Hochschule für Technik Rapperswil

Loop Analysis and Transformation towards STL Algorithms

Master Thesis: *Fall Semester 2010*

Author
Pascal Kesseli

Supervisor
Prof. Peter Sommerlad

Abstract

Loops and iterations have always been a traditional error source in programming. "Off-by-one" errors, where an iteration is executed once too often or once too few, top the most common program errors. C++ provides STL algorithms to simplify the most common tasks accomplished using loops, helping to avoid these kinds of errors. Unfortunately, STL algorithms are not as commonly used as they could be.

This master thesis describes the development of an Eclipse C++ Development Tools (CDT) plug-in, which encourages and supports developers to use STL algorithms. The plug-in provides semi-automatic recognition and transformation of compatible loops to corresponding STL algorithms. To achieve this, tree pattern matching algorithms are applied to the processed abstract syntax trees (AST). The final plug-in features transformation of *for_each* and *find/find_if*-compatible *for* and *while* loops to equivalent STL algorithm function calls. The respective loop body is transformed into a corresponding functor. The user can select either a C++0x lambda expression, a TR1 *bind* expression or a C++98 *bind1st*/*bind2nd* expression.

Based on the foundation created with this project, there exist multiple extension possibilities. Additional algorithms, such as *generate* and *transform*, would greatly increase the number of use cases served. Furthermore, introducing trivial *explicit function* and *explicit functor class* functor transformations can increase the plug-in's flexibility even further. Lastly, the tree pattern matching engine, with all its benefits, should definitely be provided to the Eclipse end users. Various applications, from *tree pattern search masks* to *semi-automatic pattern definition based on existing codes*, could support programmers in all of their daily tasks and challenges.

Contents

1. Introduction

One of the very first things prospective programmers are being taught, after having written their first "Hello World" to the console, are the possibilities and proper use of loops and iterations. And it is done so not by coincidence: Loops are the very one programming expression that separate our modern software environments from traditional finite automatons. They bring us Turing-completeness. Unfortunately, they also introduce another property to our programs: They enable our code to never halt [Sip06, p.137-147,153-154].

Loops and iterations have thus always been a potential source of errors and complexity in programming, and there exist different systems and methods to meet these challenges. One of these measures in the C++ programming language was the introduction of the Standard Template Library (STL), which - among other things - attempted to allay the complexity of loops by mapping them to expressive, standardized algorithms. Even though the use of STL algorithms provides remarkable benefits, as explained in section 1.2.4, they are unfortunately not as frequently used as they could be. Reasons for this unfavorable situation are manifold and explained to further detail in section 1.2.4, but its implications can be summarized into one sentence:

Large portions of today's C++ code can be simplified to a great deal by replacing custom loops with STL algorithms.

This assumption represents the very essence of this master thesis' contents and objectives, which will be explained further in the following of this introduction. This chapter also includes a brief overview of the functional range of the C++ Standard Template Library, to provide readers unfamiliar with its possibilities with a context for the rest of the document.

1.1. Task description

To establish a thorough definition of this master thesis' goals and objectives, the task description is stated within three sections: The initial position and current situation (1.1.1), the definition of the actual problem to be addressed (1.1.2) and the nominal objectives of the thesis (1.1.3).

1.1.1. Initial position

Among C++ experts, the use of STL algorithms over custom loops is strongly emphasized [Mey01, Baj01, Lov10]. Nevertheless, there still exist countless instances in current C++ code where a manifold of different loop types could be replaced by an equivalent STL algorithm, as chapter 2 points out. Automatic code transformation systems often focus on reducing time-consuming tasks for programmers rather than improving the code quality [WY07], which in turn is usually accomplished by refactoring features. While their functionalities have grown remarkably over the past few years, major C++ Integrated Development Environments (IDEs) such as Eclipse C++ Development Tools (CDT) and Visual Studio currently lack the features to automatically map existing loops to STL algorithms.

1.1.2. Problem definition

Introducing STL functionality into legacy code is today a mainly manual task. It requires a conscious interpretation of a given loop's semantics, the knowledge about the available STL functionalities and an equivalent reformulation of the code in terms of an STL algorithm. These three requirements illustrate also the main reasons for the current lack of STL usage:

- Analyzing the code manually is expensive and may simply not be considered worthwhile
- STL algorithms require almost always advanced C++ concepts such as functors and iterators, which occur to be too little-known among even professional C++ programmes (see also section 1.2.4)
- Manually rephrasing a loop as STL algorithm may introduce new errors to the code

1.1.3. Objectives of thesis

Section 1.1.2 explained that manual transformation of loops to STL algorithms is expensive, challenging and error-prone. The possibilities, requirements and implementation options of automatic loop transformations will therefore be analyzed in the scope of this master thesis. The nominal objectives of this task are listed in the following sections.

Statistical analysis of existing code

Before evaluating and proposing possible transformation techniques, a statistical analysis of existing C++ code from different sources shall provide a representative context on the loops and code constructs to be transformed. These occurrences shall be separated into groups with similar semantical and syntactical properties that allow the application of common recognition and transformation techniques.

Research and formulate transformation systems

Based on the findings of the previous code analysis, feasible transformation techniques for the respective groups shall be proposed and evaluated. These systems may span a wide area of algorithms ranging from provably equivalent transformations to best-effort code substitutions.

Implement proposed systems

The most promising and beneficial transformation systems shall be implemented as an Eclipse CDT plug-in, be it that one such implementable technique has been found. The chosen implementation shall be available as a refactoring feature.

1.1.4. Expressive examples

Section 1.2 provides a detailed explanation of the STL features and structures focussed on in this master thesis. Nevertheless, let at this point follow some specific examples of concrete custom loop to STL transformations that should help express the goals of this project to more detail.

Searching an element by equality

Given any standard container and an element of the contained type, one may want to search an entry within the container that is equal to the value (e.g. to verify that aforesaid element is contained).

A custom, intuitive way of achieving this may be expressed by the following lines of code:

```cpp
vector<Person> p = getPersons();
Person inDemand("Raphael", "Weiss");
const Person *result = 0;
for (vector<Person>::size_type i = 0; i < p.size(); ++i) {
  if (p.at(i) == inDemand) {
      result = &p.at(i);
  }
}
if (result) {
  cout << *result << endl;
}
```

Listing 1.1: Custom search loop

One might agree that the solution in listing 1.1 is as straightforward and simple as it could ever be. It does not take more than some minutes to truly comprehend all possible results above code excerpt could yield. Yet, before judging, let us consider an equivalent expression as an STL algorithm:

```cpp
vector<Person> p = getPersons();
Person inDemand("Raphael", "Weiss");
vector<Person>::const_iterator result = find(p.begin(), p.end(), inDemand);
if (result != p.end()) {
  cout << *result << endl;
}
```

Listing 1.2: STL *find*

Listing 1.2 illustrates how the *find* algorithm reduces the five lines of the custom *for* loop to one expressive call to *find*. It can do so because the custom *for* loop is actually only a poor copy of what has already been implemented in the STL. Intuitively, pragmatic programmers tend to reuse existing functionalities instead of rewriting them, avoiding errors and providing their algorithms with descriptive names. There is, as some readers may probably already argue, of course a catch in this mentality, for which to point out we introduce some changes to the original code:

```
vector<Person> p = getPersons();
Person inDemand("Raphael", "Weiss");
const Person *result = 0;
for (vector<Person>::size_type i = 0; i < p.size(); ++i) {
  if (p.at(i) == inDemand) {
    result = &p.at(i);
  } else {
    registerSomewhere(p.at(i));
  }
}
if (result) {
  cout << *result << endl;
}
```

Listing 1.3: Custom search and register looop

In this case, a simple *find* is no more sufficient to replace the code in listing 1.3 for it does no more only implement a simple search through the container. The possibilities, limits and very purposes that lay within the STL will thus be explained in section 1.2.

Searching an element by a certain property

Listing 1.4 instantiates the very same example as listing 1.1, except that a person with a matching first name is demanded.

```
vector<Person> p = getPersons();
string inDemand = "Raphael";
const Person *result = 0;
for (vector<Person>::size_type i = 0; i < p.size(); ++i) {
  if (p.at(i).getFirstName() == inDemand) {
    result = &p.at(i);
  }
}
if (result) {
  cout << *result << endl;
}
```

Listing 1.4: Custom search-by-property looop

This example can again be reformulated using the STL algorithm *find_if*. *find_if* requires us to pass not only the range to be searched but also a predicate function that takes an element of the list as argument and returns true or false, depending on whether the element fulfills the predicate. This predicate can be passed as simple function or as a functor, which was implemented in listing 1.5 by using *bind*. Again, even though the transformed code does still present itself as very expressive, it also becomes apparent that the use of STL functions exceeds the capabilities of C++ novices.

```
vector<Person> p = getPersons();
auto isTom = bind(equal_to<string>(), "Tom", bind(&Person::getFirstName, _1));
vector<Person>::iterator result = find_if(p.begin(), p.end(), isTom);
if (result != p.end()) {
  cout << *result << endl;
}
```

Listing 1.5: STL *find_if*

Searching a container using a predicate is in fact a very generic way of finding elements and can be applied to almost any criterion based upon which should be searched. It remains to state that using C++0x lambda expressions, predicates can be defined far simpler than done so in listing 1.5, as shown in listing 1.6.

```
auto isTom = [](Person p) { return p.getFirstName() == "Tom"; };
```

Listing 1.6: C++0x predicate

The features relevant to this master thesis that have been introduced by C++0x will be illustrated in section 1.3.

Printing each element of a list

As a last example, a short code excerpt writing all elements of the container to the console will be transformed.

```
for (vector<Person>::size_type i = 0; i < p.size(); ++i) {
  cout << p.at(i) << endl;
}
```

Listing 1.7: Custom loop to print each element

The semantics of the code in listing 1.7 can be rephrased by a practical reinterpretation of classical iterators called *ostream_iterator*. An *ostream_iterator* represents a wrapper to a given output stream implementing the *iterator* interface. Aforesaid implementation writes all data assigned to the current iterator position out to the referenced output stream. This structure allows us to implement the *"print each element"*-functionality in terms of a *copy* algorithm:

```
copy(p.begin(), p.end(), ostream_iterator<Person> (cout, "\n"));
```

Listing 1.8: Print each element using copy

Applying the transformation from listing 1.7 to listing 1.8 again requires a profound knowledge of the tools available in the STL as well as the expertise to use them. These few examples tried to emphasize how practical and helpful a plug-in implementing these transformations automatically could be. Relieving the programmer of the burden to recognize the *"print each element"*-semantics as a possible stage for *copy* and *ostream_iterator* could ease the use of STL algorithms significantly and widen their acceptance.

1.2. The Standard Template Library (STL)

The Standard Template Library (STL) is a software library and represents a subset of the C++ Standard Library. The following sections provide a short overview of the library in general and the contained sub-libraries *Algorithms*, *Iterators* and *Containers*.

1.2.1. Overview

The STL provides reusable and adaptible implementations of containers, iterators, algorithms and functors. These classes can be used with any built-in or user-defined type that supports the necessary operations (e.g. copy-constructibility). This flexibility is achieved by the use of templates, which provide compile-time polymorphism.

1.2.2. Algorithms library

Provided by the <*algorithm*> header, the programming components implemented in the Algorithms library allow C++ programs to perform algorithmic operations on containers and any other compatible sequence type. Its functions can be categorized in *Non-modifying sequence operations*, *Mutating sequence operations*, *Sorting and related operations* and *C library algorithms.*

Non-modifying sequence operations

Non-modifying sequence operations usually serve to retrieve data from ranges (e.g. *find_if*) or apply a common operation on the elements of a range (e.g. *for_each*, *count*). There exist exactly 13 *non-modifying sequence oeprations* and their effect, return values and complexities can be found in the final C++ committee draft [ISO10b] [ISO10b, p.844]. The most important algorithms in context of this document are listed explicitly in the following:

- *for_each*
  - *signature*:
    template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
  - *requirements*: *Function* must meet the requirements of *MoveConstructible*
  - *effects*: Applies *f* to the result of dereferencing every iterator in the range [*first,last)*, starting from first and proceeding to *last - 1.* [Note: If the type of first satisfies the requirements of a mutable iterator, *f* may apply nonconstant functions through the dereferenced iterator.—end note]
  - *returns*: std::move(f).
  - *complexity*: Applies *f* exactly *last - first* times.
  - *remarks*: If *f* returns a result, the result is ignored.

[ISO10b, p.855]

- *find*

  - *signature*:
    template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last, const T& value);

    template<class InputIterator, class Predicate>
    InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

    template<class InputIterator, class Predicate>
    InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
  - *returns*: The first iterator *i* in the range [*first,last)* for which the following corresponding conditions hold: **i == value*, *pred(*i) != false*, *pred(*i) == false*. Returns *last* if no such iterator is found.
  - *complexity*: At most *last - first* applications of the corresponding predicate.

  [ISO10b, p.855-856]

- *count*

  - *signature*:
    template<class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type count(InputIterator first, InputIterator last, const T& value);
    template<class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type count_if(InputIterator first, InputIterator last, Predicate pred);
  - *effects*: Returns the number of iterators *i* in the range [*first,last)* for which the following corresponding conditions hold: **i == value*, *pred(*i) != false.*
  - *complexity*: Exactly *last - first* applications of the corresponding predicate.

  [ISO10b, p.857]

Mutating sequence operations

Mutating sequence operations allow the manipulation of complete ranges of generic data types. The STL holds 13 mutating sequence operations ranging from simple *copy* and *remove* operations to very flexible algorithms such as *transform* [ISO10b, p.861] [ISO10b, p.844].

Sorting and related operations

The last category of the <algorithm> library is represented by *Sorting and related operations*, which perform both sorting and localization tasks on containers. Typical examples are *sort*, which allows sorting a given range, *is_sorted*, which verifies whether a given range is already sorted, and *binary_search*, which localizes an element within a given range, if present. [ISO10b, p.861] [ISO10b, p.868-872].

1.2.3. Containers library

The Containers library provides the user with a set of type-safe container classes that allow a unified form of organizing collections of information. Both sequence and associative containers are included and explained in all their forms in the final C++ committee draft [ISO10b, p.710]. While explaining all available container types at this point would lead beyond the scope of this introduction, only the *std::vector* class is introduced in this section. *std::vector* may be viewed as the default container type for C++ projects and is used frequently throughout this document for examples and explanations.

vector

A vector is random-access sequence container. It supports constant time insert and erase operations at the end of the container (namely *push_back* and *pop_back*). Elements in a vector are stored contiguously, meaning that the following condition holds for all data stored in vectors: $\forall n(0 \leq n < v.size() \rightarrow \&v[n] = \&v[0] + n)$. The most important and defining vector operations are:

- void push_back
  (const T &item) Inserts a new item at the end of the vector.
- void pop_back
  Removes the last item of the vector.
- T &operator [](size_type index)
  Provides direct access to the requested index, without enforcing and boundary control on the provided index. Invalid memory accesses are thus possible.
- T &at(size_type index)
  Provides boundary-controlled access to the requested index, throwing an instance of *std::range_check*, should the provided index exceed the vector's boundaries.

1.2.4. Benefits and liabilities

Most C++ experts embrace the use of the STL as a basis for every prospective C++ programmer [Mey01, Baj01, Lov10]. In constrast of this overwhelming acceptance by the language's developers and experts, there also exist strong viewpoints against the use of the STL at all [Wil05]. Therefore, this section tries to provide a neutral view on a limited selection of both benefits and liabilities within the use of STL constructs.

Benefits

- Standardized and excessively tested code
  STL algorithms and class templates have been tested and evaluated in various environments and programming situations. Few software libraries can thus pretend to fulfill their specifications as exactly and precise as the STL does.

- Interoperability
  Programs relying on the same template-based constructs are able to exchange data simpler and in a more type-safe manner than if they all used different container implementations.

- Development effort save
  Using STL constructs save software developers the effort of implementing these functionalities themselves. The STL is available in all C++ environments and thus introduces no explicit library dependencies to satisfy.

Liabilities

- Complexity
  Programmers unfamiliar with C++ find some design aspects of the STL bewildering, e.g. the lack of a common super class for all container types. Furthermore, even trivial tasks such as iterating over each element of a container using *for_each* require at least basic understanding of templates, iterators and functors. This provides beginners with a relatively steep learning curve.

- Genericity
  Instead of being implemented as member functions, most algorithms are implemented as template-based free functions, applicable to a vast variety of types. While this represents a very efficient design aspect, it also requires developers to e.g. understand an std::string as a sequence of characters and calling *sort(str.begin(), str.end())* instead of *str.sort()* [Wil05].

1.3. C++ and C++0x

C++ and especially C++0x provide some concepts and features essential to the usage of STL algorithms that developers from other areas may not be used to. The features used throughout the rest of this document are thus explained in the following of this section.

1.3.1. Argument binding

Section 1.2 introduced the possibilities and functions the STL provides to C++ programmers. One of its features, which enjoys the main focus of this master thesis, are the included algorithm implementations. Almost all STL algorithms consist of one or more iterator ranges to be processed and a function or predicate pointer to be applied to the traversed items. This pattern, however, can in some situations present itself as quite restrictive, as shown in listing 1.9.

```
void registerPerson(const Registrar &, const Person &);
// ...
vector<Person> v = getPersons();
Registrar registrar;
for_each(v.begin(), v.end(), registerPerson);
```

Listing 1.9: for_each and functions with more than one argument

The code from listing 1.9 will not compile. Given the signature of *register*, we must provide a corresponding registrar for each person. However, for_each is limited to functions taking only a single argument of the type contained in the container (in this example *Person*). To circumvent this limitation, we can replace the function pointer by a C++ functor class, as listing 1.10 illustrates.

```
void registerPerson(const Registrar &, const Person &);
// ...
vector<Person> v = getPersons();
Registrar registrar;
class doRegister {
private:
  const Registrar &r;
public:
  doRegister(const Registrar &r) : r(r) {}
  void operator()(const Person &p) {
    registerPerson(r, p);
  }
};
for_each(v.begin(), v.end(), doRegister(r));
```

Listing 1.10: for_each and functor

This process of creating a separate functor class that holds the second argument as a member variable is called *argument binding*. Since this pattern provides a remarkable amount of boilerplate code, argument binding has been facilitated by the introducation of the template-based *bind* functor.

```
void registerPerson(const Registrar &, const Person &);
// ...
vector<Person> v = getPersons();
Registrar registrar;
for_each(v.begin(), v.end(), bind(registerPerson, registrar, _1));
```

Listing 1.11: for_each and bind

Listing 1.11 illustrates the usage of the *bind* environment. The functor's constructor takes the respective function (*registerPerson*), the arguments to bind (*registrar*) and the argument placeholders remaining for the function call. In 1.11, only the placeholder for the first argument (*_1*) is used. The result is a functor taking one argument and calling *registerPerson* with *registrar* and the given argument [Mad05].

1.3.2. Lambdas

In standard C++ and particularly when using STL algorithms such as *for_each* or *sort*, developers often need to instantiate functor objects implementing a certain logic. Creating explicit classes for these functors is sometimes remunerative, if they are of general use and referenced multiple times throughout the project. In many other situations, however, these objects represent only one particular instance of code within one function, and developers would thus prefer to define them as near as possible to their actual usage. Lambda expressions have been introduced to C++0x for this very reason [Mic10, p.7-10]. Lambda functions are defined as shown in listing 1.12.

```
[](int x, int y) { return x + y; }
```

Listing 1.12: Simple lambda expression

The implicit return type of the function in listing 1.12 is *decltype(x + y)* (see 1.3.3). This return type can be stated explicitly using the *trailing-return-type* syntax.

```
[](int x, int y) -> int { return x + y; }
```

Listing 1.13: Simple lambda expression with trailing return type

Furthermore, lambda functions are able to reference identifiers declared outside the lambda body. To do so, *closures* defined between square brackets in the declaration of the lambda are used. Listing 1.14 explains their semantics.

```
[]       //no variables captured. Using one will result in a compilation error.
[x, &y]  //x captured by value, y captured by reference
[&]      //any external variable is implicitly captured by reference
[=]      //any external variable is implicitly captured by value
[&, x]   //x explicitly captured by value. Others captured by reference
[=, &z]  //z explicitly captured by reference. Others  captured by value
```

Listing 1.14: Lambda closures semantics

1.3.3. Type inference

In order to use a variable in standard C++, its type must be explicitly specified. In template-based environments, however, identifying these specific types is sometimes not trivial or even possible, as listing 1.15 demonstrates.

```
template<class T>
void someGenericAlgorithm(const T &t) {
    T::iterator it = t.begin();
    TYPE value = *it;  // What type to use here?
}
```

Listing 1.15: Difficult explicit type specification

For situations like these, the two keywords *decltype* and *auto* provide relief. Using *decltype*, the programmer is able to deduce an expressions type at compile-time and use it in one's own declarations.

```
template<class T>
void someGenericAlgorithm(const T &t) {
    T::iterator it = t.begin();
    decltype(*it) value = *it;
}
```

Listing 1.16: Decltype example

The keyword *auto* provides a very similar functionality, but can only be used in a declaration with associated initializer, since that initializer's type is used for the declaration. This behavior is illustrated in listing 1.17, where the declarations of *value1* and *value2* are equivalent.

```
template<class T>
void someGenericAlgorithm(const T &t) {
    T::iterator it = t.begin();
    decltype(*it) value1 = *it;
    auto value2 = *it;
}
```

Listing 1.17: Auto vs. decltype example

1.4. Natural limits of code analysis

The halting problem, one of the most renowned theorems of computer science, proofs that a computer program in general cannot be analyzed against even the most simple properties (e.g. "halts" or "does not halt") [Sip06, p.173-181]. Whether or not a loop implements an STL algorithm, which is the main topic of this thesis, also represents one such unprovable property. This implies that the problem approached during this master thesis cannot be fully solved. The following sections are dedicated to proving this statement by applying the matter of loop transformation to the halting problem.

1.4.1. Assumptions

In order to transform a loop to an STL algorithm, it is necessary to identify the range traversed by the loop. Loops that do not process an explicit range can be extended by a virtual one. Listings 1.18 and 1.19 illustrate such an extension by a virtual range.

```
while (true) {
    doSomething();
}
```

Listing 1.18: Loop without explicit range

```
int i = 0;
while (true) {
    doSomething();
    ++i;
}
```

Listing 1.19: Same loop extended by a virtual range

This adaption can be applied without loss of generality. In order to transform the loop body to a lambda expression, *break*, *return* and other range-affecting statements would also need to be expressed by the traversed range, as listings 1.20 and 1.21 illustrate.

```
for (int i = 0; i < 10; ++i) {
    if (i == 7) {
        break;
    }
    cout << i << endl;
}
```

Listing 1.20: Loop with range-affecting statements

```
int nums[] = { 0, 1, 2, 3, 4, 5, 6 };  // range resulting due to "break;"
for_each(nums, nums + 7, [](int x) {
    cout << x << endl;
});
```

Listing 1.21: for_each call resulting from range-affecting statements

Above two measures guarantee that:

- there are no (range-affecting) statements in the loop body that forbid a transformation to a lambda expression
- there is a valid range to be processed by a *for_each* call

Based on these two conditions, every *for* loop can be transformed to a *for_each* call as illustrated by figures 1.22 and 1.23.

```
for (init();condition();step()) {
    body();
}
```

Listing 1.22: General for loop

```
init();
for_each(begin, end, [](T x) {
    condition();
    body();
    step();
});
```

Listing 1.23: General for_each call

The question is now whether such a transformed form can always be found. Its answer strictly depends on the question of whether the range processed by a loop can always be properly determined.

1.4.2. Proof of non-existence of a general algorithm

We assume it would be possible to always automatically deduce the (possibly virtual) range of a loop. This would directly allow to disambiguate infinite loops from finite loops. Every loop having a range of the form $[x, \infty)$ or $[x, -\infty)$ would in that case represent an infinite loop. The possibility to identify halting loops from not halting loops, however, would solve the halting problem.

It is thus not possible to automatically identify the (virtual) range of every loop and thus automatically transforming every compatible loop to a matching STL function call is impossible.

1.4.3. Implications for this project

The theoretical impossibility shown in section 1.4.2 should advise us not to focus on a fully-automated transformation of every possible loop construct, since that would be infeasible. Instead, the idea behind this master thesis is creating a semi-automatic system to facilitate these transformations for the programmer. Best effort transformation systems that may need some manual user interaction, and analysis algorithms that recognize commonly known patterns of transformable loops can still provide a tremendous benefit to C++ developers. The notions from section 1.4.2 thus in no way, shape or form discourage the creation of a loop-to-STL transformation plug-in. They merely show how to do it and which features and functionalities to focus on.

2. Analysis

This chapter provides a detailed dissection of this master thesis' different motivations and circumstances. It first states a summary of general conditions and properties one has to face when dealing with loops in the C++ language. Afterwards, a use-of-potential analysis describes which loop categories and relative algorithms are most worthwhile to be covered by this project.

2.1. Properties of loops in C++

Before exploring the various possibilities of how C++ loops can be analyzed and transformed, let first be explained what syntactical and semantical elements these statements are allowed to bear due to the C++ language specification [ISO10a]. This will lead to some strict implications as to how far an analysis module can or should be implemented.

2.1.1. Syntactical properties

The C++ syntax definition in the format of a Backus-Naur-Form (BNF) provides a solid overview of what kind of structures and nodes are expected to be found in the whereabouts of a for loop in C++. We will thus explain an excerpt of the C++ BNF definition to describe what a potential C++ loop analysis system at least needs to be able to cope with in order to at least implement some amount of genericity.

While

The following BNF grammar describes how *while* statements are defined in C++0x [Mar10]. Please note that Terminals are written in bold letters. Furthermore, the used grammar dates to 2nd of July, 2010. Since C++0x still remains a working draft up to this point, changes and divergencies may still occurr.

while $\rightarrow$
- **while (** condition **)** statement

condition $\rightarrow$
- expression |
- attribute-specifier$_{opt}$ type-specifier-seq declarator = initializer-clause |
- attribute-specifier$_{opt}$ type-specifier-seq declarator braced-init-list

expression $\rightarrow$
- assignment-expression |
- expression **,** assignment-expression

assignment-expression $\rightarrow$
- conditional-expression |
- logical-or-expression assignment-operator initializer-clause |
- throw-expression

// all possible types of expressions to follow

statement $\rightarrow$
- labeled-statement |
- attribute-specifier$_{opt}$ expression-statement |
- attribute-specifier$_{opt}$ compound-statement |
- attribute-specifier$_{opt}$ selection-statement |
- attribute-specifier$_{opt}$ iteration-statement |
- attribute-specifier$_{opt}$ jump-statement |
- declaration-statement |
- attribute-specifier$_{opt}$ try-block

For

Using equally top-level BNF elements, the *for* loop specification presents itself as follows:

for $\rightarrow$
- **for (** for-init-statement condition$_{opt}$ **;** expression$_{opt}$ **)** statement

for-init-statement $\rightarrow$
- expression-statement |
- simple-declaration

Consequences

Both *for* and *while* loops allow a *statement* within their bodies, which includes any possible C++ statement and equivalents to an arbitrarily complicated code block, as section 2.1.2 illustrates. For a code analysis system, it thus only matters what kind of statement is present - and whether it is simple enough to be replaced by one or more calls to *bind* or whether a lambda functor is necessary. Furthermore, they both use a *condition* within their header brackets - mandatory for *while* and optional in *for*. However, in C++, each and every *expression* may serve as a condition, as indicated in the BNF definition.

The other two elements of the classic *for* loop are defined as *for-init-statement* and - once more - *expression.* Taking into account that *for-init-statement* may be either of *expression-statement* or *simple declaration*, this leaves the programmer with almost any possible freedom on what to include into a loop statement in C++. There exists a slight restriction inherent to expressions that statements are not confined to, which is explained in section 2.1.2. The influence of this finding on this project is elaborated further in 2.1.3.

2.1.2. Semantic properties

One important thing to realize before analyzing the predication of section 2.1.1 is the difference between a *statement* and an *expression.* The Microsoft Developer Network (MSDN) library defines these terms as follows [Cor10a, Cor10b]:
"Expressions are sequences of operators and operands that are used for one or more of these purposes:

- *Computing a value from the operands.*
- *Designating objects or functions.*
- *Generating "side effects." (Side effects are any actions other than the evaluation of the expression — for example, modifying the value of an object.)*

C++ statements are the program elements that control how and in what order objects are manipulated. This section includes:
[. . .]

- *Categories of Statements*
  - *Null statements. These statements can be provided where a statement is required by the C++ syntax but where no action is to be taken.*
  - *Compound statements. These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever a single statement may be used.*
  - *Selection statements. These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.*
  - *Iteration statements. These statements provide for repeated execution of a block of code until a specified termination criterion is met.*
  - *Jump statements. These statements either transfer control immediately to another location in the function or return control from the function.*
  - *Declaration statements. Declarations introduce a name into a program. (Declarations provides more detailed information about declarations.)"*

The list of statement types is illustrative, but not complete. The very important type *Expression statement* is missing. This type also forms the very reason why expressions may be considered more "powerful" than statements - a statement can consist of a list of many expression, imposing semantic order and control on them. The C++ final committee draft [ISO10b] confirms this description:

"An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects." [ISO10b, p.83]

"Except as indicated, statements are executed in sequence. [...] Iteration statements specify looping." [ISO10b, p.125-128]

To summarize above implications, let again be said that *statements* can incorporate any number and type of loops, which deems complete analysis impossible, as explained in the following of this section. *Expressions*, on the other hand, may be arbitrarily complex and can even hold conditional assignments. Loops, however, may not be incorporated in an *expression*. Unfortunately, this does not mean that they can be analyzed any further than statements, as listing 2.1 illustrates:

```cpp
function<int(int)> fibonacci;
(fibonacci = [&](int z) {
  if (z >= 2) {
    return fibonacci(z - 1) + fibonacci(z - 2);
  }
  if (z == 1) {
    return 1;
  }
  return 0;
})(10);
```

Listing 2.1: Recursive expression

Listing 2.1 shows that, even without previously defined functions, *expressions* can implement recursive calls, which provides equally much functionality as loop statements do [HS08]. For this project's analysis module, the following facts must hence always remain considered:

- Loop bodies can represent complete programs themselves ("turing-complete systems", [Sip06, p.141-143]). Accordingly, they can not be automatically analyzed against properties of their implemented logic (e.g. "implements a find_if-algorithm", see also "The halting problem" [Sip06, p.173-181]). Even though a fitted analysis module can perform very well and recognize most practical patterns of today's C++ code, there may always exist even trivial cases that the module fails to interpret.

- Using *function call expressions* or, as listing 2.1 describes, using functor constructions, *expressions* provide the same functionality as *statements* do. The additional constraints they are bound to (see section 2.1.1) make them no more analyzable than *statements*.

2.1.3. Conclusion

Sections 2.1.1 and 2.1.2 provide the basic knowledge to comprehend what a prospective analysis module can face when analyzing a *while* or *for* loop. *While* statements must be analyzed against the following questions:

- Is the *condition* expression an iteration step or based upon an iteration step within the body?
- Can the body be expressed as a (possibly composed) *bind* expression or is a lambda functor necessary?

The *for* analysis consists of rather similar ideas:

- Is the *for-init-statement* something more sophisticated than a *simple declaration* or an *assignment expression* operation?
- Is the iteration *expression* implementable by means of an iterator?
- Can the iteration *condition* be expressed in terms of an iterator comparison?
- Can the body be expressed as a (possibly composed) *bind* expression or is a lambda functor necessary?

Depending on the answers to these questions, the transformation algorithms can apply the following actions to achieve STL compatibility:

- Move the *for-init-statement* before the STL algorithm call.
- Extract the *simple declaration* or *assignment expression* r-value and pass it as the first iterator to the algorithm.
- Extract the *condition* argument that is not equal to the iteration variable and pass it as the second iterator to the algorithm.
- Transform non-STL based iteration structures to STL iterators 2.4.1.

Above actions present of course only a very generic overview of the refactoring actions possible (and necessary) for this project. Actual refactorings will, however, presumably not be implementable using generic aspects, since genericity usually provides no added expressiveness, as the *for_iterator* example in listing 2.14 indicates. Instead, the analysis and transformation algorithms will have to cater specific instances of loops in order to transform them to their most expressive STL equivalent.

2.2. Use-of-potential analysis

As a matter of practice, this project faces from its very beginning two limiting factors. One of them is the number of STL algorithms, that amounts to over 60 functions ranging from binary sorting to permutation generation. The second factor is implied by the fact that loop initializers and their corresponding code blocks may not only contain single variables and single statements, respectively. Instead, both initializers and iterated code block can call arbitrarily complex functions, establishing a Turing-complete system of their own (see also listing 2.2). As chapter 1.4 pointed out, such a system can never truly be completely analyzed by an automated algorithm.

```
AnyType t = 0;
for (initManyVariables(); veryComplexPredicate(); affectManyVariables()) {
  perform(&t);
  simple(&t);
  findIf(&t);
}
```

Listing 2.2: Arbitrarily complex loops

This statement implicates that, no matter how thoroughly our transformation system for a certain STL algorithm may be, there will always exist loop constructions that can not be transformed by it. So e.g. even though a certain loop may implement no more than a simple *find_if*, it can be too complex to analyse and recognize as such. The project is thus bound to focus on a certain set of algorithms and a corresponding set of loop instances that should be analyzed for appropriate transformations. This section strives to identify the algorithms and loop types that, based on the analysis of a sufficiently large code base, are expected to provide the greatest potential in terms of usage frequency and feasibility.

2.2.1. Considered categories of transformations

At this point, a summary of the different types of transformations taken into account during this use-of-potential analysis is stated.

Find and find_if

Loops implementing the functionality of *find* and in this context also *find_if* often exhibit the following properties:

- Search range (index- or iterator-based)
- Predicate
- Result

During this analysis, loops presenting themselves similar in one or more ways to the following constructs are thus considered "*find*/*find_if* candidates":

```
for (int i = BEGIN; i < NUM_ELEMENTS; ++i) {
  if (CONDITION(elements[i])) {
    RESULT = elements[i];
    break;
  }
}
```

Listing 2.3: *Find*/*find_if* candidates

for_each

In combination with C++0x lambda functors, *for_each* may be the most flexible of all STL algorithms. Almost any *for* loop, even index-based ones, can be mapped to a *for_each* call with corresponding lambda functor. To apply index-based loops to this pattern as well, a transformation to iterators is necessary, as explained in section 2.4.1. Preferred candidates to a *for_each* transformation are loops having only a single statement in their bodies. Using lambda functors, however, virtually any for loop can be transformed. Listings 2.4 through 2.6 illustrate these two cases and their characteristics.

```
for (ITERATOR_TYPE it = BEGIN; it != END; ++it) {
  it->FUNCTION();
}
```

Listing 2.4: Simple classic *for* pattern

```
for_each(BEGIN, END, mem_fun_ref(&TYPE::FUNCTION));
```

Listing 2.5: *for_each* corresponding to 2.4

```
for (ITERATOR_TYPE it = BEGIN; it != END; ++it) {
  it->FUNCTION();
  cout << *it << endl;
}
```

Listing 2.6: Extended classic *for* pattern

```
for_each(BEGIN, END, [](TYPE &t) {
  t.FUNCTION();
  cout << t << endl;
});
```

Listing 2.7: *for_each* corresponding to 2.6

count/count_if

Compared to other algorithms such as *for_each* or transform, *count* and *count_if* provides the very narrow, yet often used functionality of counting elements in a range satisfying a certain predicate. During analysis, loops similar in one or multiple properties to the following constructs are possible *count/count_if* candidates:

```
SIZE_TYPE count;
for (ITERATOR_TYPE it = BEGIN; it != END; ++it) {
  if (PREDICATE(*it)) {
    ++count;
  }
}
SIZE_TYPE count = count_if(BEGIN, END, &PREDICATE);
```

Listing 2.8: *Count/count_if* candidates

2.2.2. Code base analysis results

Based upon the characteristics defined in section 2.2.1, a substantial code basis of various different open source projects is searched for these properties. The target of this task is to establish a profound estimate of potential for the respective STL functions in order to assign priorities of implementation based on their frequencies of use and their feasibilities.

Projects

- Blender
  Blender is a cross-platform, free open source 3D content creation suite published under GNU General Public License.
- CGAL
  The Computational Geometry Algorithms Library (CGAL) offers, as its name suggests, algorithms and data structures for various geometric calculations.
- MySQL
  A project which requires no further introduction. MySQL is one of the most popular free databases available.
- Flight gear
  Flight gear is an open source flight simulator, which recently matured to version 2.0.
- Mozilla Firefox
  Currently allocating 30-60% of today's browser market, Mozilla Firefox is one of the world's most frequently used web browser.

Overview

To establish a broad idea of what kinds of loops exist in the examined projects, a course-grained search based on regular expressions has been performed. Since regular expressions provide too few support to identify certain nestings (e.g. "if statement within for loop"), code format conventions of the different projects have been used to augment the results, so the nesting level has been deduced e.g. from the number of leading tabs and spaces.

		Blender	CGAL	MySQL	Flight gear	Firefox
for loops		3153	1265	2926	565	8947
	begin/end	526	322	0	18	79
	0/size	362	36	271	236	1042
while loops		332	312	1617	105	5633
	head	173	30	88	2	386
	body	72	71	225	20	987
print		7	34	0	14	2
for_each		524	153	421	88	1176
count_if		19	6	16	1	94
find_if		284	100	250	42	623

Table 2.1.: Results of regular expression-based analysis

The different expressions that lead to the results in table 2.1 are listed in the following:

- for loops
  Regular expression matching all for statements:

```
^\s*for\s*\([^;]*;[^;]*;[^)]*\)
```

  Examples:

```
for(int i = 0; i < 10; ++i)
for (unsigned int i = 0; i < vec.size(); ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
while(x != y)
```

- for loops (begin/end)
  Expression matching only for loops starting with “begin()” and ending with “end()”.

```
for\s*\([^;]*begin\s*\(\s*\)[^;]*;[^;]*end\s*\(\s*\)[^;]*;[^)]*\)
```

  Examples:

```
for(int i = 0; i < 10; ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
for(int begin = 0; begin != end; ++begin)
while(x != y)
```

- for loops (0/size)
  Expression matching only for loops beginning at “0” and ending at “size()” or “count” respectively.

```
for\s*\([^;]*0[^;]*;[^;]*(size|count)\(\)[^;]*;[^)]*\)
```

Examples:

```
for(int i = 0; i < vec.size(); ++i)
for(val = 0; val < max_size; ++i)
for(int z = 0; z < count; ++z)
for(int i = begin; i < count; ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
while(x != y)
```

- while loops
  Pattern to recognize any while loop, even in combination with "do".

```
while\s*\(((?!\)\r?\n).)*?\)
```

Examples:

```
for(int i = 0; i < vec.size(); ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
while(x != y)
while(true)
while(x != y)
do {
} while(true);
```

- while loops (head-counting)
  Adapted "while loops" pattern, recognizing only while statements with increment or decrement statement in their condition.

```
\s*while\s*\([^\n]*?(\+\+|\-\-)[^\n]*?\)
```

```
while\s*\(((?!\)\r?\n).)*?\)
```

Examples:

```
for(int i = 0; i < vec.size(); ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
while(x != y)
while(x++ != y)
while(--value)
do {
} while(true);
do {
} while(--x >= 22);
```

- while loops (body-counting)
  Adapted "while loops" pattern, recognizing only while statements with nested

increment or decrement statements.

```
(\s*)while\s*\([^\r\n]*\)\s*{?\r?\n?(^\1\s+.*\r?\n){0,2}(^\1\s
    +.*(\+\+|--).*\r?\n)+(^\1\s+.*\r?\n){0,2}
```

Examples:

```
for(int i = 0; i < vec.size(); ++i)
for(;;)
for(vector::<int> it = vec.begin(); it != vec.end(); ++it)
while(x != y) {
        ++x;
}
while(x++ != y) {
        int x = z + --y;
}
while(x++ != y) {
        cout << x << endl;
}
do {
        ++x;
} while(x >= 22);
```

- print
  This specific pattern searches for loops printing complete collections to an output stream, which can be achieved very expressively by using *copy* and *ostream_iterator*s.

```
(\s*)for\s*\(((?!\1[^s]).)*\)\s*{?\r?\n\1\s+.*(cout|clog|cerr|os)\s
    +<<.*\r?\n\1([^\s]+|\r?\n)
```

Examples:

```
for(int i = 0; i < vec.size(); ++i) {
        cout << vec.at(i) << endl;
}
for(;;) {
        clog << "Hello World!" << endl;
}
for(vector::<int> it = vec.begin(); it != vec.end(); ++it) {
        os << *it << endl;
}
while(x != y) {
        clog << random() << endl;
}
for (;;) {
        cout << x << endl;
}
for (;;) {
        cout << x << endl;
        cout << somethingElse() << endl;
}
```

- for_each
  Apart from the various *for* loops processed by the previous expressions, this pattern focuses on trivial *for* loops containing only one line, thus rendering themselves perfectly suited for a *for_each* and *bind* construct.

```
(\s*)for\s*\((((?!\1[^s]).)*\)\s*{?\r?\n\1\s+.*\r?\n\1([^\s]+|\r?\n)
```

Examples:

```
for (unsigned int i = 0; i < vec.size(); ++i) {
        addToList(i);
}
for (vector<Person>::iterator it = list.begin(); it != list.end(); ++it) {
        it->call();
}
for (vector<Person>::iterator it = list.begin(); it != list.end(); ++it) {
        it->call();
        register(*it);
}
```

- count_if
  One of the simpler and yet powerful and broadly usable algorithms is expressed in the following pattern. It is designed slightly more restrictive than it had to be, which should allow compensation for the cases in which the counting within a loop is only an accompanying task to fulfill the actual logic of the code - and extracting the counting from the loop would break aforesaid logic.

```
(\s*)for\s*\([^;]*;[^;]*;[^)]*\)((?!\1[^s]).)*?(\1\s+)if((?!\3[^s]).)
    *?\3\s+.*?(\+\+|--|\+=|-=)[^;]*
```

Examples:

```
for(;;) {
        if(true) {
                ++count;
        }
}
for(;;) {
        ++count;
}
for(;;) {
        if(true) {
                doIt();
        }
}
```

- find_if
  Pattern to recognize an assignment or “break” expression within an “if” node, which itself again is nested within a “for” node.

```
(\s*)for\s*\([^;]*;[^;]*;[^)]*\)((?!\1[^s]).)*?(\1\s+)if((?!\3[^s]).)
    *?\3\s+.*?(=|(break))[^;]*
```

Examples:

```
for(vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        if(isPrime(*it)) {
                result = *it;
```

```
    }
}
for(it = vec.begin(); it != vec.end(); ++it) {
    if(isPrime(*it)) {
        break;
    }
}
for(it = vec.begin(); it != vec.end(); ++it) {
    if(isPrime(*it)) {
        doSomethingElse();
    }
}
for(it = vec.begin(); it != vec.end(); ++it) {
    result = *it;
}
```

2.2.3. Estimated potential

The results of section 2.2.2 provide a certain ambivalence. On the one hand, projects like *Blender* and *CGAL* suggest that there exists at least some room for improvements through STL algorithms, as *for* loops spanning from *begin* to *end* can at least be replaced by *for_each* and a lambda. Even though the regular expressions-based analysis provides anything but guarantee for its results, depending on how reliable we estimate them, it remains safe to say that 15-25% of the loops in Blender and CGAL could be replaced by STL algorithms. However, for every *Blender* and *CGAL* project, there exist examples like *MySQL*, where the STL functions *begin()* and *end()* are only called as much as 51 times, and never in the direct context of a *for* loop. This indicates a very low adoption of the STL in general, and abominations like a name space called "mySTL" support this suspicion.

Summarizing both the motivating and disenchanting results of this analysis, a cautious estimation of 10% of all loops in today's C++ code can be replaced by appropriate STL algorithms. Applying this idea to the processed example projects, incorporating a total of no less than 24855 loops, and let us assume that an STL algorithm can on average replace three lines of code by one, this amounts to roughly 5000 lines of code saved by the use of STL algorithms. While this analysis provides a brief and meaningful impression of the possible benefits of this project, it is of course despite its various sources neither representative nor complete. But it states two facts very clearly: There is room for STL algorithms in today's projects, and they can reduce the amount of code in a non-trivial manner.

2.3. Analysis techniques

Section 1.4 pointed out the various difficulties one may encounter when trying to automatically interpret and manipulate the semantics of existing programs. While there exist no all-embracing solutions against this backdrop, there are two promising approaches, namely "Code pattern recognition" and "Natural metadata interpretation", that have been examined during this master thesis and will be explained in the following of this section.

2.3.1. Pattern-based analysis

When refactoring manual STL algorithm implementations to effective STL function calls, there exist certain patterns in these implementations that tend to repeat themselves. One example may be that, when programmers manually search through a container instead of using *find* or *find_if*, it is highly probable that they create slightly altered forms of the code lines illustrated in listings 2.9 through 2.11.

```
int data[] = { 0, 1, 2, 3, 4, 5 };
for (it = data; it != data + 6; ++it) {
  if (*it == 3) {
    break;
  }
}
if (it != data + 6) {  // check whether an element has been found
  // ...
}
```

Listing 2.9: *find_if break* pattern

```
const int *findIfReturn(const int *begin, const int *end, int value) {
  for (const int *it = begin; it != end; ++it) {
    if (*it == value) {
      return it;
    }
  }
  return end;
}
```

Listing 2.10: *find_if return* pattern

```
unsigned char line[] = { 'a', 's', 'd', 'f' };
const unsigned char *c = line + 4;
for (c = line; c != line + 4 && *c == 's'; ++c) {
}
```

Listing 2.11: Embedded *find_if* pattern

Section 2.2 evaluated a series of open source projects against a set of such patterns and showed that analysis techniques based on these *code patterns* may indeed recognize a large portions of potential STL function call candidates. Unfortunately, this analysis did also point out that common lexical regular expressions are not sufficient for a reliable recognition of these patterns. Instead, at least the additional structural information of the code's abstract syntax tree (AST) should be exploited, leading to the idea of *tree patterns* becoming a necessity. The following subsection will thus address the tree pattern matching problem.

Pattern matching in trees

Hoffmann and O'Donnell illustrated in their research paper *"Pattern Matching in Trees"*[HO82] that tree pattern matching is in its implementation complexity comparable to lexical pattern matching. They even provided a procedure to map the tree pattern matching problem to the lexical pattern matching problem in general. Lu

Wuu, Lu and Yang followed up on these findings in their work *"A Simple Tree Pattern-Matching Algorithm"* [WtLY00] and exemplified that these algorithms' computational complexity does not exceed the limits of today's commonly available performance standards and may indeed find application in actual software projects.

Both papers address, however, only the recognition of exact subtrees within a tree and omit the possibility of metadata augmentation for more elaborate pattern specifications (such as e.g. repetitions in lexical regular expressions). The natural language processor Tregex (`http://nlp.stanford.edu/software/tregex.shtml`) provides a good overview of which metadata relationships are helpful in creating expressive tree regular expressions [LA05, p.2]. Of the various possibilities implemented there, the following relationships are estimated critical for successful pattern matching in the abstract syntax tree:

- Domination
  One tree or node dominates another one, i.e. represents an ancestor of the latter. Immediate domination represents a parent-child relationship, general domination allows for an indefinite number of intermediate nodes. Figure 2.1 illustrates this relationship.

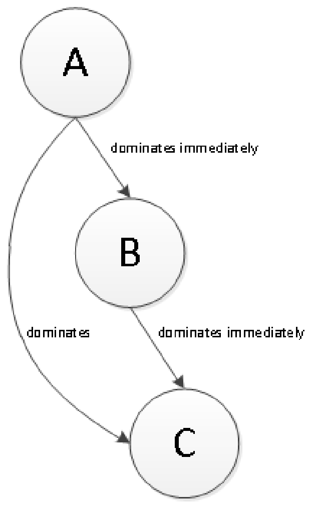


Figure 2.1.: Domination and immediate domination

- Sibship
  Subnodes of the same parent are identified as right or left siblings of each other. Immediate siblings allow furthermore no intermediate nodes between them. The relationship is displayed in detail in figure 2.2.

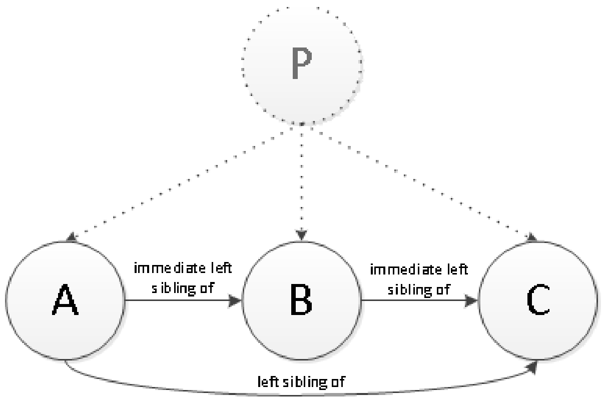


Figure 2.2.: Sibship and immediate sibship

- Logical connections
  A subtree can be required to satisfy multiple patterns at the same time or to satisfy any one of a set of arbitrary patterns. One example of such a pattern combination is shown in figure 2.3.

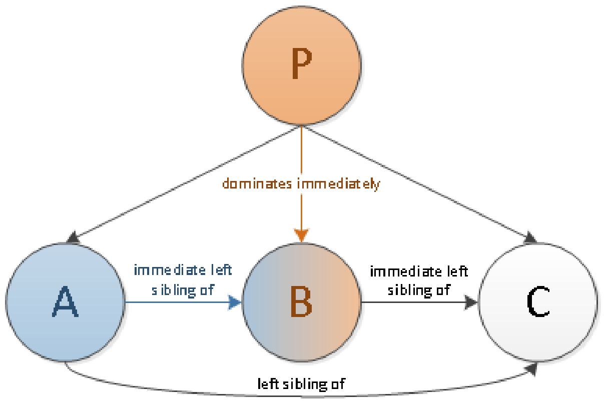


Figure 2.3.: B is immediately dominated by P **AND** is the immediate right sibling of A

The requirements and possibilities identified in this chapter find their respective implementation described in section 3.3.1.

2.3.2. Natural metadata interpretation

Natural metadata interpretation represents an approach to use meta information apart from the actual source code, that does not have to be inserted explicitly for the sake of analysis, but may be found sporadically in any code (thus "natural" metadata, a neologism introduced during this thesis and used as such during the rest of this thesis). To explain the benefits and occurrences of natural metadata, introducing an example of non-natural metadata first may be helpful. For the upcoming release of version 7 of the Java programming language Ali, Correa, Ernst and Papi filed a Java specification request (JSR) named "The Checker Framework: Custom pluggable types for Java"[ACEP10]. The paper proposed the introduction of independently controllable compiler extensions that would specifically target the semantics of a set of newly introduced annotations. Examples of these annotations include:

- Nullness annotations
  @Nullable indicates that a variable may indeed reach a value of *null* and should be checked for such before dereference. @NonNull indicates that *null* is excluded from the domain of this variable and it may be safely accessed. The nullness checker will cause a compiler error when detecting the assignment of a (possibly) *null* value to the variable.

- Mutability annotations
  @ReadOnly types provide only non-modifying access. A reference ammended by this annotation may not be used to modify its referent. @Mutable marks the very opposite case. Should the programmer use modifying methods (or methods not marked as non-modifying) of a @ReadOnly object, the immutability checker will issue corresponding warnings and errors.

- Tainting annotations
  The tainting checker prevents trust errors by marking values as tainted or untrusted. Tainted values stem from arbitrary, possibly malicious sources, such as user input or unvalidated data. These values must be sanitized or "untainted" before further using them. One example of such a tainted situation might be a user input that should be searched for in the database. Before querying for the value, however, it must be untainted, by e.g. escaping all parts of the input that could be interpreted as SQL commands, thus effectively prohibiting SQL injection.

One of the core problems of this master thesis is the fact that turing-complete systems cannot be automatically analyzed for their semantics. The checkers framework circumvents this issue by introducing metadata outside of the scope of the actual turing-system: the annotations. The very same approach could be chosen for the goals of this thesis to analyze loops for their equivalence to STL algorithms, as listing 2.12 illustrates.

```
int data[] = { 0, 1, 2, 3, 4, 5 };
const int *it = data + 6;
// @Find
```

```
for (it = data; it != data + 6; ++it) {
  if (*it == 3) {
    break;
  }
}
if (it != data + 6) {
  // ...
}
```

Listing 2.12: Annotations for simplified analysis

In contrary to anything said about the complexity of semantic analysis in the course of this document, identifying the loop in listing 2.12 as an implementation of *find_if* would be a trivial task. For the scope of this project, however, the presence of such specific, non-natural metadata cannot be assumed. What would be imaginable, however, is a slightly altered scenario shown in listing 2.13.

```
int data[] = { 0, 1, 2, 3, 4, 5 };
// The default value, if the desired value was not found
const int *it = data + 6;
// Searches the container for the requested value
for (it = data; it != data + 6; ++it) {
  if (*it == 3) {
    break;
  }
}
if (it != data + 6) {
  // ...
}
```

Listing 2.13: Code comments as metadata source

Looking for keywords like *found* or *search* in the context of *find_if*, as an example, code comments could provide a valuable resource for natural metadata describing the semantics of the actual code. This approach, however, was not followed during the progress of this master thesis and may be the topic of future works in this field.

2.4. Transformation techniques

This section explains which principles have been applied programmatically when transforming the various parts of the processed loops. It focuses on general concepts, e.g. “how to replace iterator-based by value-type operations”, rather than the actual implementation of these concepts within the final plug-in.

2.4.1. Index-based access to iterators

One unnegotiable difference between all STL algorithms and plain *for* statements is STL’s restriction to and focus on iterators. The STL mainly targets operations on containers, whereas *for* loops may use arbitrary statements for initialization, iteration step and break condition. That may seem to leave a transformation system with a certain gap of functionality. Yet in theory, an iterator is merely an interface and can be implemented far beyond the enumeration of a list, as listing 2.14 indicates.

```
for (init(); finished(); next()) {
  // ...
}
for_iterator it([]() { init() }, []() { if(finished()) {
        return *this = for_iterator()
    } },
    []() { next() });
for_each (it, for_iterator(), [](for_iterator::value_type it) {
  // ...
});
```

Listing 2.14: Iterator and *for* statement bridging

In practice, of course, the use of such a *for_iterator* is highly questionable, as it provides little to no more expressiveness than a plain *for* statement. The only use it could provide is applying other STL algorithms than *for_each* to a range of numbers - which is usually more easily accomplished using a *counting_iterator* construct, as shown in listing 2.15;

```
int result = -1;
for (int i = 0; i < 99999; ++i) {
  if (satisfiesCertainCondition(i)) {
    result = i;
  }
}
counting_iterator<int> begin(0), end(99999), result;
result = find_if(begin, end, &satisfiesCertainCondition);
```

Listing 2.15: Counting iterator (boost.org)

This project focuses on eliminating custom implementations of STL algorithms. This effort is primarily directed towards iterator-based loops. Transforming index-based accesses to iterator-based accesses, as seen in listing 2.16, is not part of the official task description and thus considered optional.

```
vector<int> v = getData();
for (vector<int>::size_type i = 0; i < vec.size(); ++i) {
  if (lookingFor(vec.at(i))) {
    // ...
  }
}
find_if(vec.begin(), vec.end(), &lookingFor);
for (vector<int>::size_type i = 7; i < 25; ++i) {
  if (lookingFor(vec.at(i))) {
    // ...
  }
}
find_if(vec.begin() + 7, vec.begin() + 25, &lookingFor);
```

Listing 2.16: Index access to iterators

Even for those rather simple examples above, it must be stated that the second transformation is considered highly hazardous and not equivalent to the original, since *vector*'s *at* function would throw a *range_check* exception, whereas the transformed code would result in an illegal memory access in case of a container having less than 25 elements. *Counting_iterator* based solutions to apply STL algorithms to numeric values are again, since not part of the official task description, considered strictly optional in the scope of this project.

The remaining transformations thus will have to fulfill the requirement of being able to link the index value of 0 to *vec.begin()*, index value of *vec.size()* to *vec.end()* and recognise their usage in terms of a vector access using *vector::at* or *vector::operator []* as an equivalent to iterator access. This will require a transformation algorithm to inspect initialization, iteration step and break condition of counting *for* loops and determine whether they can be statically deduced to the corresponding values or methods respectively. Listing 2.17 illustrates a situation in which such a static deduction is bound to fail and a respective transformation thus cannot be applied.

```cpp
vector<int>::size_type begin, end;
cin >> begin;
cin >> end;
for (vector<int>::size_type i = begin; i < end; ++i) {
  if (lookingFor(vec.at(i))) {
    // ...
  }
}
```

Listing 2.17: Example case of impossible static deduction

Let once more be stated that the transformation techniques analyzed in this complete section are not part of the thesis' task description and will consequently only approached if spare time is at hand.

2.4.2. Iterators to value_type

As remarked throughout the entirety of this document, removing iterator-based access by an equivalent value_type based access is a desired improvement of the programming code. This is primarily true beacuse we strive for the code to be as simple and understandable as possible. A short code example in listing 2.18 may explain this issue further.

```cpp
for (int i = 1; i <= 10; ++i) {
  cout << i << endl;
}
for (int i = (11 - 10), j = 99; i < j - 88;) {
  int *p = &i;
  cout << *p << endl;
  i *= 2;
  i += 2;
  i /= 2;
}
```

Listing 2.18: Simplicity vs. semantic power

Both code excerpts in listing 2.18 implement the same logic and both of them represent perfectly legal C++ code. As for the second example, C++ offers the freedom to place also non-trivial expressions as *for-init-statement* and *condition*, and the *iteration step* may very well be handled within the body as well. Furthermore, C++ offers pointer arithmetics that have proven themselves useful in countless instances. The question that remains is in fact the very one that motivated this whole master thesis: Should one use these extended capabilities and semantic ways of expression the language offers if they are not necessary? Introducing pointers to this short code example offers countless new ways of expressing the desired logic - as well as countless new opportunities for mistakes. Simplifying code usually means either omitting overhead syntax or removing unnecessary semantic power. One expressive example of the latter measure are C# LINQ expressions. Listings 2.19 and 2.20 show a for loop that has been replaced by a LINQ statement.

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
IList<int> lowNums = new List<int>();
foreach (int element in numbers)
{
    if (element < 5)
    {
        lowNums.Add(element);
    }
}
```

Listing 2.19: Foreach "lower-than-5" example

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var lowNums =
  from n in numbers
    where n < 5
    select n;
```

Listing 2.20: LINQ "lower-than-5" example

Most programmers agree that the implementation in 2.20 is the more expressive one. That is to some respect a surprising statement, since the *foreach* variant requires less knowledge of extra keywords and can even be understood by novice programmers. Nevertheless, if a logic can be implemented both as *for* loop or *LINQ statement*, the latter option usually presents itself remarkably simpler. This is due to the fact that SQL statements (apart from PL/SQL, CTE and Windowing [Fet09, p.47-53]) and LINQ statements accordingly provide less functionality than the C# programming language itself. Consequently, SQL is also more easily learnable than the full C# language. This reduced functionality, however, bears limitations as well so that there exist iterative logics that cannot be expressed as LINQ query. One such example is given in listing 2.21.

```
IList<int> result = new List<int>();
for (int i = 0; i < numbers.Length / 2; ++i)
{
  result.Add(numbers[i]);
    result.Add(numbers[numbers.Length - i - 1] + numbers[i]);
}
```

Listing 2.21: Loop without LINQ equivalent

Removing semantic power, as explained in listings 2.19 through 2.21, therefore also implies removing capabilities from the code. While the question of whether we should use pointers to implement the counting functionality in listing 2.18 will probably be answered unanimously with “no” by most programmers, the same question can be asked for iterators as well.

```cpp
vector<int> vec = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
  if (it != vec.begin()) {
    clog << (*it - *(it - 1)) << endl;
  }
}
for_each(vec.begin(), vec.end(), [](int i) {
  // How to implement above logic?
});
```

Listing 2.22: Additional iterator semantics

Listing 2.22 illustrates a situation where iterators indeed prove themselves useful and necessary. The rest of this documentation shows a habit of denouncing iterators as tedious and error-prone, since it was written during the effort of searching and replacing the instances of iterator usages where this indeed may apply. However, the additional layer of complexity iterators introduce does serve a purpose - it allows us to access elements depending on the current iterator position. This functionality is used in listing 2.22 and represents a perfectly expressive code example with no overhead semantic. If this position information is not necessary to implement our logic, however, the usage of iterators becomes very similar to the second example in listing 2.18. A prospective analysis module that is to decide whether a certain logic can be implemented by means of a lambda functor or an STL algorithm should thus also consider iterator-specific semantics (such as increasing or decreasing the iterator position within the loop body) and judge based on this information that the usage of iterators may indeed be necessary and cannot be replaced.

3. Implementation

The purpose of this chapter is to provide both a structural as well as a logical description of the actual implementation of the analysis and transformation plug-in developed during this master thesis. The Architecture section (3.2) will thus present the different modules and packages of the program, while the rest of this chapter serves the explanation of the logical behavior and algorithms implemented by this plug-in.

3.1. Overview

Before describing any part of the final implementation in more detail, a general overview of the program and its information flow is shown first. Figure 3.1 provides a comprehensive summary of all involved components.

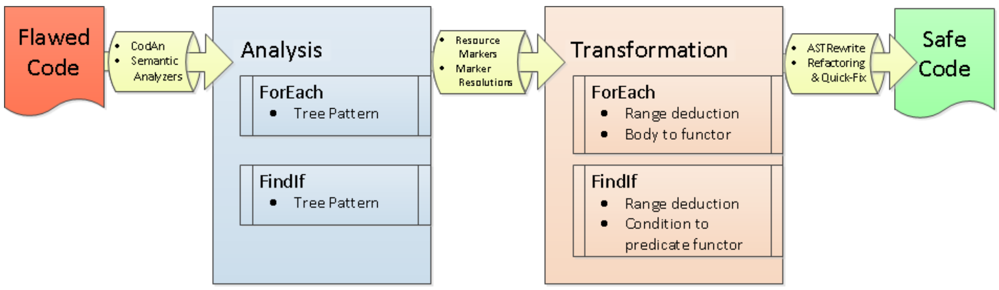
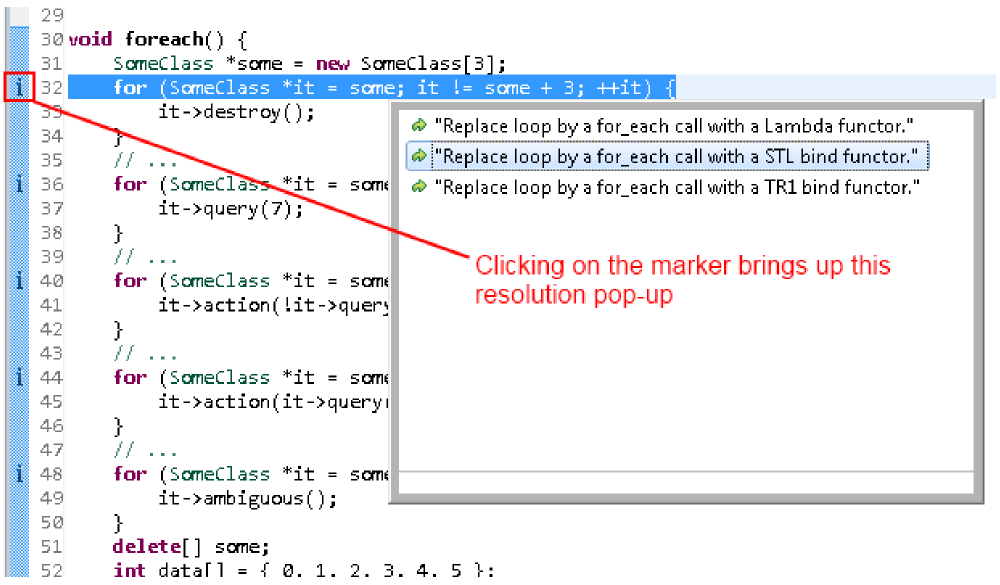


Figure 3.1.: Functionality overview

Generally speaking, the source code in question is first analyzed by the Analysis module and its *Semantic analyzers* (3.2.1). These analyzers verify the code against certain properties and flaws. The two most important properties also displayed in figure 3.1 are “convertible to for_each” or “convertible to find_if” respectively. Any loop satisfying that property is marked by an Eclipse *resource marker*. Resource markers, on the other hand, can be connected by so-called *marker resolutions* to be displayed if a user desires so. Figure 3.2 shows a screenshot of *resource marker* and its relative *marker resolutions* displayed when the programmer clicked on the marker symbol.



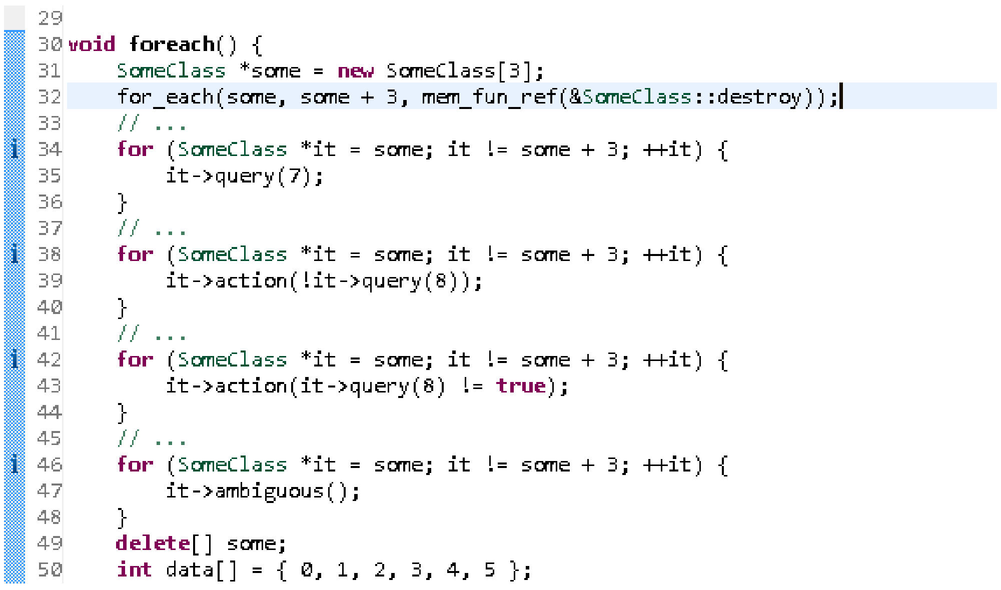
```
void foreach() {
    SomeClass *some = new SomeClass[3];
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->destroy();
    }
    // ...
    for (SomeClass *it = some
        it->query(7);
    }
    // ...
    for (SomeClass *it = some
        it->action(!it->query
    }
    // ...
    for (SomeClass *it = some
        it->action(it->query
    }
    // ...
    for (SomeClass *it = some
        it->ambiguous();
    }
    delete[] some;
    int data[] = { 0, 1, 2, 3, 4, 5 };
```

"Replace loop by a for_each call with a Lambda functor."
"Replace loop by a for_each call with a STL bind functor."
"Replace loop by a for_each call with a TR1 bind functor."

Clicking on the marker brings up this resolution pop-up

Figure 3.2.: Markers and marker resolutions

If selected by a user, these marker resolutions trigger the second major module of the plug-in: The transformation module. Depending on the information attached to the marker itself, the appropriate transformation algorithm converting bodies to functors and iterator values to ranges. Eventually, if the conversion rendered successful, the Eclipse refactoring framework is used to rewrite the source code and replace the respective loop by the newly created *for_each* or *find_if* function call. Figure 3.3 shows the same code snippet after the transformation has been applied.



```
void foreach() {
    SomeClass *some = new SomeClass[3];
    for_each(some, some + 3, mem_fun_ref(&SomeClass::destroy));
    // ...
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->query(7);
    }
    // ...
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->action(!it->query(8));
    }
    // ...
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->action(it->query(8) != true);
    }
    // ...
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->ambiguous();
    }
    delete[] some;
    int data[] = { 0, 1, 2, 3, 4, 5 };
```

Figure 3.3.: Same code snippet after transformation

3.2. Architecture

Describing module by module, this section is dedicated to providing insight in the structural composition and the logical data flows of the implemented plug-in project. The structures and dependencies displayed in section 3.1 as an overview will be covered in the following sections in more detail. The major focus in this section will be architectural properties of the plug-in, such as class diagrams and dependency views. For the actual algorithms used in the analysis and transformation module, please see sections 3.3 and 3.4 respectively.

3.2.1. Analysis

The analysis package is dominated by two major concepts: The tree pattern matching package explained in section 2.3.1 and the semantic analyzers infrastructure. Both concepts will be explained in the forthcoming of this section.

Tree pattern matching package

Based on the requirements defined in section 2.3.1, the actual implementation of the described patterns interface is illustrated in figure 3.4.

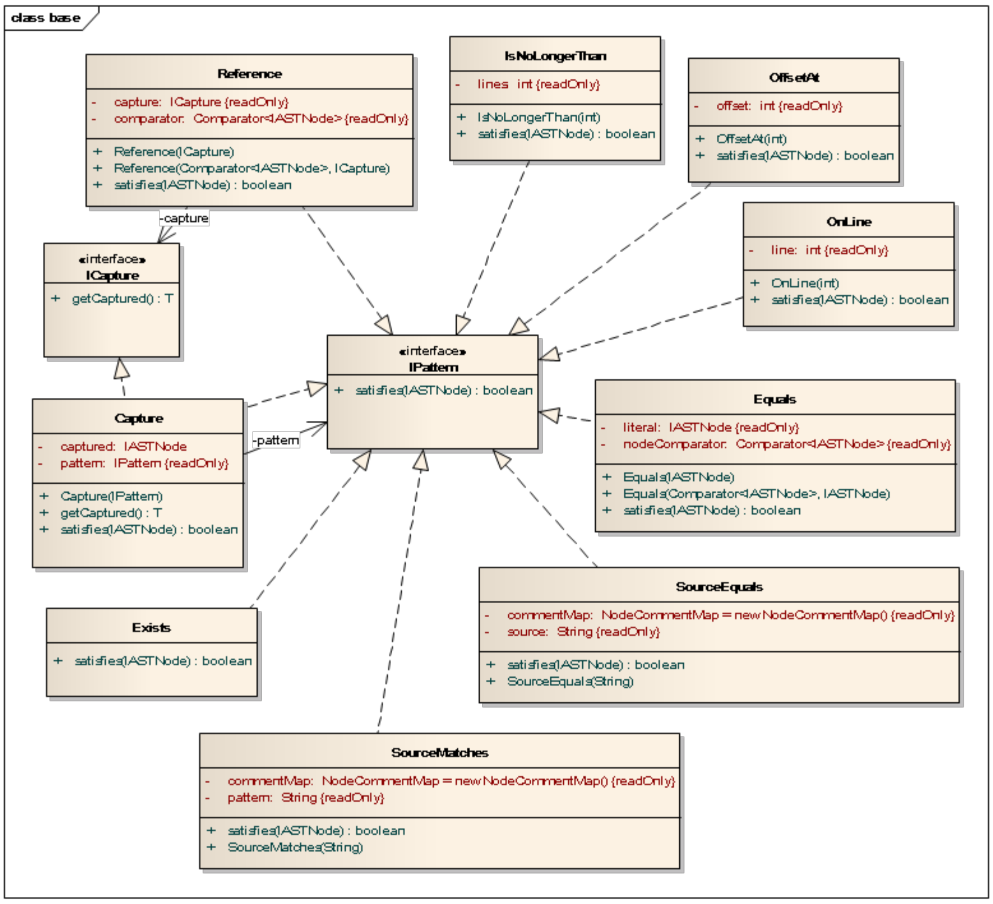

class base
Reference
- capture: ICapture {readOnly}
- comparator: Comparator<IASTNode> {readOnly}
+ Reference(ICapture)
+ Reference(Comparator<IASTNode>, ICapture)
+ satisfies(IASTNode) : boolean
IsNoLongerThan
- lines: int {readOnly}
+ IsNoLongerThan(int)
+ satisfies(IASTNode) : boolean
OffsetAt
- offset: int {readOnly}
+ OffsetAt(int)
+ satisfies(IASTNode) : boolean
OnLine
- line: int {readOnly}
+ OnLine(int)
+ satisfies(IASTNode) : boolean
-capture
«interface»
ICapture
+ getCaptured() : T
«interface»
IPattern
+ satisfies(IASTNode) : boolean
Capture
- captured: IASTNode
- pattern: IPattern {readOnly}
+ Capture(IPattern)
+ getCaptured() : T
+ satisfies(IASTNode) : boolean
-pattern
Equals
- literal: IASTNode {readOnly}
- nodeComparator: Comparator<IASTNode> {readOnly}
+ Equals(IASTNode)
+ Equals(Comparator<IASTNode>, IASTNode)
+ satisfies(IASTNode) : boolean
Exists
+ satisfies(IASTNode) : boolean
SourceEquals
- commentMap: NodeCommentMap = new NodeCommentMap() {readOnly}
- source: String {readOnly}
+ satisfies(IASTNode) : boolean
+ SourceEquals(String)
SourceMatches
- commentMap: NodeCommentMap = new NodeCommentMap() {readOnly}
- pattern: String {readOnly}
+ satisfies(IASTNode) : boolean
+ SourceMatches(String)

Figure 3.4.: Basic patterns package (ch.hsr.ifs.ds8.analysis.patterns.base)

The base package lays the foundation for the various following implementations of the *IPattern* interface. One of these basic concepts are capturing and referencing, which are implemented in this package using the two wrapper classes *Capture* and *Reference*. Any pattern can be wrapped by a *Capture* pattern, matching the same nodes, but storing the matched node for later access. Equally, each capture can be used with *Reference* object, to require a node equal to the previously matched one to appear again.

Listing 3.1 shows how an *IPattern* can be used in the actual Java code.

```
IASTNode node = nodeFactory.newBreakStatement();
IPattern pattern = new IsInstanceOf(IASTBreakStatement.class);
pattern.satisfies(node); // true
```

Listing 3.1: *IPattern* usage example

Every pattern can furthermore be combined using logical connector patterns. The respective package is shown in figure 3.5.

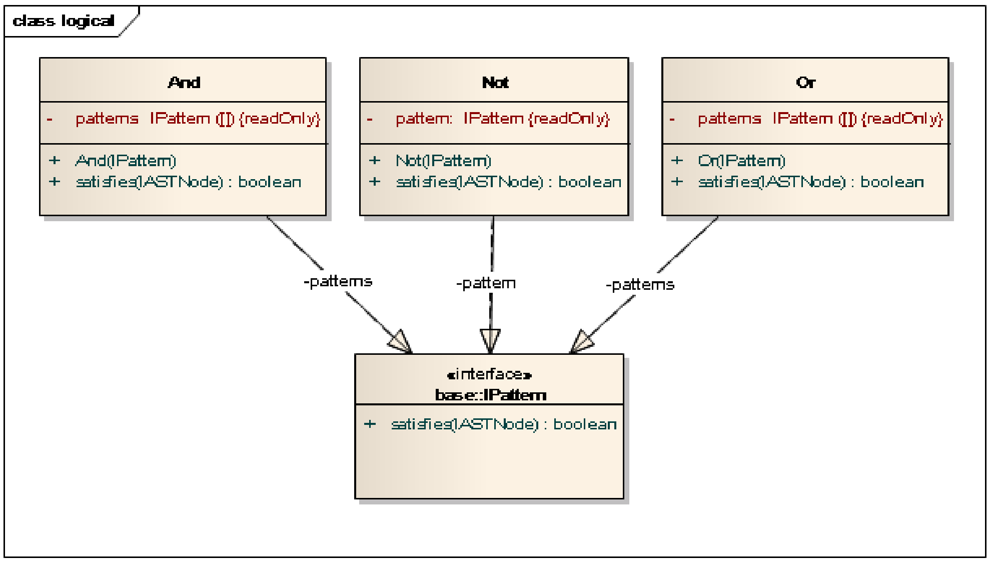

class logical
And
- patterns : IPattern ([]) {readOnly}
+ And(IPattern)
+ satisfies(IASTNode) : boolean
Not
- pattern: IPattern {readOnly}
+ Not(IPattern)
+ satisfies(IASTNode) : boolean
Or
- patterns : IPattern ([]) {readOnly}
+ Or(IPattern)
+ satisfies(IASTNode) : boolean
-patterns
-pattern
-patterns
«interface»
base::IPattern
+ satisfies(IASTNode) : boolean

Figure 3.5.: Logical patterns package (ch.hsr.ifs.ds8.analysis.patterns.logical)

Using these logical combinations, another requirement stated in section 2.3.1 can be fulfilled. These requirements are:

- Ability to express domination
- Ability to express sibship
- Ability to combine pattern elements

By combining the available patterns using these logical connectors, very flexible combinations can be created to identify desired nodes within an abstract syntax tree. Using these tools, a wide variety of semantic patterns have been implemented suitable for various analysis and AST-based search tasks. An excerpt of these patterns is shown in figure 3.6.

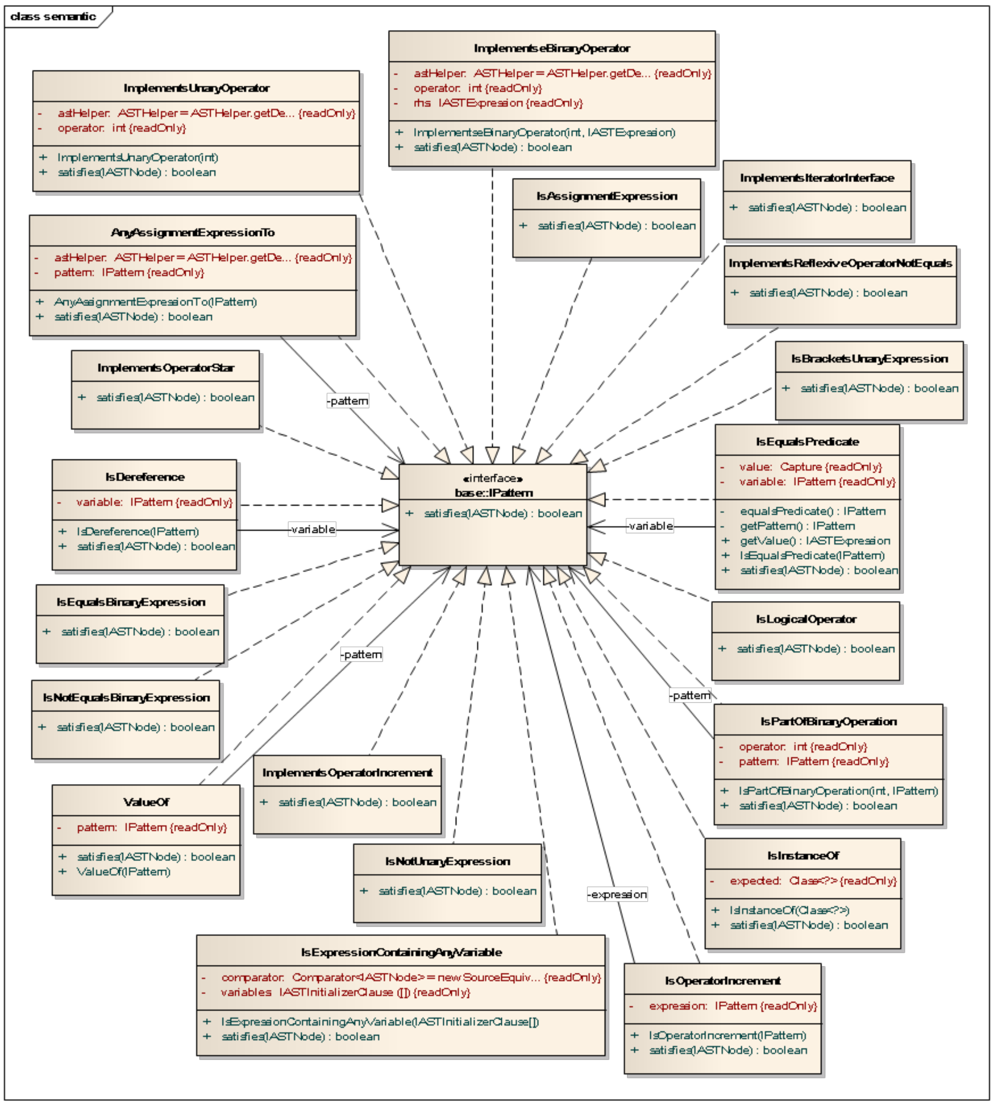

class semantic
ImplementseBinaryOperator
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- operator: int {readOnly}
- rhs: IASTExpression {readOnly}
+ ImplementseBinaryOperator(int, IASTExpression)
+ satisfies(IASTNode) : boolean
ImplementsUnaryOperator
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- operator: int {readOnly}
+ ImplementsUnaryOperator(int)
+ satisfies(IASTNode) : boolean
ImplementsIteratorInterface
+ satisfies(IASTNode) : boolean
IsAssignmentExpression
+ satisfies(IASTNode) : boolean
AnyAssignmentExpressionTo
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- pattern: IPattern {readOnly}
+ AnyAssignmentExpressionTo(IPattern)
+ satisfies(IASTNode) : boolean
ImplementsReflexiveOperatorNotEquals
+ satisfies(IASTNode) : boolean
IsBracketsUnaryExpression
+ satisfies(IASTNode) : boolean
ImplementsOperatorStar
+ satisfies(IASTNode) : boolean
-pattern
«interface»
base::IPattern
+ satisfies(IASTNode) : boolean
IsEqualsPredicate
- value: Capture {readOnly}
- variable: IPattern {readOnly}
- equalsPredicate() : IPattern
- getPattern() : IPattern
+ getValue() : IASTExpression
+ IsEqualsPredicate(IPattern)
+ satisfies(IASTNode) : boolean
IsDereference
- variable: IPattern {readOnly}
+ IsDereference(IPattern)
+ satisfies(IASTNode) : boolean
-variable
-variable
IsEqualsBinaryExpression
+ satisfies(IASTNode) : boolean
IsLogicalOperator
+ satisfies(IASTNode) : boolean
-pattern
IsNotEqualsBinaryExpression
+ satisfies(IASTNode) : boolean
-pattern
IsPartOfBinaryOperation
- operator: int {readOnly}
- pattern: IPattern {readOnly}
+ IsPartOfBinaryOperation(int, IPattern)
+ satisfies(IASTNode) : boolean
ImplementsOperatorIncrement
+ satisfies(IASTNode) : boolean
ValueOf
- pattern: IPattern {readOnly}
+ satisfies(IASTNode) : boolean
+ ValueOf(IPattern)
IsNotUnaryExpression
+ satisfies(IASTNode) : boolean
IsInstanceOf
- expected: Class<?> {readOnly}
+ IsInstanceOf(Class<?>)
+ satisfies(IASTNode) : boolean
-expression
IsExpressionContainingAnyVariable
- comparator: Comparator<IASTNode> = new SourceEquiv... {readOnly}
- variables: IASTInitializerClause ([]) {readOnly}
+ IsExpressionContainingAnyVariable(IASTInitializerClause[])
+ satisfies(IASTNode) : boolean
IsOperatorIncrement
- expression: IPattern {readOnly}
+ IsOperatorIncrement(IPattern)
+ satisfies(IASTNode) : boolean

Figure 3.6.: Semantic patterns package (ch.hsr.ifs.ds8.analysis.patterns.semantic)

An expressive example of how these elements can be used to express an actual pattern recognizing transformable loops is the *find_if* pattern. Displayed in a simplified form, the pattern recognizes all *for* loops that immediately dominate bodies (*compound statements*) holding an *if* statement which itself dominates either a *break* or *return* statement. Figure 3.7 shows this example graphically.

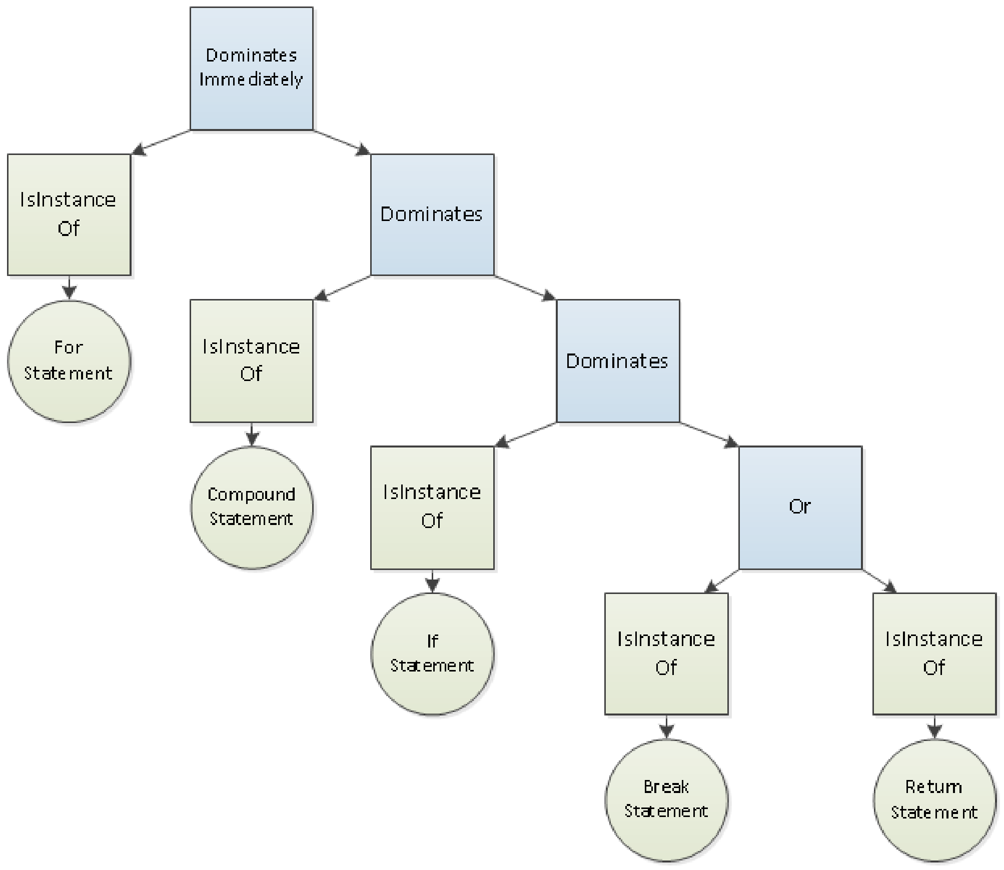


Figure 3.7.: Find_if pattern example

Semantic analyzers

The semantic analyzers represent the core of the analysis module. Using the algorithms described in section 3.3, they verify a given code segment and nodes within the abstract syntax tree offending their topic. As one example, the *ForEachSemanticAnalyzer* would skim a given translation unit for any *for* or *while* loop traversing a determinable range of iterators (section 3.4 explains the respective deduction and analysis algorithms to more detail). Figure 3.8 provides a brief overview of how these analyzers are interconnected.

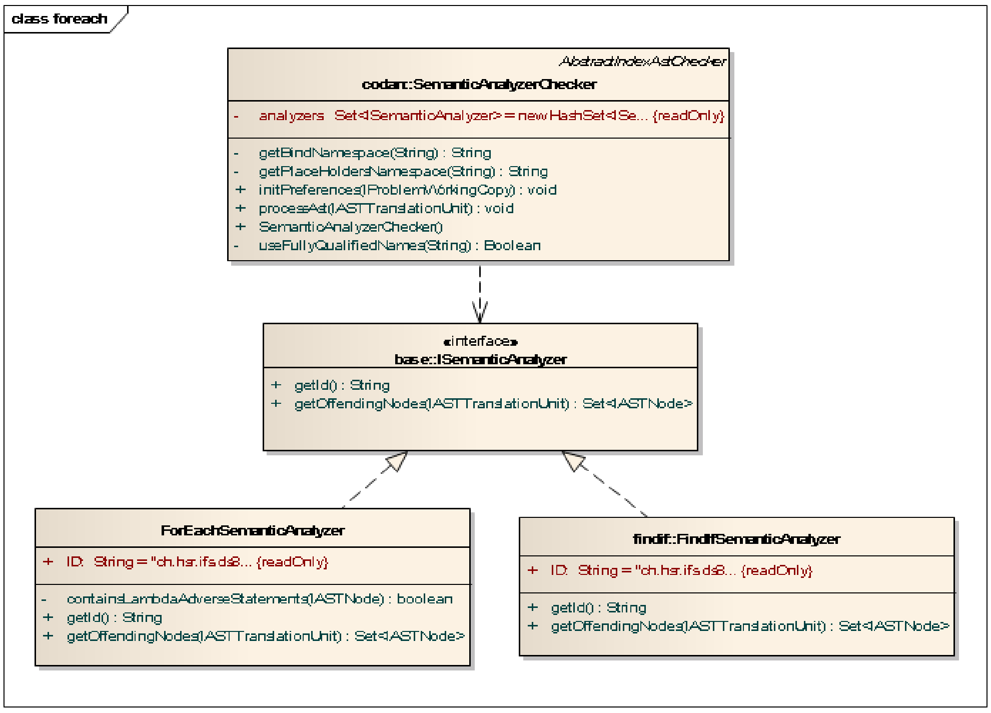

class foreach
AbstractIndexAstChecker
codan::SemanticAnalyzerChecker
- analyzers Set<ISemanticAnalyzer> = new HashSet<ISe... {readOnly}
- getBindNamespace(String) : String
- getPlaceHoldersNamespace(String) : String
+ initPreferences(IProblemWorkingCopy) : void
+ processAst(IASTTranslationUnit) : void
+ SemanticAnalyzerChecker()
- useFullyQualifiedNames(String) : Boolean
«interface»
base::ISemanticAnalyzer
+ getId() : String
+ getOffendingNodes(IASTTranslationUnit) : Set<IASTNode>
ForEachSemanticAnalyzer
+ ID: String = "ch.hsr.ifs.ds8... {readOnly}
- containsLambdaAdverseStatements(IASTNode) : boolean
+ getId() : String
+ getOffendingNodes(IASTTranslationUnit) : Set<IASTNode>
findif::FindIfSemanticAnalyzer
+ ID: String = "ch.hsr.ifs.ds8... {readOnly}
+ getId() : String
+ getOffendingNodes(IASTTranslationUnit) : Set<IASTNode>

Figure 3.8.: Semantic Analyzers (ch.hsr.ifs.ds8.analysis.base)

A semantic analyzers must, in order to implement its interface, provide client classes with a set of *IASTNode*s matching its implemented rule. In the case as displayed in figure 3.8, this happens to be a class using the CodAn framework (`http://wiki.eclipse.org/CDT/designs/StaticAnalysis`) to display the reported nodes as user interface markers. Of course, this class could be replaced by any other desired implementation not using the CodAn framework. *ISemanticAnalyzer* thus represents a seam point for the analysis package and avoids cumbersome dependencies to other frameworks at all cost.

Listing 3.2 shows the explicit interface specification of *ISemanticAnalyzers*, while listing 3.3 illustrates how client classes (e.g. a *CodAn Checker*) could use this interface to report problems.

```
/**
 * {@link ch.hsr.ifs.ds8.analysis.base.ISemanticAnalyzer Semantic analyzers}
 * represent the central interface between the DeepSpace-8 analysis module and
 * Eclipse UI {@link org.eclipse.core.resources.IMarker markers}. Problem
 * reporters use these analyzers to find offending
 * {@link org.eclipse.cdt.core.dom.ast.IASTNode nodes} in a translation unit
 * and mark their position with the
 * {@link ch.hsr.ifs.ds8.analysis.base.ISemanticAnalyzer analyzer}'s id.
 */
public interface ISemanticAnalyzer {
    /**
```

```
     * @return The textual problem id this analyzer addresses.
     */
    public String getId();

    /**
     * @param unit
     *            The {@link org.eclipse.cdt.core.dom.ast.IASTTranslationUnit
     *            IASTTranslationUnit} to be analyzed.
     * @return A {@link java.util.Set Set} of all
     *         {@link org.eclipse.cdt.core.dom.ast.IASTNode nodes} within
     *         <code>unit</code> that offend this semantic analyzer's code
     *         rules and should be refactored.
     */
    public Set<IASTNode> getOffendingNodes(IASTTranslationUnit unit);
}
```

Listing 3.2: *ISemanticAnalyzer* interface

```
public void processAst(IASTTranslationUnit ast) {
    for (ISemanticAnalyzer analyzer : analyzers) {
        Set<IASTNode> offendingNodes = analyzer.getOffendingNodes(ast);
        for (IASTNode node : offendingNodes) {
            reportProblem(analyzer.getId(), node);
        }
    }
}
```

Listing 3.3: *ISemanticAnalyzer* usage example

3.2.2. Transformation

This section is dedicated to explaining the various algorithms found present in the final implementation of the transformation module.

Range

The range deduction and transformation module is purposed to analyze transformable *loop* statements in the code, dissect initialization statements and break conditions of the respective constructs. From this information, the module tries to deduce an equivalent range of iterators that can be used with STL algorithms to replace the given *loop* statement. Listing 3.4.1 marks the three main analysis artifacts of this module: range begin, range end and the iteration variable.

```
vector<int> vec = getData();
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
    // ...
}
```

Listing 3.4: Range deduction

The module itself can be summarized into three sub-parts: the *IFirstDeducer*s, *ILastDeducer*s and *IIterationExpressionDeducer*s. The respective class diagrams are shown in figures 3.9, 3.10, 3.11 and 3.12.

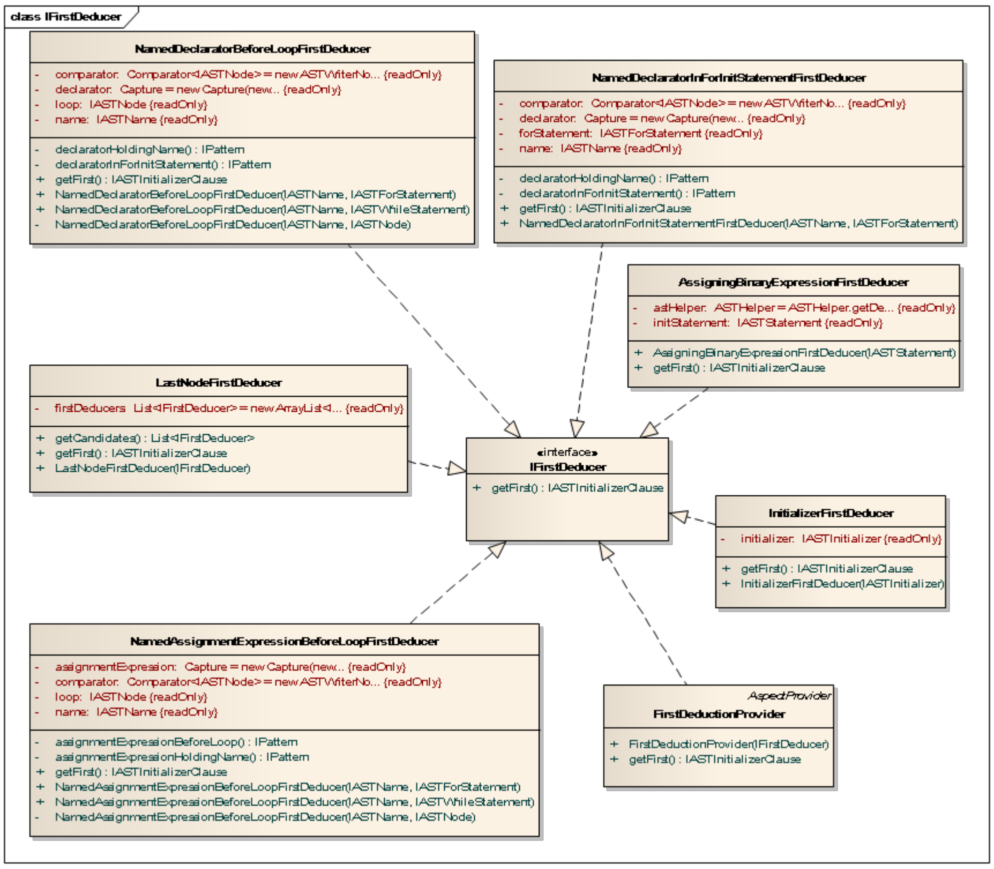

class IFirstDeducer
NamedDeclaratorBeforeLoopFirstDeducer
- comparator: Comparator<IASTNode> = new ASTWriterNo... {readOnly}
- declarator: Capture = new Capture(new... {readOnly}
- loop: IASTNode {readOnly}
- name: IASTName {readOnly}
- declaratorHoldingName() : IPattern
- declaratorInForInitStatement() : IPattern
+ getFirst() : IASTInitializerClause
+ NamedDeclaratorBeforeLoopFirstDeducer(IASTName, IASTForStatement)
+ NamedDeclaratorBeforeLoopFirstDeducer(IASTName, IASTWhileStatement)
- NamedDeclaratorBeforeLoopFirstDeducer(IASTName, IASTNode)
NamedDeclaratorInForInitStatementFirstDeducer
- comparator: Comparator<IASTNode> = new ASTWriterNo... {readOnly}
- declarator: Capture = new Capture(new... {readOnly}
- forStatement: IASTForStatement {readOnly}
- name: IASTName {readOnly}
- declaratorHoldingName() : IPattern
- declaratorInForInitStatement() : IPattern
+ getFirst() : IASTInitializerClause
+ NamedDeclaratorInForInitStatementFirstDeducer(IASTName, IASTForStatement)
AssigningBinaryExpressionFirstDeducer
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- initStatement: IASTStatement {readOnly}
+ AssigningBinaryExpressionFirstDeducer(IASTStatement)
+ getFirst() : IASTInitializerClause
LastNodeFirstDeducer
- firstDeducers List<IFirstDeducer> = new ArrayList<I... {readOnly}
+ getCandidates() : List<IFirstDeducer>
+ getFirst() : IASTInitializerClause
+ LastNodeFirstDeducer(IFirstDeducer)
«interface»
IFirstDeducer
+ getFirst() : IASTInitializerClause
InitializerFirstDeducer
- initializer: IASTInitializer {readOnly}
+ getFirst() : IASTInitializerClause
+ InitializerFirstDeducer(IASTInitializer)
NamedAssignmentExpressionBeforeLoopFirstDeducer
- assignmentExpression: Capture = new Capture(new... {readOnly}
- comparator: Comparator<IASTNode> = new ASTWriterNo... {readOnly}
- loop: IASTNode {readOnly}
- name: IASTName {readOnly}
- assignmentExpressionBeforeLoop() : IPattern
- assignmentExpressionHoldingName() : IPattern
+ getFirst() : IASTInitializerClause
+ NamedAssignmentExpressionBeforeLoopFirstDeducer(IASTName, IASTForStatement)
+ NamedAssignmentExpressionBeforeLoopFirstDeducer(IASTName, IASTWhileStatement)
- NamedAssignmentExpressionBeforeLoopFirstDeducer(IASTName, IASTNode)
AspectProvider
FirstDeductionProvider
+ FirstDeductionProvider(IFirstDeducer)
+ getFirst() : IASTInitializerClause

Figure 3.9.: ch.hsr.ifs.ds8.transformation.range.IFirstDeducer

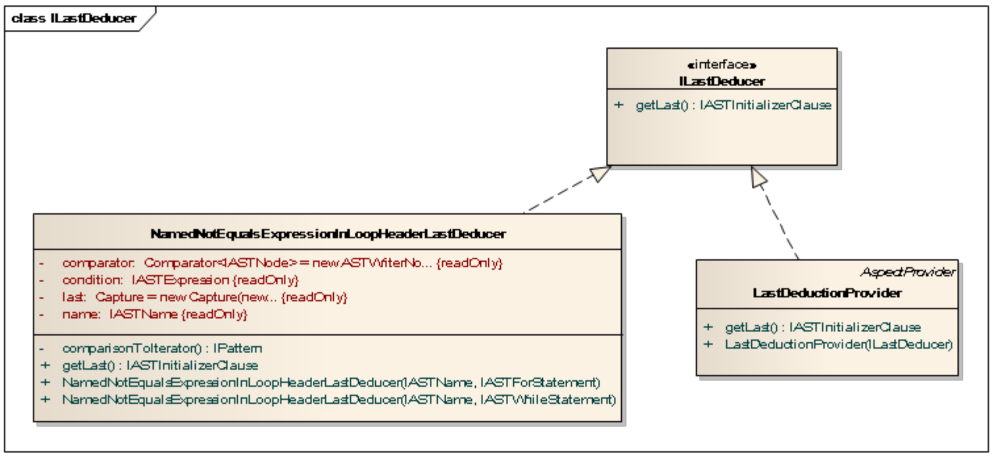

class ILastDeducer
«interface»
ILastDeducer
+ getLast() : IASTInitializerClause
NamedNotEqualsExpressionInLoopHeaderLastDeducer
- comparator: Comparator<IASTNode> = new ASTWriterNo... {readOnly}
- condition: IASTExpression {readOnly}
- last: Capture = new Capture(new... {readOnly}
- name: IASTName {readOnly}
- comparisonToIterator() : IPattern
+ getLast() : IASTInitializerClause
+ NamedNotEqualsExpressionInLoopHeaderLastDeducer(IASTName, IASTForStatement)
+ NamedNotEqualsExpressionInLoopHeaderLastDeducer(IASTName, IASTWhileStatement)
AspectProvider
LastDeductionProvider
+ getLast() : IASTInitializerClause
+ LastDeductionProvider(ILastDeducer)

Figure 3.10.: ch.hsr.ifs.ds8.transformation.range.ILastDeducer

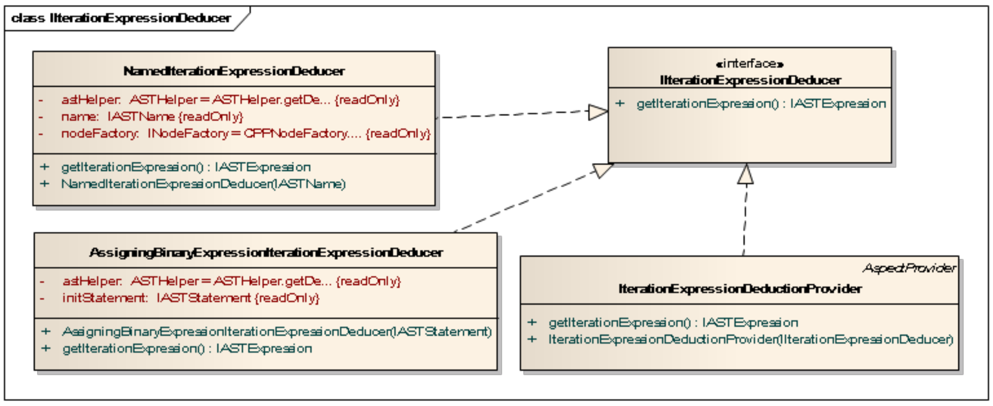

class IIterationExpressionDeducer
NamedIterationExpressionDeducer
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- name: IASTName {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
+ getIterationExpression() : IASTExpression
+ NamedIterationExpressionDeducer(IASTName)
«interface»
IIterationExpressionDeducer
+ getIterationExpression() : IASTExpression
AssigningBinaryExpressionIterationExpressionDeducer
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- initStatement: IASTStatement {readOnly}
+ AssigningBinaryExpressionIterationExpressionDeducer(IASTStatement)
+ getIterationExpression() : IASTExpression
AspectProvider
IterationExpressionDeductionProvider
+ getIterationExpression() : IASTExpression
+ IterationExpressionDeductionProvider(IIterationExpressionDeducer)

Figure 3.11.: ch.hsr.ifs.ds8.transformation.range.IIterationExpressionDeducer

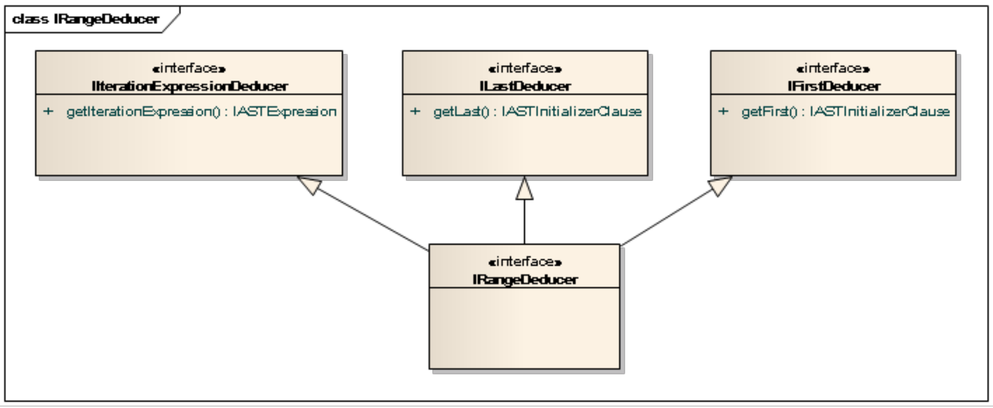

class IRangeDeducer
«interface»
IIterationExpressionDeducer
+ getIterationExpression() : IASTExpression
«interface»
ILastDeducer
+ getLast() : IASTInitializerClause
«interface»
IFirstDeducer
+ getFirst() : IASTInitializerClause
«interface»
IRangeDeducer

Figure 3.12.: ch.hsr.ifs.ds8.transformation.range.IRangeDeducer

Important to notice in figures 3.9 through 3.12 is the fine-grained interface structure, even including partial range deduction mechanisms such as *IFirstDeducer*, *ILastDeducer* and *IIterationExpressionDeducer*. Each of these interfaces are implemented by multiple code analysis algorithms (e.g. deducing the last iterator from an *equality expression* forming the *loop*'s break condition). All these deduction algorithms can be combined using *DeductionProviders*, that try all available algorithms and use the first one that successfully determines the respective range element. This design decision was made under the assumption that there will exist many algorithms to deduce iterator ranges and each of them will only fit certain instances and fail in others. The concept of a *DeductionProvider* furthermore eliminates the necessity of deciding for a concrete algorithm without knowing its capabilities. What cases of a *loop* can or cannot be dissected using a given algorithm is a matter to the specific algorithm itself only and needs not to concern client classes thanks to this design aspect.

Iterator

Transforming iterator-based to value_type-based access is, as this document's analysis chapter pointed out, a central aspect when eliminating custom instances of STL algorithms. Therefore, this issue was addressed in the implementation by a separate module dedicated to iterator transformation rules and techniques.

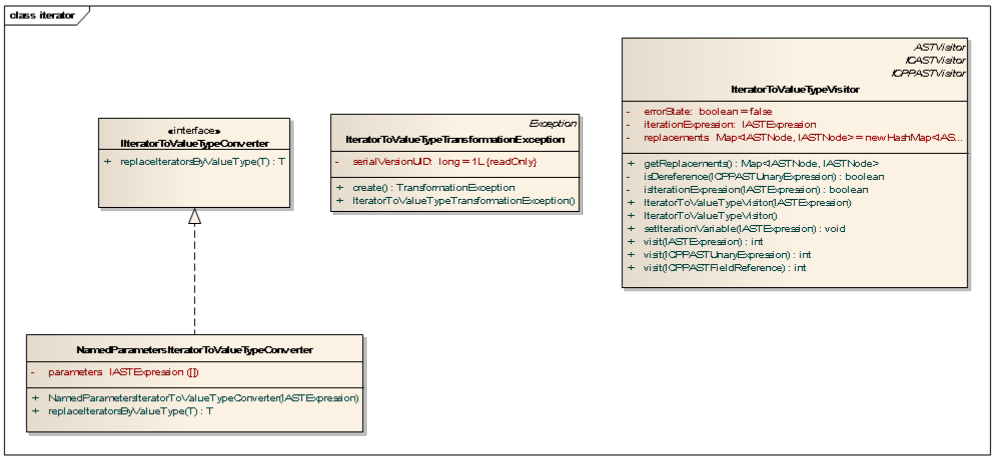

class iterator
«interface»
IIteratorToValueTypeConverter
+ replaceIteratorsByValueType(T) : T
Exception
IteratorToValueTypeTransformationException
- serialVersionUID: long = 1L {readOnly}
+ create() : TransformationException
+ IteratorToValueTypeTransformationException()
ASTVisitor
ICASTVisitor
ICPPASTVisitor
IteratorToValueTypeVisitor
- errorState: boolean = false
- iterationExpression: IASTExpression
- replacements: Map<IASTNode, IASTNode> = new HashMap<IAS...
+ getReplacements() : Map<IASTNode, IASTNode>
- isDereference(ICPPASTUnaryExpression) : boolean
- isIterationExpression(IASTExpression) : boolean
+ IteratorToValueTypeVisitor(IASTExpression)
+ IteratorToValueTypeVisitor()
+ setIterationVariable(IASTExpression) : void
+ visit(IASTExpression) : int
+ visit(ICPPASTUnaryExpression) : int
+ visit(ICPPASTFieldReference) : int
NamedParametersIteratorToValueTypeConverter
- parameters: IASTExpression ([])
+ NamedParametersIteratorToValueTypeConverter(IASTExpression)
+ replaceIteratorsByValueType(T) : T

Figure 3.13.: ch.hsr.ifs.ds8.transformation.iterator

A central role is accorded to the *IIteratorToValueTypeConverter* interface. It provides a generic *replaceIteratorsByValueType* that accepts any IASTNode, removes any iterator occurrence by a value_type and returns the node again. This replacement functionality is implemented by means of an *IteratorToValueTypeVisitor* that traverses the given node and replaces any child nodes identified as iterator accesses by their value_type equivalent. Listings 3.5 and 3.6 show one instance of such an iterator to value type transformation. Matching code segments in the original and the transformed block are highlighted with the same color.

```
vector<string>::iterator it = getIterator();
{
  cout << *it << endl;
  cout << it->size() << endl;
}
```

Listing 3.5: Body with iterator semantics

```
vector<string>::iterator it = getIterator();
{
  cout << it << endl;
  cout << it.size() << endl;
}
```

Listing 3.6: Body with value type semantics

NamedParametersIteratorToValueTypeConverter in this context represents the most straightforward implementation of *IIteratorToValueTypeConverter*, where the respective iterator expressions to be replaced are known and can be passed as an argument.

Functor

Chapter 1 pointed out that STL algorithms only allow *functions* or *functors* as arguments. A very important focus is thus to be set on the transformation of statements within a loop to equivalent *function* or *functor*, respectively. This can be achieved using the options pointed out in section 1.3 - namely *argument binding*, *lambda expressions* or of course *explicit functions or functor classes*. This idea represents the very purpose of the *functor* module. This package's *IteratorStatementToLambdaFunctorConverter* illustrated in figure 3.14 serves two purposes. On the one hand, it transforms iterator-based accesses previously included in the original loop statement by value_type-based operations. It does so using the iterator-transformation package. Afterwards, the new statement is wrapped into a lambda expression taking an argument of the value_type of the iteration expression and returns this whole assembly as the result functor. Listings 3.7 and 3.8 show a comprehensive example of these two transformation steps. Matching code segments in the original and the transformed block are highlighted with the same color.

```
vector<string>::iterator it = getIterator();
// vector<string>::iterator::value_type = std::string
{
  cout << *it << endl;
  cout << it->size() << endl;
}
```

Listing 3.7: Original compound statement to transform to a lambda

```
[](std::string & it) {
    cout << it << endl;
    cout << it.size() << endl;
}
```

Listing 3.8: Resulting lambda expression

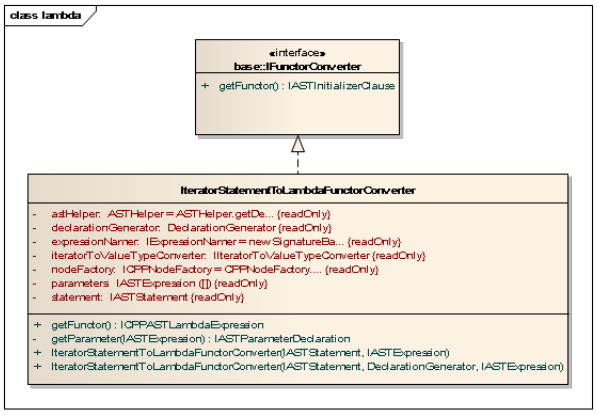

class lambda
«interface»
base::IFunctorConverter
+ getFunctor() : IASTInitializerClause
IteratorStatementToLambdaFunctorConverter
- astHelper: ASTHelper = ASTHelper.getDe... {readOnly}
- declarationGenerator: DeclarationGenerator {readOnly}
- expressionNamer: IExpressionNamer = new SignatureBa... {readOnly}
- iteratorToValueTypeConverter: IIteratorToValueTypeConverter {readOnly}
- nodeFactory: ICPPNodeFactory = CPPNodeFactory.... {readOnly}
- parameters: IASTExpression ([]) {readOnly}
- statement: IASTStatement {readOnly}
+ getFunctor() : ICPPASTLambdaExpression
- getParameter(IASTExpression) : IASTParameterDeclaration
+ IteratorStatementToLambdaFunctorConverter(IASTStatement, IASTExpression)
+ IteratorStatementToLambdaFunctorConverter(IASTStatement, DeclarationGenerator, IASTExpression)

Figure 3.14.: ch.hsr.ifs.ds8.transformation.functor.lambda

Other implementations within the package are *SingleExpressionStatementToTr1BindFunctor* and *SingleExpressionStatementToStdBindFunctor* shown in figure 3.15 and figure 3.16 respectively. They allow the conversion of single expression statements (e.g. a function call or a binary expression) to an equivalent bind expression. This process is illustrated in the examples in listings 3.9 through 3.9. Matching code segments in the original and the transformed block are highlighted with the same color.

```
vector<string>::iterator it = getIterator();
{
  it->reserve(100);
}
```

Listing 3.9: Original compound statement to transform to a bind expression

```
vector<string>::iterator it = getIterator();
bind2nd(mem_fun_ref(&std::string::reserve), 100)(*it);
```

Listing 3.10: Resulting STL bind expression

```
vector<string>::iterator it = getIterator();
bind(&std::string::reserve, _1, 100)(*it);
```

Listing 3.11: Resulting TR1 bind expression

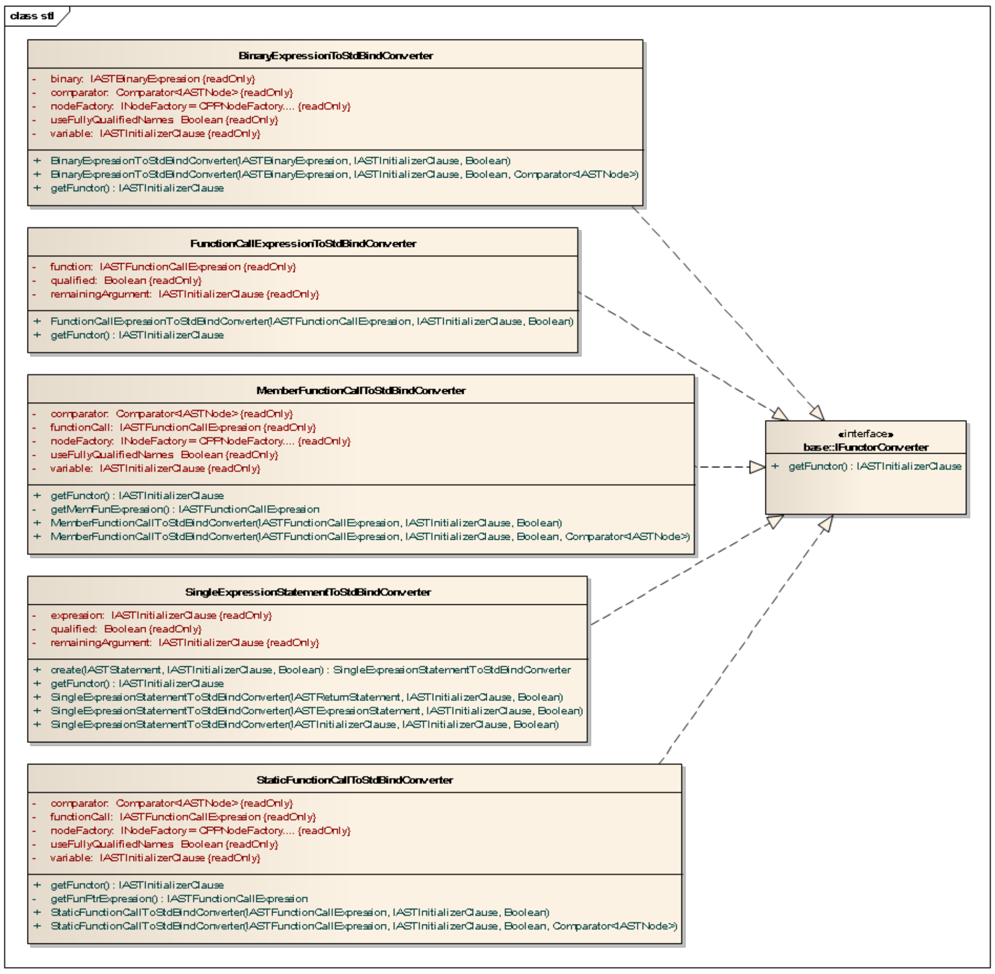

class stl
BinaryExpressionToStdBindConverter
- binary: IASTBinaryExpression {readOnly}
- comparator: Comparator<IASTNode> {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- useFullyQualifiedNames Boolean {readOnly}
- variable: IASTInitializerClause {readOnly}
+ BinaryExpressionToStdBindConverter(IASTBinaryExpression, IASTInitializerClause, Boolean)
+ BinaryExpressionToStdBindConverter(IASTBinaryExpression, IASTInitializerClause, Boolean, Comparator<IASTNode>)
+ getFunctor() : IASTInitializerClause
FunctionCallExpressionToStdBindConverter
- function: IASTFunctionCallExpression {readOnly}
- qualified: Boolean {readOnly}
- remainingArgument: IASTInitializerClause {readOnly}
+ FunctionCallExpressionToStdBindConverter(IASTFunctionCallExpression, IASTInitializerClause, Boolean)
+ getFunctor() : IASTInitializerClause
MemberFunctionCallToStdBindConverter
- comparator: Comparator<IASTNode> {readOnly}
- functionCall: IASTFunctionCallExpression {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- useFullyQualifiedNames Boolean {readOnly}
- variable: IASTInitializerClause {readOnly}
+ getFunctor() : IASTInitializerClause
- getMemFunExpression() : IASTFunctionCallExpression
+ MemberFunctionCallToStdBindConverter(IASTFunctionCallExpression, IASTInitializerClause, Boolean)
+ MemberFunctionCallToStdBindConverter(IASTFunctionCallExpression, IASTInitializerClause, Boolean, Comparator<IASTNode>)
«interface»
base::IFunctorConverter
+ getFunctor() : IASTInitializerClause
SingleExpressionStatementToStdBindConverter
- expression: IASTInitializerClause {readOnly}
- qualified: Boolean {readOnly}
- remainingArgument: IASTInitializerClause {readOnly}
+ create(IASTStatement, IASTInitializerClause, Boolean) : SingleExpressionStatementToStdBindConverter
+ getFunctor() : IASTInitializerClause
+ SingleExpressionStatementToStdBindConverter(IASTReturnStatement, IASTInitializerClause, Boolean)
+ SingleExpressionStatementToStdBindConverter(IASTExpressionStatement, IASTInitializerClause, Boolean)
+ SingleExpressionStatementToStdBindConverter(IASTInitializerClause, IASTInitializerClause, Boolean)
StaticFunctionCallToStdBindConverter
- comparator: Comparator<IASTNode> {readOnly}
- functionCall: IASTFunctionCallExpression {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- useFullyQualifiedNames Boolean {readOnly}
- variable: IASTInitializerClause {readOnly}
+ getFunctor() : IASTInitializerClause
- getFunPtrExpression() : IASTFunctionCallExpression
+ StaticFunctionCallToStdBindConverter(IASTFunctionCallExpression, IASTInitializerClause, Boolean)
+ StaticFunctionCallToStdBindConverter(IASTFunctionCallExpression, IASTInitializerClause, Boolean, Comparator<IASTNode>)

Figure 3.15.: ch.hsr.ifs.ds8.transformation.functor.bind.stl

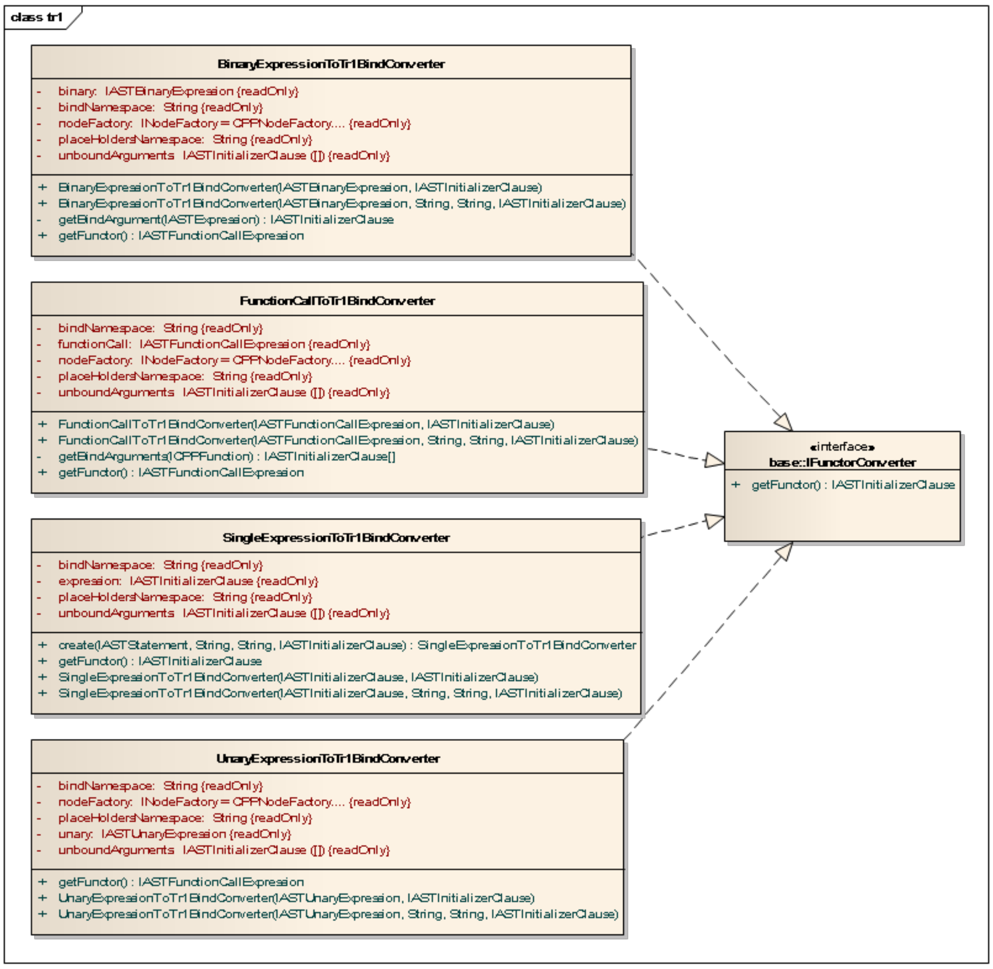

class tr1
BinaryExpressionToTr1BindConverter
- binary: IASTBinaryExpression {readOnly}
- bindNamespace: String {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- placeHoldersNamespace: String {readOnly}
- unboundArguments: IASTInitializerClause ([]) {readOnly}
+ BinaryExpressionToTr1BindConverter(IASTBinaryExpression, IASTInitializerClause)
+ BinaryExpressionToTr1BindConverter(IASTBinaryExpression, String, String, IASTInitializerClause)
- getBindArgument(IASTExpression) : IASTInitializerClause
+ getFunctor() : IASTFunctionCallExpression
FunctionCallToTr1BindConverter
- bindNamespace: String {readOnly}
- functionCall: IASTFunctionCallExpression {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- placeHoldersNamespace: String {readOnly}
- unboundArguments: IASTInitializerClause ([]) {readOnly}
+ FunctionCallToTr1BindConverter(IASTFunctionCallExpression, IASTInitializerClause)
+ FunctionCallToTr1BindConverter(IASTFunctionCallExpression, String, String, IASTInitializerClause)
- getBindArguments(ICPPFunction) : IASTInitializerClause[]
+ getFunctor() : IASTFunctionCallExpression
«interface»
base::IFunctorConverter
+ getFunctor() : IASTInitializerClause
SingleExpressionToTr1BindConverter
- bindNamespace: String {readOnly}
- expression: IASTInitializerClause {readOnly}
- placeHoldersNamespace: String {readOnly}
- unboundArguments: IASTInitializerClause ([]) {readOnly}
+ create(IASTStatement, String, String, IASTInitializerClause) : SingleExpressionToTr1BindConverter
+ getFunctor() : IASTInitializerClause
+ SingleExpressionToTr1BindConverter(IASTInitializerClause, IASTInitializerClause)
+ SingleExpressionToTr1BindConverter(IASTInitializerClause, String, String, IASTInitializerClause)
UnaryExpressionToTr1BindConverter
- bindNamespace: String {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- placeHoldersNamespace: String {readOnly}
- unary: IASTUnaryExpression {readOnly}
- unboundArguments: IASTInitializerClause ([]) {readOnly}
+ getFunctor() : IASTFunctionCallExpression
+ UnaryExpressionToTr1BindConverter(IASTUnaryExpression, IASTInitializerClause)
+ UnaryExpressionToTr1BindConverter(IASTUnaryExpression, String, String, IASTInitializerClause)

Figure 3.16.: ch.hsr.ifs.ds8.transformation.functor.bind.tr1

Other possible converters not implemented during the scope of this project are converters to new functor classes or new functions having the transformed statements as body. Listings 3.12 and 3.13 repeat the previous examples to illustrate how prospective follow-up projects should implement such a transformation.

```
void doReserve(std::string &it) {
  it.reserve(100);
}
// ...
vector<string>::iterator it = getIterator();
doReserve(*it);
```

Listing 3.12: Resulting free function

```
class Reserver {
public:
  void operator()(std::string &it) {
    it.reserve(100);
  }
};
// ...
vector<string>::iterator it = getIterator();
Reserver()(*it);
```

Listing 3.13: Resulting functor class

Foreach

The first instance putting all previous transformation techniques together and instantiating an actual STL algorithm transformation is instigated by the *foreach* module.

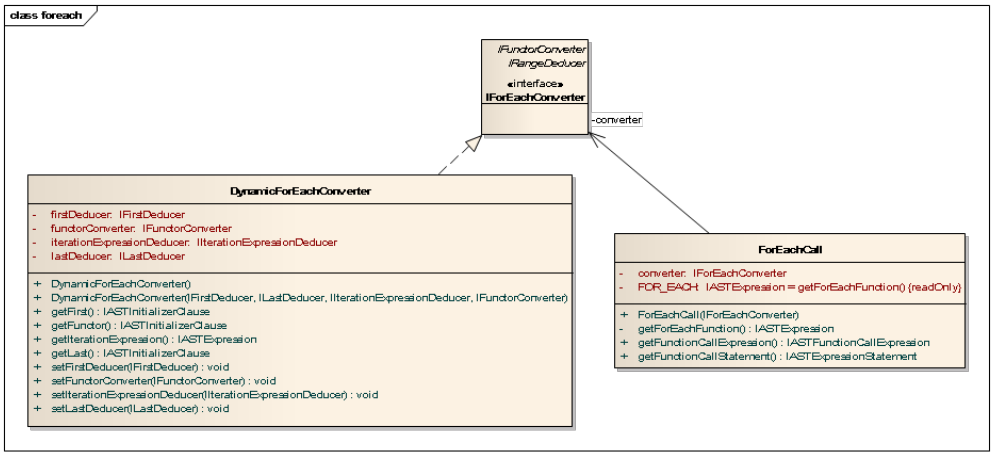

class foreach
IFunctorConverter
IRangeDeducer
«interface»
IForEachConverter
-converter
DynamicForEachConverter
- firstDeducer: IFirstDeducer
- functorConverter: IFunctorConverter
- iterationExpressionDeducer: IIterationExpressionDeducer
- lastDeducer: ILastDeducer
+ DynamicForEachConverter()
+ DynamicForEachConverter(IFirstDeducer, ILastDeducer, IIterationExpressionDeducer, IFunctorConverter)
+ getFirst() : IASTInitializerClause
+ getFunctor() : IASTInitializerClause
+ getIterationExpression() : IASTExpression
+ getLast() : IASTInitializerClause
+ setFirstDeducer(IFirstDeducer) : void
+ setFunctorConverter(IFunctorConverter) : void
+ setIterationExpressionDeducer(IIterationExpressionDeducer) : void
+ setLastDeducer(ILastDeducer) : void
ForEachCall
- converter: IForEachConverter
- FOR_EACH: IASTExpression = getForEachFunction() {readOnly}
+ ForEachCall(IForEachConverter)
- getForEachFunction() : IASTExpression
+ getFunctionCallExpression() : IASTFunctionCallExpression
+ getFunctionCallStatement() : IASTExpressionStatement

Figure 3.17.: Foreach transformation (ch.hsr.ifs.ds8.transformation.foreach)

An *IForEachConverter* is the combination of an *IRangeDeducer* and an *IFunctorConverter*. Thus, given a respective loop, *IForEachCoverters* are able to deduce both a begin and end iterator equivalent to the range processed by the loop and instantiate a functor implementing the same logic as the loop body. The helper class *ForEachCall* uses this information to construct a *call statement* to *for_each* with these very same properties as function parameters. To achieve this, the *DynamicForEachConverter* class can be configured with instances of these very same interfaces and uses them to perform its transformation. Listings 3.14 and 3.14 illustrate this process, highlighting matching code segments in the original and transformed block with the same color.

```
vector<string> data = getData(); for (vector<string>::iterator it = data.begin();
     it != data.end(); ++it) {
  it->clear();
}
```

Listing 3.14: Original for_each candidate

```
for_each(data.begin(), data.end(), bind(&std::string::clear, _1));
```

Listing 3.15: Resulting for_each call

Find and find_if

The second major example of a transformation package summarizing multiple transformation techniques to implement an STL algorithm is the *find* and *find_if* package.

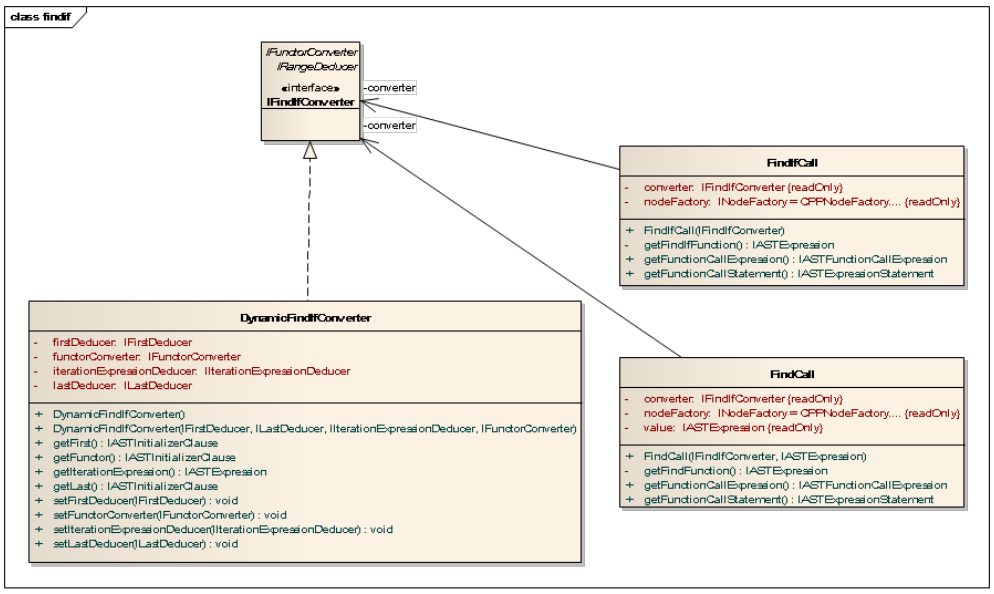

class findif
IFunctorConverter
IRangeDeducer
«interface»
IFindIfConverter
-converter
-converter
FindIfCall
- converter: IFindIfConverter {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
+ FindIfCall(IFindIfConverter)
- getFindIfFunction() : IASTExpression
+ getFunctionCallExpression() : IASTFunctionCallExpression
+ getFunctionCallStatement() : IASTExpressionStatement
DynamicFindIfConverter
- firstDeducer: IFirstDeducer
- functorConverter: IFunctorConverter
- iterationExpressionDeducer: IIterationExpressionDeducer
- lastDeducer: ILastDeducer
+ DynamicFindIfConverter()
+ DynamicFindIfConverter(IFirstDeducer, ILastDeducer, IIterationExpressionDeducer, IFunctorConverter)
+ getFirst() : IASTInitializerClause
+ getFunctor() : IASTInitializerClause
+ getIterationExpression() : IASTExpression
+ getLast() : IASTInitializerClause
+ setFirstDeducer(IFirstDeducer) : void
+ setFunctorConverter(IFunctorConverter) : void
+ setIterationExpressionDeducer(IIterationExpressionDeducer) : void
+ setLastDeducer(ILastDeducer) : void
FindCall
- converter: IFindIfConverter {readOnly}
- nodeFactory: INodeFactory = CPPNodeFactory.... {readOnly}
- value: IASTExpression {readOnly}
+ FindCall(IFindIfConverter, IASTExpression)
- getFindFunction() : IASTExpression
+ getFunctionCallExpression() : IASTFunctionCallExpression
+ getFunctionCallStatement() : IASTExpressionStatement

Figure 3.18.: FindIf transformation (ch.hsr.ifs.ds8.transformation.findif)

The *IFindIfConverter* again demands the implementation of an *IRangeDeducer* that almost all loop-to-STL transformations require. Furthermore, it expects an implementation of an *IFunctorConverter* to be present. Different from the converter found present in the *for_each* conversion, however, this functor converter will identify condition statements within the transformed loop and express them as a functor expression to be used as predicate within a *find_if* call. This encompasses basically the same techniques for functor transformation described previously in this section, with the addition of e.g. introducing a *return statement* when creating a lambda functor. Listings 3.16 and 3.17 show this transformation, highlighting matching code segments in the original and transformed block with the same color.

```
vector<string> data = getData() vector<string >::iterator it;
for (it = data.begin(); it != data.end(); ++it) {
  if (it->empty()) {
    break;
  }
}
```

Listing 3.16: Original for_each candidate

```
vector<string> data = getData();
vector<string >::iterator it;
it = find_if(data.begin(), data.end(), bind(&std::string::empty, _1));
```

Listing 3.17: Resulting find_if call

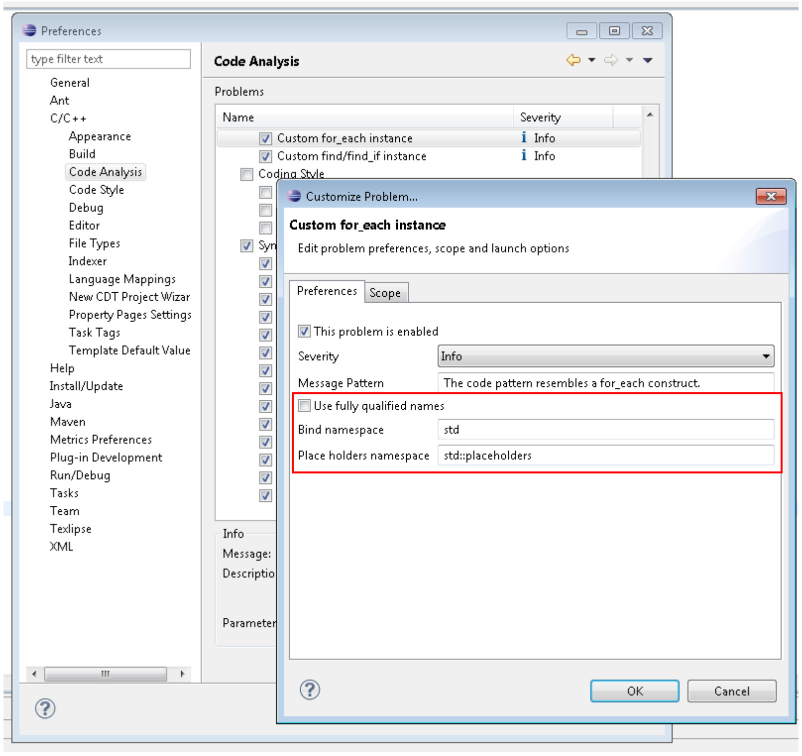
3.2.3. Eclipse integration

There exist two seam points this project's plug-in implementation uses to interact with the Eclipse Framework: The Eclipse static code analysis framework (*CodAn*, `http://wiki.eclipse.org/CDT/designs/StaticAnalysis`) and the Eclipse *IMarkerResolution* interface. Both will be explained in detail in the following subsections.

CodAn Framework

The CodAn project basically provides a unified interface for static code analysis plug-ins. Instead of working with resources and loading their contents manually, plug-in developers are free to work directly on abstract syntax trees (AST, *IASTTranslationUnit*). Same is true when the plug-in developer desires to place markers in the resources once the static code analysis plug-in has detected an issue in one of the traversed AST's nodes. Usually, the plug-in developer would need to determine the node's actual location within the original file in order to create a marker. CodAn, however, allows placing markers directly on AST nodes. In fact, if developers do not explicitly wish to do so, they do not need to interact with raw *IResources* at all.

Furthermore, CodAn provides a seamless UI integration for configuring the newly created static code checker. Developers only need to state the name and data type of properties they would like their users to be able to configure and CodAn will automatically generate the respective UI elements. Figure 3.19 illustrates such automatically generated UI elements based on one *Boolean* and two *String* properties.



Name	Severity
✓ Custom for_each instance	Info
✓ Custom find/find_if instance	Info

Custom for_each instance

Edit problem preferences, scope and launch options

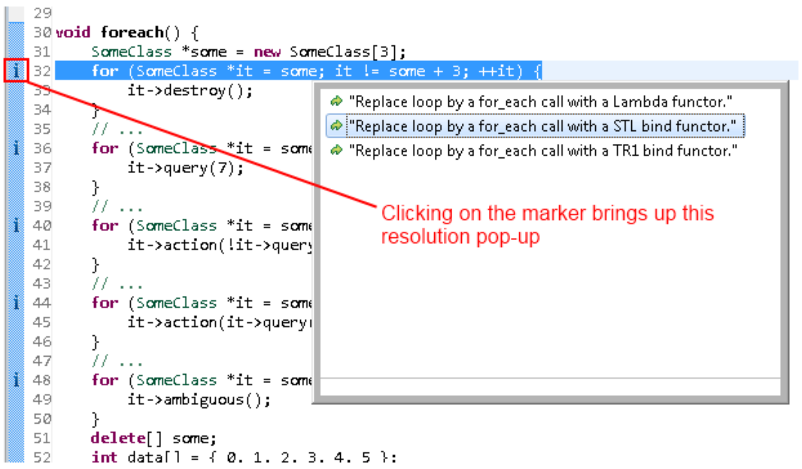
✓ This problem is enabled

Severity	Info
Message Pattern	The code pattern resembles a for_each construct.
☐ Use fully qualified names	
Bind namespace	std
Place holders namespace	std::placeholders

Figure 3.19.: Automatically generated CodAn UI elements.

IMarkerResolution

Once resource markers have been placed as explained in the previous section, seasoned Eclipse users will realize that this is only half the story. Eclipse provides markers indicating errors or warnings with quick-fixes to resolve the marked problems. Programmatically speaking, these quick-fixes relate to *IMarkerResolutions*, which themselves are generated by *IMarkerResolutionGenerators*. In fact, the complete transformation module described in section 3.4 eventually boils down to a set of *IMarkerResolution* implementations. *IMarkerResolutionGenerators* are registered as extension points in the *plugin.xml* descriptor and should return the correct quick-fixes (or *IMarkerResolutions*) based on the given problem id. Figure 3.20 shows an example of this resolution provision in the UI.



```
void foreach() {
    SomeClass *some = new SomeClass[3];
    for (SomeClass *it = some; it != some + 3; ++it) {
        it->destroy();
    }
    // ...
    for (SomeClass *it = some
        it->query(7);
    }
    // ...
    for (SomeClass *it = some
        it->action(!it->query
    }
    // ...
    for (SomeClass *it = some
        it->action(it->query
    }
    // ...
    for (SomeClass *it = some
        it->ambiguous();
    }
    delete[] some;
    int data[] = { 0, 1, 2, 3, 4, 5 };
```

"Replace loop by a for_each call with a Lambda functor."
"Replace loop by a for_each call with a STL bind functor."
"Replace loop by a for_each call with a TR1 bind functor."

Clicking on the marker brings up this resolution pop-up

Figure 3.20.: Each quick-fix relates to a respective *IMarkerResolution* implementation

3.3. Code analysis module

Based on the identified analysis techniques in section 2.3, the following sections describe how the pattern-based approach has been instantiated in the actual program. The metadata-based approaches have been omitted in the current version for they have not been estimated beneficial enough for further investigation (directive from professor P.Sommerlad, November 18 2010).

3.3.1. Pattern matching

To fit existing structures within the Eclipse plug-in framework and to provide the pattern analysis functionality in all areas manipulating the analyzed program's abstract syntax tree, the tree regular expressions package is expected to meet an interface specification as simple and easily usable as the following:

```
/**
 * General interface for tree patterns used to verify whether a certain node
 * fits a given set of distinguishing features.
 */
public interface IPattern {
    /**
     * Verifies whether <code>node</code> fits the distinguishing features
     * satisfying this pattern instance.
     *
     * @param node
     *           The {@link org.eclipse.cdt.core.dom.ast.IASTNode node} in
     *           question.
     * @return <code>true</code> if <code>node</code> satisfies the pattern,
     *         <code>false</code> otherwise.
     */
    public boolean satisfies(IASTNode node);
}
```

Listing 3.18: Desired regular expressions interface

Furthermore, the following pattern implementations are deemed mandatory for a successful analysis of the use cases described in section 1.1.3:

- Domination
  A node or tree can be required to dominate (i.e. be an ancestor of) a specific subtree of nodes, possibly satisfying requirements of their own. This hierarchical information is the one critical part the abstract syntax tree surpasses the analysis information provided by a lexical representation.
- Binary predicates
  Two nodes or trees must be comparable for equality or other predicates in terms of various criteria. Such criteria may range from textual equality of the resulting source code to node type and field equality.
- Sibship
  The order of elements on the same layer must be identifiable and nodes must be describable as left or right siblings of each other. This feature gains importance e.g. when trying to identify the current value of a variable before the first execution of a loop. Hence, the left siblings of the loop need to be identified and analyzed.

The following subsections will describe a first attempt of implementing the previously described requirements using synthetical metadata nodes within the abstract syntax tree. This approach failed to meet all the desired requirements, which is explained to more detail there. Afterwards, the successful second implementation approach is explained in the remaining subsections.

Failed approach of metadata nodes

The first draft implementation of the patterns package was based upon the idea that the abstract syntax tree should be amended by additional, synthetical metadata nodes to form the actual tree regular expression. This approach is comparable to classic regular expressions based on text data extended by metadata characters such as *.*, *** or *?*. A possible tree pattern might thus look as described in figure 3.21.

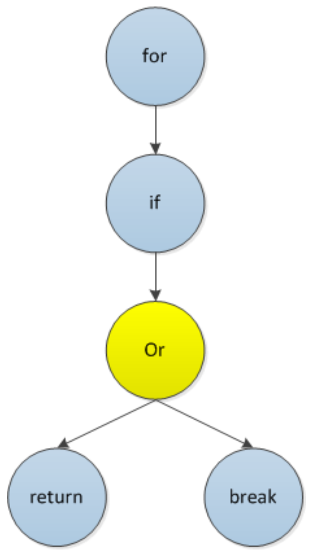


Figure 3.21.: Simple metadata nodes pattern for find_if

This idea provided very promising results for patterns as simple as the previously described one. However, as soon as non-deterministic patterns with wildcards and repetitions enter the picture, the metadata approach proves itself far less applicable. Figure 3.22 represents one such more elaborate case.

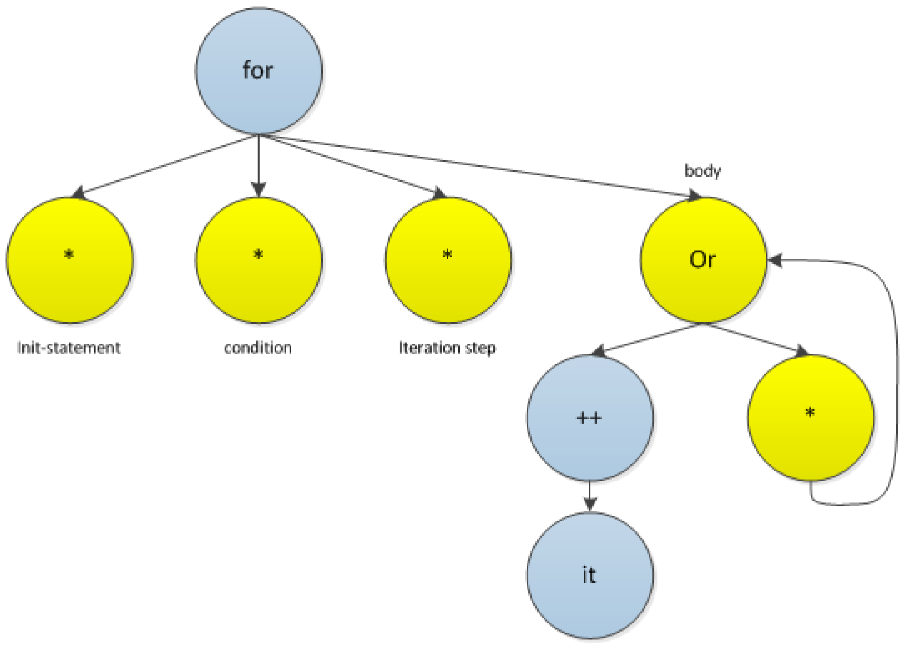


Figure 3.22.: Extended metadata nodes pattern disqualifying loops unsuitable for a conversion to for_each. Please note that *** instead of *.* was chosen as a wildcard character in this figure for the sake of readability.

The pattern described in figure 3.22 is satisfied if the given *for* loop contains an increment expression on the iteration expression within the body. Any node satisfying this pattern thus cannot be transformed to an STL function call. Identifying nodes that match patterns like this, however, may become increasingly difficult due to the following problems:

- It is not clear whether wildcards should match single nodes or complete subtrees. Matching subtrees would represent an implicit repetition of its own and would be difficult to express in combination with explicit repetitions. Matching single nodes, however, ends up being an oversimplification and requiring pattern specifications to be significantly more complicated than as seen in figure 3.22.
- Even for the simple case of a “contains ++it”-regular expression displayed in figure 3.22, a non-deterministic pattern specification containing a possibly infinite loop is necessary. Algorithms that preprocess the regular expression statement before applying it to a node would need to be aware of this possibility, making their implementation unnecessarily complicated.
- The last and most critical issue not directly depicted in figure 3.22 is the question whether repetitions should relate to additional siblings to follow or instead

additional child nodes to be allowed. Figure 3.23 illustrates this ambivalence (please note that **.** identifies a wild card and **{n}** a repetition of n elements). When finally asking the question whether a repeated child may have siblings as well, the impression occurs that repetitions are a very unexpressive concept in tree patterns.

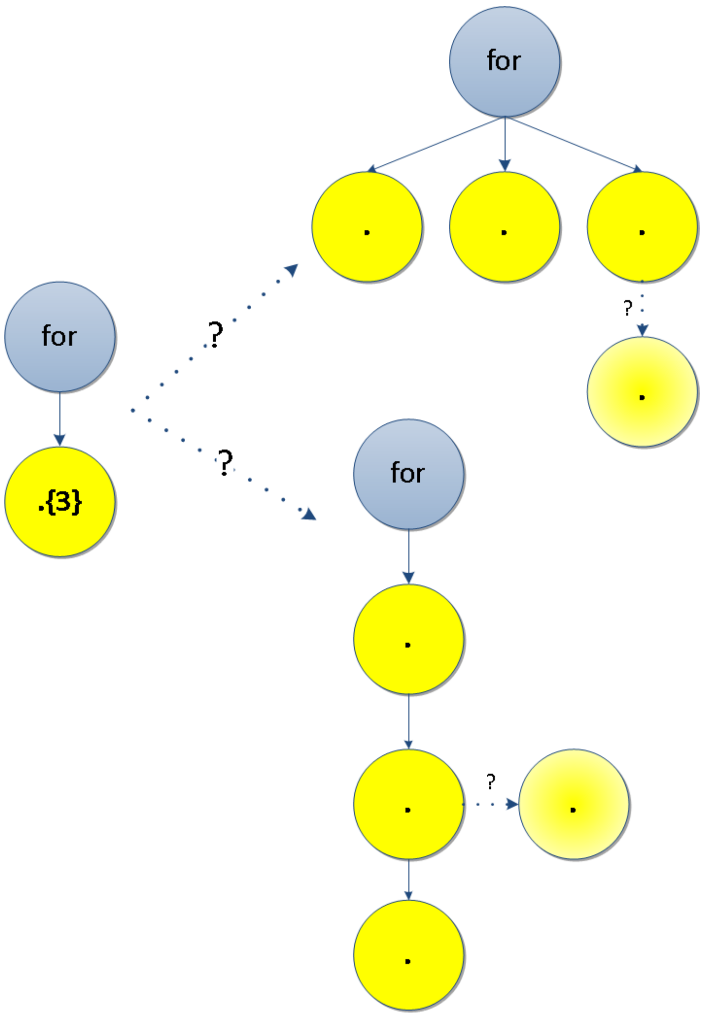


Figure 3.23.: Ambivalence of repetitions in trees

Even though there may indeed exist feasible solutions to the problems described in this section concerning metadata nodes, their implementation has been dropped for the scope of this project. Instead the focus of this project's regular expressions implementation was shifted from repetitions and metadata nodes over to another, very promising concept: the pattern domination.

Pattern domination

Please note: "Pattern domination" is a term introduced during this thesis to describe this very specific implementation of tree pattern matching. While tree pattern matching is a concept often addressed in (natural) language processing [LA05], we have been unable to identify software patterns equivalent to the one instantiated with this implementation. Thus the software pattern "Tree pattern domination" or "Pattern domination" respectively is introduced and referred to as such during the rest of this report.

Repetitions focus on the idea of allowing a series of nodes between the actual nodes of interest. $A.{*}B$, for example, would allow any content between the two nodes A and B. This idea, however, puts the focus in an unnatural manner on the content to be eventually ignored. The more important question in this context would be whether B should be a sibling or a child of A - and whether it should be so immediately or whether intermediate nodes are allowed. Pattern domination replaces the unnatural repetition relationships from the tree regular expressions and introduces new, by far more expressive relations to the pattern:

- A << B
  A dominates B, i.e. B is a descendant of A.
- A < B
  A immediately dominates B, i.e. B is a child of A.
- A << (B < C)
  A dominates B, which itself immediately dominates C.
- A $++ B
  A is a left sibling of B.
- A $+ B
  A is an immediate left sibling of B.

Above list represents a subset of the relations implemented in Tregex [LA05, p.2].

The idea of one node dominating another one is promising, but by itself not sufficient to represent all required patterns, as figure 3.24 illustrates.

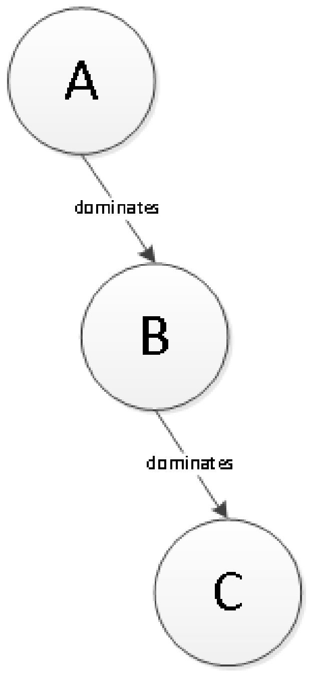


Figure 3.24.: Nested domination pattern

If a pattern can only express that a node A dominates a node B and a node B equally dominates a node C, the question remains how these two patterns should be connected, i.e. how node B should be guaranteed to be exactly the same node, so that the original pattern as a whole is satisfied. While this example might be implemented by an additional identity equality check, it points out that hierarchical relationships may be required between patterns of nodes instead of nodes only. Figure 3.25 shows this combination problem explicitly.

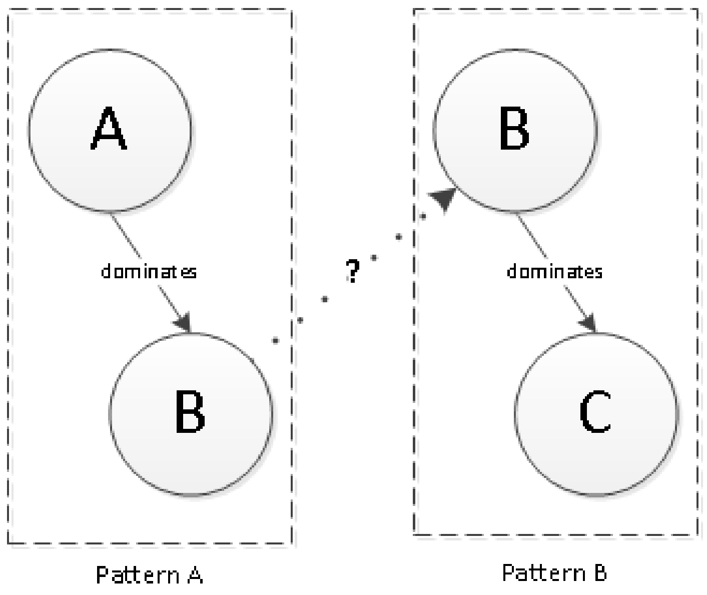


Figure 3.25.: Pattern combination problem

The solution to this combination problem chosen in the implementation of this project was to completely drop the concept of one node dominating another and instead design structural relationships in a way that node patterns dominate other node patterns. A pattern as illustrated in figure 3.24 might thus be represented in a pattern form as shown by figure 3.26.

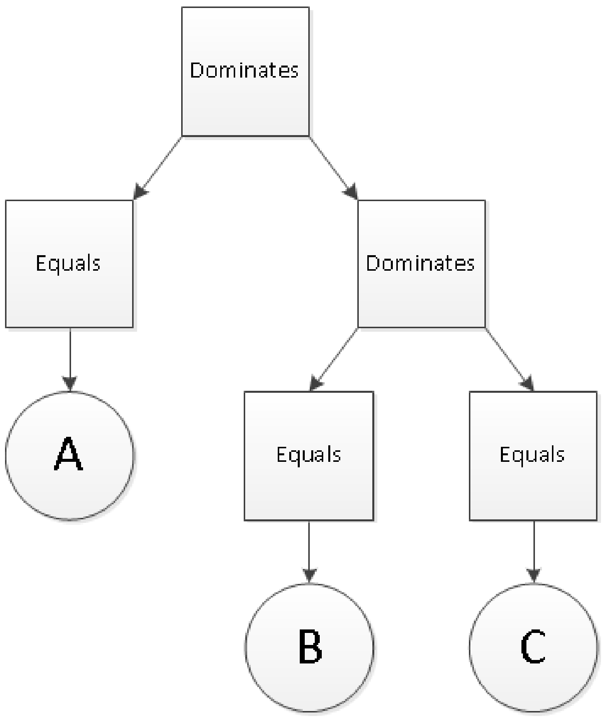


Figure 3.26.: Pattern domination-design

Capturing and Referencing

Another concept borrowed from textual regular expressions is Capturing and Referencing. Capturing allows to store matched portions of the complete expression and extract them after the evaluation. This functionality proofs useful when trying to extract data of a certain pattern from a large amount of data. Referencing, on the other hand, allows to access previously captured portions and use them in further expression specifications. One example of such a reuse is to require a previously matched subportion to occur again later in the complete data. Listing 3.19 provides an example of a textual regular expression with one capturing group marked by parentheses and one reference to that capturing group (\1). Listing 3.20 illustrate how this regular expression would match against actual text.

```
.*(‘‘empty’’\.).*\1
```

Listing 3.19: Textual regex with captures and references

```
This sentence ends with “empty”.
This one does not.
This, however, does again end with “empty”.
```

Listing 3.20: Matched text

The very same functionality is available for any pattern in the tree patterns package. Every pattern can be wrapped by a Capture decorator, storing the matched node in case of a successful match. Equally, references are available as adapted version of equality comparison and represent themselves patterns that can be matched against. Figure 3.27 displays an example of a pattern featuring one capture and one respective reference.

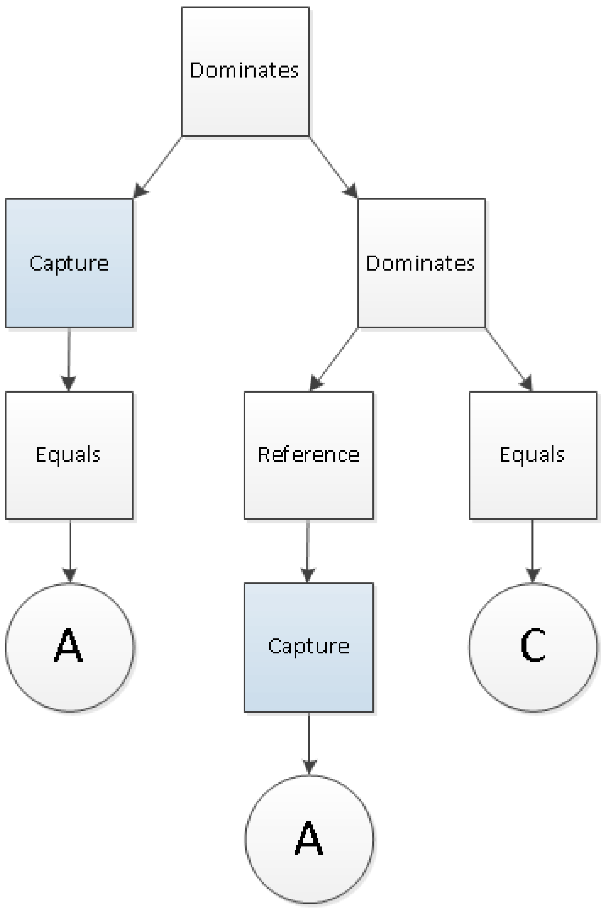


Figure 3.27.: Capturing and referencing

Please note that the two capture nodes are highlighted by the same color to illustrate that these are indeed references to the very same object. This implementation also requires all references to captures to appear during in-order traversal strictly after their respective capture nodes.

Backtracking inversion

While the concept of decorator classes for capturing and referencing described in section 3.3.1 provides and elegant and flexible implementation of such functionality, it provides one natural drawback. Using two different wrapper nodes for capturing and referencing leaves them only very loosely coupled to each other. Explicitly, the match of a capturing node is completely independent of the match of a corresponding reference. This may represent a very undesirable situation, if one e.g. tries to search a C++ program for a variable name declared within a for loop init-statement and subsequently used within the loop body. The declaration would refer to a capturing node, and the usage within the body of the loop would correspond to a reference. Listing 3.21 illustrates one such code snippet.

```
vector<int> vec = getData();
for (vector<int>::iterator it = vec.begin(), it2 = vec.begin(); it2 != vec.end
    (); ++it2) {
    cout << *it2 << endl;
}
```

Listing 3.21: Situation for backtracking inversion

A naive pattern to match the declaration and subsequent usage within the body shown in listing 3.21 may be implemented as illustrated in figure 3.28.

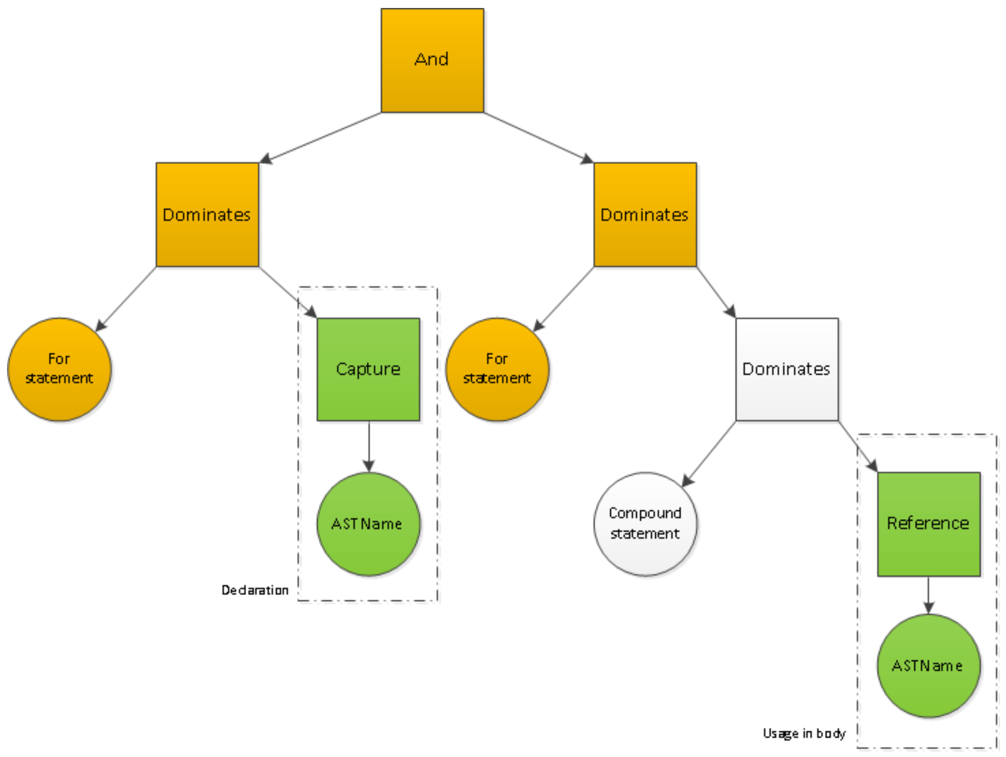


Figure 3.28.: Naive referencing

It is important to notice, however, that the pattern from figure 3.28 will not match the code in listing 3.21. This is due to the fact that the two branches of the *And* node are independent from each other. The left branch will search for an *IASTName* and find it by capturing the name *"it"*, whereafter the right branch is traversed. There, however, it occurs that *"it"* cannot be located within the body and the match will fail. In this situation, the *And* node's left branch has no incentive whatsoever to backtrack to its previous capturing of *"it"* and search for another *IASTName* descendant, namely *"it2"*. The backtracking inversion technique was introduced to provide this incentive.

Within the current tree patterns implementation, there exists one measure to impose natural backtracking functionality: Hierarchy. If a child pattern fails to match, the *Dominates* or *DominatesImmediately* pattern will search for another descendant and try to match it against the respective sub-pattern. As long as the reference is a descendant of the capture, backtracking works flawlessly. In the example provided in listing 3.21 and figure 3.28, the capturing and referencing take place on the same level within the pattern - naturally so, because the *for-init-statement* and the loop body are located on the same level below the *for-statement* within the AST as well. Using backtracking inversion and the *isDominatedBy* pattern, however, it is possible to introduce an artificial hierarchy to our pattern enabling the backtracking behavior. Figure 3.29 describes this concept.

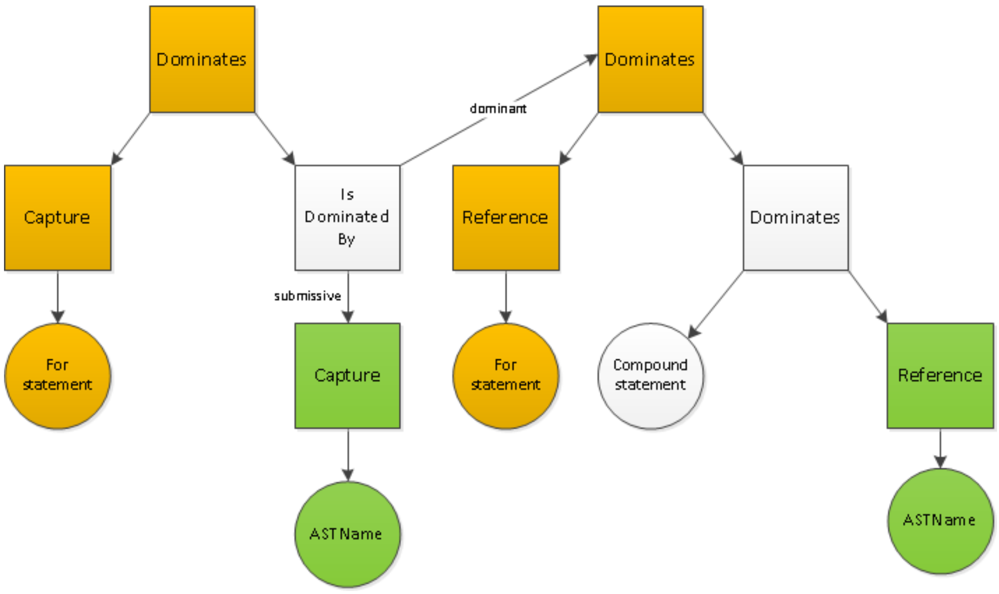


Figure 3.29.: Backtracking inversion

Thanks to *isDominatedBy* and the additional reference to the original *for statement*, the reference node becomes a descendant of the capture node and backtracking will work as desired.

For_each tree pattern implementation

Figure 3.30 illustrates a simplified form of the tree pattern used to recognize *for_each* candidates. Pleas note that structural metadata nodes, such as *Dominates*, have been illustrated as arrow connectors. The shown pattern matches iterator variables declared before the loop that are incremented once within the loop and dereferenced at least once within the body. The final pattern in the plug-in code is extended by several alternatives (e.g. having the iterator declaration in the for-init statement), but their general make-up remains the same.

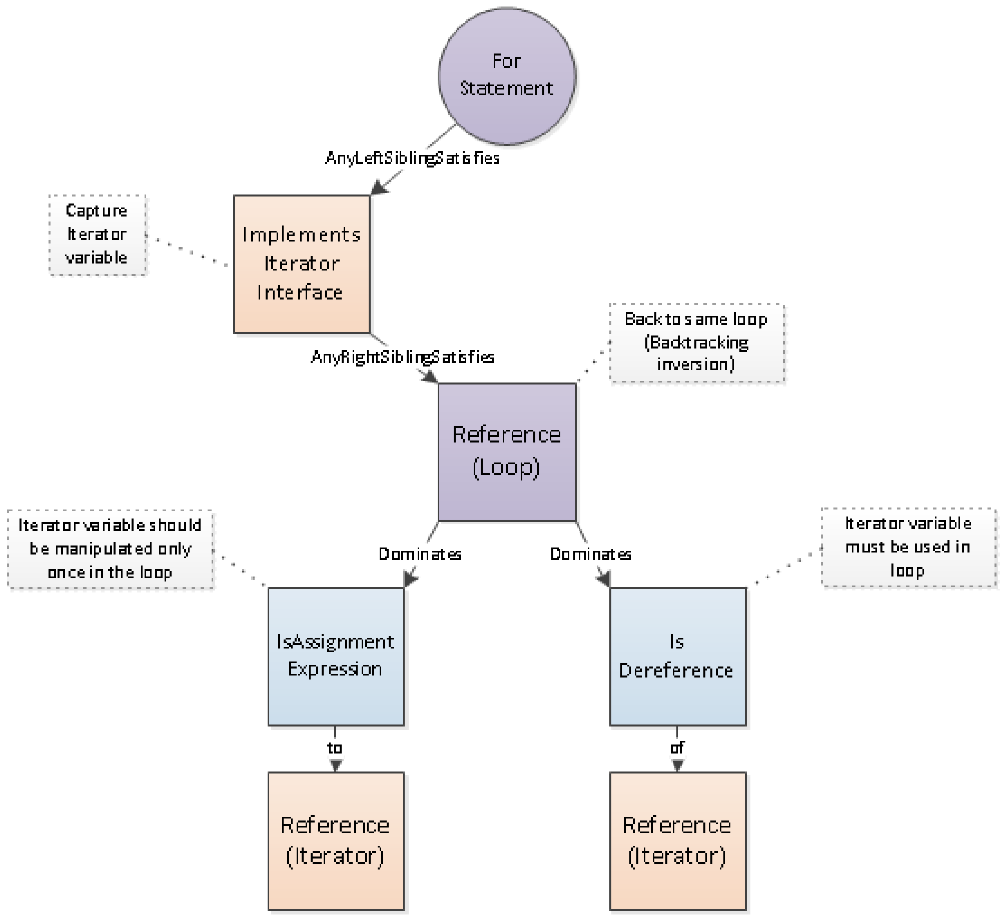

For Statement
AnyLeftSiblingSatisfies
Capture Iterator variable
Implements Iterator Interface
AnyRightSiblingSatisfies
Back to same loop (Backtracking inversion)
Reference (Loop)
Iterator variable should be manipulated only once in the loop
Dominates
Dominates
Iterator variable must be used in loop
IsAssignment Expression
Is Dereference
to
of
Reference (Iterator)
Reference (Iterator)

Figure 3.30.: For_each tree pattern

Find_if tree pattern implementation

In figure 3.31 a simplified version of the *find_if* pattern is shown. It recognizes *for loops* dominating *if statements* which themselves dominate either a *break* or *return statement*. In the final plug-in code, several variations of this pattern are used for the eventual recognition, but the general implementation method remains as described in the figure.

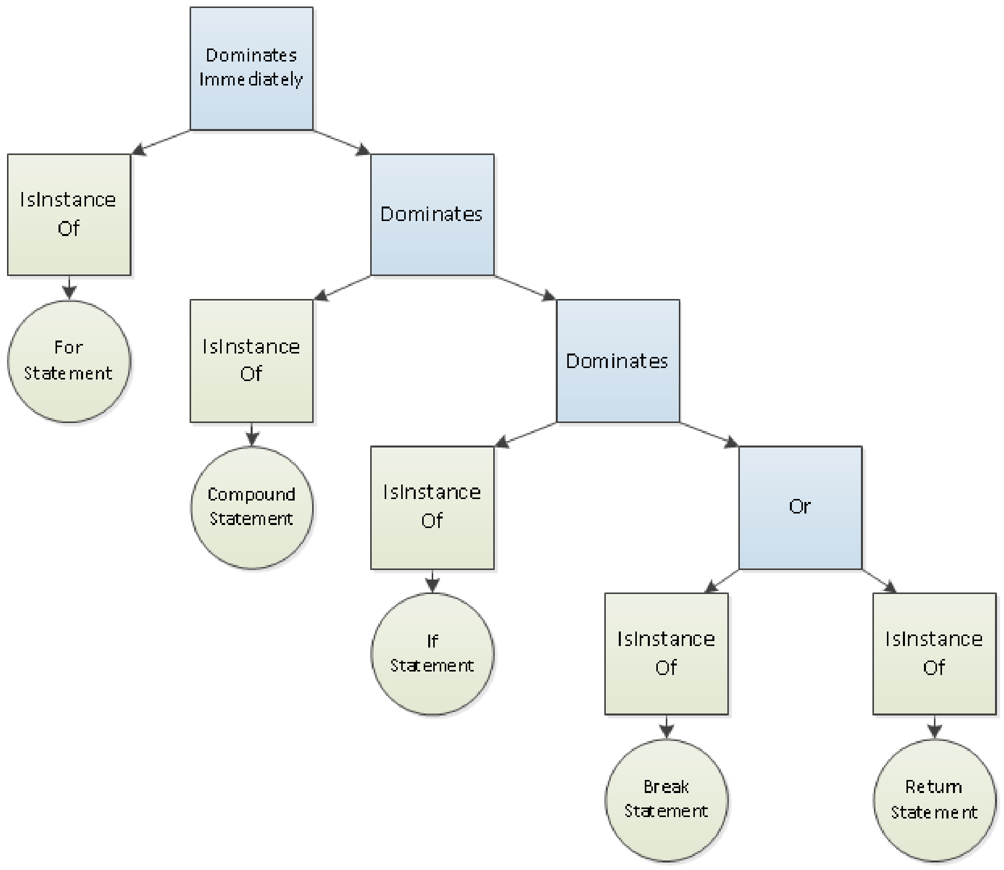


Figure 3.31.: Find_if tree pattern

3.3.2. Iterator compatibility

When trying to identify variables suitable to be used as iterators within a C++ program, it remains important to notice that in C++, other than in Java, there exists no explicit interface for iterators. Instead, all STL algorithms accepting iterators are template-based and implicitly demand certain operators to be present. The STL algorithm *for_each*, as an example, uses the following operators in its implementation when used with a type T:

- operator != (T)
- operator ++
- operator *

These three operators thus also exactly represent the three operations the current analysis system's implementation expects to be present in order to consider a variable an iterator variable candidate.

3.4. Code transformation module

Once a loop and possibly some of its siblings have been identified as candidates for an STL function call refactoring, the second major module of this thesis' program steps into action. The transformation module identifies the range covered by the loop and transforms its body to an equivalent functor to be used with the STL function call.

3.4.1. Range deduction

Loops working on containers or other sets of values iterate over a more or less implicit range of elements. This range begins with the actual value of the iteration variable before the first execution of the loop body and is terminated by the first iteration variable value that no more satisfies the loop's continuation condition. In feature-rich languages such as C++, however, these two values are not always trivial to deduce, which is why the range deduction module's architecture assumed the presence of many different deduction algorithms (see section 3.2.2).

The most important approach to this deduction in the scope of this thesis is the idea of searching for initialization values and assignment expressions to the iteration expression before the loop (or possibly within the initialization statement in a *for* loop) and select the last value as the begin of the range. The terminating iteration expression value may be extracted from the loop's condition expression, which is expected to be a comparison expression between the iteration expression and the terminating value. A very simple approach, this idea is able to determine the range of a variety of loop statements correctly and before expanding this deduction algorithm to more complex predicates, index-based access to iterator transformation may provide refactoring support for far more loop instances than the support for arbitrarily complex iterator range deduction (see also 2.4.2).

3.4.2. Iterator to value type access

When transforming an iterator-based loop's *body* statement to a lambda functor, the need to replace iterator accesses by value type-based operations becomes apparent. Iterators embrace a pointer akin syntax of dereferencing (*) and member access (→). The code examples in listings 3.22 and 3.23 illustrates this issue.

```
vector<string> vec;
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  cout << *it << endl;
  it->clear();
}
```

Listing 3.22: Iterator access syntax

```
for_each(vec.begin(), vec.end(), [](string &it) {
    cout << it << endl;
  it.clear();
});
```

Listing 3.23: Transformed iterator access

The two loops in listing 3.22 and 3.23 and are equivalent and represent the mapping this project's transformation module is expected to introduce. However, there exist transformations that may be considered equally trivial, but must not be performed by the system. Listing 3.24 presents these cases.

```
void registerValue(const string &);  // overload #1
void registerValue(const vector<string>::iterator &);  // overload #2
vector<string> vec;
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  registerValue(it);  // uses overload #1
}
for_each(vec.begin(), vec.end(), [](string &it) {
  registerValue(it);  // uses overload #2
});
```

Listing 3.24: Disallowed iterator transformations

At first glance, a prospective transformation system will find itself bewildered by the usage of *it* without a data access operator (* or →). It would for all intents and purposes be possible to advise the system to determine whether there exists an overload of *registerValue*, accepting also the *value_type* of the iterator. Since *registerValue* could, however, implement completely different semantics for the two overloads or one of the overloads could even get changed by the unsuspecting developer, equivalence of the two loops in listing 3.24 is no more guaranteed. Transformation systems that desire to focus on loops and have no interest in significantly changing a developer's class members should thus resign upon such code constructs. There exist other cases that shape up as very inopportune to emphvalue_type-based access, as listing 3.25 shows.

```
vector<string> vec;
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  if (increment()) {
    ++it;
  ]
  if (decrement()) {
    --it;
  }
  // ...
}
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  if (it != vec.begin()) {
    clog << *it << endl;
  ]
}
for (vector<string>::iterator it = vec.begin() + 1; it != vec.end(); ++it) {
  clog << (*it + *(it - 1)) << endl;
}
```

Listing 3.25: Unremunerative transformations

The loops in listing 3.25 are allowedly exaggerated examples, but they clearly represent the semantic loss faced when moving from iterators to *value_types*, as described in section 2.4.2. To still provide a suitable transformation, any such system would have to introduce additional complexity (e.g. in form of a counting variable), which is why these cases are not considered worthwile in the this project's implementation.

3.4.3. Function parameter types deduction

When creating functors and functions during a transformation, the plug-in must be able to deduce their parameter types depending on the loop to be transformed. This usually involves the deduction of the iteration variable type and the return type of its *operator ** function, if the iteration variable represents a pointer or iterator type. The Eclipse SDK provides each *expression node* within the abstract syntax tree with a function *getExpressionType*. For any given *expression node*, this method returns the correct *IType* definition of the relative C++ type. When creating functors and functions, the plug-in relies on this functionality by applying the following steps:

- Deduce the iteration variable *var* (either a pointer or iterator type, see 3.26)
- Create a synthetic, unary expression "**var*" (see 3.27, step #2)
- Retrieve the *IType* object using *getExpressionType* (see 3.27, step #3)
- Construct the parameter declaration using the *DeclarationGenerator* interface

The above approach is as generic as the *iterator* interface and thus can process any range and iteration types that can be handled by STL algorithms. Listings 3.26 and 3.27 illustrate this behavior.

```
vector<string> vec;
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  // "it" is deduced to be the iterationExpression
}
```

Listing 3.26: Function parameter type deduction (source)

```
IASTNode parent = iterationExpression.getParent();
IASTExpression expression = iterationExpression.copy();
ICPPASTUnaryExpression operatorStar = new CPPASTUnaryExpression(
     IASTUnaryExpression.op_star, iterationExpression);  //step #2
operatorStar.setParent(parent);
IType valueType = operatorStar.getExpressionType();  //step #3
```

Listing 3.27: Function parameter type deduction (plug-in code)

3.4.4. Function parameter types in template environments

Section 3.4.3 pointed out techniques to identify function parameter types in general. There exist situations in the C++ programming language that do not allow the explicit deduction of the target type name, as listing 3.28 illustrates.

```
template<class T>
void doSomething(T &container) {
  for(typename T::iterator it = container.begin(); it != container.end(); ++it
     ) {
    doIt(*it);
  }
}
```

Listing 3.28: Infeasible Function parameter type deduction

When transforming above loop to a call to *for_each*, the plug-in finds itself in the uncomfortable situation of being unable to deduce the type name of the expression **it*. If *doSomething* was called using a *vector<string>*, the respective type would be *string*. If it was called using a *vector<int>*, the type would expand to *int*. Since the respective type cannot be deduced using a definitive type name, the transformation needs to rely on the C++ compiler and defer the type deduction to it, as shown in listing 3.29 [ISO10a, dcl.type.simple, section 4].

```
template<class T>
void doSomething(T &c) {
  for_each(c.begin(), c.end(), [](decltype(*container.end()) &it) {
    doIt(it);
  });
}
```

Listing 3.29: Defer type evaluation to compiler

3.4.5. Associate equivalent type names

Apart from the notions of sections 3.4.3 and 3.4.4, there remains a simple, but significant extension to the type name deduction process. While the evaluation of an expression against a certain IType provides a general and universally applicable technique, the results may not always prove to be favorable, as listing 3.30 explains.

```
vector<string> data;
vector<string>::iterator it = data.begin();
cout << *it << endl;
// getExpressionType returns "std::basic_string<char, std::char_traits<char>,
    std::allocator<char> >"
```

Listing 3.30: Expanded type names by *getExpression*

Strictly speaking, the type deduced in 3.30 is perfectly correct and can be compiled in this form as well without any errors. However, it would obviously proof simpler and more understandable to use the typedef *string* instead of the explicit base type. Thus, the plug-in searches the current translation unit for any typedef that fits the explicit type and uses it, should it render shorter than the original.

4. Conclusion

When reviewing the initial task description of this project, explaining that the solution should be programmed as an Eclipse plug-in, "if one such implementable technique is found", it becomes obvious that no one at the beginning of this projected expected it to actually prosper and unfold as it has. The following sections will thus reflect on the results the project has put forth and where there still exists room for further development.

4.1. Results

During this master thesis, the areas of *semantic code analysis*, *code transformation* and the details of an appropriate *Eclipse integration* have been explored. The following sections highlight the progress made during this project in the respective areas. The resulting program's development name was *DeepSpace-8* and is referred to as such in the following or by its abbreviation *DS-8*.

4.1.1. Semantic analysis

Despite the very harsh limitations semantic code analysis faces in today's programming world (see section 1.4), the project has shown that analysis techniques such as tree pattern matching or to some extent also natural metadata analysis can proof themselves invaluable tools in focused areas of program analysis. The most important drawback faced when applying tree pattern matching is the fact that there exist in principle an infinite amount of false negatives the system fails to recognize. On the other hand, if a respective code matches the tested pattern, experiences made during this project suggest that false positives and faulty transformations hardly ever occur. This leads to the conclusion that pattern-based analysis serves best when trying to avoid the repetition of known errors rather then the recognition of new weaknesses in our code.

4.1.2. Code transformation

In contrary to code analysis, actual code transformation has shown to be indeed a deterministic and thoroughly implementable task. Its possibilities, however, remain bound to the capabilities of the respective code analysis modules. Once a loop e.g. has been identified as a *find_if* candidate and its range has been positively determined, the actual transformation of the loop body to a bind- or lambda-based predicate and the transformation of the loop itself to a *find_if* function call proved surprisingly trivial.

4.1.3. Eclipse integration

Before the beginning of this project, the Eclipse refactoring plug-in has not yet fully explored the subject of functor conversions. As an example, the refactoring plug-in's *ExpressionWriter* first needed an enhancement to support the newly introduced lambda expressions. Furthermore, too many type- and name-related operations within Eclipse have been implemented on text-based algorithms that render useless for advanced transformation tasks. There exist at least three implementations converting an *IASTName* to a "::"-separated, fully-qualified name, but not a single one converting it to a *ICPPASTQualifiedName*. This lack exists probably due to the fact that string-based operations are essentially easier to program than proper ones using *IASTNames*. The DS-8 project should set a good example of how proper, AST-based implementations can be achieved and how the existing, string-based operations can be reused using rewriting and re-parsing methods.

4.1.4. Actual plug-in functionality

To conclude this summary of the achieved results during this project, the following list once more explicitly states the features and concepts elaborated during this master thesis:

- Analysis
  - Recognition on possible for_each candidates based on pre-defined patterns. (3.3.1)
  - Recognition on possible find/find_if candidates based on pre-defined patterns. (3.3.1)
  - Recognition of compound statements convertible to lambda functors. (3.2.2)
  - Recognition of compound statements convertible STL or TR1 functors using argument binding. (3.2.2)
  - Deduction of iterator ranges traversed by iterator-based loops. (3.4.1)
  - Identification of variables suitable for usage as iterators. (3.3.2)
- Transformation
  - Transformation of iterator-based compound statements to value type-based compound statements. (3.4.2)
  - Transformation of compound statements to lambda functors. (3.2.2)
  - Transformation of compound statements to STL or TR1 functors using argument binding. (3.2.2)
  - Extended deduction of expression types, including the search for equivalent, shorter typedefs. (3.4.5)
  - Replacement of loops by for_each function calls. (3.2.2)

  - Replacement of loops by find/find_if function calls. (3.2.2)
- Eclipse integration
  - Support operations for proper, AST-based name and type resolution (in contrary to the existing, text-based solutions) (3.4.3)
  - Support operations for providing text representations of nodes (e.g. type declarations) with the necessary environment to make up a full translation unit and be parsed using the CDT parser. (3.4.3, 3.4.5)
  - Framework and easily extensible interfaces for semantic analyzers developed in future projects. (3.2.1)

4.2. Goal achievement

This section is dedicated to comparing mandatory and optional goals stated in the original task description and judge their fulfillment.

4.2.1. Mandatory

The list below states that all mandatory goals have been achieved and verified:

- Statistical analysis of existing code
  - ☑ Textual search in open source projects for matching loops
  - ☑ Analysis of the results to project use-of-potential analysis
- Research and formulate transformation systems
  - ☑ Pattern matching analysis
  - ☑ ASTRewrite-based transformation
- Implement proposed systems
  - ☑ iterator-to-value-type transformation
  - ☑ loop body to lambda conversion
  - ☑ for_each conversion

4.2.2. Optional

Furthermore, once optional goals are attended, it shows that with few exceptions also these optional taks have been completed. The only topics left out were the *explicit function/functor class conversion*, which was considered simple enough to be implemented in a term project. Apart from that remain only the *generate and transform conversions*, which have been dropped in favor of the various *bind conversions.* They too, however, should provide no issues to be implemented as follow-up term projects when using the *for_each and find/find_if* implementations as a reference.

- Research and formulate transformation systems
    - ☑ Natural metadata-based analysis
- Implement proposed systems
    - ☑ typedef resolution
    - ☑ loop body to TR1 bind conversion
    - ☑ loop body to STL bind conversion
    - ☐ loop body to explicit function/functor class conversion
    - ☑ find_if conversion
    - ☑ find conversion
    - ☐ generate conversion
    - ☐ transform conversion

4.2.3. Summary

Putting all the features listed in sections 4.2.1 and 4.2.2 together, this master thesis yielded a fully functional Eclipse code analysis plug-in. When activated, users get prompted as soon as any loop the user has written resembles a *for_each* or *find_if* algorithm. They then have the choice to automatically transform these loops to respective STL function calls. While doing so, they get to select the kind of functor should used for the STL call. The three supported kinds in this context are Lambda functors, STL bind1st & bind2nd functors as well as TR1 bind functors.

4.3. Outlook

Apart from having instantiated a working and usable implementation of loop-to-STL transformations, this master thesis also provided the basis for a variety of follow-up studies and tasks that find themselves listed in the following sections.

4.3.1. Implement skipped features

Section 4.2.2 listed some optional tasks that have been skipped during this project. These include namely:

- loop body to explicit function/functor class conversion
- generate conversion
- transform conversion

Based on the existing *for_each* and *find/find_if* implementations, the *generate* and *transform* implementation provide an excellent task for students yet unfamiliar with Eclipse CDT plug-in development. It is safe to assume that they both can be completed during one respective term project.

4.3.2. Follow-up on natural metadata analysis

While the concept of natural metadata analysis has been defined and evaluated in theory in section 2.3.2, no statistical analysis of its possible benefits or practical implementation have followed this evaluation. Future code analysis projects may want to further explore this topic and how this metadata may be used to augment the possibilities and reliability of various refactoring tasks.

4.3.3. Broaden usage of existing concepts

The analysis infrastructure and transformation techniques established during this master thesis may provide valuable benefits for other areas of the refactoring plug-in. Most custom search visitors frequently used in test cases, as an example, could easily be refactored to use a *PatternSearchVisitor*, greatly reducing the resulting amount of code. Furthermore, a follow-up project ot this thesis should focus on whether some transformation algorithms could be called explicitly by the user, say to replace a marked statement by a lambda or STL/TR1 bind functor. Such features could provide remarkable benefits to usability and broad acceptance of these concepts. Since they would not require any implementation but a UI integration of existing transformation features, such a follow-up project would even be suitable for small term projects.

4.3.4. Textual syntax for pattern definition

Other tree pattern implementations such as Tregex support a textual syntax for pattern definition [LA05, p.2]. These syntaxes usually feature a textual representation for each pattern class, so e.g. << would refer to domination, *$*++ to sibship. An according pattern syntax implementation for the DS-8 patterns module may simplify the introduction of new analysis patterns substantially. On a MSc-level project, this task may represent a very promising follow-up project to the current implementation.

4.3.5. Semi-automatic pattern generation for existing code

The implemented patterns for *for_each* and *find_if* recognition have been created based on effective code snippets. While doing so, it has appeared that these patterns very much resemble their code templates. It is imaginable that a semi-automatic transformation of a given algorithm implementation to a relative tree pattern is possible. Semi-automatic, because the user would need to amend the generated pattern by metadata information, such as e.g. replacing an effective variable name by a wildcard. This task would bear two challenging areas: the pattern generation from a given abstract syntax tree and the creation of a suitable user interface providing sophisticated graphical support for this process. Both tasks, probably placed best within a master-level project, would simplify the creation of future analysis modules significantly.

4.3.6. Efficient pattern search algorithm

The current implementation of the pattern search is, as practical as it may be, a very trivial approach based on the visitor pattern. Its complexity strictly amounts to $O(n)$ where n would represent the number of nodes in the abstract syntax tree. Based on the work of Wuu, Lu and Yang [WtLY00], the implementation of a better performing algorithm could be covered as bachelor thesis or master term project. Since performance has not yet been an issue with the current tree pattern matching algorithm during experiments, that project would probably be conducted for the sport of it rather than for actual, practical purposes.

4.3.7. Pattern search mask in Eclipse

As pointed out in chapter 2, tree patterns provide tremendous benefits for plug-in developers, allowing them to traverse trees and extract parts of it very elegantly. But not only plug-in developers, but also Eclipse CDT end-users could use this functionality in various situations. An extension of the default Eclipse search mask, possibly using the syntax recommended in section 4.3.4, could simplify many extended search tasks where simple textual regular expressions fail.

4.3.8. Just about any other plug-in!

While reviewing the goals achieved during this project, I have no doubt that the patterns module has been the one driving force that assured its eventual success. After the patterns module had been implemented, not a day passed without finding another situation or refactoring possibility in any of the plug-in modules, where a *PatternSearchVisitor* could reduce two dozen lines of code to merely three. Just as programmers working in the field of text recognition and manipulation should at the very beginning get accustomed to textual regular expressions, tree pattern matching should be a starting point for every developer planning to take on the Eclipse CDT and its abstract syntax trees.

4.4. Personal review and acknowledgement

A bit more than five months ago, we approached the DS-8 project exceptionally humble, not sure what to expect in results of such a challenging task. Looking back at these past months, it was indeed uncertain whether or not this project would yield any usable product at all. Many of the concepts elaborated during this master thesis have hardly been attended by anyone before, especially not in an Eclipse environment. Many parts of the resulting program, from tree pattern matching algorithms to lambda expression support in Eclipse, needed to be created from scratch, for no comparable functionality was available at that point. Even worse, the available algorithms and framework features never seemed to fit the project's actual requirements, e.g. providing string-based operations where proper abstract syntax trees were necessary.

Nevertheless, there also have been positive aspects to the Eclipse environment found present. The unsatisfying framework operations could eventually be reused by re-parsing their textual results into actual AST entities. Additionally, the *CodAn* framework represented one of the few Eclipse resources that fitted the project's requirements like a glove, greatly simplifying the integration of the created algorithms into Eclipse. At this point I would also like to thank Emanuel Graf, Lukas Felber, Mirko Stocker and the whole IFS team at HSR Rapperswil for always extending a helping hand in order to master the Eclipse plug-in development environment. Thanks also to Michael Rüegg for his elaborate review of this report and professor Dr Josef Joller for his support in redefining the tree patterns package. And finally, I also thank my professor, Peter Sommerlad, who supported and encouraged me during this project from the very beginning.

If there are any lessons to be learned from this master thesis, it is the fact that semantic code analysis lives up to its reputation of being an unsolvable and therefore infinitely complex task. And the fact that, nevertheless, no one should shy from this task, since projects such as DS-8 prove that there can still be remarkable results and practical benefits to gain.

A. Development environment installation

The following sections provide a complete installation guide for all relevant systems and programs used during this master thesis. This description is intended to support future students in preparing their working environment and to allow them to focus on the actual task at hand.

A.1. SVN server

Striving to meet the technological standards of a University of Applied Sciences, this project emphasizes the use of an SSL-enabled access to an HTTP-based SVN server. Based on the following installation guides, Apache 2 has been used as a basis for aforesaid configuration:

- `http://ubuntuforums.org/showthread.php?t=51753`
- `https://bugs.launchpad.net/ubuntu/+source/apache2/+bug/77675/+attachment/88984/+files/apache2-ssl.tar.gz`

A.2. Hudson build server

During the early phases of this master thesis, it remained still unsure whether there would even follow an actual implementation of the researched transformation techniques. It was sure, however, that if there were any development tasks, they would have to comply to the test-driven continuous development methodology HSR Rapperswil is renowned for. Against this backdrop, Hudson CI was chosen as build server software for its formidable integration with the Apache Maven build system (see also A.5) (`http://hudson-ci.org/`).

A.3. Trac

To provide access to shared documentation entities and progress reports, the open source project managament and bug-tracking tool TRAC has been used (`http://trac.edgewall.org/`).

A.4. Eclipse PDE

The Eclipse plug-in development environment (PDE) all by itself is a very simple construct. It basically represents a default Eclipse platform having additional plug-ins useful for Eclipse plug-in development installed. Apart from the very simple installation, the only matter to be aware about in this context is which PDE version to select for development.

A.4.1. Automatic dependency resolution

When developing an Eclipse plug-in from within Eclipse, it is important to notice that many plug-ins and libraries the newly created plug-in might depend on are already loaded within the running Eclipse environment. This allows Eclipse to satisfy them automatically wherever possible. While this may provide remarkable convenience to the user, it is also a source of potential errors when developing in a team with different Eclipse versions installed or when trying to create the plug-in on a build server, where no Eclipse installation is present and this automatic resolution fails. Version mismatches can lead to anomalities where the build may succeed on one computer, but fail on others or the build server.

Eclipse proposes multiple solutions for dependency-aware development:

- Use a separate Eclipse installation as a target platform, which has to be exactly the same for all contributors
- Provide exact dependency specifiers in the MANIFEST.MF file for every dependency present - even Eclipse core plug-ins.
- Agree on a specific CVS version level both as development environment and target platform and use repository-based dependency resolution

Using separate target platforms introduces additional complexity to the project as well as more room for version mismatch-based errors. Additionally, it provides little to no support for running an automated build server. Exact dependency specifiers in the MANIFEST.MF unnecessarily increase the complexity of the dependency hierarchy and are error-prone when it comes to indirect dependencies, where again version mismatches may occur.

The most conservative and successful conclusion in this context is the idea that Eclipse's automatic dependency resolution by itself is very reliable and comfortable. The only necessities are assuring that the build environment complies to the resolved libraries and that a build server without Eclipse installed has access to them. This strategy is called repository-based dependency resolution and implemented by the Eclipse Tycho plug-in [Son10].

A.4.2. Selecting a version

According to the findings in the previous subsection, it is important to download an Eclipse PDE version from the Eclipse download page that is as compatible as possible to a recent source code version in the Eclipse CVS repository. As this master thesis focuses on developing Eclipse CDT plug-ins, the repository in question will be the Eclipse CDT repository (`:pserver:anonymous@dev.eclipse.org:/cvsroot/tools`). To download the initial set of all necessary plug-ins for CDT development, Eclipse CDT developers have provided a project set file that automatically downloads and installs the plug-ins as workspace projects when opened with Eclipse (`http://www.eclipse.org/cdt/psf/cdt-main.psf`).

After having created these plug-ins projects, their CVS version can be switched using the "Team" context menu, as figure A.1.

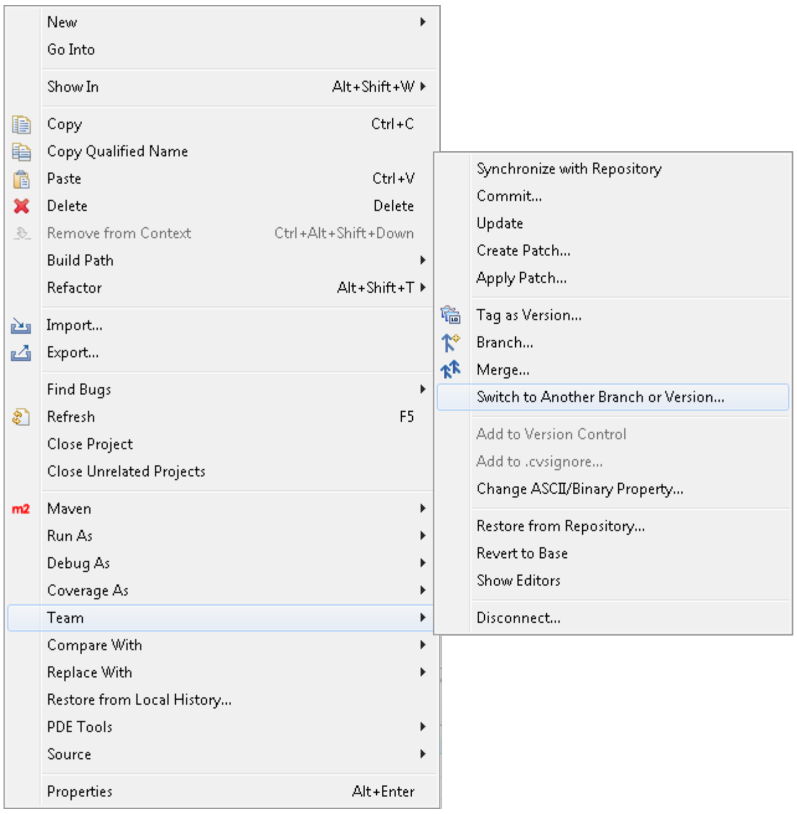
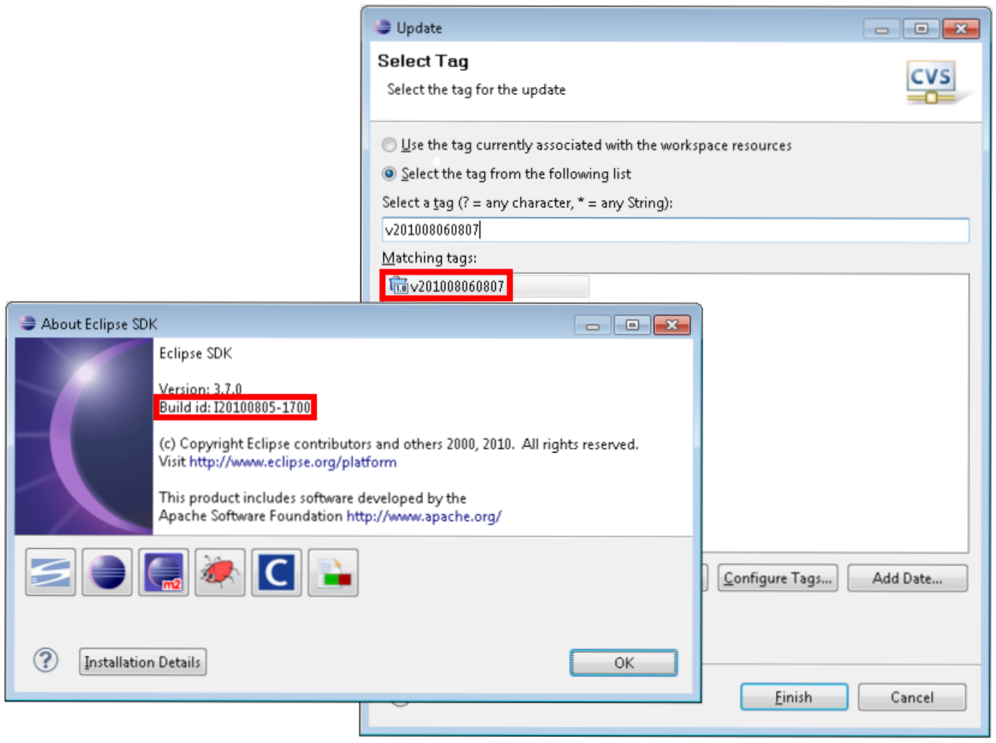


Figure A.1.: Switch CVS version

As mentioned before, the version selected here should be as recent as possible and yet as compatible as possible to a downloadable Eclipse PDE distribution (`http://download.eclipse.org/eclipse/downloads/`). "Compatible" in this context describes version number similarity. Figure (A.2) provides an example of an Eclipse installation where a very close match has been achieved.



Select Tag

Select the tag for the update

( ) Use the tag currently associated with the workspace resources

(●) Select the tag from the following list

Select a tag (? = any character, * = any String): v201008060807

Matching tags: v201008060807

Eclipse SDK

Version: 3.7.0
Build id: I20100805-1700

(c) Copyright Eclipse contributors and others 2000, 2010. All rights reserved.
Visit http://www.eclipse.org/platform

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

Figure A.2.: Eclipse installation with very close match between program and CVS version

Once downloaded and successfully installed, Eclipse will use the installed projects to satisfy its dependencies. Note that the method described in this section actually requires an Eclipse PDE version already installed for browsing the CVS repository for compatible versions. This PDE installation is afterwards replaced by the chosen one. Alternatively, any other CVS browser can be used to determine the desired version.

A.4.3. Additional plug-ins

The additional Eclipse plug-ins listed in table A.1 have been used for all development tasks during this master thesis:

Name	Description	URL
Subclipse	Subclipse is an Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE.	`http://subclipse.tigris.org/update_1.6.x`
m2eclipse	Maven build system integration for Eclipse.	`http://m2eclipse.sonatype.org/sites/m2e`
FindBugs	Static analysis program to identify bugs in Java code.	`http://findbugs.cs.umd.edu/eclipse`
EclEmma	Free Java code coverage tool for Eclipse.	`http://update.eclemma.org`
Eclipse Metrics	Code Metrics and dependency analysis tool.	`http://metrics.sourceforge.net/update`

Table A.1.: Additional Eclipse plug-ins used

A.5. Tycho

As introduced in section A.4.2, Tycho is a repository-based dependency resolution and build environment for Eclipse plug-ins [Son10]. It is implemented as Apache Maven plug-in (`http://maven.apache.org/`) and provides the following features to an Eclipse project:

- Dependency resolution using Maven and Eclipse (p2) repositories (`http://wiki.eclipse.org/Equinox_p2_Getting_Started`)
- Integrated Eclipse plug-in tests including Surefire test report generation (`http://maven.apache.org/plugins/maven-surefire-plugin/`)
- Plug-in feature generation and publication
- Eclipse update site compilation

These remarkable features come by the negligible price of adhering strictly to the Tycho project layout. Unfortunately, Tycho has just yet been accepted as official Eclipse project and is thus still very poorly documented. A complete description of the required project layout and meta-files is therefore provided in the following subsections.

A.5.1. Project layout

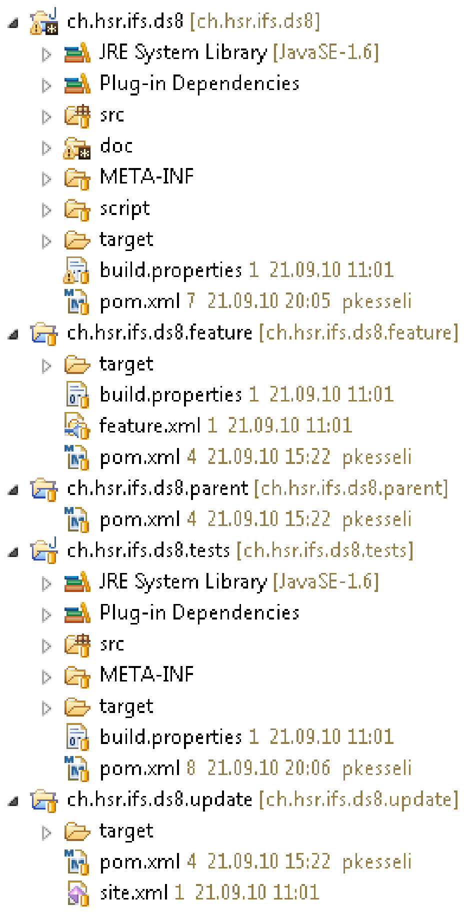
A full Tycho-based build requires your project to be shaped in the following manner:

- Parent

  - Meta-data-only Eclipse project without any natures (even Java), summarizing all participating sub-projects in a single POM-file for Maven to run.

- Main plug-in
  - “Eclipse plug-in” project, containing the plug-in under development.
- Plug-in tests
  - Test project containing all unit tests for the main plug-in. This project should be of the “Eclipse plug-in” project type and runnable as Eclipse plug-in unit test.
- Feature
  - “Eclipse feature” project, referencing the main plug-in project.
- Update site
  - Referencing both the main plug-in and feature projects, this artifact of type “Eclipse update site” project generates a directly accessible update site for convenient installation.

Figure A.3 provides a graphical example of a Tycho-based Eclipse project layout.



```
ch.hsr.ifs.ds8 [ch.hsr.ifs.ds8]
    JRE System Library [JavaSE-1.6]
    Plug-in Dependencies
    src
    doc
    META-INF
    script
    target
    build.properties 1  21.09.10 11:01
    pom.xml 7  21.09.10 20:05  pkesseli
ch.hsr.ifs.ds8.feature [ch.hsr.ifs.ds8.feature]
    target
    build.properties 1  21.09.10 11:01
    feature.xml 1  21.09.10 11:01
    pom.xml 4  21.09.10 15:22  pkesseli
ch.hsr.ifs.ds8.parent [ch.hsr.ifs.ds8.parent]
    pom.xml 4  21.09.10 15:22  pkesseli
ch.hsr.ifs.ds8.tests [ch.hsr.ifs.ds8.tests]
    JRE System Library [JavaSE-1.6]
    Plug-in Dependencies
    src
    META-INF
    target
    build.properties 1  21.09.10 11:01
    pom.xml 8  21.09.10 20:06  pkesseli
ch.hsr.ifs.ds8.update [ch.hsr.ifs.ds8.update]
    target
    pom.xml 4  21.09.10 15:22  pkesseli
    site.xml 1  21.09.10 11:01
```

Figure A.3.: Example project featuring Tycho layout

A.5.2. Maven POM configuration files

Each of the previously describe projects requires a separate pom.xml descriptor for being built by Maven. Fortunately, none of these descriptors contains more than half a dozen configuration properties and look almost the same for every possible project. The different descriptors are thus listed in the following and their contents explained where applicable.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ch.hsr.ifs</groupId>
  <artifactId>ch.hsr.ifs.ds8.parent</artifactId>
  <version>0.0.1</version>
  <packaging>pom</packaging>
  <properties>
    <tycho-version>0.9.0</tycho-version>
  </properties>
  <repositories>
    <repository>
      <id>helios-release</id>
      <url>http://download.eclipse.org/releases/helios/</url>
      <layout>p2</layout>
    </repository>
    <repository>
      <id>helios-cdt-release</id>
      <url>http://download.eclipse.org/tools/cdt/releases/helios/</url>
      <layout>p2</layout>
    </repository>
  </repositories>
  <modules>
    <module>../ch.hsr.ifs.ds8</module>
    <module>../ch.hsr.ifs.ds8.tests</module>
    <module>../ch.hsr.ifs.ds8.feature</module>
    <module>../ch.hsr.ifs.ds8.update</module>
  </modules>
  <build>
    <plugins>
      <plugin>
        <groupId>org.sonatype.tycho</groupId>
        <artifactId>tycho-maven-plugin</artifactId>
        <version>\${tycho-version}</version>
        <extensions>true</extensions>
      </plugin>
      <plugin>
        <groupId>org.sonatype.tycho</groupId>
        <artifactId>target-platform-configuration</artifactId>
        <version>${tycho-version}</version>
        <configuration>
          <resolver>p2</resolver>
          <environments>
            <environment>
              <os>linux</os>
              <ws>gtk</ws>
              <arch>x86</arch>
            </environment>
            <environment>
              <os>linux</os>
              <ws>gtk</ws>
              <arch>x86_64</arch>
            </environment>
            <environment>
              <os>win32</os>
              <ws>win32</ws>
              <arch>x86</arch>
            </environment>
            <environment>
              <os>win32</os>
              <ws>win32</ws>
              <arch>x86_64</arch>
            </environment>
            <environment>
              <os>macosx</os>
              <ws>cocoa</ws>
              <arch>x86_64</arch>
            </environment>
          </environments>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing A.1: Tycho parent POM example

Listing A.1 shows an example parent POM configuration, whereof the following fields have been highlighted in red:

- Bundle version
  - Chosen version number. Should match the version specifiers in MANIFEST.MF.
- Tycho version
  - Defined as an environment variable, this version number is assured to be the same in all sub-configuration files and in this example equals to 0.9.0, at the time of writing Tycho's latest release candidate.
- Repositories
  - Depending on the type and version of the plug-in in development, additional Eclipse p2 repositories should be configured here, allowing Tycho to implement its automatic dependency resolution.
- Modules
  - Only a matter of formality, all collaborating POMs for this build are to be registered in this section with there relative paths.
- Environments
  - All desired target platforms are to be configured in this section. Tycho will then build the plug-in for each platform using the platform-dependent Eclipse packages and check for build warnings and errors.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
      org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.ds8.parent</artifactId>
    <version>0.0.1</version>
    <relativePath>../ch.hsr.ifs.ds8.parent</relativePath>
  </parent>
  <groupId>ch.hsr.ifs</groupId>
  <artifactId>ch.hsr.ifs.ds8</artifactId>
  <version>0.0.1.qualifier</version>
  <packaging>eclipse-plugin</packaging>
</project>
```

Listing A.2: Tycho main plug-in POM example

The configuration file in listing A.2 presents itself already more straightforward than its predecessor. Important fields are highlighted in red and described in the following:

- Parent
  - A pendant to the parent POM's "modules" field, this setting should provide a link to the project's parent POM.
- Packaging
  - Tycho uses Eclipse-specific packaging types, each accounting for a specific Eclipse artifact. The packaging type for the main plug-in is called eclipse-plugin.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.ds8.parent</artifactId>
    <version>0.0.1</version>
    <relativePath>../ch.hsr.ifs.ds8.parent</relativePath>
  </parent>
  <groupId>ch.hsr.ifs</groupId>
  <artifactId>ch.hsr.ifs.ds8.tests</artifactId>
  <version>0.0.1.qualifier</version>
  <packaging>eclipse-test-plugin</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.sonatype.tycho</groupId>
        <artifactId>maven-osgi-test-plugin</artifactId>
        <version>${tycho-version}</version>
        <configuration>
          <testSuite>ch.hsr.ifs.ds8.tests</testSuite>
          <testClass>ch.hsr.ifs.ds8.tests.TestSuite</testClass>
          <useUIHarness>true</useUIHarness>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing A.3: Tycho test plug-in POM example

Listing A.3 illustrates an example of a Tycho test project configuration. Settings to notice by possible users are highlighted in red and listed below:

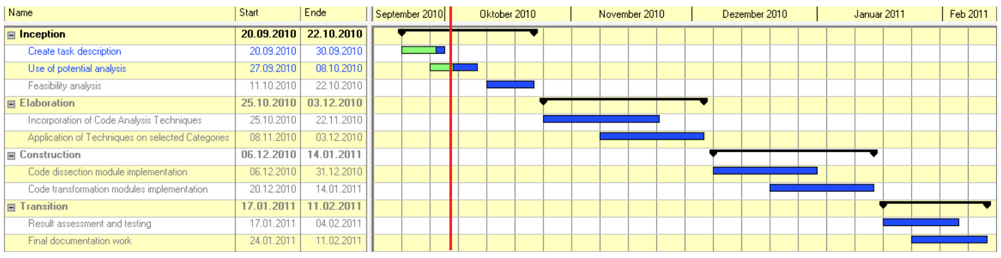
- Parent
  - A pendant to the parent POM's "modules" field, this setting should provide a link to the project's parent POM.
- Packaging
  - The Tycho eclipse-test-plugin package automatically creates a set of surefire XML reports from the TestSuite stated in the build configuration.
- TestSuite

  - Represents a reference to the eclipse-plugin-test module's name, i.e. should equal "artifactId"

- TestClass
  - Java class reference to your JUnit test suite class

B. Project management

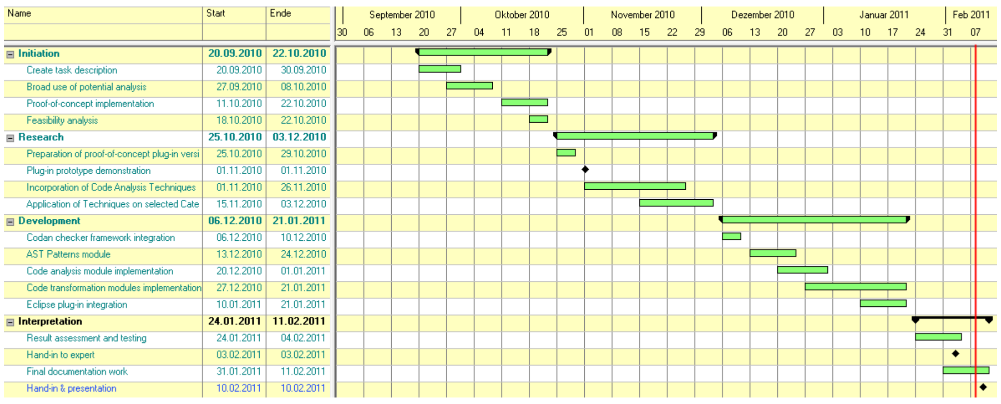
Given the fact that this project set foot in relatively unknown ares, a large portion of the planned work load has been allotted towards research tasks. In fact, the initial project plan proposed equal shares of research and implementation tasks, as shown by the project plan snapshot in figure B.1.



Name	Start	Ende
Inception	**20.09.2010**	**22.10.2010**
Create task description	20.09.2010	30.09.2010
Use of potential analysis	27.09.2010	08.10.2010
Feasibility analysis	11.10.2010	22.10.2010
Elaboration	**25.10.2010**	**03.12.2010**
Incorporation of Code Analysis Techniques	25.10.2010	22.11.2010
Application of Techniques on selected Categories	08.11.2010	03.12.2010
Construction	**06.12.2010**	**14.01.2011**
Code dissection module implementation	06.12.2010	31.12.2010
Code transformation modules implementation	20.12.2010	14.01.2011
Transition	**17.01.2011**	**11.02.2011**
Result assessment and testing	17.01.2011	04.02.2011
Final documentation work	24.01.2011	11.02.2011

Figure B.1.: Project plan snapshot (September 30, 2010)

This estimation proved to be very accurate. The relation between research and implementation work shifted slightly towards more implementation work as the project evolved, since many research tasks represented explorative work that issued practical benefits for the implementation part as well. However, this did not affect the overall picture of the work load distribution, as the last iteration meeting's project plan snapshot in figure B.2 illustrates. Both research and implementation tasks remained throughout the whole project two equally important working areas.



Name	Start	Ende
Initiation	**20.09.2010**	**22.10.2010**
Create task description	20.09.2010	30.09.2010
Broad use of potential analysis	27.09.2010	08.10.2010
Proof-of-concept implementation	11.10.2010	22.10.2010
Feasibility analysis	18.10.2010	22.10.2010
Research	**25.10.2010**	**03.12.2010**
Preparation of proof-of-concept plug-in versi	25.10.2010	29.10.2010
Plug-in prototype demonstration	01.11.2010	01.11.2010
Incorporation of Code Analysis Techniques	01.11.2010	26.11.2010
Application of Techniques on selected Cate	15.11.2010	03.12.2010
Development	**06.12.2010**	**21.01.2011**
Codan checker framework integration	06.12.2010	10.12.2010
AST Patterns module	13.12.2010	24.12.2010
Code analysis module implementation	20.12.2010	01.01.2011
Code transformation modules implementation	27.12.2010	21.01.2011
Eclipse plug-in integration	10.01.2011	21.01.2011
Interpretation	**24.01.2011**	**11.02.2011**
Result assessment and testing	24.01.2011	04.02.2011
Hand-in to expert	03.02.2011	03.02.2011
Final documentation work	31.01.2011	11.02.2011
Hand-in & presentation	10.02.2011	10.02.2011

Figure B.2.: Project plan snapshot (February 8, 2011)

HSR Rapperswil awards a Master thesis with 27 ECTS-credits, which should amount to approximately 810 working hours. Split over a period of 20 weeks, this results to a minimum of 40.5 hours per week. The effective time invested into this project exceeds this minimum easily: Approximately 930 working hours have been invested into this master thesis. Figure B.3 shows this relation graphically.

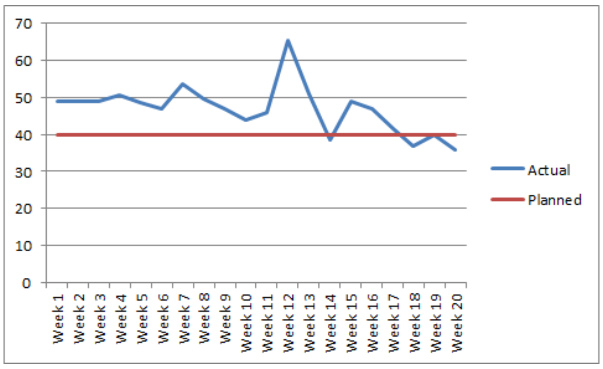


Figure B.3.: Working hours per week

Figure B.3 features two remarkable work load peaks in weeks 12 and 15. The first one relates to the implementation of the tree patterns package, which has been considered too challenging to be implemented during this project. Thanks to one student's stubbornness and a tremendous increase in work load, this has been proven to be untrue. The second peak in week 15 relates to the "evaluation" of *bind* transformations, which in the end became a fully-fledged implementation of both TR1 *bind* and STL *bind1st*/*bind2nd* transformations.

Concludingly, it is safe to say that the project plan has been very accurate and the work load played out almost exactly as predicted. Nevertheless, without the extra time invested into this project, the plug-in would not provide the remarkable features it does now. Therefore, I leave it to Leonard Bernstein [Thi10] to conclude this section:

"To achieve great things, two things are needed: a plan, and not quite enough time."

Bibliography

[ACEP10] Mahmood Ali, Telmo Correa, Michael D. Ernst, and Matthew M. Papi. The checker framework: Custom pluggable types for java. Technical report, University of Washington, 2010.

[Baj01] Samir Bajaj. C++ and STL: Take advantage of STL algorithms by implementing a custom iterator. *MSDN Magazine*, April 2001. http://msdn.microsoft.com/en-us/magazine/cc301955.aspx.

[Cor10a] Microsoft Corporation. Expressions (C++). Website access October 11 2010, 2010. http://msdn.microsoft.com/en-us/library/625x66bt.aspx.

[Cor10b] Microsoft Corporation. Statements (C++). Website access October 11 2010, 2010. http://msdn.microsoft.com/en-us/library/bzzyh1y4.aspx.

[Fet09] David Fetter. High performance SQL with postgreSQL 8.4. Website access November 2 2010, 2009. http://assets.en.oreilly.com/1/event/27/High%20Performance%20SQL%20with%20PostgreSQL%20Presentation.pdf.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matchin in trees. Technical report, Purdue University, 1982.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, second edition, August 2008.

[ISO10a] ISO. Programming languages - C++. Technical report, International Organization for Standardization, 3 2010.

[ISO10b] ISO. Working draft, standard for programming language C++. Technical report, International Organization for Standardization, 11 2010.

[LA05] Roger Levy and Galen Andrew. Pattern matchin in trees. Technical report, University of Edinburgh and Microsoft Research Team, 2005.

[Lov10] Tim Love. CUED talk: C++ and the STL (standard template library). Technical report, University of Cambridge, January 2010. http://www.eng.cam.ac.uk/help/tpl/talks/C++.html.

[Mad05] John Maddock. TR1 by subject - function object binders. Website access October 10 2010, 2005. http://www.boost.org/doc/libs/1_39_0/doc/html/boost_tr1/subject_list.html#boost_tr1.subject_list.bind.

[Mar10] Alessio Marchetti. Hyperlinked C++ BNF grammar. Website access October 11 2010, May 2010. http://www.nongnu.org/hcb.

[Mey01] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, June 2001.

[Mic10] Daniel Michel. Lambdas in modern programming languages. Technical report, Hochschule für Technik Rapperswil, 2010.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, second edition, 2006.

[Son10] Inc. Sonatype. Tycho overview, 2010. http://tycho.sonatype.org/.

[Thi10] ThinkExist. Leonard bernstein quotes. Website access February 2 2011, 2010. http://thinkexist.com/quotes/leonard_bernstein/.

[Wil05] Brian Wilson. Why C++ templates (and STL) are bad. Website access September 27 2010, 5 2005. http://www.ski-epic.com/templates_stl_rant/index.html.

[WtLY00] Hsiao-Tzu Lu Wuu, Hsiao tzu Lu, and Wuu Yang. A simple tree pattern-matching algorithm. In *In Proceedings of the Workshop on Algorithms and Theory of Computation*, 2000.

[WY07] Daniel Waddington and Bin Yao. High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64–78, September 2007. http://portal.acm.org/citation.cfm?id=1288023.