !leadingToEnumName.gets(enumName) **then**  vpLocation.add(enumeration);  **endif**  **endforeach** |

|  |  |  |
| --- | --- | --- |
| **Instruction: Field** | | |
| **Summary** | | |
| The SPL must contain the fields from all Variants. The refactoring integrates fields from all Variants into the leading Variant. In case of different initial values among the implementations, initialization blocks with conditional assignments get introduced. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Field | |
| Exclusion | * Fields with equal names but different types. | |
| **Example** | | |
| Integrates the missing field b into the leading Variant and introduces an initializer block to integrate the initializations from both Variants. | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **private int** a = 0;  } | | **public** **class** SomeClass {  **private int** a = 1;  **private int** b = 1;  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  **private int** a;  **private int** b = 1;  {  **if**(Config.CONF1.equals("Leading")){  a = 0;  }  **if**(Config.CONF1.equals("Integration")){  a = 1;  }  }  } | | |
| **Additional Parameters** | | |
|  | | |

|  |
| --- |
| **Mechanics** |
| The algorithm first adds the import of the configuration class to the containing compilation unit. It then (first foreach) builds maps to store the field objects, their positions within the parent container and their initial values to the field’s name. It also stores the field’s initial value and the ID of the Variant that it is implemented in.  It then deleted the fields of the leading Variant from the VP’s location and gererates two blocks (to initialize static and non-static fields). In the following foreach it adds the fields to the VP’s location, removes final modifiers (if applicable) and fills the initializer blocks with Variant specific conditional assignments if they have more than one initial values. Finally, it adds the initializer blocks to the VP’s location if they are not empty. |
| Class: vpLocation ← vp.location;  addConfigurationClassImportIfMissing(vpLocation.containingCompilationUnit);  Map<String,Field>: fieldToFieldName;  Map<String,Integer>: positionToFieldName;  Map<String, Set<Expression>>: initialValuesToFieldName;  Map<Expression, String>: variantIDToInitialValue;  **foreach** Variant:variant ∈ vp.variants **do**  **foreach** Field:field ∈ variant.implementingElements **do**  fieldToFieldName.put(field.name, field);  positionToFieldName.put(field.name, field.container.indexOf(field));    **if** !initialValuesToFieldName.get(field.name).contains(field.initialValue) **then**  initialValuesToFieldName.get(field.name).add(field.initialValue);  **endif**    variantIDToInitialValue.put(field.initialValue, variant.id);  **endforeach**  **endforeach**  deleteVariableFieldsFromLeading(vp);  Block: nonStaticBlock;  Block: staticBlock;  staticBlock.modifiers.add(new Final());  **foreach** String: fieldName ∈ fieldToFieldName.keys **do**  Field: field ← fieldToFieldName.get(fieldName );  int: fieldPos ← positionToFieldName.get(fieldName );  List< Expression >: initialValues ← initialValuesToFieldName.get(fieldName);    vpLocation.add(fieldPos, field);  removeFinalModifier(field);    **if** initialValues.size > 1 **then**  field.initialValue ← null;  **if** isStatic(field) **then**  createFieldConditionalInitialization(initialValues, staticBlock);  **endif**  **else**  createFieldConditionalInitialization(initialValues, nonStaticBlock);  **endelse**  **endif**  **endforeach**  **if** staticBlock.size > 0 **then**  vpLocation.add(staticBlock);  **endif**  **if** nonStaticBlock.size > 0 **then**  vpLocation.add(nonStaticBlock);  **endif** |
| **// ---------------------------------------------------------------------------**  **// Function: createFieldConditionalInitialization**  **// Creates a new conition to initialize a field.**  **Input**: VariationPoint: vp // the variation point  Block: block; // the block to add the condition to  List< Expression >: initialValues; // the initial values of for the field  Map<Expression, String>: variantIDToInitialValue // variant ids of the initial values  **Output**:  **foreach** Expression: initialValue ∈ initialValues **do**  String: variantId ← variantIDToInitialValue.get(initialValue);  String: groupId ← vp.group.id;    String: conditionString ← “SPLConfig.” + groupId + “.equals(” + variantId + ”)”;  Condition: condition;  condition.condition ← expressionFromString(conditionString);  ExpressionStatement: assignmentStatement ←  initialValueToStandaloneAssignment(initialValue);  condition.ifBlock.add(assignmentStatement);  block.add(condition);  **endforeach** |

|  |  |  |
| --- | --- | --- |
| **Instruction: Method** | | |
| **Summary** | | |
| The SPL classes must contain the methods from all Variants. Therefore, this refactoring integrates the methods from the integration Variants into the leading Variant. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Method | |
| Exclusion | * Methods with equal names but different return types. * Methods with equal names and an equal set of parameters (parameter names not taken into account). | |
| **Example** | | |
| Integrates the method with the double parameter from the integration into the leading Variant since it does not contain a method with one double parameter. | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **public void** someMethod(){...};  **public void** someMethod(**int** i){...};  } | | **public** **class** SomeClass {  **public void** someMethod(){...};  **public void** someMethod(**double** d){...};  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  **public void** someMethod(){...};  **public void** someMethod(**int** i){...};  **public void** someMethod(**double** d){...};  } | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their methods. Adds the method to the VP’s location, if it does not contain a method with the same name and an equal set of parameters. | | |
| MemberContainer: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Method:method ∈ variant.implementingElements **do**  vpLocation.add(method);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Constructor** | | |
| **Summary** | | |
| The SPL classes must contain the constructors from all Variants. Therefore, this refactoring merges the constructors from the integration Variants into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | Class | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Constructor | |
| Exclusion | Constructors with an equal set of parameters (parameter names not taken into account). | |
| **Example** | | |
| Integrates the constructor with the double parameter from the integration into the leading Variant since it does not contain a constructor with one double parameter. | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {  **public** SomeClass(){...};  **public** SomeClass(**int** i){...};  } | | **public** **class** SomeClass {  **public** SomeClass(){...};  **public** SomeClass(**double** d){...};  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  **public** SomeClass(){...};  **public** SomeClass(**int** i){...};  **public** SomeClass(**double** d){...};  } | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants. Adds the Constructors to the VP’s location if the location does not contain a Constructor with an equal set of Parameters. | | |
| MemberContainer: vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Constructor:constructor ∈ variant.implementingElements **do**  vpLocation.add(constructor);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Block** | | |
| **Summary** | | |
| The SPL Classes must contain all initializer blocks from all Variants. Therefore, this refactoring merges the bocks from the integration Variants into the leading Variant. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | MemberContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Block | |
| Exclusion |  | |
| **Example** | | |
| Integrates a block from the integration into the leading Variant since it does not contain the block. | | |
| **Leading** | | **Integration** |
| **public** **class** SomeClass {} | | **public** **class** SomeClass {  { System.out.println(); }  } |
| **Refactored SPL** | | |
| **public** **class** SomeClass {  { System.out.println(); }  } | | |
| **Additional Parameters** | | |
|  | | |
| **Mechanics** | | |
| Iterates over all integration Variants and their Blocks and adds them to the VP’s location. | | |
| vpLocation ← vp.location;  **foreach** Variant:variant ∈ vp.variants **do**  **if** variant.isLeading **then**  **continue**;  **endif**  **foreach** Block:block ∈ variant.implementingElements **do**  vpLocation.add(block);  **endforeach**  **endforeach** | | |

|  |  |  |
| --- | --- | --- |
| **Instruction: Statement in a StatementListContainer** | | |
| **Summary** | | |
| Introduces conditional statements that match the SPL configuration to execute Variant specific statements. Local variable statements whose variable is referenced outside the VP, are placed in front of the conditional statement. It is initialized with the type’s default value and the Variant specific assignment is done within the conditional statement. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | StatementListContainer | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Statement | |
| Exclusion | * Local variable declarations with equal variable name but different variable types and elements out- side the containing VP that have references to the variable. | |
| **Example** | | |
| The integration method has a different initial value for x and a different argument in the print method. Splits x into declaration and assignment and executes the Variant specific code in the conditional statements. The variable x is initialized with 0 which is the default value for the int type. | | |
| **Leading** | | **Integration** |
| **public** **void** method(){  **int** x = 1;  print("Leading:" + x);  **return** x;  } | | **public** **void** method(){  **int** x = 2;  print("Integration:" + x);  **return** x;  } |
| **Refactored SPL** | | |
| **public** **void** method(){  **int** x = 0;  **if**(Config.CONF1.equals(“Leading”)){  x = 1;  print("Leading:" + x);  }  **if**(Config.CONF1.equals(“Integration”)){  x = 2;  print("Integration:" + x);  }  **return** x;  } | | |
| **Additional Parameters** | | |
|  | | |

|  |
| --- |
| **Mechanics** |
| The algorithm first adds the import of the configuration class to the containing compilation unit. It then calculates the position at which the variable elements have to be inserted at in the VP’s location. It then wraps the statements of each Variant into a condition that evaluates the SPL configuration. If the statement is a local variable statement, whose variable is referenced out- side the current VP, it gets split into declaration and assignment whereas the declaration gets stored and the assignment is done within the If-block. It also removes the final modifier of the variable (if applicable). It then adds the created condition into the VP’s location at the previously calculated position.  It then deletes the leading Variant’s elements from the VP’s location and adds the declaration in front of the first of the generated conditions. Finally, the algorithm checks whether the VP’s location is a method and whether it has a non-void return type and all Variants have a trailing return statement. In this case, the algorithm adds a default return statement at the end of the method. |
| StatementListContainer: vpLocation ← vp.location;  addConfigurationClassImportIfMissing(vpLocation.containingCompilationUnit);  Map<String, LocalVariableStatement>: localVariableStatementsToName;  int: variabilityPositionStart ← getVariabilityPosition(vp);  int: variabilityPositionEnd ← variabilityPositionStart;  **foreach** Variant:variant ∈ vp.variants **do**  String: groupId ← vp.group.Id;  String: variantId ← variant.Id;    String: conditionString ← “SPLConfig.” + groupId + “.equals(” + variantId + ”)”;  Condition: condition;  condition.condition ← expressionFromString(conditionString);  **foreach** Statement:statement ∈ variant.implementingElements **do**  int: offset ← variant.implementingElements.size –  variant.implementingElements.indexOf(statement);  **if** statement instanceof LocalVariableStatement  && isReferencedByPostdecessor(statement, offset) **then**  LocalVariable: variable ← statement.variable;  removeFinalModifier(variable);  statement ← extractAssignment(variable);  variable.initialValue ← defaultValueForType(variable.typeReference.target);    **if** !localVariableStatementsToName.containsKey(variable.name)||variant.isLeading **then**  localVariableStatementsToName.gets(variable.name) ← statement;  **endif**  **endif**  **if** statement != null **then**  condition.ifBlock.add(statement);  **endif**  **endforeach**  vpLocation.statements.add(variabilityPositionEnd++, currentCondition);  **endforeach**  deleteLeadingVariantElementsFromVPLocation(vp);  vpLocation.addAll(variabilityPositionStart, localVariableStatementsToName.values);  **if** vpLocation instanceof ClassMethod **then**  boolean: isVoid ← returnTypeIsVoid(vpLocation);  boolean: allVariantsHaveATrailingReturn ← allVariantsHaveATrailingReturn(vp);    **if** !isVoid && allVariantsHaveATrailingReturn **then**  Return: returnStatement;  Literal: defaultValue ← defaultValueForType(vpLocation.typeReference.target);  returnStatement.returnValue ← defaultValue;  vpLocation.add(returnStatement);  **endif**  **endif** |
| **// ---------------------------------------------------------------------------**  **// Function: getVariabilityPosition**  **// Calculates the position of the variable statements.**  **Input** : VariationPoint: vp  **Output**: int: The position.  StatementListContainer: vpLocation ← vp.location;  **foreach** Variant: variant ∈ vp.variants **do**  **if** variant.isLeading **then**  Statement: firstElement ← variant.implementingElements.gets(0);  return firstElement.eContainer.indexOf(firstElement);  **endif**  **endforeach**  Statement: firstElement ← vp.variants.gets(0).implementingElements.gets(0);  int: posIntegration ← firstElement.eContainer.indexOf(firstElement);  List< Statement >: predecessors ← firstElement.eContainer.subList(0, posIntegration);  predecessors ← intersect(vpLocation, predecessors);  **if** predecessors.size == 0 **then**  return 0;  **endif**  pos ← searchFirstGroupOccurence(vpLocation, predecessors);  return pos +1; |

|  |
| --- |
| **// ---------------------------------------------------------------------------**  **// Function: searchFirstGroupOccurence**  **// Searches the first occurrence of a given list of statements in a container.**  **Input:** StatementListContainer: targetContainer, List<Statement>: predecessors  **Output:** int: If group was found, the index of the last element of the group; otherwise -1.  predecessorPos ← 0;  **for** i ← 0 to predecessors.size −1 **do**  baseElement ← targetContainer.gets(i);  predecessor ← predecessors.gets(predecessorPos);    **if** baseElement.equals(predecessor) **then**  predecessorPos++;  **endif**  **elseif** isVariabilityCondition(baseElement) **then**  int: predecessorsSubList ← predecessors.subList(predecessorPos, predecessors.size);  predecessorPos += countVariableStatements(baseElement.ifBlock, predecessorsSubList);  **endelseif**    **if** predecessorPos == predecessors.size **then**  **if** predecessor instanceof LocalVariableStatement **then**  int: posNextVarCond ← findNextVariabilityCondition(targetContainer, i);  **if** posNextVarCond ! = −1 **then**  **return** posNextVarCond;  **endif**  **endif**  return i;  **endif**  **endfor**  **return** −1; |
| **// ---------------------------------------------------------------------------**  **// Function: findNextVariabilityCondition**  **// Finds the position of the next condition that was introduced by this variability  // mechanism and returns its position. Searches elements starting at a given index.**  **Input:** StatementListContainer: targetContainer; int: startIndex  **Output:** The position. -1 if nothing found.  **for** i ← startIndex **to** targetContainer.size −1 **do**  Statement: currentStatement ← targetContainer.statements.gets(i);  **if** isVariabilityCondition(currentStatement) **then**  **return** i;  **endif**  **endfor**  **return** −1; |

|  |
| --- |
| **// ---------------------------------------------------------------------------**  **// Function: isReferencedByPostdecessor**  **// Checks whether a LocalVariableStatement's LocalVariable is referenced  // by a following element in its parent container.**  **Input** : LocalVariableStatement: localVariableStatement; int: offset  **Output**: boolean: True if a reference was found; false otherwise.  LocalVariabe: variable ← localVariableStatement.variable;  List<Statement>: containerStatements ← localVariableStatement.eContainer.statements;  int: fromIndex ← containerStatements.indexOf(localVariableStatement) + offset;  int: toIndex ← containerStatements.size;  **if** fromIndex >= toIndex **then**  **return** false;  **endif**  List<Statement>: postDecessors ← containerStatements.subList(fromIndex, toIndex);  **foreach** Statement: postDecessor: postDecessors **do**  **if** hasReferenceTo(postDecessor, variable) **then**  **return** true;  **endif**  **endforeach**  **return** false; |

|  |  |  |
| --- | --- | --- |
| **Instruction: Statement in a Condition** | | |
| **Summary** | | |
| This refactoring integrated variable else-statements in a condition. It introduces conditional statements to execute the Variant code and merges them into the leading copy. | | |
| **Preconditions** | | |
| **Location** | | |
| Element | Condition | |
| Exclusion |  | |
| **Implementing Elements** | | |
| Element | Statement | |
| Exclusion |  | |
| **Example** | | |
| The integration Variant has one additional else-if in between. For each of the Variants, we introduce an if-statement to execute the variable statement. | | |
| **Leading** | | **Integration** |
| **public** **void** method(**int** i) {  **if** (i == 0) {  print("0");  } **else** **if** (i == 2) {  print("2");  }  } | | **public** **void** method(**int** i) {  **if** (i == 0) {  print("0");  } **else** **if** (i == 1) {  print("1");  } **else** **if** (i == 2) {  print("2");  }  } |
| **Refactored SPL** | | |
| **public** **void** method(**int** i) {  **if** (i == 0) {  print("0");  } **else** **if** (Config.CONF1.equals("Leading")) {  **if** (i == 2) {  print("2");  }  } **else** **if** (Config.CONF1.equals("Integration")) {  **if** (i == 1) {  print("1");  } **else** **if** (i == 2) {  print("2");  }  }  } | | |

|  |
| --- |
| **Additional Parameters** |
|  |
| **Mechanics** |
| The algorithm starts by storing the else statement of the VP’s location and adding the configuration class import afterwards. It then iterates over all Variants and their implementation (exactly one statement in this case). It wraps the statement in a condition that evaluates the SPL configuration and places each condition in the else-branch of the previous condition. When adding the statements to the condition it checks whether the statement is a block. If so, it sets it directly as the if-block. At then end, we append the original else-statement to the last condition that we created. |
| Condition: vpLocation ← vp.location;  Statement: elseStatement ← vpLocation.elseStatement;  addConfigurationClassImportIfMissing(vpLocation.containingCompilationUnit );  String: groupId ← vp.group.Id;  Condition: previousCondition ← vpLocation;  **foreach** Variant:variant ∈ vp.variants **do**  String: conditionString ← ”SPLConfig.” + groupId + ”.equals(” + variantId + ”)”;  Condition: variabilityCondition;  variabilityCondition.condition ← expressionFromString(conditionString);    Statement: statement ← variant.statements.get(0);    **if** statement instanceOf Block **then**  variabilityCondition.statement ← statement;  **endif**  **else**  variabilityCondition.ifBlock.add(statement);  **endelse**    previousCondition.elseStatement ← variabilityCondition;  previousCondition ← variabilityCondition;  **endforeach**  previousCondition.elseStatement ← elseStatement; |