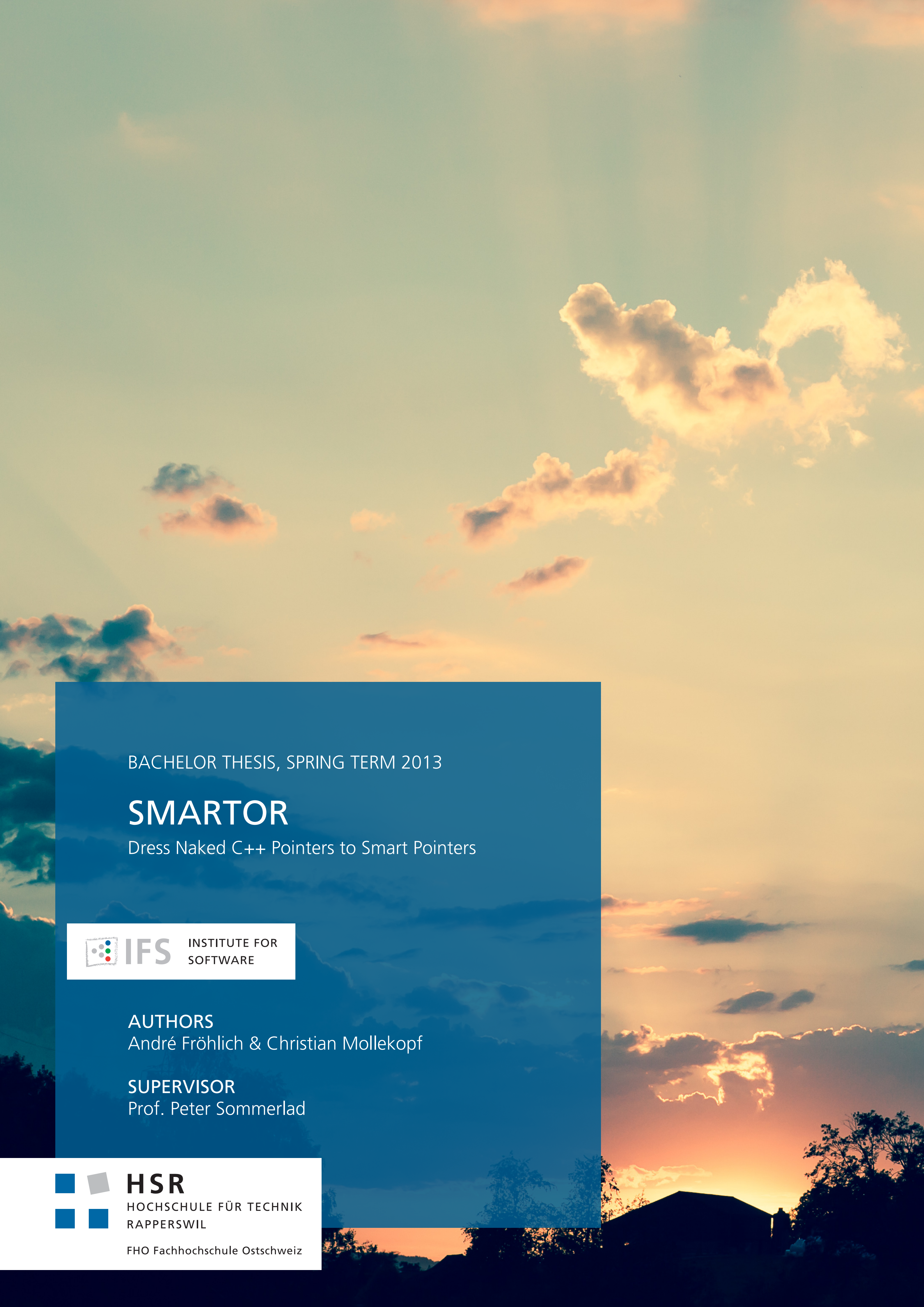


BACHELOR THESIS, SPRING TERM 2013

SMARTOR

Dress Naked C++ Pointers to Smart Pointers

IFS INSTITUTE FOR SOFTWARE

AUTHORS
André Fröhlich & Christian Mollekopf

SUPERVISOR
Prof. Peter Sommerlad

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelor Thesis

Smartor

Dress Naked C++ Pointers to Smart Pointers

André Fröhlich, Christian Mollekopf

Spring Term 2013

Supervised by Prof. Peter Sommerlad

Abstract

C++ allows the use of raw pointers as they exist in plain C. Unfortunately, they can't demonstrate the semantic role of their usage, as they are a very low level concept. Furthermore, they often lead to memory leaks and hard to understand source code.

Better alternatives in C++ are smart pointers. Smart pointers are a group of wrappers around raw pointers. They provide a facility that covers the memory management and adds semantic information on what exactly the pointer is used for. The goal of our project is to develop an Eclipse CDT plug-in assisting a developer in finding and converting raw pointers into smart pointers. In addition, it serves as an experiment on how the Scala language helps to write code that manipulates an abstract syntax tree.

Our thesis contains an analysis of pointer roles and their possible transformations into smart pointers. The plug-in assists a developer in detecting and transforming raw pointers into suitable smart pointers by displaying a marker in the editor and offering appropriate quick fixes. Using Scala resulted in cleaner and more readable source code. Scala's match statement provides a neat way to match against types thus helps avoiding conditional statements for type checking.

Management Summary

This thesis describes how error prone C++ naked pointers can be automatically converted into C++ smart pointers as well as the development of an Eclipse CDT plug-in assisting a developer on this task. This chapter gives a brief overview of the motivation, the goals and the results of this thesis.

Motivation

C++ supports the use of pointers. A pointer is a data type whose value refers directly to another value stored elsewhere in the computer memory using its address[1]. Every time a pointer is used, it has a special role describing its usage. However, it is not possible to demonstrate the role with a pointer since it is a low level concept. This leads to hard to understand code.

In C++ better alternatives are smart pointers. The use of smart pointers can demonstrate the role of the pointer and manage resources. with these problems and provides a high lever concept. A smart pointer is an abstract data type that simulates a pointer while providing automatic memory management and other features[2]. Different type of smart pointers provides different semantic meanings such as sharing a reference or transfering ownership.

[1]Wikipedia, *Pointer (computer programming)*.
[2]Wikipedia, *Smart pointer*.

Goals

The goal of this thesis is to analyse the different pointer roles and how they can be used to replace plain pointer by developing an Eclipse plug-in for this task. Many of these transformations are non-trivial and require a careful analysis. The main focus is the anlysis of possible pointer roles and their relation with newer C++ smart pointers.

Usually Eclipse plug-ins are written in Java. For this thesis we decided to examine how the Scala programming language helps to write such a plug-in as it provides a build in pattern matching mechanism. Using Scala, we hoped to be able to write cleaner and less cluttered code without the usual boiler plate Java requires. In addition we wanted to expand our knowledge with the functional programmming paradigm Scala uses.

Results

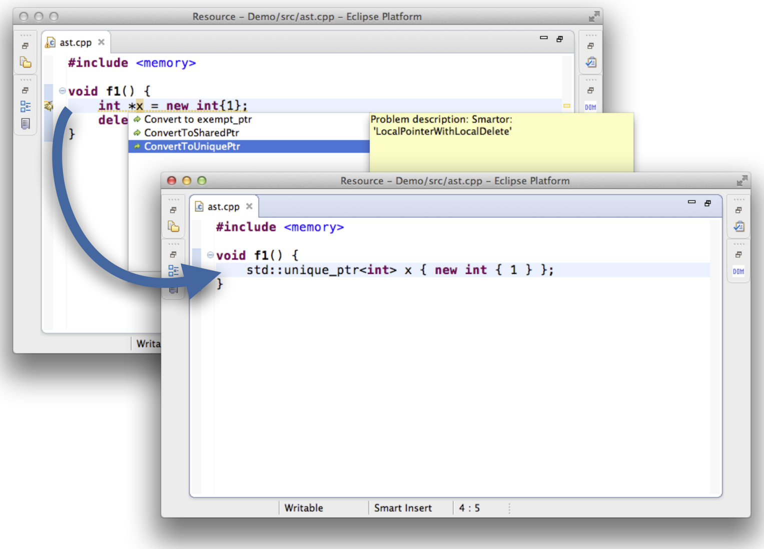
Our plug-in assists a developer to convert raw pointers into smart pointers. It is able to find occurrences and mark them in the editor using a marker. The plug-in also provides a quick fix for various cases. Activating a quick fix transforms the code to get rid of the raw pointer (1) and replaces it with the appropriate smart pointer (2). The plug-in is also able to detect and remove local deallocation code as it becomes invalid using a smart pointer.

```
int *x = new int{1};
delete x;
```

Listing 1: Before the transformation

```
int x = std::unique_ptr{new x{1}};
```

Listing 2: After the transofrmation



```
#include <memory>

void f1() {
    int *x = new int{1};
    dele
}
```

Problem description: Smartor:
'LocalPointerWithLocalDelete'

```
#include <memory>

void f1() {
    std::unique_ptr<int> x { new int { 1 } };
}
```

Conversion into a unique pointer

Further Work

Pointers to strings and arrays are special cases and would be an interesting subject for further studies and a useful extension for the plug-in. The heuristic to determine ownership can also be extended and would be a valueable addition. Furthermore, pointer tracking using symbolic execution could be greatly beneficial.

Declaration of Authorship

We declare that this thesis and the work presented in it was done by our own and without any assistance, execept what was agreed with the supervisor. All consulted sources are clearly mentioned and cited correctly. No copyright-protected materials are unauthorizedly used in this work.

Place and date	André Fröhlich
Place and date	Christian Mollekopf

Contents

1. Introduction

At the University of Applied Sciences Rapperswil a bachelor thesis is usually done by groups of two students. This chapter covers the project duration, the contents of this report and its target audience.

1.1. Project Duration

This project started with the beginning of the spring term 2013 on February 18th. The kickoff meeting was on February 20th.

This report has to be handed in by June 14th 12:00. On the same day the exhibition of the bachelor thesis takes place.

The date for the final demonstration of this thesis is set on June 24th 2013.

1.2. Report Contents

This report starts with the mandatory abstract and management summary chapters. After this introduction the objectives are described in a separate chapter. The analysis chapter shows our investigation on pointer roles and their possible transformation into smart pointers. Implementation details and notes are described in the following chapter. This reports ends with a chapter about our conclusion and personal view on this project. The appendix contains an overview over the environment the plug-in was developed in. It also contains a brief introduction to developers interested in picking up this project for further work. A user manual shows how the plug-in can be installed and used. The appendix ends with the nomenclature and bibliography.

1.3. Target Audience

Readers are expected to have the knowledge a student of the University of Applied Sciences Rapperswil usually has after study for four semesters.

Developers interested in working with our code must know some basic Scala concepts. This report does not contain an introduction to the Scala language. For a quick introduction we recommend "A Tour of Scala"[1] found on the Scala website. For a deeper introduction, we recommend "Programming in Scala"[2] by Martin Odersky, Lex Spoon and Bill Venners.

[1]scala-lang.org, *A Tour of Scala*.
[2]Odersky, Spoon, and Venners, *Programming in Scala*.

2. Objectives

2.1. Motivation

Using raw pointers in C++ is complex, error prone, and results often in hard to understand code. Manual resource management can become very complex as the number of code-path's grows, and consequently becomes a maintenance burden as even minor changes to the code could potentially break the resource management. Further, pointers can have various semantics, including C-implementation of a reference, strings, arrays, optional return values or arguments, ownership transfer, etc. Unfortunately there are only few tools for the developer to communicate the intended meaning. This results in unclear responisbilities in programming interfaces which can ultimately lead to dangling pointers, double deletes, memory leaks and alike. Smart-pointers address those problems, by giving the pointer a type, expressing the meaning, and by moving the resource-management to the library. This leads to more expressive code.

While smart-pointers have been long existing in various third-party libraries, C++11 now standardized `std::unique_ptr` and `std::shared_ptr`, removing the last barrier from widespread use of smart-pointers.

2.2. Vision

The goal of this project is to analyse in which cases raw pointers can be replaced with which smart-pointer and to develop an Eclipse CDT plug-in, that helps developers to spot existing uses of raw pointers and fix them with a corresponding smart-pointer. As many of those transformations are non-trivial, the goal is to identify possible transformations and to implement only a subset deemed feasible.

The plugin should for instance handle the following cases:

- raw pointers passed as arguments, where the function doesn't take ownership of the resource, should be converted to a configurable smart-pointer (`non_owning_ptr`), which is likely to be standardized in upcoming versions of the C++ standard.
- raw pointers passed as arguments, where the function does take ownership of the resource, should be converted to `std::unique_ptr` .
- `char *` should be converted to `std::string`, but only if the pointer owns the resource.
- non-owning `char *` should be converted to a configurable non-owning string reference type (`string_ref`), which will also be standardized in a future C++ standard version.

A further case which has to be researched is the usage of pointer-based arrays where the operators `new[]` / `delete[]` are used, as opposed to the single object `new` / `delete` .

C++11 provides several new memory management features. Move semantics have been introduced, as alternative to passing by reference respectively copying. This new feature enables replacing many uses of references and pointers, where move semantics would have been required. Among the new smart-pointers one can find `std::shared_ptr`, which is a traditional reference counted, shared pointer. `std::unique_ptr` provides a move-only pointer replacing `std::auto_ptr`, which i.e. tried to emulate move semantics using the pre-C++11 tools, resulting in somewhat unexpected/awkward behaviour. Upcoming C++ versions will enhance this further by providing pointers for non-owning pointers and non-owning string references (`string_ref`), as a replacement for `char*` .

Since third party libraries provide similar pointers, i.e. boost, the used smart-pointer types should be configurable as far as possible. This includes factory functions such as `make_shared` which would have to be adapted accordingly. Further the header files for smart-pointer definitions must be managed so the conversion can be executed fully automated.

2.3. Focus

The plug-in will be developed in Scala, a programming language with functional-programming aspects, that runs in the Java-VM and supports mixing Java and Scala code. This is required for working with the Eclipse Plug-in Framework. Scala was chosen for educational purposes and it's pattern-matching capabilities which could become useful for processing the abstract syntax tree.

The project will be developed using Maven as buildsystem and the development will be supported by a bugtracker and a fully automated buildserver for continuous integration.

A project report with management summary and abstract will be written to document the project. Further deliverables will be provided according to the HSR bachelor thesis requirements.

Example

The following example shows a function that takes a raw pointer as parameter x, modifies x's pointee using a member-function call, and returns a heap allocated resource of type T2.

```
T2* f(T1* const x)
{
    if (x)
        x->set(1);
    return new T2();
}
```

x is a non-owning reference to a object, since f contains no delete and x's pointee is modified, so it should be converted to a non_owning_ptr to make the semantics clear. By using non_owning_ptr we make explicitly clear that f doesn't take ownership and that it may modify x's pointee. Since x may be null, we also can't convert x to a reference. The return value could be either a `std::unique_ptr`, or a `std::shared_ptr` . Which one is suitable is not visible without more context, but for this example we'll assume `std::shared_ptr` is what we want. The converted result would therefore look like this:

```
std::shared_ptr<T2> f(non_owning_ptr<T1>
    const x)
{
    if (x)
        x->set(1);
    return make_shared<T2>();
}
```

In the result the semantics and responsibilities of f and the caller are much clearer. It's clear that x may be modified by f and that x is still valid after the call to f. It is also clear that the caller of f owns the returned object, and the potential memory leak (if the caller doesn't handle the return value), is fixed.

2.4. Agreement

The following parties confirm the validity of the tasks described in this chapter.

Place and date	Prof. Peter Sommerlad
Place and date	André Fröhlich
Place and date	Christian Mollekopf

3. Analysis

This chapter analyses available smart pointers and their applicability to various scenarios where raw pointers may be used.

The first section introduces the problem domain by analyzing the various roles a pointer can take and how a raw pointer could be better represented using smart pointers and other language constructs.

The second section presents a set of common smart pointers that are currently available, and explains their typical usage.

The third section analyses how suitable smart pointers for a raw pointer could be chosen and describes an approach to automatically select a smart pointer based on the ownership of the raw pointer.

Finally, the fourth section analyses a set of concrete code examples and how they could be refactored. This analysis is then meant to serve as foundation for the implementation.

3.1. Pointer Roles

Considering the following pointer declaration:

```
T *p;
```

A raw pointer can have a variety of roles, but gives no indication which roles the author of the code intended for it. It is thus left to the reader of the code to figure out what the author's intentions were. Without using smart pointers to convey the original intention, all that is left to help the reader are comments in the code and coding conventions, which are both highly error prone as they are up to interpretation and not checkable by the compiler.

A raw pointer declaration can have any role of the following list, or a combination thereof:

- reference
- reassignable reference
- handle to dynamically allocated memory
- optional
- error code
- array
- string

Note how the pointer declaration above doesn't reveal anything about the developers intentions or the roles the pointer could have, and thus all semantics are hidden in the declarations context (comments, how the pointer is used, common patterns, ...).

The following sections explain each of the mentioned roles, where they typically appear, on how they could be represented in a way that conveys the semantics using smart pointers and other intention revealing language constructs such as references.

For a description of the used smart pointers in this section see 3.2 on page 22;

Reference

```
void foo(int *ref) {
    int i = *ref;
}
```

A pointer can be the C-implementation of the C++ reference. A reference is by definition non-owning but can be const or non-const. Further can the reference refer to stack or heap allocated data structures. A reference must be initialized and may not be reassigned (the referenced value can be reassigned of course).

References typically occur as return values, function parameters and local identifiers.

C++ offers a dedicated operator for references:

```
void foo(int &ref) {
    int i = ref;
}
```

By using the reference operator it is clear that `ref` can not be NULL, doesn't own the referenced resource and that the reference can not be reassigned.

Note that a reference can become dangling if the reference lives longer than the resource it is pointing to:

```
int &getLocalVariable() {
    int x;
    return x;
}
```

While this is an obvious example to illustrate the problem, valid scenarios exist where a reference can become dangling. However, in such cases the use of a C++-Reference is not suitable, and other options such as a `std::weak_ptr` should be considered.

Reassignable reference

Since references cannot be reassigned with a new pointee, and must be initialized immediately, it is not always possible to use a C++ reference in place of a raw pointer when the pointer has the role of a reference.

```
int *ref = getRef();
for(int i = 0; i < 3; i++) {
    if (condition(i))
```

```
        ref = getRef(i);
}
*ref = getNewValue();
```

`ref` is initialized with the reference to different values based on `condition()`. Because references must be initialized immediately, and may not be assigned a new value later on, it is not possible to simply replace the use of the pointer with a reference.

In such cases `non_owning_ptr` may be better suited.

```
non_owning_ptr<int> ref  = getRef();
for(int i = 0; i < 3; i++) {
    if (condition(i))
        ref = getRef(i);
}
*ref = getNewValue();
```

`non_owning_ptr` behaves exactly like a raw pointer, but clearly shows the intent that `ref` is a non-owning reference.

Handle to dynamically allocated memory

A raw pointer may represent a handle with ownership to dynamically allocated memory. Having ownership means that the holder of the handle is responsible to release the allocated memory before disposing of the handle.

Typical examples are factory and disposal functions:

```
int *create() {
    return new int(3);
}
void dispose(int *p) {
    delete p;
}
```

The raw pointers returned from `create()` and passed to `dispose()`, own the resource but give no indication that they do.

To better communicate the semantics of the pointer, `std::unique_ptr` and `std::shared_ptr` are available, where `std::unique_ptr` holds sole ownership of the resource and `std::shared_ptr` shared ownership (for further information about the smart pointers see 3.2 on page 22).

```
std::unique_ptr<int> create() {
    return std::unique_ptr<int>(new int());
}
```

By returning a `std::unique_ptr`, it is clear that the caller gets the ownership of the resource, and that the resource is not meant to be shared. `create()` can return a `std::unique_ptr` using move semantics.

```
std::shared_ptr<int> create() {
    return std::make_shared<int>(3);
}
```

By returning a `std::shared_ptr`, it is clear that the caller gets the ownership of the resource, and that the resource is meant to be shared.

Both variants ensure by returning a resource managing smart pointer that the resource is not leaked.

`std::unique_ptr` and `std::shared_ptr` typically occur as return values, function parameters or local variables.

Note that these smart pointers not only communicate the intent of the developer, but also take over the responsibility for the resource management by ensuring that the resource is deleted when no longer required.

Optional

Raw pointers can be used for optional values that are not guaranteed to be initialized. The convention is typically that if the pointer is NULL, the value is not available.

```
void foo(int *value) {
    if (value) {
        addToList(*value);
    }
}
```

Because the `value` can be NULL, it can be used as optional value. Note that this implementation passes `value` by reference.

Optionals typically occur as return values or function parameters.

`boost::optional` provides an explicit way to mark a value as optional.

```
void foo(boost::optional<int> value) {
    if (value) {
        int i = *value;
    }
}
```

`boost::optional` indicates that `value` is indeed optional. Note that this implementation passes `value` by value.

Since smart pointers can be empty, optional can be implemented using smart pointers:

```
void foo(non_owning_ptr<int> value) {
    if (value) {
        int i = *value;
    }
}
```

An optional reference or value can be represented by a `non_owning_ptr` . `value` can be empty, indicating that the value is not available. Note that this notation doesn't give any explicit information that `value` may be empty. Also note that this example passes `value` by reference like the first example.

Error Code

Since an integer can be assigned to a pointer variable, the pointer can be misused to transport other information instead of the address of a pointee. Besides optional values, this property is typically used for return values where pointers are mixed with error codes. In its simplest variation the only error code is NULL, where NULL would be more appropriately represented by `nullptr` in C++11.

```
int *foo() {
    int i = getValue();
    if (error()) {
        return 0;
    }
    return i;
}
```

While this example looks similar to an optional return value, its semantics are slightly different. An optional return value implies that it is entierly valid that no value is returned, where an error code, such as `nullptr`, implies that the call to `foo()` should normally succeed, but may fail. In such cases an exception based error handling may be better suited, clearly distinguishing the normal operation and the error handling code-path.

More advanced versions of such error codes may transport different error codes using the pointer variable:

```
#define ERROR_FOO_FAILED -1
#define ERROR_FOO2_FAILED -2
int *foo() {
    int i = getValue();
    if (error()) {
        return ERROR_FOO_FAILED;
    }
    return i;
}
```

As this usage is a clear misuse of pointers, this pointer role is not considered further by this analysis.

Array

C++ arrays degrade to raw pointers when assigned to a raw pointer variable e.g. when passed to a function with a raw pointer parameter. Hence a raw pointer can have the role of an array.

```
int getFirst(int *array) {
    return array[0];
}
```

An array degrades to a pointer because the identifier of an array is equal to a pointer pointing to the first element of the array (i.e. `array == &(array[0])`). Note however that `array` contains no indication that it is an array, and further doesn't convey any information about its size. It is thus neither possible for the function to check for an out of bounds access nor if `array` even is an array at all.

To at least pass along the information that `array` is an array, the following notation should be preferred:

```
int getFirst(int array[]) {
    return array[0];
}
```

When dynamically allocated, arrays must be allocated using the `new[]` operator, and consequently be deleted using the `delete[]` operator. Failing to do so, and using the normal `delete` operator would lead to only the first element of the array being deleted and in leaking the rest of the array.

Arrays are a special case of a raw pointer, imposing the same problems as pointers to other resources, but with special allocation and deallocation operators. The solutions are thus largely orthogonal.

`std::unique_ptr` already supports array operators (`new[]`/`delete[]`), and `std::shared_ptr` may support them in future versions (see also Section 3.2 on page 22). Additionally `boost::shared_array` exists.

As plain arrays are a very low level concept (essentially pointer arithmetics) they should not normally be used. Instead, the C++

Standard Library[1] provides containers such as `std::vector` and `std::list`, which are in most cases preferred. The containers can be used in conjunction with one of the usual smart pointers when allocated on the heap.

Because smart pointer refactorings for arrays would be largely an orthogonal effort to the refactorings for other resources, and because the use of plain arrays in most modern C++ code is not recommeded anyways, arrays are not considered further by this analysis.

String

Strings are a special case of vectors and as such also degrade to raw pointers when passed to a raw pointer variable.

```
void print(char *string) {
    std::cout << string;
}
```

`string` doesn't convey any information if it is a pointer to a single char, an array of char's or indeed an array of chars representing a string.

The Standard C++ Library thus provides `std::string` as dedicated container for strings.

```
void print(std::string &string) {
    std::cout << string;
}
```

By using the std::string class it is immediately clear that `string` represents a string, `print()` doesn't have to delete the object, and the strings size is available within the object.

Just like a normal reference can a char pointer also be a string reference. const string references are typically used to refer to literals.

```
char const *p = "literal";
```

[1]Wikipedia, *C++ Standard Library*.

Such a literal could be wrapped using the not yet standardized `string_view`[2], which is a non-owning reference to a string.

Although it is not yet possible replace the use of char pointers with a suitable string class in all cases, it is still often possible, and future enhancements to the standard should resolve the missing parts.

String objects can be used with the usual smart pointers when allocated on the heap.

Because smart pointer refactorings for strings would be largely an orthogonal effort to the refactorings for other resources, and because the use of C-Style strings in modern C++ code is not recommeded anyways, strings are not considered further by this analysis.

3.2. Available Smart-Pointers

Various smart pointers are available to express the semantics a raw pointer can have (see table 3.1 on page 13), and to help with the resource management. This section provides an overview of available smart pointers and their semantics.

3.2.1. The purpose of smart pointers

Smart pointers have two primary purposes.

First, the smart pointers that take ownership of a resource also help with the memory management. Two main concepts exist, the sole ownership of a resource, and the shared ownership of a resource. Sole ownership means only one smart pointer ever holds ownership of the resource, and is thus responsible to release the resource before the smart pointer is destroyed. Shared ownership means multiple smart pointers share the ownership of the resource, so the resource is only released once the last smart pointer looses it's reference to it.

[2]Yasskin, *N3609: string_view: a non-owning reference to a string, revision 3*.

Second, they provide a vocabulary for the developer to express his intents in a way that is not only understandable by humans, but also a compiler (and other automated tools). This makes it easier for a developer who has to understand a piece of code, as he doesn't have to guess the roles a pointer has based on how the pointer is used. It also enables automated tools to check if the intended usage is not violated.

3.2.2. Available smart pointers and other intention revealing constructs

The following smart pointers have been considered in this analysis. `optional` is not really a smart pointer, but serves a similar purpose in terms of an intention revealing vocabulary. It was thus added for the sake of completeness.

Smart Pointer	Boost Smart Pointer	Description
std::unique_ptr	boost::scoped_ptr	takes sole ownership, manages lifetime
std::shared_ptr	boost::shared_ptr	shared ownership, manages lifetime
std::weak_ptr	boost::weak_ptr	non-owning reference to object managed by shared_ptr
std::auto_ptr	-	predecessor of std::unique_ptr
non_owning_ptr	-	a non-owning reference
optional	boost::optional	optional value, not a smart pointer
-	boost::scoped_array	scoped pointer for arrays
-	boost::shared_array	shared pointer for arrays
-	boost::intrusive_ptr	shared pointer with externalized reference count

Table 3.1.: Available Smart Pointers

3.2.3. std::unique_ptr

`std::unique_ptr` takes sole ownership of a resource, and deletes it upon destruction. It is therefore non-copyable, but supports move semantics and can thus be used in containers supporting move semantics (e.g. `std::vector`). Currently a **`new`**-expression is still required to create a resource that is wrapped by a `std::unique_ptr`, but a `make_unique` similar to `make_shared` has already been proposed[3] and will likely be included in a future version of the C++ standard[4]. For dynamically allocated arrays `std::unique_ptr` supports `std::unique_ptr<T[]>` instead of `std::unique_ptr<T>`.

3.2.4. std::shared_ptr

`std::shared_ptr` takes shared ownership of a resource, employing a reference count, and deletes the resource once it is no longer referenced. It can therefore be freely copied, while ensuring that no memory leak occurs. `std::shared_ptr` should be created using `std ::make_shared`. Support for dynamically allocated arrays has been discussed[5], but not yet proposed for the standard.

3.2.5. std::weak_ptr

`std::weak_ptr` holds a non-owning reference to an object managed by a `std::shared_ptr`. It doesn't increase the reference count and thus doesn't prevent the resource from being deleted, but tracks the deletion of the object so it doesn't result in a dangling pointer. To access the referenced object the `std::weak_ptr` must be converted to a `std::shared_ptr` first to assume temporary ownership.

[3]Lavavej, *N3588: make_unique.*

[4]Sutter, *Trip Report: ISO C++ Spring 2013 Meeting.*

[5]Hinnant, *Why isn't there a `std::shared_ptr<T[]>` specialisation?*

3.2.6. std::auto_ptr (deprecated)

`std::auto_ptr` takes sole ownership of a resource, and deletes it upon destruction. Copying an `std::auto_ptr` transfers the ownership to the target, resulting in unusual copy semantics. Due to the unusual copy semantics and the lack of move semantics, `std::auto_ptr` may not be placed in standard containers. As of C++11 `std::auto_ptr` is deprecated and replaced by `std::unique_ptr`[6].

3.2.7. non_owning_ptr

`non_owning_ptr` holds a non-owning reference to an object. It behaves exactly like a raw pointer and doesn't provide any memory management facilities. It's sole purpose is to express that the pointer doesn't hold ownership of the resource. `non_owning_ptr` should be created using `make_nonowning`. The pointer is drafted in N3514[7] as `exempt_ptr` with the associated function `make_exempt`.

3.2.8. optional

`optional` allows to transport an optional payload (value, reference, ...). The payload is accessed using the familiar operators from pointers, `*` to dereference it and `->` to access a member of the contained object directly. `optional` is not yet standardized, but should be part of a future version of the C++ standard[8].

[6]Toit, *N3337 Working Draft, Standard for Programming Language C++: The class template auto_ptr is deprecated. Note: [The class template unique_ptr 20.7.1 provides a better solution. -end note].*

[7]Brown, *N3514: A Proposal for the World's Dumbest Smart Pointer*.

[8]Sutter, *Trip Report: ISO C++ Spring 2013 Meeting*.

3.2.9. boost smart pointers

Boost is a widely used set of libraries that provide its own set of smart pointers. Some standard smart pointers, such as `std::shared_ptr` came out of boost initially[9].

boost::scoped_ptr

`boost::scoped_ptr` takes sole ownership of a resource, and deletes it upon destruction. It is non copyable and doesn't support move semantics. Similar to `std::unique_ptr<T>`.

boost::scoped_array

`boost::scoped_array` takes sole ownership of a dynamically allocated array, and deletes it upon destruction. It is non copyable and doesn't support move semantics. Similar to `std::unique_ptr<T[]>`.

boost::shared_ptr

`boost::shared_ptr` takes shared ownership of a resource, employing a reference count, and deletes the resource once it is no longer referenced. Similar to `std::shared_ptr<T>`.

boost::shared_array

`boost::shared_array` takes shared ownership of a dynamically allocated array, employing a reference count, and deletes it once it is no longer referenced.

[9]Boost, *boost::shared_ptr class template*.

boost::weak_ptr

`boost::weak_ptr` holds a non-owning reference to an object managed by a `boost::shared_ptr`. Similar to `std::weak_ptr`.

boost::intrusive_ptr

`boost::intrusive_ptr` behaves like a `boost::shared_ptr` but externalizes the reference count. This decreases usability, but can be useful if the reference count is for instance embedded in the managed object. `boost::shared_ptr` should be preferred whenever possible.

boost::optional

See `optional`.

3.3. Replacing the use of raw pointers

To replace a raw pointer with a suitable smart pointer or values and references it is first necessary to analyze which roles a raw pointer takes. There is no single correct solution for a specific raw pointer, as each pointer can take multiple roles, and many roles can be represented in multiple ways.

Considering the following use of a raw pointer:

```
void set(int *i) {
    if (i) {
        *i = 3;
    }
}
```

Because the referenced value holds no const modifier, and a value is assigned, the pointer represents a reference. The if-clause suggests that the parameter is optional, it is not clear however if the pointer is truly allowed to be NULL, or if this is just a safety measurement for errors.

Thus, this example could for instance represent an optional reference:

```
void set(boost::optional<int&>() i) {
    if (i) {
        *i = 3;
    }
}
```

In this representation the developer expressed that the function parameter is optional, and if available is a reference to an integer.

The same thing could also be expressed, though less explicitly, using `non_owning_ptr` as reference:

```
void set(boost::optional<const non_owning_ptr
    <int> > i) {
    if (i) {
        **i = 3;
    }
}
```

The `non_owning_ptr` still makes clear that `i` is a non-owning reference, but is slightly less convenient to use.

And since `non_owning_ptr` can be empty, the optional could also be implemented without boost::optional:

```
void set(const non_owning_ptr<int> i) {
    if (i) {
        *i = 3;
    }
}
```

In this version it is not explicitly communicated that the parameter is optional. Depending on other factors, such as coding conventions of the project, the simpler notation might justify this loss of semantics.

As one can see there is an almost infinite number of combinations possible, and often there is more than one “correct” solution. It is in the responsibility of the developer to communicate his intentions by using an appropriate combination of the available smart pointers after evaluating other factors such as the projects coding conventions or other “best practices”.

3.3.1. Choosing the right smart pointer for a refactoring

Because a raw pointer can be represented by various combinations of smart pointers and other intention revealing language constructs (reference, value, ...), the problem can be simplified by selecting a subset of smart pointers to implement the roles.

When looking at the options for suitable smart pointers, there are two primary candidates with which most roles can be expressed: `non_owning_ptr` and `std::unqiue_ptr`.

`std::shared_ptr` is usually not an option because there is no reason why a reference count should be required if there wasn’t one with the raw pointer. If there was a custom reference counting with the raw pointer, a refactoring would need to be able to correctly identify and remove it. `std::shared_ptr` could still be a useful offer in some cases such as a new factory function that hasn’t been used yet (if it is already used we run into the aforementioned problem that a reference count is introduced where there wasn’t one before).

`optional` is not required as its functionality can be emulated using empty smart pointers. A refactoring without `optional` is syntactically also less intrusive (the smart pointer can be used exactly like a raw pointer, `optional` requires additional adjustments).

C++ References are also implementable using `non_owning_ptr`. Further is `non_owning_ptr` syntactically less intrusive and has the additional advantage that it may be reassigned, which again removes a special case.

Values on the stack are not considered, because the same ownership semantics can be achieved using `std::unique_ptr` (sole ownership, destruction as the identifier goes out of scope). Further may the move from heap to stack be undesirable due to platform limitations such as the stack size limit. I.e. if the object was managed on the heap before, nothing has changed in that regard that would justify why the object should be managed on the stack after the refactoring. Like C++ references is a refactoring to a value also syntactically a rather intrusive change.

3.3.2. An approach for automatic determination of a suitable smart pointer

When only considering `std::unqiue_ptr` and `non_owning_ptr` for a refactoring (for the reasons lined out in the previous section), a primary point of distinction is the ownership of the pointer. It is therefore possible to automatically select the appropriate smart pointer by determining the ownership of a raw pointer.

This section analyses how the ownership of a raw pointer can be determined and highlights the difficulties inherent to the approach.

Determining ownership of a raw pointer

To automatically select an appropriate smart pointer based on the ownership of the raw pointer, ways to automatically determine ownership are required.

There are two places where a pointer is normally (see 3.3.2 on page 32) guaranteed to be owning:

- When a pointer is first initialized using **`new`**.
- When a pointer is finally deleted using **`delete`**.

In all other cases a pointer is potentially non-owning, and the only way to determine the ownership of a pointer is to track it from either the allocation or the deletion.

Between allocation and deletion the pointer may be passed around, potentially transferring ownership to other variables. Each assignment may potentially transfer ownership:

- An assignment to a variable: `int *x = p;`
- An assignment to a function parameter: `foo(p);`
- An assignment to a return value: `return p;`

However, while an assignment may transfer ownership, it is not guaranteed to do so. The assignment may instead result in a non-owning reference.

Consider the following example:

```
1 int *p = new int();
2 int *x = p;
3 foo(p);
4 delete x;
```

- line 1: `p` is guaranteed to have ownership of the resource.
- line 2: `x` MAY receive ownership of the resource, but is not guaranteed to do so. From this point on it is unclear if `x` or `p` has ownership of the resource or is just a reference.
- line 3: as in line 2, `foo()` MAY receive ownership, but is not guaranteed to do so.
- line 4: the delete on `x` gives the guarantee that `x` holds ownership.

It is also possible that a pointer looses (or receives) ownership of a pointer conditionally:

```
int *p = new int();
int *x {0};
if (i == 1) {
 x = p;
}
if (x) {
    delete x;
} else {
 9      delete p;
10  }
```

- line 1: `p` is guaranteed to have ownership of the resource.
- line 3: if the condition is true, `x` receives ownership of the pointer (although that is only clear on line 7).
- line 6: only if `x` was set the condition is true and line 7 is reached, otherwise the delete on line 9 is reached resulting in `p` having ownership.
- line 7: the delete on `x` gives the conditional guarantee that `x` holds ownership.
- line 9: the delete on `p` gives the conditional guarantee that `p` holds ownership.

The same is of course possible across function boundaries. This shows that pointer ownership has to be tracked per code path, as the same identifier can have different semantics in different code paths (because the underlying pointer can change).

Note that smart pointers are already used in parts of the code base may also give the required information if a pointer holds ownership or not.

Other Memory Management Models

There are scenarios where the assumption that the first pointer that receives the pointee from the `new`-expression receives the ownership of the object, doesn't hold true.

Consider the following example:

```
struct Object {
    Object(){ register(); }
    void register() { MemoryManagementUnit::register(this); }
};

```

```
6 Object *o = new Object();
```

We assume that `MemoryManagementUnit` is a facility that manages the lifetime of all objects registered with it. `MemoryManagementUnit::register()` takes therefore ownership of pointers passed to it. In line 2, the constructed object passes ownership to the `MemoryManagementUnit` by calling `register()`. Although `o` is initialized directly from the `new`-expression in line 6, `o` never receives ownership of the object.

Similarly, it is possible that an object passes its ownership somewhere during a function call:

```
1 struct Object {
2     void doSomething() { MemoryManagementUnit
          ::register(this); }
3 };
4
5 Object *o = new Object();
6 o->doSomething();
```

While `o` initially receives ownership in this example on line 5, it looses it (rather unexpectedly) in line 6 during the call to `doSomething()`. This ownership transfer would go unnoticed when only checking for external assignments.

It would be possible to check for this different memory management model by ensuring that the `this`-pointer is not stored anywhere from within the object.

As a real world example, such a memory management model is for instance implemented in the Qt-Framework[10] with `QObject`. With `QObject`, each object can register itself with the parent object, passing it's ownership to the parent. The parent object is then responsible for managing the lifetime of its children[11].

As this memory management model doesn't work together with the one used for smart pointers, this issue is ignored for this analysis.

[10]qt-project.org, *Qt Project*.
[11]qt-project.org, *QObject Trees & Ownership*.

Conclusion

While tracking the ownership of a pointer seems feasible at first, conditional ownership transfers and multiple assignments make the task quickly daunting. To be able to track the ownership each codepath has to be evaluated, where the number of codepaths grows exponentially with the number of if statements and potentially unlimited with loops and recursion.

The structural representation of the code provided by an abstract syntac tree allows to track identifiers in the code, but not the underlying value, which would be required to be able to track pointers. In order to be able to track pointer ownership, a codepath analysis would therefore be required.

An example, to highlight the complexity of tracking pointer ownership, would be a list containing pointers. Depending on the internal implementation, it would be very difficult to track where a pointer enters the list and where it leaves it (i.e. one would have to track the index which the pointer occupies).

While this is not generally infeasible (compilers and advanced code inspection tools can inspect codepaths), it is clearly beyond the scope of this project.

3.4. Refactoring Cases

In order to refactor the use of raw pointers into smart pointers, the following section identifies a set of starting points where raw pointers can appear in the source code, and analyses what refactorings can be applied to each starting point.

Raw pointers can appear in various places and convey various semantics. Each starting point where a raw pointer could appear is associated with a number of possible refactorings (one per smart pointer), which are then offered to the developer. Because we are not able to analyze the semantics automatically (see Section 3.3.2), it is up to the developer to choose a suitable refactoring from the options provided. It is however possible to support the developer in his choice by limiting the available refactorings using heuristics.

As the complexity for what pointers can be used can quickly go beyond what we can reasonably refactor in a way that is still useful, this analysis tries to identify a limited set of the common cases, so we can provide actually useful refactorings. This means not every possible case is covered, but a limited set of cases should be handled well. By looking at concrete cases that are existing in real-world codebases, we hope to identify a useful subset of possible refactorings.

Further is the complexity limited by only refactoring a single call-hierarchy level instead of recursively converting full call hierarchies . This is because tracking the variables is not directly supported (see also Section 3.3.2 on the preceding page), so the task would be too complex within the scope of this project. Further has this approach the advantage that the changes from a refactoring are usually applied to the code the developer is currently working on, making it easier for the developer to review the changes and also giving finer grained control over the refactored parts of the source code.

The analysis is based on the smart pointers available in Standard C++ Library. If other smart pointers (i.e. boost) should be used, a smart pointer with the same semantics has to be chosen (see Table 3.1 on page 23). For cases where no standard smart pointer exists, boost is the preferred alternative.

The following guidelines are generally applicable to all refactorings:

- Preserve the lifetime of the object: Using a smart pointer should not change the lifetime of the object.

```
void foo()
{
    MyClass *p = new MyClass();
    bar(p)
    delete p;
    MyClass p2 = new MyClass();
    delete(p2)
}
```

By turning p into a local smart pointer, both objects would live until the end of foo(), resulting in a change of lifetime for p. As this could lead to problems, and p was deleted on purpose before allocating p2, such changes of lifetime must be avoided (for instance by calling p.release()).

- Don't change semantics of the pointer, i.e., constness should be preserved.

```
MyClass const *p = new MyClass();
//becomes
std::unique_ptr<const MyClass> p { new
    MyClass() };
```

- Don't change the scope of the pointer, i.e., don't move a declaration in a for-loop to the beginning of the function.

```
for(int i = 0; i < 3; i++) {
    MyClass const *p = new MyClass();
}
//becomes
for(int i = 0; i < 3; i++) {
    std::unique_ptr<const MyClass> p {
        new MyClass() };
}
```

- When creating smart pointers, type hierarchies must be taken into account, i.e., the type used in the `new`-expression must be used to to create the smart pointer, as opposed to the type of the pointer-variable declaration, that might be using a base-type.

```
Base *p { new Derived() };
//becomes
std::unique_ptr<Base> p { new Derived()
    };
```

3.4.1. Format

The section 3.4.3 on page 39 first analyses the particularities of the chosen starting points wherer a raw pointer could be used in the source code.

3.4.4 on page 42 then analyses an example of each starting point for the case where the pointer holds ownership and once for the case where the pointer doesn't hold ownership. Each case is analysed using the following format:

- A short explanation
- A generic example based on Listing 3.1
- A specific example from a real world code base[12]
- The refactored generic example
- Limitations and edge cases of the handling of the refactoring case

This format allows to describe all considered cases with enough detail that the implementation can be based on it.

Throughout the examples the following class hierarchy is used:

```
class Vehicle;

class Car: public Vehicle
{
public:
    Car(int ps);
};

class Engine;

template <typename T>
class SpecializedCar: public Vehicle
{
public:
    SpecializedCar(int ps);
```

[12]Software, *DOOM3 Sourcecode*.

```
};
```

Listing 3.1: Vehicle class hierarchy

3.4.2. Scope and Limitations

Only `std::unique_ptr` and `non_owning_ptr` are considered for the refactorings because all roles can be expressed with them (see also section 3.3.1 on page 29).

As strings and arrays are a special case of pointers for which specific smart pointers exist, this analysis doesn't consider them (See 3.1 on page 13).

It is assumed that a single identifier always either stands for an owning pointer or a non-owning pointer, but never for both. Therefore a construct like this would not be supported:

```
int *getRef();
int *i;
if (condition()) {
    i = new int();
} else {
    i = getRef();
}
if (condition()) {
    delete i;
}
```

It is not possible to choose a suitable smart pointer for `i`, because the identifier changes the semantics in the codepaths.

Further are any problems resulting from misuse of pointers, such as assigning numbers to a pointer other than `nullptr`, not taken into account.

3.4.3. Starting points

For the analysis three different starting points were chosen:

- A local pointer variable: `int *x;`
- A pointer function parameter: `void foo(int *x);`
- A pointer return value: `int *foo();`

The starting points have been chosen because they seemed like natural starting points for a conversion (i.e. where one would want to apply a quickfix), and due to expected feasibility.

Further potential starting points that have not been analysed are:

- Global variables
- Class members

Local pointer

Local pointers are the simplest case due to their limited visibility. Therefore only the scope the local variable needs to be considered.

Function parameter

Function parameters are different from local variables because they are visible outside of the scope of the function. Therefore also the caller code needs to be considered. Note that because the function parameter is visible inside the scope of the function, the same refactorings apply as for local pointers.

If a function is modified to take a smart pointer instead of a raw pointer as argument, the signature changes. These are the implications for callers:

- There can be a type mismatch when a caller tries to pass in a raw pointer:

```
void foo(non_owning_ptr<Car> car)
{

}

Car *car;
foo(car);
```

The call to `foo()` fails because the constructor of `non_owning_ptr` is explicit.

- The compiler could choose a different overload, which may go unnoticed, leading to unexpected behaviour:

```
void foo(Vehicle *)
{
}

//before refactoring
void foo(Car *car)
{
}

//after refactoring
void foo(non_owning_ptr<Car> car)
{
}

Car *car = new Car(42);
foo(car);
```

Before the refactoring `foo(Car*)` is selected. After the refactoring `foo(Vehicle*)` is selected, because the raw pointer argument no longer matches the `non_owning_ptr` parameter.

A possible migration path for this problem would be to provide an overloaded wrapper function that takes the raw pointer, wraps it in a smart pointer, and delegates the call to the smart pointer version:

```
void foo(non_owning_ptr<Car> car)
{
}
```

```

//wrapper
void foo(Car *car)
{
    foo(non_owning_ptr<Car>(car));
}
```

This way the caller code doesn't need to be adjusted and there is no danger of a new overload being selected. Note that if all callers can be adjusted before the refactoring, this problem can be avoided entirely. This may however not be possible if the code is part of the API of a library.

Note that the same problem applies to a virtual dispatch as well:

```
struct Base {
    virtual foo(Car*);
};
struct Derived {
    //before the refactoring
    virtual foo(Car*);
    //after the refactoring
    virtual foo(non_owning_ptr<Car>);
};

Car *car = new Car(42);
Base *base = new Derived;
base->foo(car);
```

The call to `base->foo()` resulted in a call to `Derived::foo()` before the refactoring due to the virtual dispatch mechanism. After the refactoring `Base::foo()` is called instead, because `Derived::foo()` no longer matches. This problem can be solved with the aforementioned wrapper approach as migration path, or by adjusting overridden methods in the whole type hierarchy.

Note that this refactoring must be applied to both the declaration and the definition of the function.

Return value

Return values require the adaption of all callers, just like function parameters, but as the return value is not part of the function signature, there is no conflict with overloaded functions.

Note that this refactoring must be applied to both the declaration and the definition of the function.

3.4.4. Default Cases

For each starting point there are generally two options, one per target smart pointer, due to the two chosen smart pointers `std::unique_ptr` and `non_owning_ptr` as described in 3.4.2 on page 38.

The default refactoring cases are therefore the following:

- Local pointer, owning pointer
- Local pointer, non-owning pointer
- Function parameter, owning pointer
- Function parameter, non-owning pointer
- Return value, owning pointer
- Return value, non-owning pointer

This means per identified raw pointer two refactorings are potentially applicable. Because automatically determining ownership of the pointer was deemed to complex in most cases within the scope of this project (see 3.3.2 on page 30), it is up to the developer to make the appropriate choice.

As a special case, a delete within the local scope can be used as heuristic to determine that the deleted raw pointer holds ownership of the resource. In this case it is not necessary to offer the non-owning pointer refactoring to the develeoper.

3.4.5. Local pointer: owning pointer

A local pointer receives ownership if it is initialized using a `new`-expression, a function returning an owning pointer (i.e. a factory function), or an owning pointer variable.

The owning pointer must either pass ownership somewhere or delete the resource before it goes out of scope.

Example

```
void foo()
{
    Vehicle const *p { new SpecializedCar<
       Engine>(42) };
    bar(p);
    delete p;
}
```

- The pointer has a const modifier that must be preserved during conversion.
- The initialization expression of the allocated object is 42
- The pointer is then passed to `bar()`, which will result in an error after converting p to a smart pointer
- Finally the object is deleted at the end of the scope, resulting in exactly the same behaviour when using a smart pointer

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/splines.cpp:1307-1323
- function: idCameraDef::load
- variable: src

```
bool idCameraDef::load( const char *filename
    ) {
     idParser *src;
     src = new idParser( filename,
         LEXFL_NOSTRINGCONCAT |
         LEXFL_NOSTRINGESCAPECHARS |
         LEXFL_ALLOWPATHNAMES );
     if ( !src->IsLoaded() ) {
          common->Printf( "couldn't load %s\n",
               filename );
          delete src;
          return false;
     }
     clear();
     parse( src );
     delete src;
     return true;
}
```

The code shows the following characteristics:

- The if-clause results in two possible code-paths, of which both must contain a delete statement in order to get the same result after converting to a smart pointer.
- The definition of `src` is separated from its initialization.
- src is passed to `parse`, requiring either calling `src.get()` or converting the argument of parse to a smart pointer.
- Due to the delete it is ensured that `parse` didn't take ownership.

Example refactored

```
void foo()
{
     std::unique_ptr<const Vehicle> p { new
         SpecializedCar<Engine>(42) };
     bar(p.get());
}
```

The pointer can be refactored to a `std::unique_ptr`.

Because `p` is passed to `bar`, the raw pointer must be extracted using `p.get()`. Further the delete had to be removed as the `std::unique_ptr` will delete the resource as it goes out of scope.

In case a `std::make_unique` would become available the initialization would look the following:

```
std::unique_ptr<const Vehicle> p {
    make_unique< SpecializedCar<Engine> >(42)
    };
```

Limitations and Edge Cases

- The plugin could detect disposal functions, and pass the pointer to the disposal function using the smart pointers `release()` method.
- The initialization for the pointer may also occur at a later stage (no direct initialization). In this case both the definition of the pointer and the initialization must be adjusted accordingly.
- If a pointer is deleted early (before it goes out of scope), it is required to reset the smart pointer early using `unique_ptr<T>::reset(nullptr)`, in order not to change the lifetime of the pointee.
- If the pointer is reassigned, the smart pointer would have to be reassigned as well using its `reset()` method. Note that the old pointer must be released first, in order not to delete it:

  ```
  std::unique_ptr<const Vehicle> p { new
      Car(42) };
  p.release();
  p.reset(new Car(42));
  ```

  But since this code would result in the first assigned pointer to leak, the only valid scenario would be:

```
std::unique_ptr<const Vehicle> p { new
    Car(42) };
passOwnership(p.release());
p.reset(new Car(42));
```

There must be something which takes ownership of the old pointer, otherwise to code is not valid. If `passOwnership()` would expect a `std::unique_ptr` as argument (as it should), this would result in

```
passOwnership(make_unique(p.release()));
```

At this point it should become evident that it is not a good idea to reuse the pointer identifier when using an owning pointer. The plugin therefore does not handle this case.

- If a function parameter to which the pointer is passed already uses a `std::unique_ptr`, the pointer can be moved using `std::move` instead of wrapping it with a constructor call.

```
void bar(std::unique_ptr<Vehicle>);

Vehicle *p { new Car(42) };
bar(std::unique_ptr(p));
//becomes
std::unique_ptr<Vehicle> p { new Car(42)
    };
bar(std::move(p));
```

- If the pointer is initialized from a function return value, and the return value already uses a `std::unique_ptr`, the smart pointer can be copy constructed instead of using `release()`.

```
std::unique_ptr<Vehicle> create();

Vehicle *p { create().release() };
//becomes
std::unique_ptr<Vehicle> p { create() };
```

- If the pointer is passed to a function which takes ownership of the pointer, but doesn't have a smart pointer parameter, the pointer must be released using `release()` instead of get().

- If the pointer is returned from a function with a raw pointer return value, the raw pointer must be extracted using `release()`.

```
Vehicle *create()
{
    std::unique_ptr<Vehicle> p { new Car
       (42) };
    return p.release();
}
```

3.4.6. Local pointer: non-owning

A local pointer is non owning if it is initalized by a reference.

It can therefore be refactored to a `non_owning_ptr`.

Example

```
void bar(Vehicle const *);
Vehicle *getReference();

void foo()
{
    Vehicle const *p { getReference() };
    bar(p);
}
```

- The pointer has a const modifier, which must be preserved during conversion.
- `p` is initialized using a call to `getReference()`, which returns a non-owning pointer.
- `p` is then passed to `bar()`, which will result in an error after converting to a smart pointer.

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/CamWnd.cpp:53-67
- function: ValidateAxialPoints
- variable: selFace

```
void ValidateAxialPoints() {
    int faceCount = g_ptrSelectedFaces.
        GetSize();
    if (faceCount > 0) {
        face_t  *selFace = reinterpret_cast <
             face_t * > (g_ptrSelectedFaces.
            GetAt(0));
        if (g_axialAnchor >= selFace->
            face_winding->GetNumPoints()) {
             g_axialAnchor = 0;
        }
        if (g_axialDest >= selFace->
            face_winding->GetNumPoints()) {
             g_axialDest = 0;
        }
    } else {
        g_axialDest = 0;
        g_axialAnchor = 0;
    }
}
```

The code shows the following characteristics:

- The definition of `selFace` is nested in an if-clause.
- `selFace` is initialized with a reference to a `face_t` object.
- Because `selFace` is neither passed to a function nor is it deleted in `ValidateAxialPoints`, it is ensured that `selFace` doesn't hold ownership.

Example refactored

```
void bar(Vehicle const *);
Vehicle *getReference();

void foo()
{
    non_owning_ptr<const Vehicle> p {
       getReference() };
    bar(p.get());
}
```

The pointer can be refactored to a `non_owning_ptr`.

Because `p` is passed to `bar()`, the raw pointer must be extracted using `p.get()`.

In case of a later initialization or reassignment the pointer must be created using `make_nonowning`:

```
non_owning_ptr<const Vehicle> p;
p = make_nonowning(getReference);
```

Limitations and Edge Cases

- The initialization for the pointer may also occur at a later stage (no direct initialization). In this case both the definition of the pointer and the initialization must be adjusted accordingly. Note that a reassignment can be handled in the same way.
- If a function parameter to which the pointer is passed already uses a `non_owning_ptr`, the pointer can be passed directly instead of wrapping it with `make_nonowning`.

```
void bar(non_owning_ptr<Vehicle>);

Vehicle *p{new Car(42)};
bar(make_nonowning(p));
//becomes
non_owning_ptr<Vehicle> p{make_nonowning<
   Car>(42)};
bar(p);
```

- If the pointer is initialized from a function return value, and the return value already uses a `non_owning_ptr`, the smart pointer can be copy constructed instead of using `get()`.

  ```
  non_owning_ptr<Vehicle> getRef();

  Vehicle *p { getRef().get() };
  //becomes
  non_owning_ptr<Vehicle> p { getRef() };
  ```

- If the pointer is returned from a function with a raw pointer return value, the raw pointer must be extracted using `get()`.

  ```
  non_owning_ptr<Vehicle> getRef();
  Vehicle *foo()
  {
      std::non_owning_ptr<Vehicle> p {
          getRef() };
      return p.get();
  }
  ```

3.4.7. Function parameter: owning pointer

A function takes ownership of a pointer if either the function deletes the pointer itself (like a disposal function), or if the function transfers ownership somewhere.

See also 3.3.2 on page 30.

Example

```
void sell(Car*);
void destroyCar(Car* c)
{
    delete c;
}

void trySellCar(Car* car)
{
    //We can't sell this junk
```

```
    if (car->ps() <= 42) {
        destroyCar(car);
    } else {
        sell(car);
    }
}

void foo()
{
    Car *car = new Car(42);
    trySellCar(car);
}
```

`trySellCar` would transfer the ownership of the pointer to `sell` if it's ps were greater than 42, but instead passes it to the disposal function `destroyCar`. We know that `destroyCar` takes ownership of the pointer because it deletes it.

We assume that `sell` takes ownership of the pointer. If it wouldn't, car would be owning in one codepath and non-owning in the other.

Example refactored

```
void sell(Car*);
void destroyCar(Car* c)
{
    delete c;
}

void trySellCar(std::unique_ptr<Car> car)
{
    //We can't sell this junk
    if (car->ps() <= 42) {
        destroyCar(car.get());
    } else {
        sell(car.get());
    }
}

void foo()
{
    Car *car = new Car(42);
    trySellCar(std::unique_ptr<Car>(car));
}
```

The pointer can be refactord to a `std::unique_ptr`.

`destroyCar` could be removed entirely in this case, since it's apart from the `delete` empty. `car` in `foo()` must be wrapped by a `std::unique_ptr`. If `car` in `foo()` would already be a `std::unique_ptr`, `std::move` would be required to pass it to `trySellCar()` using move semantics:

```
void foo()
{
    std::unique_ptr<Car> car{ new Car(42) };
    trySellCar(std::move(car));
}
```

Limitations and Edge Cases

- The same limitations as for a local pointer apply, as the parameter behaves inside the scope of the function like a local variable.
- If the refactored function is either virtual or has overloaded functions, the workaround described in 3.4.3 on page 39 should be applied as migration path.
- If a caller already uses a `std::unique_ptr`, but extracts the raw pointer using `release()`, the pointer can be moved using `std::move` instead.

```
void bar(non_owning_ptr<Vehicle>);

std::unique_ptr<Car> p;

bar(p.release());
//becomes
bar(std::move(p));
```

3.4.8. Function parameter: Non-owning pointer

A non-owning pointer can be passed to a function as argument for example to:

- provide a reference to a data structure where complex return values can be stored.
- pass a dynamically allocated object as argument.

The pointer parameter is non-owning if:

- it was ensured that the pointer is not owning (see 3.4.7 on page 50).
- it was ensured that the caller keeps ownership of the pointer.

As this is non-trivial to detect (see 3.3.2 on page 30), it is up to the developer to indicate if the argument is indeed non-owning.

Example

```
void setupCar(Car* car)
{
    car->setup();
}

void setup()
{
    Car *car = new Car(42);
    setupCar(car);
    delete car;
}
```

The parameter `car` of `setupCar()` is non-owning as the pointer is never deleted or passed to a function taking ownership. The passed object may be modified as long as the object is not const.

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/splines.cpp:1728-1750

- function: idInterpolatedPosition::parse
- variable: src

```
void idInterpolatedPosition::parse( idParser
    *src ) {
    idToken token;

    src->ExpectTokenString( "{" );
    while ( 1 ) {
        if ( !src->ExpectAnyToken( &token ) )
            {
            break;
        }
        if ( token == "}" ) {
            break;
        }

        if ( !token.Icmp( "startPos" ) ) {
            src->Parse1DMatrix( 3, startPos.
               ToFloatPtr() );
        }
        else if ( !token.Icmp( "endPos" ) ) {
            src->Parse1DMatrix( 3, endPos.
               ToFloatPtr() );
        }
        else {
            idCameraPosition::parseToken(
               token, src);
        }
    }
}
```

The code shows the following characteristics:

- A raw pointer `src` is passed as a function argument.
- assuming `parseToken` doesn't take ownership of `src`, this function never takes ownership of the pointer.

Example Refactored

```
void setupCar(non_owning_ptr<Car> car)
{
    car->setup();
}

void setup()
{
    Car *car = new Car(42);
    setupCar(make_nonowning(car));
    delete car;
}
```

The pointer can be refactored to a `non_owning_ptr`.

Each caller must be adjusted to use `make_nonowning`, because the constructor of `non_owning_ptr` is explicit.

Limitations and Edge Cases

- The same limitations as for a local pointer apply, as the parameter behaves inside the scope of the function like a local variable.
- If a caller already uses a `non_owning_ptr`, but extracts the raw pointer using `get()`, the smart pointer can passed directly. If the pointer is extracted from another smart pointer using `get()`, it still needs to be wrapped using `make_non_owning`

3.4.9. Return value: owning pointer

A function may pass ownership with a pointer as return value.

One pattern returning an owning pointer is the factory method pattern[13].

[13] Wikipedia, *Factory Method*.

Example

```
Vehicle *createVehicle()
{
    return new Car();
}

void main()
{
    Vehicle *vehicle = createVehicle();
}
```

`createVehicle` allocates the Car object and then transfers ownership to the caller. It is therefore a factory function.

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/splines.cpp:777-787
- function: idCameraDef::startNewCamera
- variable: cameraPosition

```
idCameraPosition *idCameraDef::startNewCamera
    ( idCameraPosition::positionType type ) {
    clear();
    if (type == idCameraPosition::SPLINE) {
        cameraPosition = new idSplinePosition
            ();
    } else if (type == idCameraPosition::
        INTERPOLATED) {
        cameraPosition = new
            idInterpolatedPosition();
    } else {
        cameraPosition = new idFixedPosition
            ();
    }
    return cameraPosition;
}
```

The code shows the following characteristics:

- The if-clauses result in 3 different codepaths, of which all return an owning pointer.

Example Refactored

```cpp
std::unique_ptr<Vehicle> createVehicle()
{
    return std::unique_ptr<Vehicle>(new Car()
       );
}

void main()
{
    Vehicle *vehicle = createVehicle().
       release();
}
```

The return value can be refactored to a `std::unique_ptr`.

Each return statement must be adjusted to use `std::unique_ptr`.

Each caller must be adjusted to extract the raw pointer using `release()`.

Limitations and Edge Cases

- If a caller already uses a `std::unique_ptr` it can be initialized directly instead:

  ```cpp
  std::unique_ptr<Vehicle> vehicle{
      createVehicle() };
  ```

3.4.10. Return value: non-owning pointer

A function may return a pointer as non-owning reference.

Example

```cpp
Vehicle *getVehicle()
{
    static Car car;
    return &car;
}
```

```

void main()
{
    Vehicle *vehicle = getVehicle();
}
```

Note that this is not a factory function, but resembles something like the singleton pattern. The important difference is that `getVehicle` doesn't transfer the ownership of `car`, but only returns a reference to it.

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/splines.cpp:1606-1612
- function: idInterpolatedPosition::getPoint
- variable: startPos, endPos

```
idVec3 *idInterpolatedPosition::getPoint( int
     index ) {
    assert( index >= 0 && index < 2 );
    if ( index == 0 ) {
        return &startPos;
    }
    return &endPos;
}
```

The code shows the following characteristics:

- The if-clause results in two possible code-paths, of which both return a non-owning pointer.

Example Refactored

```
non_owning_ptr<Vehicle> getVehicle()
{
    static Car car;
    return make_nonowning(&car);
}

void main()
```

```
{
    Vehicle *vehicle = getVehicle().get();
}
```

The return value can be refactored to a `non_owning_ptr`, to indicate that no ownership is transferred to the caller.

Each return statement must be adjusted to use `make_nonowning()`.

Each caller must be adjusted to extract the raw pointer using `get()`.

Limitations and Edge Cases

- If a caller already uses a `non_owning_ptr` it can be initialized directly instead:

  ```
  non_owning_ptr<Vehicle> vehicle{
      getVehicle() };
  ```

3.4.11. Heuristic to determine ownership: local delete

If a delete is available within the local scope, it can be assumed that the pointer is owning. This assumption is not guaranteed to hold true in all cases, but is a good heuristic to determine that a pointer holds ownership of the resource.

By employing this heuristic, it is possible to only offer `std::unique_ptr` as suitable refactoring, removing `non_owning_ptr` from the available choices.

Note that while a delete of a pointer is a clear indication of ownership of the pointer, a lack thereof doesn't automatically mean that the pointer doesn't own the resource. The ownership may be transferred to a disposal function, or to the caller in case of a factory function. Further it is possible that the pointer is stored and cleaned up later on. See also 3.3.2 on page 30.

It is therefore not possible to apply this heuristic reversely.

Example

```
void foo()
{
    Car *p { new Car(42) };
    delete p;
}
```

Because `p` is deleted within the scope of this function, it can be assumed that `p` holds ownership of the resource.

Specific Example

- sourcefile: DOOM-3/neo/tools/radiant/splines.cpp:1307-1323
- function: idCameraDef::load
- variable: src

```
bool idCameraDef::load( const char *filename
   ) {
    idParser *src;
    src = new idParser( filename,
       LEXFL_NOSTRINGCONCAT |
       LEXFL_NOSTRINGESCAPECHARS |
       LEXFL_ALLOWPATHNAMES );
    if ( !src->IsLoaded() ) {
        common->Printf( "couldn't load %s\n",
            filename );
        delete src;
        return false;
    }
    clear();
    parse( src );
    delete src;
    return true;
}
```

The `delete` on `src` gives the indication that `src` is owning and could therefore be refactored to a `std::unique_ptr`.

Limitations and Edge Cases

- If the code would have multiple codepaths (for instance due to an `if`-statement), it would have to be ensured that each codepath contains a `delete`. As a simplification the plugin assumes that all codepaths contain a `delete`, if a `delete`-statement was found in the current scope.
- The plugin could detect disposal functions, and determine ownership accordingly.
- If the pointer variable is reassigned, or if the pointer passed ownership somwhere else between initialization of the pointer variable and its deletion, this heuristic could deliver wrong results.

  The following artificial example illustrates a case where the heuristic would not apply:

  ```
  1 void foo()
  2 {
  3     Car *p { getReference() };
  4     p->doSomething();
  5     p = get();
  6     delete p;
  7 }
  ```

  Although `p` represents a reference at first, it receives ownership in line 5 due to the assignment (assuming `get()` returns an owning pointer). In this case a refactoring to a `std::unique_ptr` would give the false impression that `p` holds ownership from the beginning.

3.5. Conclusion

The analysis shows, when ignoring the special cases of arrays and strings, all roles of a raw pointer can take can be expressed using `std::unique_ptr` and `non_owning_ptr`. While other language constructs would be available to represent certain roles, these two allow to refactor the code with fewer other changes required, and support refactoring only part of the code while the rest still uses raw pointers. This is important to be able to gradually convert the use of raw pointers to smart pointers, while being able to review and test each step.

Although it would be possible to automatically determine the suitable smart pointer to use, based on the ownership of the pointer, it is shown that the analysis of the ownership of a raw pointer can be very complex, because the semantics can change for the same identifier in each code path. Hence a full analysis would be too difficult to implement based on the information provided by the abstract syntax tree within the scope of this project.

Due to this complexity, fully automated refactorings adjusting all uses of a pointer automatically were not implemented. Instead the chose approach provides possible refactorings for the developer to choose, and only taking small steps in those refactorings. The developer who understands the code can then make the necessary decisions and review the applied changes.

This analysis also highlights one of the great benefits of smart pointers, the revealed semantics. While it is usually possible for a developer to guess the correct semantics of a pointer after having looked at enough context, it can be a lot of work to correctly analyze it and that work every new developer will have to do again. Smart pointers give developers the tools to express their intentions in a way that also a compiler can understand, which might be of even greater value than the automatic memory management.

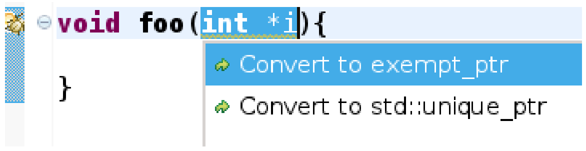
By recognizing the use of raw pointers in an existing codebase, any by providing suitable refactorings we can help developers to spot and fix the use of raw pointers. Although the plugin is not able to automatically determine the suitable smart pointer, the developer is at least supported by heuristics that work in many cases.

4. Implementation

The implementation of the smartor plug-in is split into two major parts: the checker and the quickfixes.

The checker is responsible for identifying the starting points (see Section 3.4.3 on page 39) in the code which are then marked using a marker.

The quickfix is responsible for executing a predefined refactoring. For each marker quickfixes are registered, which are then proposed to the user. If a quickfix is activated by the user, the quickfix implementation commits it's modifications to the source code.



```
void foo(int *i){

}
```

Figure 4.1.: A marker highlighting a raw pointer argument with two proposed quickfixes

The plug-in was developed using Scala, the project results in a single plug-in that can be used in Eclipse.

4.1. Checker

The checker is responsible for identifying problematic sections in the code which are then reported to the Codan framework.

The Codan framework provides an AST, which is then evaluated by the checker to identify the required parts.

The checker is implemented as single class that derives from `org.eclipse.cdt.codan.core.cxx.model.AbstractIndexAstChecker`, an extension point provided by Codan for code checking plug-ins.

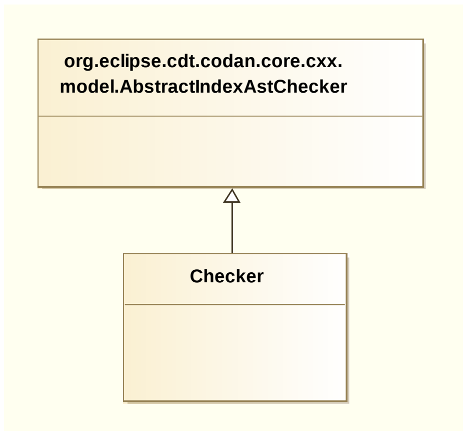

org.eclipse.cdt.codan.core.cxx.
model.AbstractIndexAstChecker
Checker

Figure 4.2.: The checker type hierarchy

The following patterns are evaluated in the given order, with patterns listed first taking precedence and the associated problem being reported:

- `ch.hsr.ifs.cdt.smartor.plugin.error.LocalPointerWithLocalDelete` :
  A declaration statement containing a raw pointer with a delete in the local scope.

  ```
  int *i = new int(3);
  delete i;
  ```

- `ch.hsr.ifs.cdt.smartor.plugin.error.LocalRawPointer`:
  A declaration statement containing a raw pointer.

  ```
  int *i = new int(3);
  ```

- `ch.hsr.ifs.cdt.smartor.plugin.error.SmartPointerArgumentToRawPointer` :

A function call statement with a smart pointer argument passed to a raw pointer parameter.

```
void foo(int *);
non_owning_ptr<int> i = make_nonowning<
    int>(3);
foo(i);
```

- `ch.hsr.ifs.cdt.smartor.plugin.error.RawPointerParameter`:
  A parameter declaration containing a raw pointer.

  ```
  void foo(int *i);
  ```

- `ch.hsr.ifs.cdt.smartor.plugin.error.RawPointerReturn`:
  A function declarator containing a raw pointer as return value.

  ```
  int *get();
  ```

The reported problems correspond to either a starting point defined in Section 3.4.3 on page 39, or a combination of a starting point and a heuristic. This approach allows quickfixes to be registered on specific combinations of starting points and heuristics. For instance is the refactoring for a `non_owning_ptr` on a local pointer not registered on `ch.hsr.ifs.cdt.smartor.plugin.error.LocalPointerWithLocalDelete`, because we know that this refactoring is not applicable in this case because the pointer has ownership of the resource.

4.1.1. Control Flow

The primary entry point to process the AST is:

```
def processAst(ast: IASTTranslationUnit):
    Unit
```

This method get's called whenever the AST must be reevaluated.

Although the framework would provide the visitor `org.eclipse.cdt.core.dom.ast.ASTVisitor` to traverse the AST, this facility was not used to make use of Scala's pattern matching instead.

```scala
private def traverseTree(node: IASTNode): Unit = {
  node match {
    case DeclarationStatement(ContainsPointer(HasLocalDelete(declarationStatment))) => placeMarkers(ProblemId.localPointerWithLocalDelete, node)
    case DeclarationStatement(ContainsPointer(parameter)) => placeMarkers(ProblemId.localRawPointer, parameter)
    case FunctionCallStatement(SmartPointerArgumentToRawPointerParameter(argument)) => placeMarkers(ProblemId.smartPointerArgumentToRawPointer, argument)
    case SinglePointerParameter(parameter) => placeMarkers(ProblemId.rawPointerParameter, parameter)
    case FunctionDeclarator(ContainsPointerNonRecursive(parameter)) => placeMarkers(ProblemId.rawPointerReturn, parameter)
    case _ => node.getChildren foreach traverseTree
  }
}
```

`traverseTree()` provides a method that recursively traverses each node in the AST, and gets called by `processAst()`. Inside this method, each node is matched using extractor objects. So instead of relying on the double dispatch mechanism of the visitor to process particular node types, Scalas matching capabilities are used to match specific patterns within the AST. This helps to keep the core logic of the Checker together in one place, with all the logic for the actual pattern matching externalized into custom extractor objects.

Each detected problem is reported using `placeMarkers`:

```scala
private def placeMarkers(problem: ProblemId.ProblemId, node: IASTNode): Unit = {
  val definition = ProblemDefinitionFactory.getProblemDefinition(problem)
  reportProblem(definition.getErrorId, node,
     definition.getProblemName)
}
```

The reported problems are finally displayed to the user as marker in the editor (see 4.1 on page 63), which allows the user to get a problem description and to trigger associated quickfixes.

4.1.2. Pattern matching

The Checker makes extensive use of Scala's pattern matching capabilities. Let us consider the following example:

```
node match {
    case DeclarationStatement(ContainsPointer
       (HasLocalDelete(declarationStatment)))
        => placeMarkers(ProblemId.
       localPointerWithLocalDelete, node)
    case _ => node.getChildren foreach
       traverseTree
}
```

This is a shortened version of `Checker.traverseTree()`. As we can see, `node` is matched using the **`match`** keyword. The **`match`**-statement is similar to a `switch`-statement popular in e.g. C++. It allows however to match the individual cases using custom extractor objects.

An extractor object must implement the `unapply()` method:

```
    def unapply(arg: T1): Option[T2]
```

`unapply()` has one argument and an optional return value. A return value of `None` indicates that the extractor object does not match, where a return value of `Some` indicates that it matches. Further is the type of the argument evaluated if it matches the matched expression.

For instance this is how the DeclarationStatement extractor is implemented:

```
object DeclarationStatement {
  def unapply(parameter:
      IASTSimpleDeclaration) = Some(parameter)
}
```

This extractor matches only if the argument is of type `IASTSimpleDeclaration`, but it returns the argument unconditionally.

Extractor objects allow for great flexibility because they can be nested:

```
node match {
    case DeclarationStatement(ContainsPointer
        (HasLocalDelete(declarationStatment)))
         => placeMarkers(ProblemId.
        localPointerWithLocalDelete, node)
...
```

`node` is first matched using `DeclarationStatement`, where `node` matches if `DeclarationStatement.unapply` returns an `Option` that is not `None`. If `DeclarationStatement` matches, its return value is passed on to the next extractor object `ContainsPointer`.

If all extractor objects match, the final return value is available in `delarationStatement`, which can then be used inside the case (in this case to place a marker).

A more advanced example than the `DeclarationStatement` is the `ContainsPointer`-extractor that recusively searches for a node of the type `IASTPointer`:

```
object ContainsPointer {
  def unapply(node: IASTNode): Option[
      IASTNode] = {
    node.getChildren.foreach { child =>
      child match {
        case x: IASTPointer => return Some(
            node)
        case x: IASTNode => unapply(x).
            foreach( ret => return Some(ret))
        case _ =>
      }
    }
```

```
      None
    }
}
```

Its `unapply()` method uses itself pattern matching to identify the type of its children, and calls itself recursively to descend in the tree of children. Because this extractor object can also not apply, althought the argument type matched, it is possible that `unapply()` returns `None` instead of `Some`. Note the shorter notation used to match a type:

```
case x: IASTPointer => return Some(node)
```

This allows to directly match for a type without writing a dedicated extractor object, but has the drawback that no extractor objects can be nested inside the expression.

4.2. Quickfix

The quickfixes apply the actual refactoring to the code when activated.

Because the quickfix class is used to associate a quickfix with a reported problem (see 4.3 on page 74), each quickfix needs to be in it's own class. Each marker has an associated problem id, for which quickfixes can be registered. This allows to decouple the detection of a potential problem, and a proposed quickfix.

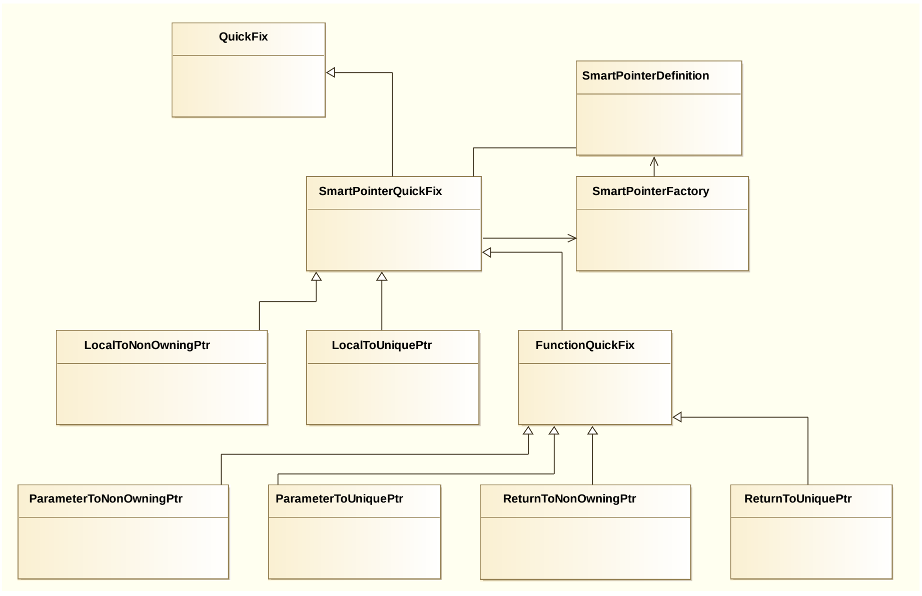
The following quickfixes were implemented in the `ch.hsr.ifs.cdt.smartor.quickfix.quickfixes` package:

- `LocalToNonOwningPtr`: Local raw pointer to `non_owning_ptr` refactoring.
- `LocalToSharedPtr`: Local raw pointer to `std::shared_ptr` refactoring (not used).
- `LocalToUniquePtr`: Local raw pointer to `std::unique_ptr` refactoring.

- `ParameterToNonOwningPtr`: Raw pointer parameter to `non_owning_ptr` refactoring.
- `ParameterToUniquePtr`: Raw pointer parameter to `std::unique_ptr` refactoring.
- `ReturnToNonOwningPtr`: Raw pointer return value to `non_owning_ptr` refactoring.
- `ReturnToUniquePtr`: Raw pointer return value to `std::unique_ptr` refactoring.
- `ExtractRawPointer`: Refactoring to extract raw pointer from a smart pointer argument using `get()` if the argument is passed to a raw pointer parameter.
- `ExtractRawPointerAll`: Mass-refactoring for `ExtractRawPointer` that is automatically applied to all uses of an identifier.

4.2.1. Architecture

To share the logic between the often similar quickfixes, a type hierarchy was introduced:



`QuickFix` is the base class for all quickfixes. It provides the framework to identify the marked statement in the sourcecode, traverse all occurrences of a variable, and to finally commit the changes to the sourcecode.

`SmartPointerQuickFix` is the base class for all quickfixes that refactor the use of raw pointers to smart pointers. It contains the code to replace initialization and assignment statements containing raw pointers with equivalents for smart pointers.

`FunctionQuickFix` is the base class for all quickfixes that modify the function itself and not only its body (i.e. parameters and return value). It contains the necessary logic to traverse all uses of the function, to be able to adjust the callers accordingly.

SmartPointerDefinition

The abstract class `SmartPointerDefinition` encapsulates the particularities of the individual smart pointers. In particular:

- The namespace the smart pointer is in.

- The name of the pointer.
- The header to include.
- Whether or not the header should be copied from the ones distributed with the plug-in.
- The initializer expressions (e.g. `std::make_unique`).

Note that two initializer expressions are available: `getInitializerExpression()` and `getEqualInitializerExpression()`. These are required because direct initialization (`getInitializerExpression()`) usually only requires the pointer, where an an assignement (`getEqualInitializerExpression()`) requires the pointer to be wrapped with the constructor call or e.g. a `std::make_unique` due to the explicit constructor.

The `SmartPointerDefinition` instances are created using the `SmartPointerFactory`, which provides a place to switch between various smart pointers (std, Boost, Qt, ...). So far only the Standard C++ Library version has been implemented though.

4.2.2. Control Flow

The primary entry point for the quickfix is the `QuickFix.modifyAST(index: IIndex, marker: IMarker)` method, which is called by the framework when the quickfix is activated. `modifyAST` employs the template method patter[1], in order to centralize most logic in the base classes and only deferring the particualrities to the subclasses.

The template method operations are:

- `getTargetStatement(node: IASTNode)`: Retrieves the primary target statement that should be refactored. This is required because the quickfixes operate on different targets such as local variables, parameters and return values. Based on the primary target statement, further target statements are identified as occurrences of the same variable.

[1]Wikipedia, *Template Method*.

- `getNewStatement(targetStatement: IASTNode, isReportedNode: Boolean): Option[IASTNode]`: Returns the refactored replacement statement for a target statement.
- `handleIncludes(r: ASTRewrite, ast: IASTTranslationUnit, destination: IFolder): Unit`: allows the subclasses to add includes to the ASTRewrite.

Applying the quickfix then involves:

- Finding the target statement:

  ```
  val targetStatement = getTargetStatement(
      astName)
  ```

- Processing all occurrences of the target statement:

  ```
  processOccurrences(targetStatement.get).
      foreach { x =>
    x._2 match {
    case null => r.remove(x._1, null)
    case _ => r.replace(x._1, x._2, null)
    }
  }
  ```

  Each statement is either removed if its replacement is null, or replaced otherwise. The replacement statements are created using `getNewStatement()`.

- Handling of includes:

  ```
  handleIncludes(r, ast, marker.getResource
      .getProject.getFolder("smartor"))
  ```

  `handleIncludes` allows subclasses to include header files. The subfolder for header includes is hardcoded to “smartor”.

- Committing the changes:

  ```
  r.rewriteAST.perform(new
      NullProgressMonitor);
  ```

  After this the changes are written back to the file.

4.2.3. Includes

The plug-in supports copying headers to a a project. This functionality is used to include headers shipped with the plug-in, such as `exempt_ptr.h`. This works especially well because many smart pointers, such as the boost implementation, are header only implementations which makes it easier to include the smart pointer implementations.

Includes are shipped in the `headerFile/ptor/` subdirectory of the plug-in, and are copied to the `smartor` subdirectory of the target project.

The handling of includes is implemented in `SmartPointerQuickFix.handleIncludes()`, which adds an **`#include`** statement to the source file if not already present and copies the header if `SmartPointerDefinition.copyHeader` is true.

To ensure that the header is found, the include path of the target project needs to be adjusted, which is done automatically by the `GnuCppIncludePathAdder`. `GnuCppIncludePathAdder` adds a new include path relative to the target workspace to the project, if not already present.

4.3. Checker-Quickfix association

The problems which can be reported by the checker are registered in plugin.xml:

```
<extension
    point="org.eclipse.cdt.codan.core.
        checkers">
    <checker
        class="ch.hsr.ifs.cdt.smartor.checker
            .Checker"
        id="ch.hsr.ifs.cdt.smartor.plugin.
            Checker"
        name="Smartor Warning Checker">
        <problem
```

```
                category="org.eclipse.cdt.codan.
                    core.categories.CodeStyle"
                defaultEnabled="true"
                defaultSeverity="Warning"
                description="Smartor Warning"
                id="ch.hsr.ifs.cdt.smartor.plugin
                    .error.LocalRawPointer"
                name="Smartor"
                messagePattern="Smartor: &apos;&
                    apos;{0}&apos;&apos;">
            </problem>
        </checker>
</extension>
```

Each problem has an `id`, which is used by the checker's `reportProblem()` method.

Each quickfix can then be registered to such an `id` using the `problemId` attribute in plugin.xml:

```
<extension
        point="org.eclipse.cdt.codan.ui.
            codanMarkerResolution">
        <resolution
            class="ch.hsr.ifs.cdt.smartor.
                quickfix.ConvertToUniquePtr"
            problemId="ch.hsr.ifs.cdt.smartor.
                plugin.error.localRawPointer">
        </resolution>
</extension>
```

If the checker now reports a problem, the associated quickfixes are offered to the user. Further can all problems individually be turned on or off in the configuration.

4.4. Tests

The project was developed using a test driven approach. Therefore a set of JUnit testcases has been implemented. All tests are in a separate project `ch.hsr.ifs.smartor.test` to avoid dependencies of the main plug-in on the test infrastructure. The tests have been written in Scala and reuse the test infrastructre of codan.

Because each test class can only test one quickfix, one test per quickfix had to be implemented.

4.5. Review

Unfortunately we were not able to implement all analysed features before end of the project. However, most of the more difficult problems such as node traversal, and finding callers of functions have been solved.

To document missing parts, unittests have been written for all missing features.

Note that the `ExtractRawPointer` and `ExtractRawPointerAll` refactorings are no longer required, because the raw pointer is now automatically extracted during the smart pointer refactorings. Instead a refactoring for owning smart pointers to transform the call to `get()` to `release()` would be required. For this `ExtractRawPointer` and the corresponding check in the checker can serve as basis.

The code is otherwise well structured, easy to extend and work with and should thus be a good starting point for further work. The architecture was designed primarily to share logic among similar quickfixes. Flexibility has been introduced where required, e.g. by encapsulating the smart pointer knowledge in `SmartPointerDefinition`.

4.6. Scala

The plug-in was written completely in Scala, as a learning experience and to make use of Scala features such as the pattern matching, which we hoped to be useful when analyzing the AST.

Scala is a programming language that runs in the JVM, is fully object-oriented, supports aspects of functional programming, and has a focus on a concise syntax. For an introducation to Scala "A Tour Of Scala"[2] can be recommended.

This section is going to highlight a couple of features that were found to be particularly useful.

4.6.1. Seamless integration with Java

Scala integrates seamlessly with java as Scala can call java code, and java can call Scala code. We were thus able to write our project fully in Scala, although the base classes we had to extend and the whole eclipse framework are pure java code. This works because Scala code compiles to bytecode for the JVM, just like normal java code.

4.6.2. val/var

Scala allows to specify identifiers as either `val` or `var`.

`var` is a normal variable, which can be reassigned and changed as we're used to from e.g. Java.

`val` is an immuatble variable, which cannot be reassigned, while the value remains mutable.

```scala
var v1: MyType = new MyType
v1 = new MyType //valid
val v2: MyType = new MyType
v2 = new MyType //invalid
v2.modify() //valid
```

By trying to avoid `var` and using `val` whereever possible it becomes much easier to track variables. Note how this effectively removes the problem of identifiers changing semantics as described in Section 3.3.2 on page 34.

[2]scala-lang.org, *A Tour of Scala*.

4.6.3. Pattern Matching

The pattern matching supported by Scala's `match` keyword, is a great feature that is applicable in a lot of cases and was used throughout the codebase. A nice example is down-casting of values:

```
x match {
  case name: IASTName => doSomething
  case node: IASTNode => doSomething
  case _ => println("Not a name nor a node")
}
```

This notation provides a concise and extensible way to downcasting, while still handling errors in the default case (`case _`).

For a detailed example how pattern matching was used in the implementation of the checker using extractor objects see 4.1.2 on page 67

4.6.4. Exception handling

A try-catch block has a return value just like any other function. It is thus possible to do something like this:

```
val ast = try {
  val tu: ITranslationUnit =
      getTranslationUnitViaEditor(marker);
  tu.getAST(index, ITranslationUnit.
      AST_SKIP_INDEXED_HEADERS);
} catch {
  case e: CoreException=> e.printStackTrace()
      ; return;
}
```

Because the last value in an expresssion is automatically the return value of the expression, `ast` is automatically assigned with the result of `tu.getAST()` if no exception occurs.

4.6.5. Option

The `Option` construct is extensively used throughout the code. It allows to transport an optional value, similar to `boost:optional`. This has the same semantic advantages, but the Scala version also provides a some nice syntactic possibilities:

- `None` represents an empty option, `Some` a non-empty one.
- Instead of testing the option using an if statement, foreach can be used:

  ```
  option.foreach{ val =>
    doSomething(val)
  }
  ```

- If the path were the option is emtpy needs to be handled, matching is more suitable:

  ```
  option match {
    case Some(val) => doSomething(val)
    case None =>
  }
  ```

4.6.6. foreach

`foreach` provides a nice notation to iterate over various containers such as `Option` and `Array`:

```
array.foreach{ val =>
  doSomething(val)
}
```

4.6.7. Concise notation

Scala supports a very concise notation by allowing to skip many unnecessary symbols such as semicolons, braces in function calls and variable identifiers. To give you an idea:

- Semicolons can be omitted when at the end of a line:
- Overridden functions don't require the full function declaration:

  ```
  //A function with a String as return value
  override def getLabel = "Quickfix Label"
  //A function with a Option[IASTNode] as return value
  override def getTargetStatement(node: IASTNode) = Some(node)
  ```

  Note that in the second example the argument only needs to be specified because `node` is used in the assigned expression.
- Braces can be ommited when calling getters:

  ```
  marker.getResource.getProject.getFolder("smartor")
  ```

This greatly helps to reduce the boilerplate code that needs to be written and to remove the visual clutter.

4.6.8. Java Conversions

Although it is possible to maintain a mixed maven project using the Scala-maven-plugin — that means using Scala and Java source files in one project — we decided to go with a pure Scala way for simplicity reasons. For that reason we had to convert some source files we work with into Scala.

The conversion was not very difficult and more or less a one to one translation from Java source code statements into Scala. To deal with collection classes the use of JavaConversions from Scala was benefical.

```
    import collection.JavaConversions._
    val includePaths = includeOption.getIncludePaths().toSeq + includePath
```

The snippet shows how implicit conversions for Java are imported. `includeOption` is a reference to an instance of `IOption`, a Java interface provided by the eclipse framework. After the import statement Scala is able to implicitly convert Java collections into Scala and vice versa. This way we didn't have to clutter the source code with type conversions.

4.6.9. Conclusion

Overall the use of Scala in this project has been a great experience. We gained many insights into different programming patterns than what we are used to from C++ and Java, and the integration between Java and Scala was smooth.

Especially the pattern matching helped to write code that is concise, reliable and easy to extend. While not all things are very intuitive (e.g. how Maps are handled), there were no insurmountable Problems, and it was overall a pleasure to use.

Because we were new to Scala, it certainly slowed us down in the beginning, but we think this learning experience payed off.

We can clearly recommend Scala for Eclipse plugins as we did face any Scala related issues and the resulting code was an improvement on what we could have done using Java.

5. Conclusion

This chapter reviews the results of our work and gives an outlook how project can be continued. It also includes our personal statements.

5.1. Accomplishments

During this bachelor thesis we examined how a plug-in can assists a developer at finding raw pointers and convert them into smart pointers. The following achievements are notable:

- The thesis contains an analysis of pointer roles as well as their possible conversions into smart pointers.
- Although the ownership tracking approach was not successful it analyses where the problems lie, which could be useful for future work on the topic.
- We created an Eclipse CDT plug-in that finds raw pointer issues and marks them in the editor. The developer may choose to manually resolve the issue or invoke a quick fix we provided for various cases. The quick fix automatically converts the pointer into a suitable smart pointer.
- The plug-in has support for smart pointers introduced with C++11 as well as a newer smart pointer proposal that is going to be implemented in a future C++ version.

- We verified that Scala is benefical in many areas of our project. The pattern matching mechanism resulted in cleaner and more readable source code. It is a neat way to match against types and thus helps avoiding conditional statements for type checking.

5.2. Future Work

There are multiple options to continue this project:

- The handling of all special cases can be finished based on the already implemented test cases.
- Pointer to strings and pointer to arrays provide a field of special cases for further analysis and would be also a useful addition to the plug-in.
- The heuristic to determine ownership can be improved and extended.
- Pointer tracking using symbolic execution would allow to make large scale refactorings throughout uses of a raw pointer.

5.3. Personal Statements

This section contains our personal statements on this project.

5.3.1. André Fröhlich

The subject of our thesis gave me once more the opportunity to develop an Eclipse plug-in and to intensively deal with one aspect of the C++ programming language. In addition, this project gave me the chance to deal with Scala and functional programming, what is not part of the regular curriculum at the University of Applied Sciences Rapperswil.

I could learn a lot about C++ again. Christian Mollekopf had more experience in C++ and better sense for the pointer issue as I did. I was able to greatly benefit from working with him. I also benefited from the supervision of Prof. Peter Sommerlad, who often gave us valuable inputs. I used to see smart pointers only as better alternative to manual memory management using raw pointer. However, at the project I learned to appreciate the semantic expressiveness of smart pointers.

I did enjoy learning Scala. Some elements of Scala also helped me to better understand concepts of other programming languages. Mirko Stocker's knowledge of Scala is immense and the possibility to get his help was very useful. I was able to develop a better sense for functions without side effects and the use of constant variables. Scala also helps to write clearer, less bloated code. I can imagine using Scala instead of Java for future projects.

During the project we had often problems with our build system using Maven and Tycho. It was not always easy to solve the problem and it took a lot of time that we would have better used for something else.

In the second last meeting, two weeks before the end of the project, we found out that we did not understand our task, as we should. For me, this was a very frustrating experience. We falsely assumed that smart pointers are primarily a better way for memory management as raw pointers in C++. We knew that it is possible to demonstrate the semantic meaning of pointers as we also analyzed the various pointer roles. However, we have failed to understand that the roles should be used as the starting point for our analysis. Of course, we have the remaining time as good as possible in order to correct our mistake.

Finally, the project was pretty exhausting, but I was able to learn a lot. The topic is very interesting and the Scala a language was enrichment both for the project, as well as for my experience.

5.3.2. Christian Mollekopf

As our second project using eclipse as a platform and maven as buildsystem it was somewhat frustrating to see that we were running again in many setup and buildsystem problems. Although most problems could be solved in the end and often proved to be a real issue in a configuration file or the other, the errors often appeared without any specific action taken and seldomly with a useful error description. This cost a lot of time during the project, that would have been better spent otherwise. It was however a familiar experience working with the eclipse frameworks with its (overly?) loose coupling.

As we established in our second last meeting, that we apparently didn't have the same understanding of our task as our advisors was obviously also not necessarily what we had hoped for. According to our task description our target was to analyse how raw pointers can be refactored into smart pointers, and provide suitable refactorings. There are however different approaches to that. Before the project my personal understanding of what a smart pointer is for, was the memory management, and never really the revealing of semantics. I though of it really just as a way for the compiler to enforce that there is no memory leak. Coming from that line of thought, there are only two major smart pointers: `std::unique_ptr` and `std::shared_ptr`, and naturally I also assumed that the primary motivation for using smart pointers was to get rid of the error prone manual memory management.

Of course I started to see the value of the semantics, especially as `non_owning_ptr` came into the picture, and we suddenly started to discuss things like `boost::optional` (which I would never have introduced to the project, due to my understanding). However, I've seen the semantics as "bonus points" while the primary target still was to remove the manual resource management. Of course I should have realized that this is not what was expected, but at this point I was already deep down the rabbit hole called "tracking pointer ownership". The ownership of a pointer is indeed an important feature to be able to automatically refactor a memory management system, but not so much to to attach semantics to a pointer (where we either need no tracking at all, or only have to track within a very limited scope). In the end we wanted to solve a problem which was much to difficult within the scope of this project, and didn't realize that this wasn't the problem we were expected to solve. We also realized too late that the problem we wanted to solve was much too difficult.

After realizing, we unfortunately only had two weeks left, within which we basically had to rewrite the whole analysis, as the approach we started with didn't really fit anymore. For instance it didn't make sense anymore to try to identify specific patterns in the code which can be refactored (which we did because we wanted identify specific constellations in which we can determine the ownership), if all we really had to do is find declarations and offer all available refactoring options. While the "new" task is a lot simpler and far more straightforward, it still requires a lot of time to implement, which we didn't have at this point in time. Nevertheless we tried our best to still achieve something useful within that timeframe and hope the result goes now more in the expected direction.

However, there were also very positive aspects in the project. I enjoyed a lot to work with Scala, which was a breath of fresh air in the Java landscape. I think Scala has a lot of nice concepts, and although I'm far from really understanding the full capabilities of the language, I still got a nice insight and would definitely consider to write my next project where I'm forced to interact with Java using Scala.

I also ended up thinking a lot about programming languages in terms of semantics of expressions, where even seemingly trivial things, like the identifier not being the same as the underlying variable, and why it can get messy if the underlying value can change, really started to become clear to me. I got to appreaceate the conveyed semantics by various language constructs and smart pointers more, `optional` being one of constructs I always made a circle around so far, and where I now can think of loads of places where I wished it was used (default constructed values really can't be the solution to that problem).

Overall, the project ended up being a bit strenuous, but still proved to be a great learning experience. Unfortunately we couldn't get everything to the state where we would have liked it, but I think we still laid a foundation that a future project could build upon.

A. Organisation

Every software development project has to have some sort of organisation. This chapter gives an overview on the approach we used and how the project was planed. It shows how the plan actually turned out.

A.1. Approach

The regulation of our school limits the time we can and should spent on this project. Therefore the scope of our project has to be open. Much of the organisation is specified by the supervisor of the thesis. Prof. Sommerlad favored weekly meetings in which the next steps as well the previous are discussed.

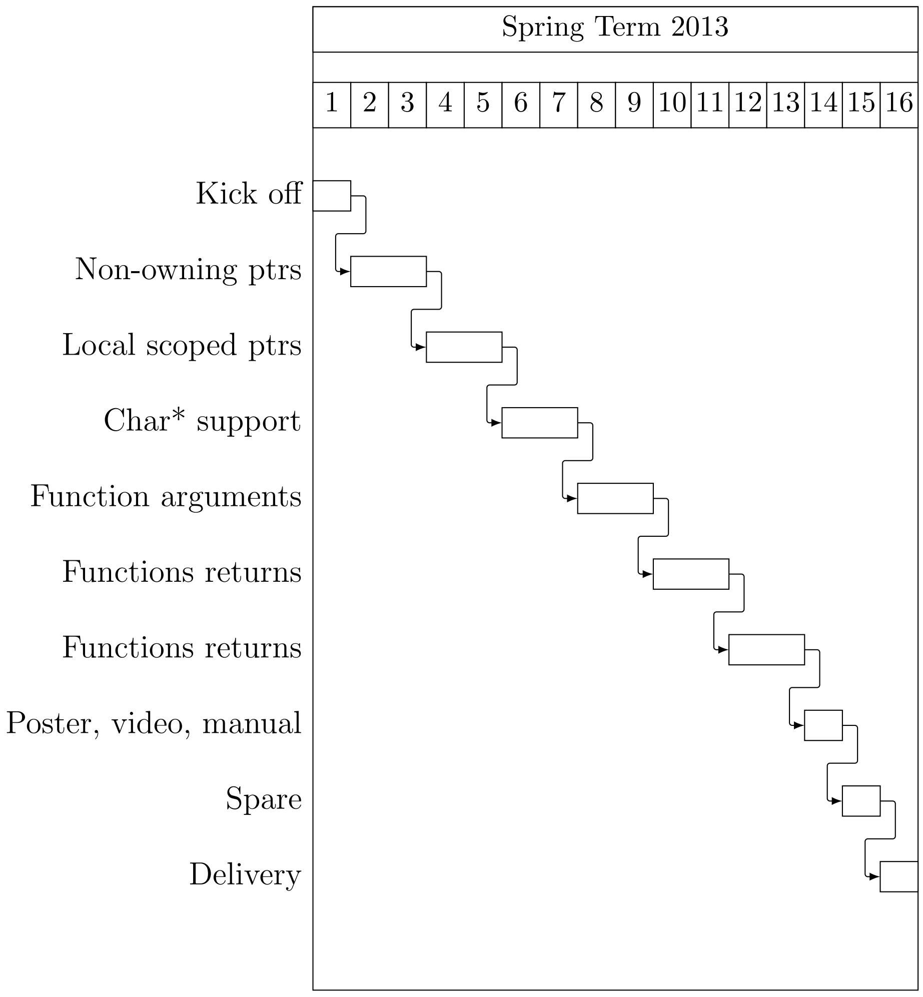
Once a week — usually on monday afternoon — we met with professor Sommerlad and his assistant in the working room of the Institue for Software at the University of Applied Sciences Rapperswil. On the agenda is to tell him what happend in the past week, discuss open problems and agree on the next steps. This leads to an incremental software development approach.

For collaboration we decided to use Redmine. It provides support for issue tracking, displaying gantt charts, document management and a project wiki.

A.2. Project Plan

Although the next steps are discussed week by week, we created a project plan first. It serves as a rough guide helping ous to determine our next steps every week.

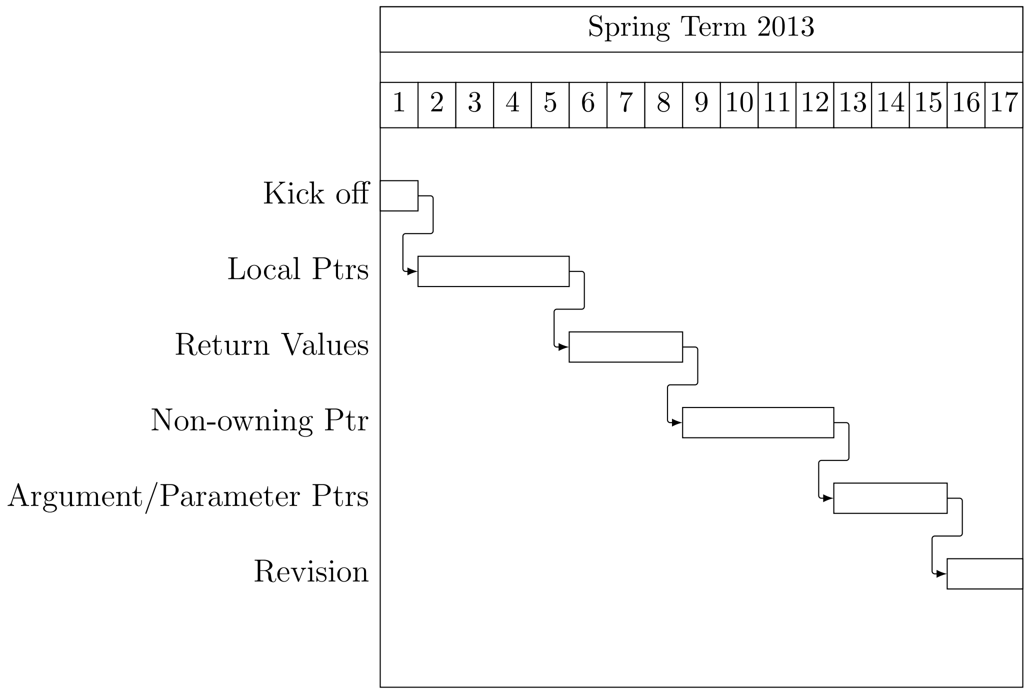
A.2.1. Intended Plan



Of course we couldn't follow the plan as there are many unexpected outcomes during the development. With the weekly meetings we could flexible change our course and still having a direction.

The plan shows we falsely assumed this semester had fourteen weeks as usual but actually it had fifteen weeks.

A.2.2. Actual Plan



At the beginning we have used more time, because we had problems with the build system and also worked out our objectives. In addition, we had little experience with Scala.

In the second last meeting we realized that we did not understand our task, as we should. We falsely assumed that smart pointers are primarily a better way for memory management as raw pointers in C++ and built our analysis on that assumption. In the last two weaks our main focus was to revise our work.

A.3. Time Report

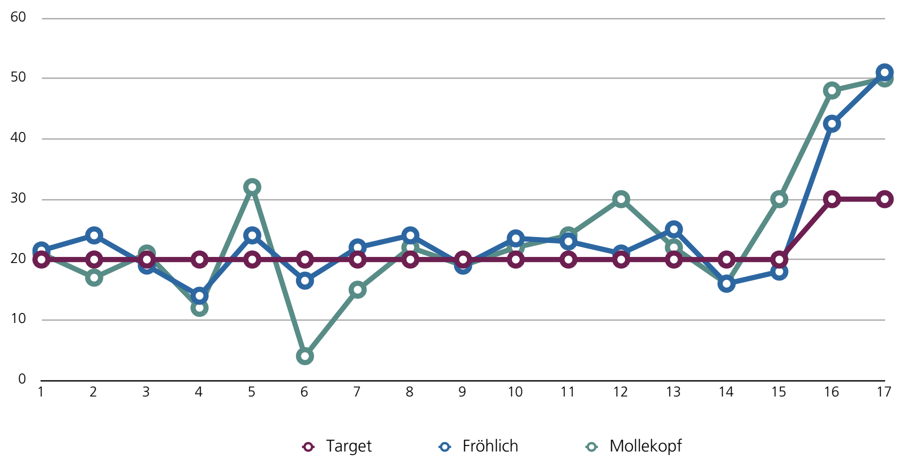


Figure A.1.: Time spent per week

The figure shows how much hours we spent working on this project per week. Time shortfalls due to illness or vacation were compensated over the whole project duration.

The effort for this module should correspond to 12 ECTS credit points or 360 hours of work. The total time spent on this thesis per person:

- **André Fröhlich** 404 hours
- **Christian Mollekopf** 405 hours

These numbers include an estimation of the effort that was done after the print of this thesis.

B. Environment

Creating Eclipse plug-ins is a complex task. This chapter gives an overview about the tools used in this project. It also severs as a brief introduction to the relevant areas of the Eclipse framework.

B.1. Tools

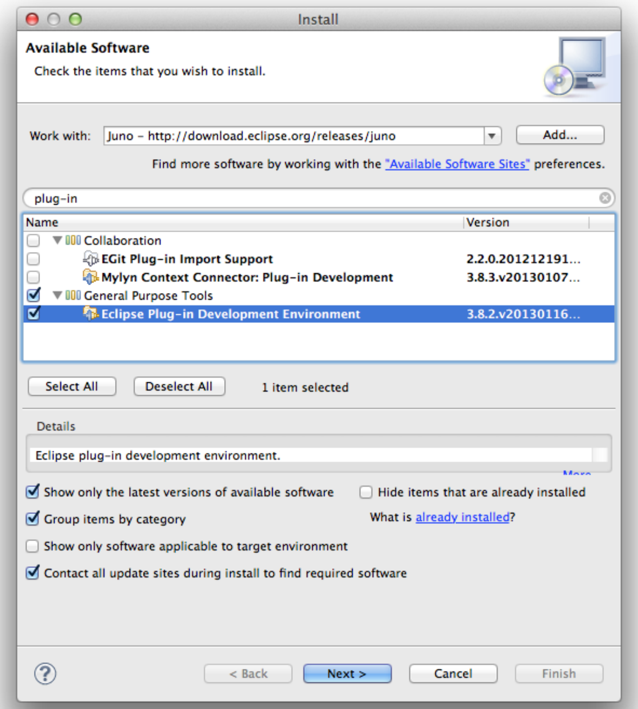
- **Eclipse C/C++ Development Tooling** is a project based on the Eclipse plattform which provides a fully functional C and C++ integrated development environment.
- **Eclipse** is a software development environment with an extensible plug-in system.
- **Git** is a distributed source code management system.
- **Jenkins** is an open-source continous integration server.
- **Maven** is a build manager for Java projects.
- **Redmine** a web application for project management.
- **Tycho** is set of Maven plug-ins and extensions for building Eclipse plugins.
- **GCC 4.8** is used to compile C++ files and support the new C++11 standard.
- **LaTeX** is the document markup language we used to write this thesis.

B.2. Eclipse Plug-ins

To develop an Eclipse plug-in using Scala several plug-ins are required. This section shows what plug-ins we used and what they are good for.

B.2.1. Installation

Where not otherwise specified, we used the Eclipse Juno update site. To install a plug-in simply go to "Help → Install New Software...".


Install

Available Software

Check the items that you wish to install.

Work with: Juno - http://download.eclipse.org/releases/juno

Find more software by working with the "Available Software Sites" preferences.

plug-in

Name	Version
☐ Collaboration	
☐ **EGit Plug-in Import Support**	2.2.0.201212191...
☐ **Mylyn Context Connector: Plug-in Development**	3.8.3.v20130107...
☑ General Purpose Tools	
☑ **Eclipse Plug-in Development Environment**	3.8.2.v20130116...

1 item selected

Details

Eclipse plug-in development environment.

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

Figure B.1.: The Eclipse dialog to install new software.

The figure shows the Eclipse dialog to install new software. Under "Work with:" the default Juno update site is selected. The screenshot also shows how to use the search field. While using the search field, it may happen that Eclipse responds slowly.

This dialog can also be used to install software from other update sites. Ether the address can be entered directly or may be managed using the "Add" button.

B.2.2. Eclipse Plug-in Development Environment

If you don't use the Eclipse RCP package you need to install this plug-in to get the environment for developming Eclipse plug-ins.

B.2.3. Scala IDE for Eclipse

The Scala IDE for Eclipse is a powerful plug-in providing a complete IDE to develop in Scala.

The update site for Eclipse Juno is: http://download.scala-ide.org/sdk/e38/scala210/stable/site

B.2.4. C/C++ Development Tools

You don't have to install this plug-in into your plug-in development Eclipse instance, but it may be convenient. If it isn't installed it can be provided using a .target file.

B.2.5. Jeeeyul's Eclipse Themes - Chrome

Originally, this plug-in is designed to make Eclipse Juno more beautiful. As it often happens during the development of Eclipse plug-ins that more then one Eclipse instance is running, we use this plug-in to give each instance a different appearance.

It can be installed using this update site: `https://raw.github.com/jeeeyul/eclipse-themes/master/net.jeeeyul.eclipse.themes.updatesite`

After the installation it can be activated in the Eclipse preferences under “General → Appearance”. Once it is active it can be customized in “General → Appearance → Chrome Theme”.

B.2.6. IFS CDT Testing

This plug-in helps to test Eclipse CDT plug-ins.

The update site is found on: `http://dev.ifs.hsr.ch/updatesites/cdttesting/juno`

B.2.7. Issues

Run Configurations

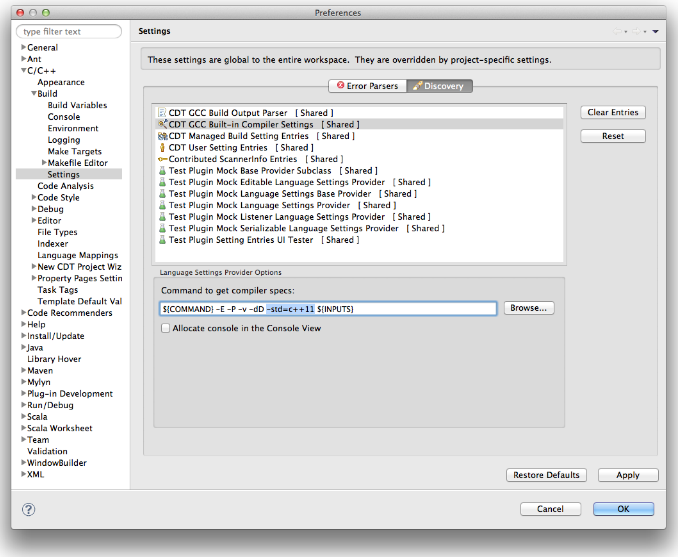
To avoid the “Setup Diagnostics” dialog while testing the plugin, it is recommended that the Scala IDE isn’t launched with the plug-in. This can be achieved by editing the “Run Configurations”. In the “Plug-ins” tab “Launch with” should be changed to “plug-ins selected below only”. Disable all unnecessary plug-ins like those of the Scala IDE. However, a Scala library is however needed and the plug-in providing it must remain enabled.

As an alternative it is also possible to select “org.eclipse.platform.ide” instead of “org.eclipse.sdk.ide” in the “Run Configuration” in the “Main” tab under “Run a product”.

Alternatively a .target file can be used. After the activation its configuration is workspace wide active.

Eclipse CDT C++11 support

Support for C++11 is not provided out of the box by the Eclipse CDT version we used. This section describes what may be necessary.



These settings are global to the entire workspace. They are overridden by project-specific settings.

- CDT GCC Build Output Parser [Shared]
- CDT GCC Built-in Compiler Settings [Shared]
- CDT Managed Build Setting Entries [Shared]
- CDT User Setting Entries [Shared]
- Contributed ScannerInfo Entries [Shared]
- Test Plugin Mock Base Provider Subclass [Shared]
- Test Plugin Mock Editable Language Settings Provider [Shared]
- Test Plugin Mock Language Settings Base Provider [Shared]
- Test Plugin Mock Language Settings Provider [Shared]
- Test Plugin Mock Listener Language Settings Provider [Shared]
- Test Plugin Mock Serializable Language Settings Provider [Shared]
- Test Plugin Setting Entries UI Tester [Shared]

Command to get compiler specs:

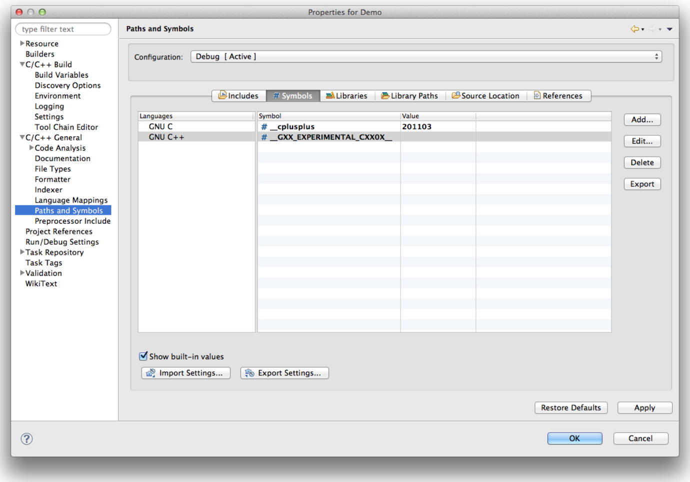
```
${COMMAND} -E -P -v -dD -std=c++11 ${INPUTS}
```

☐ Allocate console in the Console View

Figure B.2.: The addition of the -std=c++11 option is required.

The compiler options can be altered in the preferences. Under "C/C++ Build → Setting" in the "Discovery" tab, the "CDT GCC Built-in Compiler setting" can be specified. It is required to add the "-std=c++11" option as showed in figure B.2.

Depending on the version of GCC it can be required to add aditional symbols. Under "C/C++ General→ Path and Symbols" in the "Symbols" tab, the symbols can be edited as shown in figure B.3.



Paths and Symbols

Configuration: Debug [Active]

Languages	Symbol	Value
GNU C	# __cplusplus	201103
GNU C++	# __GXX_EXPERIMENTAL_CXX0X__	

☑ Show built-in values

Figure B.3.: Additional symbols can be defined.

B.2.8. Target File

When the plug-in is run Eclipse launches a new instances with the plug-in and its dependencies. A .target file allows to specify all required plug-in dependencies a plug-in needs to launch. Eclipse automaticaly downloads them if necessary. It is an alternative to configuring run configurations ins Eclipse. The target file can be activated by opening it and clicking on "Set as Target Platform".

B.3. Eclipse Static Code Analysis

Eclipse provides a framework for static code analysis called Codan (CODe ANalysis). It allows plug-ins to provide checkers which perform real time analysis on the code[1].

[1]Eclipsepedia, *CDT/designs/StaticAnalysis*.

B.4. Eclipse Abstract Syntax Tree

An abstract syntax Tree (AST) represents the syntactic structure of source code[2]. Eclipse CDT provides an interface to its internal syntax tree data structure, allowing plug-in developers to parse and manipulate the source code.

Interface

Eclipse provides a rich set of Java interfaces to access its abstract syntax tree. The following list is a selection and should give you an overview of the fundamental interface we used for the project.

- **IASTNode** Root node of the abstract syntax tree
  - **IASTAttributeOwner** AST node with attributes
    - **IASTDeclarator** Base interface for a declarator
    - **IASTSimpleDeclaration** Simple declaration
    - **IASTStatement** Root node for statements
      - **IASTCaseStatement** case statement
      - **IASTCompoundStatement** A block of statements
      - **IASTDeclarationStatement** Statment for declaration
      - **IASTDoStatement** do statement
      - **IASTExpressionStatement** Expression statement
      - **IASTForStatement** for statement
      - **IASTIfStatement** if statement
      - **IASTReturnStatement** return statement
      - **IASTSwitchStatement** switch statement

[2]Wikipedia, *Abstract Syntax Tree*.

IASTWhileStatement while statement

IASTDeclaration Root node for declaration

IASTFunctionDefinition Function declaration

IASTSimpleDeclaration Simple declaration

IASTInitializer Initializer for a declarator

IASTEqualsInitializer Initializer with equals sign (=)

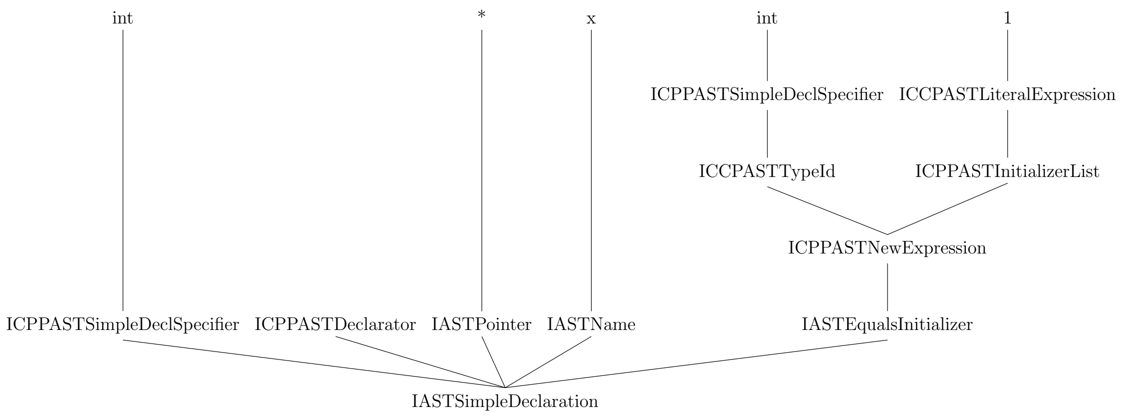
IASTInitializerList Braced initializer list

Example

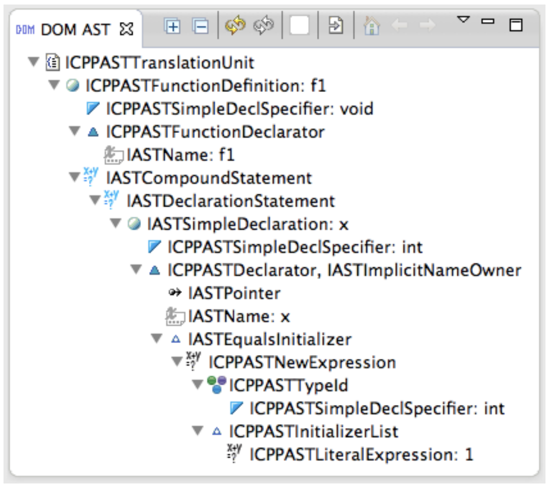
This example gives you a basic understanding how the abstract syntax tree in Eclipse works. The declaration is compound a simple declaration specifier containing the data type and a declarator containing all the other details. It is itself compound of multiple elements: one for the pointer, one for the name and one for the equals initializer. You can imagine this works somehow like a russian matryoshka doll. With the Scala match expression we do the pattern matching to navigate through the abstract syntax tree.

```
int *x = new int {1};
```

The code line above shows a simple statement written in C++. The tree below shows how this code is stored internaly.



The names are rather long because of the vast amount of available interfaces. With some basic knowledge of C++ it is not too difficult to understand what interface represents what part of the code. However, the AST DOM view is very handy to find the appropriate interface.


DOM AST

```
ICPPASTTranslationUnit
  ICPPASTFunctionDefinition: f1
    ICPPASTSimpleDeclSpecifier: void
    ICPPASTFunctionDeclarator
      IASTName: f1
    IASTCompoundStatement
      IASTDeclarationStatement
        IASTSimpleDeclaration: x
          ICPPASTSimpleDeclSpecifier: int
          ICPPASTDeclarator, IASTImplicitNameOwner
            IASTPointer
            IASTName: x
            IASTEqualsInitializer
              ICPPASTNewExpression
                ICPPASTTypeId
                  ICPPASTSimpleDeclSpecifier: int
                ICPPASTInitializerList
                  ICPPASTLiteralExpression: 1
```

Figure B.4.: The DOM AST View

If the view is not visible initially you can use the “Show View” Dialog under “Window”, “Show View”, “Other” to get it. Clicking on a node highlights the correspondent part of the code.

B.5. Testing

Automated testing with JUnit is possible by extending the CheckerTestCase class. It provides an interface to load code and run a checker over it.

```
def testLocalRawPointer(): Unit = {
  loadCodeAndRunCpp("void foo() { int *x =
      new int(3); delete x; }")
  checkErrorLine(1, ErrorId)
}
```

For Java developers the test facility provides with `getAboveComment()` a way to load C++ code from the comment above the test method which is particularly usefull for larger code snippets with multiple lines. Since the Eclipse test facility is written for Java developers, the `getAboveComment()` Method doesn't work for our Scala test classes. Fortunately, this is also not necessary since Scala supports multi-line strings.

```
def testLocalRawTwoDelete(): Unit = {
  loadCodeAndRunCpp(
   """void foo() {
         int *y;
         delete y;
         int *x = new int(3);
         delete x;
     }""")
  checkErrorLine(2, ErrorId)
}
```

B.6. Build System and Continous Integration

Building Eclipse plug-ins may be a complex and difficult task. We have opted for the more modern approach using Maven. To do continous integration we decided to use Jenkins which regularly build the Maven project.

B.6.1. Tycho

There two main approaches to build Eclipse plug-ins: Apache Ant with PDE and Apache Maven with Tycho. We used the more modern build technqiue using Maven and Tycho.

Maven Tycho is a set of Maven plugins. There is no installation required as Maven automaticaly download the plugin the first time it needs it.

Normally all building information is in the Maven POM file. However, Eclipse RCP application and plug-ins are using a manifest file for that. Tycho is able to sychronize these two approachs bringing both together. To do so it extends the maven dependency model.

Many eclpse artifacts are not stored in maven repositories but in p2 repositories instead. With Tycho Maven is able to use these p2 repositories to solve building dependencies. Tycho is using a manifest first approach.

Metadata

All Maven configuration data is stored in the pom.xml file. Metadata may be distributed over several files:

- bundle manifest (META-INF/MANIFEST.MF)
- build.properties
- Feature.xml
- .product files
- .target files
- category.xml

B.6.2. Documentation

To build this thesis we used a LaTeXplugin for Maven called LaTeX Mojo. This plug-in also helps to generate a simple LaTeXproject skeleton:

```
mvn archetype:create \
    -DarchetypeGroupId=org.codehaus.mojo \
    -DarchetypeArtifactId=latex-maven-
       archetype \
    -DarchetypeVersion=1.1 \
    -DgroupId=ch.hsr.ifs \
    -DartifactId=thesis
```

The document can be build using the latex:latex Maven goal:

```
cd thesis
mvn latex:latex
```

There is additional documentation about this plug-in the project website: http://mojo.codehaus.org/latex-maven-plugin/.

B.6.3. Jenkins

Jenkins is an open-source continous integration tool written in Java[3]. We setup two projects, one for the plug-in and antoher one for the documentation. Since both projects are build using Maven the setup is straightforward. In order that Jenkins is able to fetch the latest sources we use the Git Jenkins plug-in.

[3]**Eclipse:Jenkins**.

C. Developer Guide

This chapter guides through the process of setting up a development environment to extend the plug-in described in this thesis.

C.1. Java, Scala and other tools

For developing the following tools are required:

- Java
- Scala
- Git
- Maven

For more information about the tools, refer to the environment section of this appendix.

C.2. Installing the Eclipse IDE

Eclipse can be downloaded from eclipse.org . Any package can be used as the needed plug-ins can be installed later. A good choice is the “Eclipse IDE for Java Developers” oder the “Eclipse for RCP and RAP Developers” package. It is also possible to use the Scala-IDE Elipse distrubtion that can be found on scala-ide.org.

C.3. Plug-ins

There is a set of plug-ins we recommend.

- C/C++ Development Tools
- Eclipse for RCP and RAP Developers
- Tycho Project Configuratiors
- Scala IDE for Eclipse
- IFS CDT-Testing Feature
- Jeeyul's Themes

Jeeyul's Themes allows to modify the appearance of Eclipse. It is helpful to give multiple Eclipse instances different colors in order to easily distinguish them.

C.4. Importing the project

The source is avaiable as a Git repository. A virtual server is avaible that provides a central repository. Since the life time of the server is limited, this repository may no longer available. A copy of the repository is also provided on the compact disc of this thesis.

The repository contains the source of the plug-in as well as of the documentation.

C.5. Source overview

The source is distributed over multiple folders.

- **ch.hsr.ifs.cdt.smartor** contains the plug-in source project
- **ch.hsr.ifs.cdt.smartor.feature** contains the plug-in featuer project

- **ch.hsr.ifs.cdt.smartor.test** contains the plug-in test project
- **ch.hsr.ifs.cdt.smartor.p2repository** contains the plug-in p2repository
- **documentation** contains the LaTeXdocumentation

The root folder as well the subfolders are Maven projects, containing its `pom.xml` file. Note that the documentation is also built using Maven.

C.5.1. Manifest and plugin.xml

Dependencies are set in the `MANIFEST.MF` and `plugin.xml` files. Eclipse provides an “Plug-in Manifest editor” helping with editing these files.

C.5.2. Target file

The target is used to store the plug-ins required to run the plug-in. When it is activated the plug-ins listed in it doesn’t need to be installed in the Eclipse instance that is used to develop the plug-in. Target files are activated workspace wide and not per project.

C.5.3. Markers

Markers are part of the Eclipse CDT Codan project. As part of the static code analysis framework they provide an interface to mark passages of the source in an editor. Multiple markers per line are possible as they are stacked.

The entry point for markers is the `Checker` class located in the `ch.hsr.ifs.cdt.smartor.checker` package. It extends the `AbstractIndexAstChecker` class and overrides the `processAst` Method. Scalas **`match`** Statement is used to distinguish the differnt cases.

Possible problem cases are defined in the `Problemdefinitions.scala` file in the `ch.hsr.ifs.cdt.smartor` package.

Testing is done with the `SmartorCheckerTest` class. It is located in the `ch.hsr.ifs.cdt.smartor.test` package of the test project and extends the `CheckerTestCase` class and is a JUnit test case.

C.5.4. Quick fixes

The framework also provides a facility to provide quick fixes for markers. Activating these gives an user the opportunity to transform the reported problem.

The entry point for quick fixes is the the `QuickFix` class located in the `ch.hsr.ifs.cdt.smartor.quickfix` package. The package also contains subclasses providing the different quick fix functionality for each case.

D. User Manual

This chapter shows the installation process of the Smartor plug-in, how it is used and its known issues. It is assumed that the reader is familar with using Eclipse.

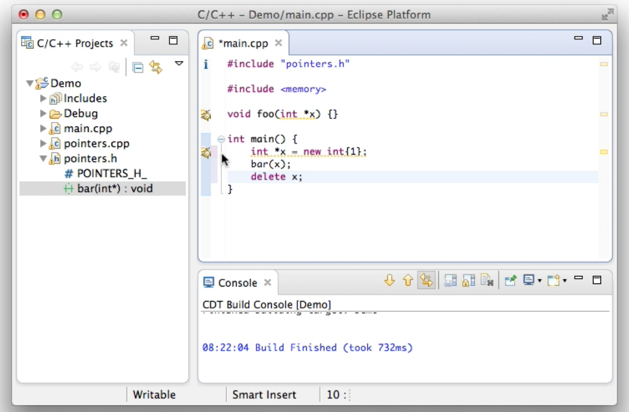
D.1. Installation

The following steps guides you through the installation process.

1. Start Eclipse
2. Select “Help→Install New Software ...”
3. Type or paste the URL of the update site into “Work with:”
4. Enable the checkbox to select the plug-in
5. Click the finish button.

D.2. Guide

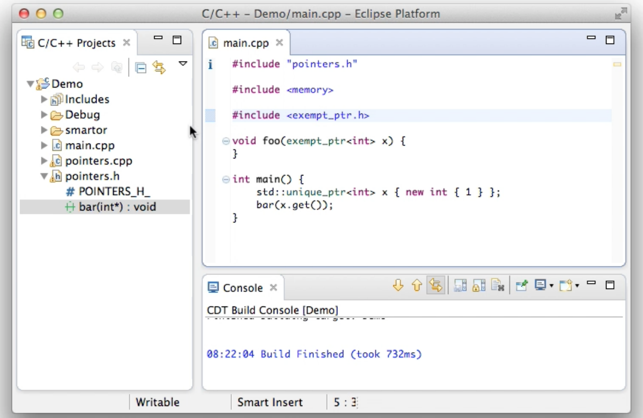
The plug-in is active after the installation. While writing C++ files markers are triggered by the use of raw pointers. The occurrence gets a yellow underline and on the left side of the editor a icon is placed to indicate the issue. It is possible to resolve the issue manually or the activate a quickfix using the icon in the editor or using a shortcut — usually **Strg+1** or **Cmd+1** on Mac.



```
#include "pointers.h"

#include <memory>

void foo(int *x) {}

int main() {
    int *x = new int{1};
    bar(x);
    delete x;
}
```

```
CDT Build Console [Demo]

08:22:04 Build Finished (took 732ms)
```

Figure D.1.: The plug-in displays raw pointer issues.



```
#include "pointers.h"

#include <memory>

#include <exempt_ptr.h>

void foo(exempt_ptr<int> x) {
}

int main() {
    std::unique_ptr<int> x { new int { 1 } };
    bar(x.get());
}
```

```
CDT Build Console [Demo]

08:22:04 Build Finished (took 732ms)
```

Figure D.2.: The plug-in removes delete statements, calls the get() function and adds include statements if needed.

As shown in figure D.2 the plug-in can also add include statements and in the case of a non-owning pointer it also adds the needed header file to the project and setups the include path in the project preferences. It also removes delete statements and uses the get() function of a smart pointer if needed.

D.3. Known Issues

The plug-in only operates as part of eclipes code analysis. To manually trigger the analysis process, right click on a file in the “Project Explorer” and select “Run C/C++ Code Analysis”.

The functionality to convert pointers is limited by the cases mentioned in the Analysis section of this thesis.

E. Nomenclature

AST Abstract syntax tree — represents the source code as tree data structure.

Best Practice A method or technique that yields results superior those achieved with other means.

Callee The callee is the function that is being called.

Caller The caller is the code that calls this function.

Checker Responsible for placing markers.

Codan Eclipse's static *cod*e *an*alysis project.

Coding Convention A set of guidelines that recommend a programming style.

Dangling Pointer A pointer that does not point to a valid object.

Eclipse RCP A platform for building rich client applications.

Empty Smart Pointer A smart pointer that points to no object (shall not be dereferenced).

Indentifier An identifier in the source code (e.g. in `int x;` is x the identifier).

Intention Revealing Language Construct Language constructs such as references, `boost::optional` or smart pointers.

JVM Java Virtual Machine.

Manifest file Contains project information for building Eclipse RCP applications and plug-ins and can be used with Maven through Tycho.

Marker Marks an issue in the editor.

Maven A build system with dependency managment and a plug-in system.

Move Semantics C++11 feature that models the move of a value, rather than a copy or a reference.

Non-owning Pointer A pointer role where the pointer does not owne the resource.

Null Smart Pointer A smart pointer that doesn't own any pointer.

OSGi Open Services Gateway initiative — a module system for the Java platform.

Owning Pointer A pointer role where the pointer own the resource.

POM Project Object Model — contains project information for Apache Maven

Pointee The object/resource a pointer points to.

Pointer Provides direct access to a memory address on a low level.

Quick fix Possible solution to an issue like a small refactoring.

RAII Resource Acquisition Is Initialization - A programming idiom that deals with resource allocation and deallocation.

Refactoring Altering code without changing its behavior.

Shared Pointer A smart pointer for shared ownership.

Smart Pointer simulates a pointer but can reveal pointer role semantics and provide memory management.

Symbolic execution Analysis of programs by tracking symbolics.

Target file Allows to specify all required plug-in dependencies that a plug-in needs to launch.

Tycho A Maven plug-in for building Eclipse RCP applications and plug-ins.

Unique Pointer A smart pointer for single ownership.

F. Bibliography

Boost. *boost::shared_ptr class template*. [Online; accessed 28-04-2013]. 2013. URL: http://www.boost.org/doc/libs/1_53_0/libs/smart_ptr/shared_ptr.htm.

Brown, Walter E. *N3514: A Proposal for the World's Dumbest Smart Pointer*. [Online; accessed 28-04-2013]. 2012. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3514.pdf.

Eclipsepedia. *CDT/designs/StaticAnalysis*. [Online; accessed 25-05-2013]. 2013. URL: http://wiki.eclipse.org/index.php?title=CDT/designs/StaticAnalysis&oldid=283283.

Hinnant, Howard. *Why isn't there a `std::shared_ptr<T[]>` specialisation?* [Online; accessed 28-04-2013]. 2013. URL: http://stackoverflow.com/a/8947700.

Lavavej, Stephan T. *N3588: make_unique*. [Online; accessed 28-04-2013]. 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3588.txt.

Odersky, Martin, Lex Spoon, and Bill Venners. *Programming in Scala*. Second Edition, Updated for Scala 2.8. Walnut Creek, Calif: artima, 2010. ISBN: 978-0981531649.

qt-project.org. *QObject Trees & Ownership*. [Online; accessed 25-05-2013]. 2013. URL: http://qt-project.org/doc/qt-4.8/objecttrees.html.

— *Qt Project*. [Online; accessed 25-05-2013]. 2013. URL: http://qt-project.org/.

scala-lang.org. *A Tour of Scala*. [Online; accessed 06-06-2013]. 2013. URL: http://www.scala-lang.org/node/104.

Software, IDE. *DOOM3 Sourcecode*. [Online; accessed 25-03-2013]. 2013. URL: https://github.com/id-Software/DOOM-3/.

Sutter, Herb. *Trip Report: ISO C++ Spring 2013 Meeting.* [Online; accessed 28-04-2013]. 2013. URL: `http://isocpp.org/blog/2013/04/trip-report-iso-c-spring-2013-meeting`.

Toit, Stefanus Du. *N3337 Working Draft, Standard for Programming Language C++: The class template auto_ptr is deprecated. Note: [The class template unique_ptr 20.7.1 provides a better solution. -end note].* 2012.

Wikipedia. *Abstract Syntax Tree.* [Online; accessed 25-05-2013]. 2013. URL: `http://en.wikipedia.org/wiki/Abstract_syntax_tree`.

Wikipedia. *C++ Standard Library.* [Online; accessed 25-05-2013]. 2013. URL: `http://en.wikipedia.org/wiki/C%2B%2B_Standard_Librar`.

Wikipedia. *Factory Method.* [Online; accessed 25-05-2013]. 2013. URL: `http://en.wikipedia.org/wiki/Factory_method_pattern`.

Wikipedia. *Pointer (computer programming).* [Online; accessed 01-06-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Pointer_(computer_programming)&oldid=554897073`.

— *Smart pointer.* [Online; accessed 01-06-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Smart_pointer&oldid=556415178`.

Wikipedia. *Template Method.* [Online; accessed 25-05-2013]. 2013. URL: `http://en.wikipedia.org/wiki/Template_method_pattern`.

Yasskin, Jeffrey. *N3609: string_view: a non-owning reference to a string, revision 3.* [Online; accessed 28-04-2013]. 2013. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3609.html`.