Eclipse CDT refactoring overview and internals

Michael Rüegg

Institute For Software
University of Applied Sciences Rapperswil

Parallel Tools Platform (PTP) Workshop, Chicago
September, 2012

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

Refactoring Basics

Properties and Goals of Refactorings

- *“Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component.”* [Fowler1999]
- Goal of refactorings: Increase understandability and modifiability
- Focus on structural changes, strictly separated from changes to functionality
- Functionality preservation: guarantee that a refactoring does not introduce any bugs or invalidates any existing tests or functionality
- Manual refactorings are time-consuming and error-prone
  ⇒ Automatic refactorings in IDE’s can help
- Atomic vs. composite refactorings
- Flexibility through composing larger refactorings from smaller ones

Refactoring Basics

Automated Refactorings

- Requirements for automated refactorings:
  - are behavior-preserving when preconditions are satisfied
  - are only applicable if the context makes sense
  - are fast
  - allow a preview of the changes to occur
  - are undoable
  - preserve formatting and comments
- Typical steps during automated refactorings:
  1. Parsing of the program source to retrieve an Abstract Syntax Tree (AST)
  2. Program analysis with the AST to ensure preconditions are satisfied
  3. AST is transformed with the refactoring and presented in source format
- Challenge: refactorings have to consider the syntax and the semantics of the underlying programming language!

Refactoring Basics

Preserving Behaviour

- *“Refactorings always yield legal programs that perform operations equivalent to before the refactoring.”* [Opdyke1992]
- Opdyke identified a set of syntactic and semantic program properties which can be easily violated if explicit checks are not done
- Examples of these properties are *unique superclass* (single-inheritance languages), *distinct class names* (nested classes are not considered), *type safe assignements*, *semantically equivalent references and operations*, etc.
- Opdyke uses program properties to describe preconditions of low-level refactorings
- Example *Create empty class*:
  $\forall$ class $\in$ Program.classes, class.name $\neq$ newClassName.
- High-level refactorings: Behavior preservation of those refactorings is proven in terms of the lower level refactorings used to compose it

Refactoring Basics

Refactoring C++ Code

- Static type information and naming resolution makes program analysis and refactoring easier compared to dynamic languages
- But: C++ is complex (largely due to its history and evolution from C)
- Programs that make use of C++ machine-level language features such as pointers, `sizeof(object)` or cast operations are difficult to subsequently refactor [Opdyke1999]
- Even worse: usage of preprocessor
- *"In retrospect, maybe the worst aspect of Cpp is that it has stifled the development of programming environments for C. The anarchic and character-level operation of Cpp makes nontrivial tools for C and C++ larger, slower, less elegant, and less effective than one would have thought possible."* — Bjarne Stroustrup

Refactoring Basics

Example C++ Refactoring Challenges

- *Extract interface* refactoring: What can go wrong when we try to extract an interface from `Die`?

```
#include <cstdlib>
struct Die { // extract an interface
  int roll() const {
    return rand() % 6 + 1;
  }
};
struct AlwaysSixDie : Die {
  int roll() const {
    return 6;
  }
};
// Interface:
struct IDie {
  virtual ~IDie() {}
  virtual int roll() const =0;
};
```

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

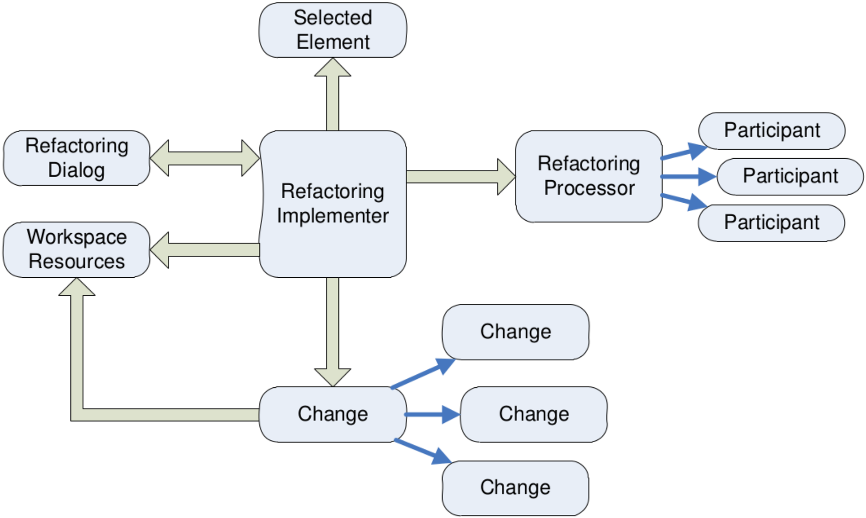
Overview LTK

What is LTK?

- Refactoring Language Toolkit (LTK) - a language neutral API for refactorings
- Used by Java Development Tools (JDT), C/C++ Development Platform (CDT) and others
- Consists of two plug-ins:
  - `org.eclipse.ltk.core.refactoring`
  - `org.eclipse.ltk.ui.refactoring`
- Most of the functionality of LTK is implemented in abstract classes which follow the template method pattern

Overview LTK

Elements of LTK



Selected Element

Refactoring Dialog

Refactoring Implementer

Refactoring Processor

Participant

Participant

Participant

Workspace Resources

Change

Change

Change

Change

Figure : Source [Widmer06]

Overview LTK

Refactoring Lifecycle Overview

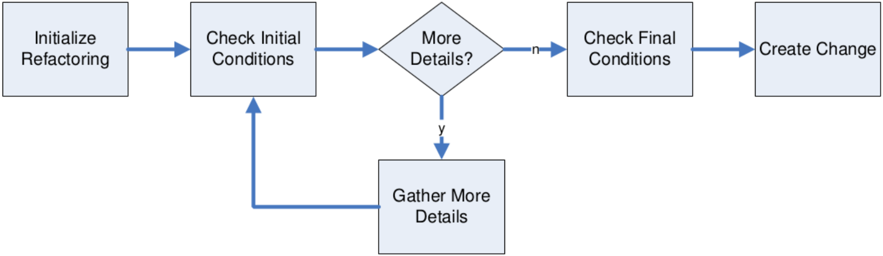


Figure : Source [Widmer06]

Overview LTK

LTK Refactoring Initializing

- Implement `org.eclipse.ui.IActionDelegate` and use extension points in `plugin.xml`
- `void selectionChanged(IAction, ISelection)`
  Enable / disable refactoring based on current selection
  ⇒ only trivial checks to prevent bad user experience!
- `void run(IAction)`
  Is executed when the user activates an available refactoring
  ⇒ use this to initialize a refactoring (e. g., selection, source file)

Overview LTK

Checking Preconditions and Transformation

- Base class for all LTK refactorings:
  `org.eclipse.ltk.core.refactoring.Refactoring`
- Checking Preconditions:
  - `RefactoringStatus checkInitialConditions(IProgressMonitor)`
    Based on the users selection we check the refactorings precondition without additional user input
  - `RefactoringStatus checkFinalConditions(IProgressMonitor)`
    Perform precondition checks that take the entered user information into account
  - `checkFinalConditions` is always called *after* calls to `checkInitialCOnditions` and *before* `createChange`
- Transformation: `Change createChange()`
  Creates a change object encapsulating all changes to be performed on the workspace ⇒ yields
  `org.eclipse.ltk.core.refactoring.Change`

Overview LTK

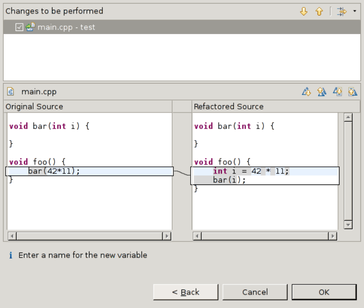
LTK Refactoring Status

- `org.eclipse.ltk.core.refactoring.RefactoringStatus`
- Used to communicate the result of the precondition checking to the refactoring framework
- `INFO`: For informational only
- `WARNING`: The refactoring can be performed, but the user could not be aware of problems or confusions resulting from the execution
- `ERROR`: The refactoring can be performed, but the refactoring will not be behavior preserving and / or the partial execution will lead to an inconsistent state (e. g., compile errors)
- `FATAL`: The refactoring cannot be performed, and execution would lead to major problems
- *Source: JavaDoc comments*

Overview LTK

Refactoring UI

- `org.eclipse.ltk.ui.refactoring.RefactoringWizard` (encapsulates the wizard itself)
- `org.eclipse.ltk.ui.refactoring.RefactoringWizardPage` (individual pages the wizard consists of)
- Every wizard inherits a standard preview page as well as a final page with a progress bar



Original Source:

```
void bar(int i) {
}
void foo() {
    bar(42*11);
}
```

Refactored Source:

```
void bar(int i) {
}
void foo() {
    int i = 42 * 11;
    bar(i);
}
```

Enter a name for the new variable

Overview LTK

Participants and Scriptable Refactorings

- A refactoring participant can participate in the condition checking and change creation of a refactoring processor
- Reason: Refactorings that change several source files may have impact on some of the other integrated tools
- Examples: Renaming classes in PDE, setting breakpoints in a debugger, consistency of C function declarations and JNI bindings
- Two scenarios for scriptable refactorings:
  - Reapplying refactorings on a previous version of a code base
  - Composing large and complex refactorings from smaller refactorings
- `org.eclipse.ltk.core.refactoring.RefactoringDescriptor` and `RefactoringContribution`

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

CDT Refactoring Support

CDT Refactoring History

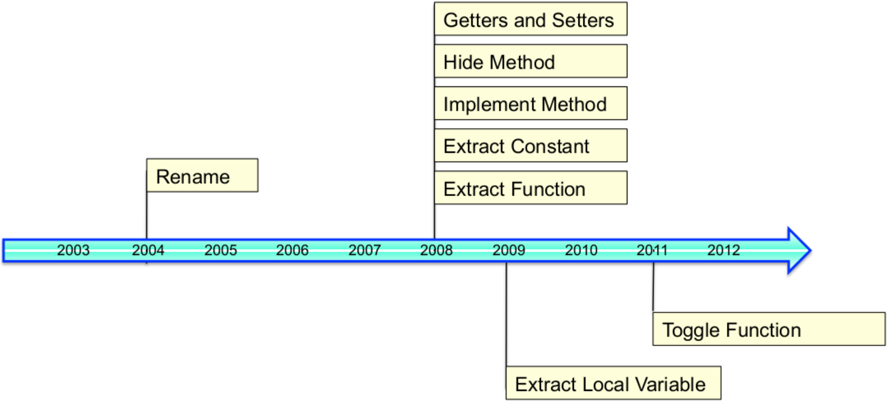
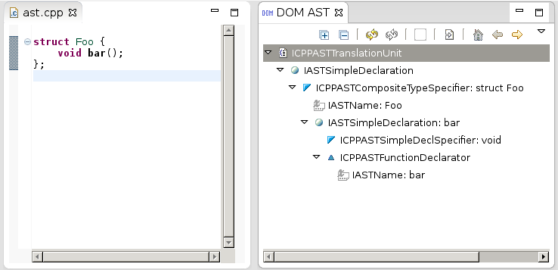


Figure : Source [Prigogin2012]

CDT Refactoring Support

Translation Unit

- A trainslation unit (TU) is a source file with all included headers
- A TU is represented by the interface `org.eclipse.cdt.core.model.ITranslationUnit`
- The root node of the AST has the type `org.eclipse.cdt.core.dom.ast.IASTTranslationUnit`
- C and C++ AST nodes are separated (e. g., `IASTUnaryExpression` and `ICPPASTUnaryExpression`); C has ~60, C++ ~90 nodes
- `org.eclipse.cdt.core.dom.ast.IASTNode` is the parent interface of all nodes in the AST



```
struct Foo {
    void bar();
};
```

```
ICPPASTTranslationUnit
  IASTSimpleDeclaration
    ICPPASTCompositeTypeSpecifier: struct Foo
      IASTName: Foo
      IASTSimpleDeclaration: bar
        ICPPASTSimpleDeclSpecifier: void
        ICPPASTFunctionDeclarator
          IASTName: bar
```

CDT Refactoring Support

Obtaining an AST

- Refactoring CDT base class: `org.eclipse.cdt.internal.ui.refactoring.CRefactoring`
- `ITranslationUnit` has a `getAST()` method which creates the `IASTTranslationUnit` for the TU
- Creates a new AST every time it is called ⇒ Better: `CRefactoring`'s `IASTTranslationUnit getAST(ITranslationUnit, IProgressMonitor)`
- This uses the AST cache of `CRefactoringContext`: `Map<ITranslationUnit, IASTTranslationUnit>`
- `CRefactoringContext` inherits from `org.eclipse.ltk.core.refactoring.RefactoringContext` and is a disposable context for C / C++ refactoring operations
- The context object has to be disposed after use ⇒ Failure to do so may cause loss of index lock!
- No problem when we execute a refactoring with `run()` of CDT's `RefactoringRunner`

CDT Refactoring Support

Querying the AST

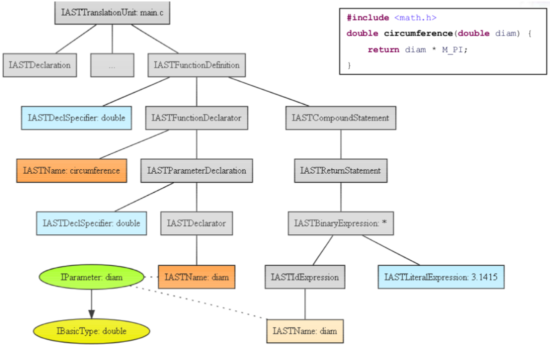
- The AST can be traversed in two ways:
  1. By calling `IASTNode`'s `getParent()` and `getChildren()` ⇒ cumbersome — we want to decouple the data from the operations that process the data
  2. By using the visitor design pattern; subtype of `org.eclipse.cdt.core.dom.ast.ASTVisitor`
     - `ASTVisitor` has overloaded `visit(IASTXXX)` methods for each node type
     - Each node class has an `accept(ASTVisitor)` method (defined in `IASTNode`) ⇒ calls `visit(this)`
     - Example visitor to collect all names:

```
class ASTNameVisitor extends ASTVisitor {
   List<IASTName> names = new ArrayList<IASTName>();
   {
     this.shouldVisitNames = true;
   }
   @Override
   public int visit(IASTName name) {
      names.add(name);
      return PROCESS_CONTINUE;
   }
}
```

CDT Refactoring Support

Binding resolution

- A binding encapsulates all the ways an identifier is used in a program
- Binding resolution is the process of searching the AST for usages of an identifier and generates an object of `IBinding`
- To get an `IBinding`, we call `resolveBinding()` on a name node (`IASTName`)



```
#include <math.h>
double circumference(double diam) {
    return diam * M_PI;
}
```

Figure : Source [Schorn2009]

CDT Refactoring Support

Index retrieval and querying

- The index contains information about:
  - References to macros and global declarations
  - Include directives and macro definitions
  - Bindings for each name
  - File-locations for each declaration, reference, include and macro definition
- Use `getIndex()` of `CRefactoring` to obtain the `org.eclipse.cdt.core.index.IIndex` because it makes sure to properly acquire and release the read lock for you (note: index will be for *all workspace projects*)
- `IIndex` contains methods to lookup program entities:

```
IIndexName[] findReferences(IBinding binding)
IIndexName[] findDeclarations(IBinding binding)
IIndexName[] findDefinitions(IBinding binding)
// all use:
IIndexName[] findNames(IBinding binding, int flags)
```

- Index usage: local changes vs. global changes

CDT Refactoring Support

Building and modifying AST nodes

- Use `CPPNodeFactory` to create new AST nodes (Abstract factory pattern)
- Example to create a namespace definition node:

```
ICPPASTNamespaceDefinition createNewNs(String ns) {
  CPPNodeFactory c = CPPNodeFactory.getDefault();
  IASTName n = c.newName(ns.toCharArray())
  return c.newNamespaceDefinition(n);
}
```

- Note that the original AST is frozen ⇒ therefore changes can only be applied on a copy of the AST (or on the sub-tree under change)
- Example how to make a decl specifier `const`:

```
IASTDeclSpecifier makeConst(IASTDeclSpecifier d) {
  IASTDeclSpecifier n = d.copy(CopyStyle.withLocations);
  n.setConst(true);
  return n;
}
```

CDT Refactoring Support

CDT's AST Rewrite

- Use `ASTRewrite` to modify code by describing changes to AST nodes
- `checkFinalConditions` of `CRefactoringContext` calls at its end `collectModifications(IProgressMonitor, ModificationCollector)`
- From there, we can obtain an `org.eclipse.cdt.core.dom.rewrite.ASTRewrite`
- Obtain an `ASTRewrite` for the currently active TU in the editor:

```
void collectModifications(IProgressMonitor pm,
                          ModificationCollector mc) {
  IASTTranslationUnit ast = getAst(tu);
  ASTRewrite r = mc.rewriterForTranslationUnit(ast);
```

- `ASTRewrite` provides the following methods:

```
void remove(IASTNode n, TextEditGroup eg)
ASTRewrite replace(IASTNode n, IASTNode repl,
                   TextEditGroup eg)
ASTRewrite insertBefore(IASTNode p, IASTNode insPoint,
                        IASTNode newN,TextEditGroup eg)
```

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

Refactoring Integration Tests in Eclipse CDT

- Refactoring tests in text files specify pre- and postconditions
- Example tests for *Remove Class* refactoring:

```
//!Not referenced local class should be removed
//@A.cpp
void foo() {
  class /*$*/A/*$$*/{};
}
//=
void foo() {
}

//!Error when not a class
//@.config
expectedInitialErrors=1
//@A.cpp
int /*$*/a/*$$*/;
```

Outline

1. Refactoring Basics
2. Overview LTK
3. CDT Refactoring Support
4. Refactoring Testing
5. Example: “Remove Class”

Example: “Remove class”

- Low-level refactoring *Delete unreferenced class* [Opdyke1992]
- Arguments: class C
- Preconditions: $\mathit{referencesTo}(C) = \emptyset \wedge \mathit{subclassesOf}(C) = \emptyset$
- ⇒ The class being deleted from the program is unreferenced; thus, all program properties are trivially preserved
- Example:

```
struct A {};   // A cannot be removed
};
struct C {};   // C cannot be removed
};
struct B: A { // B can be removed
  C c;
};
```

- ⇒ DEMO

Thanks for your attention!

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

- **IFS Institute for Software**, `http://ifs.hsr.ch`
- **CUTE** — *Green Bar for C++*, `http://cute-test.com`
- **Includator** — *Static Include Analysis for Eclipse CDT*, `http://includator.com`
- **Linticator** — *Flexe/PC-Lint Integration for Eclipse CDT*, `http://linticator.com`
- **SConsolidator** — *SCons Build Support for Eclipse*, `http://sconsolidator.com`
- **Mockator** — *Seams and Mock Objects for Eclipse CDT*, `http://mockator.com`

Bibliography I

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison Wesley, 1999.

Tobias Widmer.
Unleashing the Power of Refactoring.
Eclipse Corner Articles, 2006.

Michael Petito.
Eclipse Refactoring.
EE564, 2007.

Markus Schorn
Using CDT APIs to programmatically introspect C/C++ code.
EclipseCon, 2009

Mike Kucera
Parsing and Analyzing C/C++ code in Eclipse.
McMaster University, March 2012

Bibliography II

- J. van den Bos
  Refactoring (in) Eclipse.
  Master Software Engineering, 2008
- William F. Opdyke
  Refactoring Object-Oriented Frameworks.
  Ph.D. Thesis, 1992
- William F. Opdyke
  Refactoring C++ Programs.
  Article, 1999
- Alejandra Garrido and Ralph Johnson
  Challenges of Refactoring C Programs.
  ACM article, 2002
- Sergey Prigogin
  C++ Refactoring - Now for Real.
  EclipseCon, 2012