Eclipse Refactoring

Michael Petito
Clarkson University
EE564 Spring 2007

Introduction

The architecture and structure of any useful body of code continually evolves to address new requirements and resolve bugs. Developers often find it necessary to re-organize such code as a part of this maintenance. A powerful toolset exists within Eclipse to help developers quickly make these modifications: refactoring. Refactoring is the process of modifying a software system while preserving its external behavior. A typical refactoring might be decomposed into many small behavior preserving transformations that as a whole produce a larger and higher level, yet still behavior preserving, change. Eclipse provides a general-purpose API for implementing refactorings that can be applied to any Eclipse workspace elements, from text resources to whole projects. Several plugins for Eclipse draw upon the API to implement refactorings for specific languages; here we will focus on refactorings implemented for Java.

Eclipse Refactoring API

The Eclipse refactoring API, part of the Language Toolkit (LTK), is implemented within the `org.eclipse.ltk.core.refactoring` and `org.eclipse.ltk.ui.refactoring` plug-ins since R3.0. The former provides classes that abstract the process of refactoring and define high level interfaces used by participating objects; the later defines a wide array of UI elements common to most refactorings for a consistent look and feel. While the UI elements are important for gathering input and presenting the user with feedback and previews, of particular interest here are the API components that facilitate code manipulation.

Refactoring Model

The API for refactoring provides a process-level abstraction upon which specific refactorings may be built. Figure 1 shows the elements of this abstraction at a very high level. Here, arrows between elements represent dependencies. For example, a *Refactoring Implementer* requires knowledge of (ie. a reference to) some *Selected Element* to understand what the refactoring is being applied to. Similarly, a *Refactoring Dialog* and *Implementer* must exchange information to show such UI elements as interactive refactoring previews.

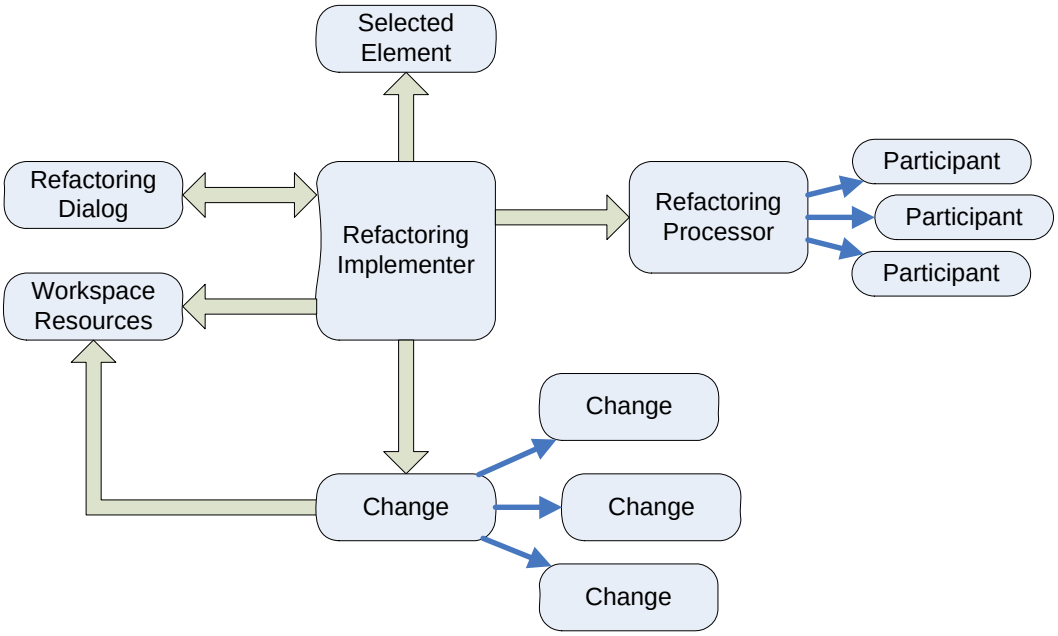

Figure 1: Refactoring API Elements

Once a refactoring has been initiated, an implementer of that refactoring is used to coordinate condition checking, gathering details about the refactoring, and ultimately producing a series of changes that may be applied to the Eclipse workspace to accomplish the refactoring. This implementer must extend the abstract class `org.eclipse.ltk.core.refactoring.Refactoring`. The life-cycle for this class is shown in Figure 2.

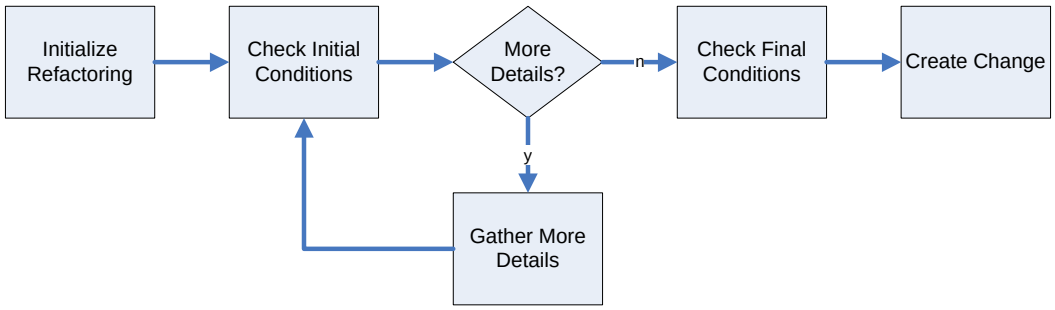

Figure 2: Refactoring Life-Cycle

The refactoring process begins by initializing a refactoring instance. This includes passing it contextual information, such as which workspace elements are selected, and sometimes even behavioral information, such as a processor specialized for the given context. The refactoring must then check that the given context is reasonable for the type of refactoring. This check is defined in overrided implementations of the `Refactoring` instance method `checkInitialConditions`. It may not be the case, however, that all information is presently available to begin execution. More information may be required from the user and can be gathered through various UI components. This process is continued until all necessary information has been gathered. At this point, the refactoring becomes responsible for ensuring that its subsequent

execution would produce code that is semantically equivalent. This occurs during `checkFinalConditions`, where the refactoring may iterate over various representations of the elements to be refactored, including AST representations of code. The entire process of checking initial- and final- conditions is referred to as pre-condition checking.

Classes derived from `Refactoring` are finally responsible for producing an instance of `org.eclipse.ltk.core.refactoring.Change` that describes the entire effect of the refactoring. Changes may affect any workspace element(s) necessary to accomplish the refactoring and may themselves be composed of smaller changes.

A refactoring may use any methods suitable for producing the required workspace changes. However, it may also utilize an additional process abstraction provided by the LTK to help produce these changes. A processor / participant model is provided to help coordinate a refactoring processor and zero or more refactoring participants. Refactorings that utilize this abstraction must be derived from `org.eclipse.ltk.core.refactoring.participants.ProcessorBasedRefactoring`. The processor / participant model requires that the refactoring load a `RefactoringProcessor` and any `RefactoringParticipants` that are suitable for the current context. An extension point may be provided here so that additional processors or participants may be dynamically registered and participate in the refactoring.

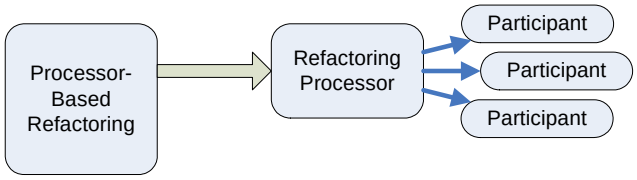

Figure 3: Processor / Participant Refactoring Model

Each participant may implement parts of the initial- or final- condition checking and/or change generation, while the `RefactoringProcessor` is itself responsible for aggregating the contributions of the participants.

Describing Workspace Changes

A refactoring ultimately produces a single the abstract class `Change` that describes the workspace changes necessary to accomplish the refactoring. Implementers of refactorings must re-use or implement their own classes derived from `Change` to specify the behavior of a change. Changes must support modifying both saved and unsaved resources.

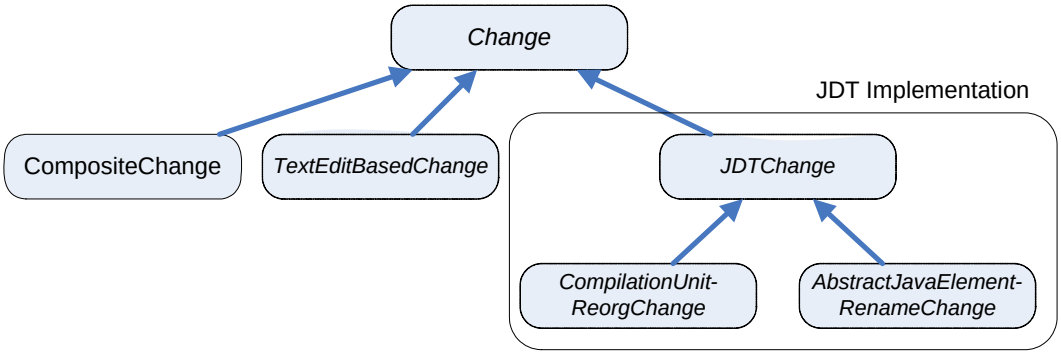

Figure 4: Change Type Heirarchy

A hierarchy of abstract changes is presented in Figure 4. Inheritance and composition of such change types allows developers to define complex changes incrementally, simplifying the code of any given piece. A single change, typically a composite change, is produced by the invocation of the `Refactoring` instance method `createChange`.

When applying changes to the workspace, execution is typically managed by the class `PerformChangeOperation`. Performing a change produces a new instance of `Change` that may be used to undo the operation.

Performance

Performance of a refactoring can be an important issue when considering the issuance of refactorings against large workspaces. A typical refactoring may need to load and manipulate an AST representation of each body of code that is to be modified. API documentation suggests that refactorings should only maintain one instance of such large objects in memory at any given time to reduce memory complexity.

For pervasive code refactorings, it seems that even more memory may be required to represent the final `Change` hierarchy. Each `Change` instance may itself be small, but refactorings are typically composed of many small changes, perhaps down to the level of text resource modifications. Further, since it is already necessary to enumerate elements of the code during pre-condition checking, in practice most or all of this change hierarchy is generated early in the refactoring life-cycle and persisted. Even if such a large change hierarchy may fit in memory, once applied, a complementary hierarchy of approximately equal size must be generated to support the Eclipse undo model and persisted for the duration of the undo history!

Implementation History

Prior to Eclipse R3.0, the refactoring API did not exist within the LTK and was instead a JDT specific implementation beginning with R2.0. To provide an API for implementing refactorings in other languages, a set of interfaces and abstract classes used within the JDT refactoring packages were somewhat extended and moved to the LTK. While a few method names changed during this move, the underlying architecture for implementing refactorings has remained consistent since R2.0. One exception to this is the refactoring processor / participant model. This process abstraction was introduced so that language

neutral refactorings that utilize language or context specific processors might be implemented. The classes present in the LTK and corresponding classes that existed in the R2.0 JDT UI plug-in are shown in Table 1.

LTK Class	Selected Methods	R2.0 JDK Class	Selected Methods
Refactoring		**IRefactoring**	
	checkInitialConditions		checkPreconditions
	checkFinalConditions		
	createChange		createChange
		Refactoring	
			checkActivation
			checkInput
RefactoringProcessor			checkPreconditions
	isApplicable		
	checkInitialConditions		
	checkFinalConditions		
	createChange		
	postCreateChange		
RefactoringParticipant			
	initialize		
	checkConditions		
	createChange		
RefactoringStatus		**RefactoringStatus**	
Change		**Change**	
	isEnabled		isActive
	isValid		
	getAffectedObjects		getModifiedLanguageElement
	perform		perform

Table 1: LTK and Corresponding R2.0 Classes

Other external effects, such as the transition from Java 2 to Java 5 with the inclusion of generics, has not had an impact on the design. While the developers seem to be aware of generics (since they include inline comments for most collections describing their type in generics notation), they have not used generics. The introduction of generics has, however, led to the implemtation of an Infer-Type-Arguments refactoring. It also seems that other Java 5 advanced language features such as enumerated types are not presently used even within head revisions of the refactoring API.

One likely reason for creating the refactoring LTK was so that refactorings for all languages may be handled as common supertypes by the Eclipse IDE. This, of course, works well as common methods of invocation (also contained within the LTK) may be used for any refactorings. This is much more effective and robust than using something along the lines of reflection to try to invoke refactorings based on some assumedly common method signatures. Instead, with the approach taken by the LTK, the compiler will verify that refactorings implement the common `Refactoring` class interface and all that is needed at runtime is a successful type cast.

JDT Refactoring Implementation

There are several plug-ins for Eclipse to support Java development. In particular, the plug-in `org.eclipse.jdt.ui` contains several Java refactoring implementations spread over tens of packages. See *Appendix B: JDT Refactorings by Type* for a complete list of refactorings by package.

It is presently unclear why `org.eclipse.jdt.ui` contains the implementation for JDT refactorings. The Eclipse refactoring API is broken down very clearly within the LTK into "core" and "ui" packages; the JDT implementation squeezes both components into the JDT UI plug-in. An approach consistent with the LTK might be to create two new JDT plug-ins: `org.eclipse.jdt.core.refactoring` and `org.eclipse.jdt.ui.refactoring`.

There are possible reasons why the JDT refactoring implementation may have been kept within the `org.eclipse.jdt.ui` plugin. First, it may have been difficult to sort out all of the plug-ins that the refactoring implementation would rely on as it has a high degree of efferent coupling. Given that the refactoring implementation also includes UI components, the JDT *core* plug-in would lack the necessary UI dependencies. The JDT UI plug-in, with the addition of the LTK plug-ins, would have a sufficient set of dependencies to support refactoring. Another reason may be that moving the refactoring implementation into separate plug-ins could introduce cyclic dependencies both between the new core and UI refactoring plug-ins, and between those plug-ins and the JDT plug-ins themselves. Again because of the degree of coupling, it may have been simplest to avoid these issues altogether and include the refactoring implementation within the JDT UI plug-in.

That code composing the JDT refactoring implementation is also significantly larger than its counterpart in the LTK. Since the LTK is language and refactoring neutral, many of its classes have abstract methods that must be implemented by refactoring implementations to define specific behaviors. The overwhelming majority of the code in the JDT is implementing the refactoring methods for checkInitialConditions, checkFinalConditions and createChange (either directly or through RefactoringProcessors tailored to specific element types), as well as subtyping the Change class to define new change behaviors.

Implementation Style

While the JDT refactorings are very effective from the perspective of the user, their code base is difficult to decipher. The underlying Eclipse refactoring API is very straightforward and well documented via Javadocs. However, lengthy undocumented and uncommented routines abound in the JDT specific implementation. Refer to *Appendix A: Code Statistics* for some specific code metrics. If it weren't for documentation inherited from the LTK classes that the JDT refactorings extend, it may very well be impossible to understand the code.

Table 2: Selected SEF Methods
addGetterSetterChanges
checkArgName
checkCompileErrors
checkFinalConditions
checkInHierarchy
checkInitialConditions
checkMethodNames
computeUsedNames
createChange
createEdits
createFieldAccess
createGetterMethod
createModifiers
createSetterMethod
initialize
makeDeclarationPrivate
mappingErrorFound
processCompilerError

As an example, consider the refactoring class *SelfEncapsulateFieldRefactoring*. This defines a refactoring where all read or write references to a field may be proxied through a corresponding accessor or mutator method. Most of the refactoring computation is performed during *checkFinalConditions*, a method defined by the LTK for ensuring that all pre-conditions for the refactoring (semantic or otherwise) are satisfied prior to applying the refactoring. Here, *checkFinalConditions* is 70 lines long without a single inline, block, or Javadoc comment. It interleaves pre-condition checking, change hierarchy generation, *and* progress bar manipulation as it references tens of undocumented and uncommented helper methods. The only hint a maintenance developer would have to the functionality provided by each of these methods is their name and argument types. To provide a feel for how difficult this might be, a list of methods for the *SelfEncapsulateFieldRefactoring* is provided in Table 2.

One interesting observation about the refactorings provided for the JDT is that *few* of them use the processor / participant refactoring model provided by the LTK. This may explain why these classes were missing from those that existed in R2.0 as shown in Table 2. Where the processor / participant model is used, a specialized processor is loaded to perform the refactoring based on the type of Java element being refactored. However, there is no logic within the refactoring itself to determine which processor must be used. For example, the *JavaMoveRefactoring* class and its supertype *MoveRefactoring* rely on a *MoveProcessor* constructor parameter to define the entire refactoring process for a specific Java element type! The appropriate *MoveProcessor* is selected and instantiated by the helper class *RefactoringExecutionStarter*, which is referenced by several JDT refactoring UI packages. Other refactorings are implemented in similar ways, including the various rename refactorings. In this way, parts of the underlying refactoring logic have been relocated away from their respective core refactoring classes and pushed nearer to the UI components.

Exception Handling

As refactorings can and are applied to large systems of code, there is plenty of opportunity for something to go wrong or an otherwise exceptional state to arise. However, the LTK does not define any refactoring-specific exceptions or conventions for handling unexpected conditions. Since each refactoring is responsible for its own pre-condition checking (which may return a fatal failure status code), the refactoring API assumes that a successful pre-condition check is sufficient for a successful refactoring application.

The only checked exceptions thrown by any of the refactoring LTK and the JDT classes for refactoring is *org.eclipse.core.runtime.CoreException*. Core

exceptions may contain information that describes the nature of the error. However, *none* of the exception handlers within the JDT reference this information in their handling routines in order to account for the exceptional state. Instead, within the JDT implementation, exception handling tends to be either logging the exception and allowing the routine to quietly fail or to generate a new exception containing the status information of the original. Exceptions that arise may leave the refactoring in an inconsistent state that will generate more exceptions later. For example, many exceptions leave fields set to *null* and further processing is unable to proceed correctly. These exceptions will then, in effect, propagate up the call stack, perhaps to methods within the LTK, or worse, to methods within the UI modules where appropriate handling will be difficult. Consider the `SelfEncapsulateFieldRefactoring` class. It does reference the checked exception `JavaModelException` seven times:

- checkArgName catches it, ignores it and the continues,
- createChange catches it, logs it; and continues,
- the constructor, createFieldAccess, createModifiers, and initialize(IField) methods throw it (unhandled)
- initialize(RefactoringArguments) catches it and returns a generic failed status.

The same class also references `CoreException` seven times. In *all* of these cases it simply ignores the `CoreException` that may be generated and passes it up the call stack. This example is representative of the Java refactorings and demonstrates a lack of appropriate exception handling. There is no effort to examine the cause of an exception near its generation and subsequently handle it. Perhaps it is the case that there is no action that can be taken for the exceptions that may be generated. However, in either case, it would have been useful to at least speculate (through comments or perhaps informative status codes) why the exception occurred given the current context of the refactoring and why it cannot be handled.

Another issue might arise even if exceptions were not generated or handled locally. Refactorings are responsible for pre-condition checking, including verifying correctness and preserving code semantics, prior to the application of the refactoring. However, given the life-cycle model for classes derived from `Refactoring`, it seems that temporal issues may arise. For example, after pre-condition checking has been completed, but before the refactoring is applied, changes to affected resources might cause the refactoring to fail or worse, complete incorrectly. Such scenarios might include cases where a resource is modified outside of Eclipse, resources become locked by another application, or access to resources is lost due to transient hardware conditions (such as network connectivity). Another possibility is that the refactoring relies on code that simply has bugs! In any of these cases, the corresponding exceptions might propagate across component and module boundaries where their meaning, and ultimately any hope of handling them gracefully, will be lost.

Experiences with the Eclipse Refactoring Code Base

Delving into the source code of a large application to which I have not contributed has been an interesting experience. It was fairly easy to locate the Refactoring API in the LTK and understand its inner-workings through the included Javadocs. The references cited here also helped somewhat. While the API is simple enough to understand in concept, its usage in the JDT was made significantly more complex by lack of documentation. In other words, components that would be re-used by future development (the API) were well documented while everything else (the JDT) was not. I wonder how common this is for open-source projects. Certainly, in either case, someone would notice rather quickly if an API that should be useful is too poorly documented and thus can't be easily used. During maintenance of the JDT code, however, does the lack of comments and documentation present itself as an issue? My experience in "closed-source" projects so far has been that *any* code that is not well documented becomes an issue in later development cycles.

Conclusion

The Eclipse Refactoring API and the corresponding implementation for the JDT has proven very powerful for restructuring large systems of code. The API provided by the LTK certainly can provide a foundation for a wide variety of refactorings in any language. While both sets of code have been evolving over time, there are several areas identified here that could use improvement. Some of these issues relate to the runtime behavior of the refactorings; yet these issues have not surfaced in practice. Perhaps the most pressing issue is the maintainability of such code that is highly coupled and largely undocumented.

Sources

Leif Frenzel: *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*, http://www.eclipse.org/articles/Article-LTK/ltk.html

Tobias Widmer: *Unleashing the Power of Refactoring*, http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html

Appendix A: Code Statistics

A collection of statistics computed by Metrics 1.3.6 for Eclipse (http://metrics.sourceforge.net/) for some relevant libraries.

Metric	R3.2
`o.e.jdt.internal.ui.refactoring.*`	
Lines of Code	19,241
Method Lines of Code (Avg)	7.9
Method Lines of Code (Max)	126
Number of Classes	213
`o.e.jdt.internal.corext.refactoring.*`	
Lines of Code	69,668
Method Lines of Code (Avg)	7.4
Method Lines of Code (Max)	175
Number of Classes	491
`org.eclipse.ltk.core.refactoring`	
Lines of Code	8,287
Method Lines of Code (Avg)	7.1
Method Lines of Code (Max)	192
Number of Classes	213

Appendix B: JDT Refactorings by Type

Refactoring ID	Package (1)	Refactoring Type
o.e.jdt.ui.convert.anonymous	code	ConvertAnonymousToNestedRefactoring
o.e.jdt.ui.extract.constant	code	ExtractConstantRefactoring
o.e.jdt.ui.extract.method	code	ExtractMethodRefactoring
o.e.jdt.ui.extract.temp	code	ExtractTempRefactoring
o.e.jdt.ui.inline.constant	code	InlineConstantRefactoring
o.e.jdt.ui.inline.method	code	InlineMethodRefactoring
o.e.jdt.ui.inline.temp	code	InlineTempRefactoring
o.e.jdt.ui.introduce.factory	code	IntroduceFactoryRefactoring
o.e.jdt.ui.introduce.indirection	code	IntroduceIndirectionRefactoring
o.e.jdt.ui.introduce.parameter	code	IntroduceParameterRefactoring
o.e.jdt.ui.promote.temp	code	PromoteTempToFieldRefactoring
o.e.jdt.ui.replace.invocations	code	ReplaceInvocationsRefactoring
o.e.jdt.ui.infer.typearguments	generics	InferTypeArgumentsRefactoring
o.e.jdt.ui.rename.field	rename	RebaneFieldProcessor
o.e.jdt.ui.rename.compilationunit	rename	RenameCompilationUnitProcessor
o.e.jdt.ui.rename.enum.constant	rename	RenameEnumConstProcessor
o.e.jdt.ui.rename.java.project	rename	RenameJavaProjectProcessor
o.e.jdt.ui.rename.local.variable	rename	RenameLocalVariableProcessor
o.e.jdt.ui.rename.method	rename	RenameMethodProcessor
o.e.jdt.ui.rename.package	rename	RenamePackageProcessor
o.e.jdt.ui.rename.resource	rename	RenameResourceProcessor
o.e.jdt.ui.rename.source.folder	rename	RenameSourceFolderProcessor
o.e.jdt.ui.rename.type.parameter	rename	RenameTypeParameterProcessor
o.e.jdt.ui.rename.type	rename	RenameTypeProcessor
o.e.jdt.ui.copy	reorg	JavaCopyProcessor
o.e.jdt.ui.delete	reorg	JavaDeleteProcessor
o.e.jdt.ui.self.encapsulate	sef	SelfEncapsulateFieldRefactoring
o.e.jdt.ui.change.method.signature	structure	ChangeSignatureRefactoring
o.e.jdt.ui.change.type	structure	ChangeTypeRefactoring
o.e.jdt.ui.extract.interface	structure	ExtractInterfaceRefactoring
o.e.jdt.ui.extract.superclass	structure	ExtractSuperTypeRefactoring
o.e.jdt.ui.move	structure	JavaMoveRefactoring
o.e.jdt.ui.move.inner	structure	MoveInnerToTopRefactoring
o.e.jdt.ui.move.method	structure	MoveInstanceMethodRefactoring
o.e.jdt.ui.move.static	structure	MoveStaticMembersProcessor
o.e.jdt.ui.pull.up	structure	PullUpRefactoring
o.e.jdt.ui.push.down	structure	PushDownRefactoring
o.e.jdt.ui.use.supertype	structure	UseSuperTypeRefactoring
	(1) Under org.eclipse.jdt.internal.corext.refactoring	