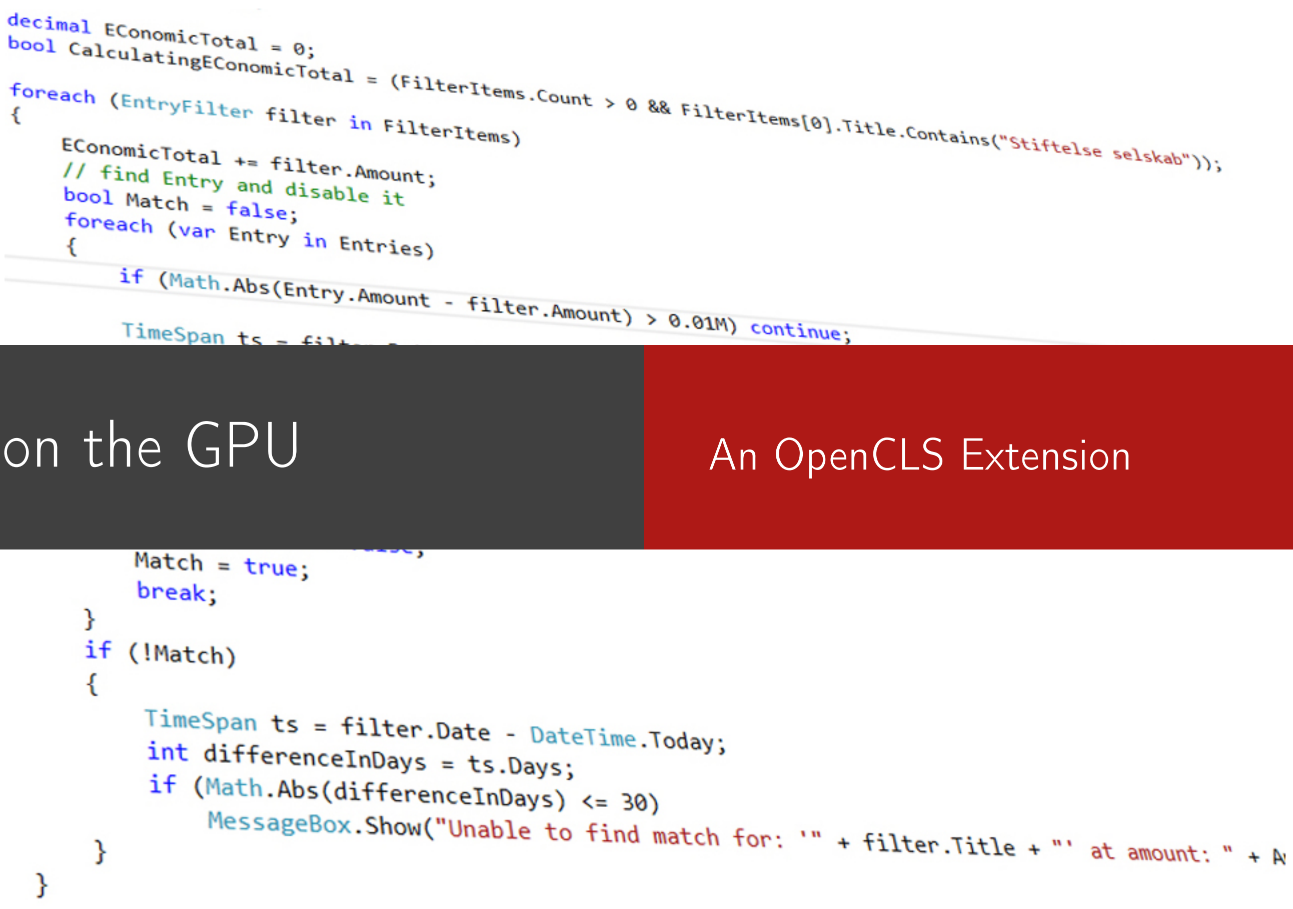


```
decimal EConomicTotal = 0;
bool CalculatingEConomicTotal = (FilterItems.Count > 0 && FilterItems[0].Title.Contains("Stiftelse selskab"));

foreach (EntryFilter filter in FilterItems)
{
    EConomicTotal += filter.Amount;
    // find Entry and disable it
    bool Match = false;
    foreach (var Entry in Entries)
    {
        if (Math.Abs(Entry.Amount - filter.Amount) > 0.01M) continue;

        TimeSpan ts = filt

        Match = true;
        break;
    }
    if (!Match)
    {
        TimeSpan ts = filter.Date - DateTime.Today;
        int differenceInDays = ts.Days;
        if (Math.Abs(differenceInDays) <= 30)
            MessageBox.Show("Unable to find match for: '" + filter.Title + "' at amount: " + A
    }
}
```

OOP on the GPU

An OpenCLS Extension

13. June 2016

DPT1011F16

10th Semester Project

AALBORG UNIVERSITET
Student Report

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://cs.aau.dk

Title: OOP on the GPU, an OpenCLS Extension

Theme: Programming Technology

Project period:
10th Semester, Spring 2016

Project group:
DPT1011F16

Participants:
Mads Holm
Minh Hieu Nguyen
Stefan Holst Christiansen
Jonas Kaerlev

Supervisor:
Lone Leth Thomsen

Page count: 102

Appendix count: 85

Finished on Jun 13th 2016

Synopsis:

This report details the development of OpenCLS 2, an extension of the OpenCLS 1 project. The OpenCLS 2 project aims to reduce split codebases for GPU development, by allowing CPU code (C#) to be compiled to GPU code (OpenCL C) and executed on the GPU. The focus of OpenCLS 2 is to extend the set of features that the OpenCLS projects support, by implementing support for object-oriented programming (OOP) through the three OOP pillars: inheritance, encapsulation and polymorphism. This is achieved through the *Roslyn* platform and the OpenCL wrapper *CLOO*. The Roslyn platform is utilized to build a syntax tree for C# code, which OpenCLS 2 iterates through to generate OpenCL C code. The OpenCL C code is then executed on the GPU using CLOO. The implementations of inheritance, encapsulation and polymorphism are tested through unit tests, design pattern tests, all of which OpenCLS 2 passes. OpenCLS 2 achieves similar performance to related work, and better performance than the CPU by a factor of 3.5 for large computation and without host-device communication.

Signatures:

Mads Holm

Minh Hieu Nguyen

Stefan Holst Christiansen

Jonas Kærlev

Preface

This report is a 10^{th} semester master project report, written by four software engineering students at Aalborg University, with focus on programming technologies. This project is an extension of our previous project "*CPU to GPU Compiler*" [1].

This project is concluded with OpenCLS 2, an extension to the OpenCLS 1 project. The source of this extension can be found in the attached zip file. In addition, the bibliography can be found on the last pages of this report before appendix.

We would also like to thank our project supervisor for helping create a great project.

Table of Contents

Part I

Introduction

This part will describe the motivation behind this project, learning goals, development method and provide an overview of the report. This overview will contain the report structure as well as the report terminology.

1 Motivation

Modern GPUs provide significant performance increase through parallelization. This allows resource-heavy algorithms and calculations to execute faster. However, GPUs are programmed through GPU languages, such as CUDA C and OpenCL C. These are C-like languages with some additional keywords for memory and execution management. Because of this, any application seeking to utilize GPU-acceleration will contain two different codebases: the codebase for CPU execution (C, C++, C#, F#, etc.), and the codebase for GPU execution (CUDA or OpenCL).

The OpenCLS 1 project [1] aimed to leverage this by allowing C# to be compiled to OpenCL code and executed on the GPU, by providing a compiler and an API, thereby removing split codebases. The API is used for executing selected code on the GPU using the CLOO library, and hides boilerplate code necessary for OpenCL. The compiler generates OpenCL C code from C# code marked with the OpenCLS attribute, and was built using the syntax and semantic analysis tools provided by the Roslyn platform, an open-source platform created by Microsoft [2].

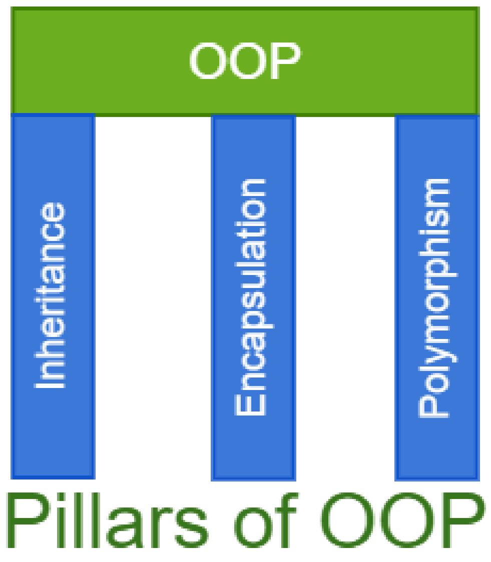


Figure 1.1: The three pillars of object-oriented programming (OOP).

The compiler supports the following features from C#: Conditional statements, functions/methods (including recursion), iterators, arithmetic expressions, member access, declarations, operators, return statements, strings and structures. In addition, the OpenCLS 1 project has implemented some elements of OOP, this includes: Definition, declaration, overloading and instantiation of objects. While the OpenCLS 1 compiler provides support for encapsulation, it provides no support for the other two OOP pillars [3], inheritance and polymorphism. [1]

OpenCLS 1 achieved performance comparable to Cudafy, a C# to CUDA and OpenCL compiler [4]. However, neither OpenCLS 1 or Cudafy achieve the same level of performance as C++ using the official OpenCL API, as can be seen in figure 1.2.

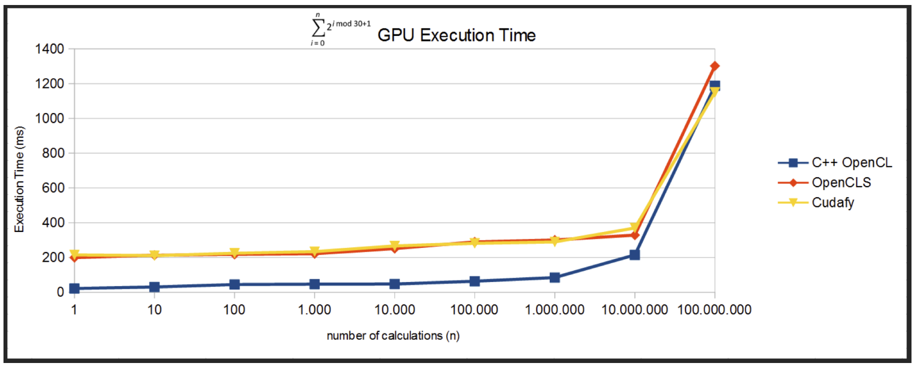


Figure 1.2: OpenCLS 1 execution time in milliseconds on the GPU compared with Cudafy [4] and C++ OpenCL.

Even though OpenCLS 1 performs worse than C++ OpenCL, it is acceptable as the performance loss is compensated with a reduction in split codebases. OpenCLS 1 did not support the three pillars of OOP and for this project we want to extend the support of OOP designs by working towards providing the remaining two OOP pillars; inheritance and polymorphism.

2 Learning Goals

The learning goal of this project is to document a problem statement and develop a solution within the field of programming technologies. This solution will be an extension of a previous semester project, and will utilize techniques and technologies relevant to the subject area. This will be achieved through the following objectives:

- Obtain an understanding of GPU development tools and examine their features and application.
- Utilize GPU terms, techniques and implementations through the development of a solution to a GPU problem statement.
- Observe advantages and disadvantages of related work.
- Measure performance advantage and validate correct execution of the solution.

These objectives will work as the foundation for the learning goal, and will be utilized to develop the solution for this project.

3 Development Method

It is impossible to predict all potential pitfalls that may appear during development of this project, as there are only few CPU-to-GPU compiler projects that this project can learn from, and these are not all fully documented. This is backed up by the fact that the initial development of OpenCLS 1 ran into limitations that were undocumented (such as the 3rd party OpenCL wrapper, *CLOO*) [1]. In order for this project to properly handle and adjust to such unknown limitations, the agile software methodology will be utilized for development [5].

The agile manifesto has goals similar to the *CDA paradigm* (Consider, Do, Adjust) by *Saras D. Sarasvathy*, where action is prioritized, and development adjusted based on what is currently possible. This is opposed to traditional waterfall methodology, where a strategy is fully conceptualized and then executed, with no consideration for discoveries, possibilities or roadblocks that may arise during development. [6]

The project will be utilizing the agile software methodology with some adjustments. The agile software methodology has a focus on customer and stakeholder collaboration, however this has been excluded as the project has no customer and no stakeholders.

It is possible to have one of the team members appointed the role of a customer proxy or a product manager who will represent the customer in the different project stages, but this will not be done as an effective customer proxy needs to be skilled in business analysis [7]. Additionally, agile software development "*favors working software over comprehensive documentation*" [5], however this project will instead adopt an adjusted version of this principle that includes and favors the academic report.

The project will also be adopting principles from the agile frameworks SCRUM [8] and XP [9] (Extreme Programming).

Feature	Pros	Cons
Pair Programming	Code is reviewed as it is written, resulting in higher quality solutions. Less refactoring required.	Code takes longer to write.
Sprints and Weekly Planned Efforts	Usable build every sprint.	Favors smaller tasks that can fit within the sprint period.
Sprint Backlog	Overview of tasks and their progress	Requires time dedicated to setup and maintenance. Requires advanced knowledge of tasks and problems.

Table 3.1: Pros and cons of SCRUM and XP features.

This choice has been made based on past experiences with several agile frameworks, where specific principles from SCRUM and XP have worked well. The project will be utilizing pair programming (from XP), sprints (from SCRUM) and sprint backlogs (also SCRUM). Every sprint (weekly planned efforts) include a supervisor meeting (whenever possible) where the state of the project is reviewed and further development is discussed. The pros and cons considered for these features can be seen in table 3.1.

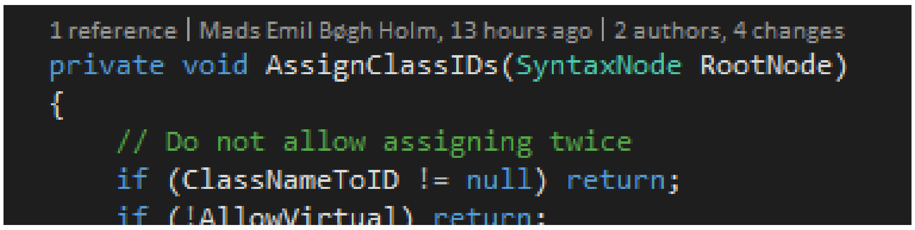
3.1 Project Tools

OpenCLS 2 will be developed using *Microsoft Visual Studio 2015*, as the OpenCLS 1 compiler, and the unit test of the compiler features, were implemented in Visual Studio 2015 [1]. The OpenCLS 1 project was developed using Visual Studio, and the project structure and solutions reflect this. This makes it attractive to develop the extension using the same environment, so that the structure can be resumed. The already made unit tests can also be used to test whether the new features in OpenCLS 2 affect the features of OpenCLS 1 and the set of unit test can be extended to test new OpenCLS 2 functionality (unit tests will be explained further in part V). Additionally it provides the Roslyn platform with syntax and semantic analysis tools and a syntax visualizer tool to inspect the syntax tree from the code that is being analyzed [10].

For OpenCLS 1, we utilized *Apache SubVersion* (*SVN*) to maintain the codebase between multiple machines. In addition to maintaining codebases, SVN also has the benefit of being able to resolve merge conflicts, which occur when changes in the same code segment are made. SVN repositories were provided by Aalborg University.

For this project, we utilize *Team Foundation Server* (TFS) which is an enterprise-grade server for teams created by Microsoft [11]. TFS provides a set of tools for project development that are essential to any software development team, such as code repositories and continuous integration. TFS provides the same range of subversion features that are available in SVN, as well as tools for agile teams, automated builds, automated testing and can be used in any language and code editor (but Visual Studio is recommended). As a result TFS is for us an upgrade from SVN. Automated builds allow us to quickly identify when a *check-in* (code submission) breaks existing implementations and prevents the project from compiling. Automated testing allows us to identify check-ins that break existing features, by giving a detailed report of which tests failed for a set of check-ins. Microsoft offers free Team Foundation Server hosting for teams with less than five team members, a requirement which our team fulfills. Microsoft Visual Studio 2015 has integrated TFS support, which allows us to review check-ins and code authors within Visual Studio (see figure 3.1). This makes it significantly easier to review changes to specific code segments.

TFS also supports being connected to other services through hooks. For this project, we utilize a PHP hook to provide additional details on builds and tests. The hook emits PHP POST with information on the build, including whether it failed or passed, and which tests were run. In our PHP implementation, we store all check-ins and builds / tests emitted by



```
1 reference | Mads Emil Bøgh Holm, 13 hours ago | 2 authors, 4 changes
private void AssignClassIDs(SyntaxNode RootNode)
{
    // Do not allow assigning twice
    if (ClassNameToID != null) return;
    if (!AllowVirtual) return;
```

Figure 3.1: Visual Studio integrated TFS.

TFS. When the PHP script receives info on a failed build, it iterates over all the check-ins since the last successful build, and generates a list of potential culprit check-ins. This list is then emailed to all project participants. The PHP code can be seen in appendix B.

4 Overview of the Report

This section will present the structure of the report, as well as explain the terminology that will be used throughout the report.

4.1 Report Structure

This report is constructed from six part:

Introduction is this current part, where we describe the motivation behind this project, the learning goals for this project, the project's development method and an overview of the report.

Analysis describes as well as analyzes the problem of this project. This part consists of five chapters: the related work for this project, an explanation of the OOP paradigm, the problem statement, the requirement for this project and a description of three OOP design patterns.

Design is the part where we design all the features that is implemented in part IV (Implementation). This part consists of four chapters: OpenCLS 2 Prerequisite (explains all the important features from the previous project), Inheritance, Encapsulation and Polymorphism. The last three chapters each describe the pillars in OOP that were explained in part II (Analysis).

Implementation is the part where we explain which features from part III (Design) we select, and how we implemented them. This part consists of four chapters: The refactoring chapter where we explain all the changes and fixes we have created for the OpenCLS 2 compiler. Additionally three chapters for each OOP pillar, as in part III (Design).

Test consists of three chapters: Unit test, design pattern test and performance test. These three chapters describes all the test that we have conducted for OpenCLS 2. In these chapters we explain how and what we test, as well as the results.

Evaluation is the part where we conclude our project, this part consists of three chapters: Reflection, Conclusion and Future Work. In the reflection chapter we reflect on our development method as well as the current status of the solution. In the conclusion we will summarize what we have done in this project, as well as describe the final result of this project. In the Future Work chapter we describe what features and subjects can be explored if we were to continue working on this project.

4.2 Terminology

OpenCLS 1: Refers to the previous OpenCLS implementation.

OpenCLS 2: Refers to the OpenCLS implementation for this project.

Users: The end-user of the OpenCLS 2 API and compiler.

Programmers: People in general who code. Not specific to OpenCLS 2.

Definition/Declaration: this project is developed in C#, and C# has no clear distinction between declaration and definition. As a result, we will for this project use the following terms when referring to any programming languages

> **Declaration:** when referring to a class or method that has a fully defined body.
>
> **Forward declaration:** when referring to a class or method declaration that does not have a complete body.

Check-ins: Code submissions in version control software.

Resolver method: A switch-case method that decide which version of an overridden method is used.

5 Summary

In this part we examine our previous semester project, the OpenCLS 1 project, as well as the programming paradigm OOP. We described the learning goals for this project and what is expected for this semester. We also explained our development method for this project as well as what kind of methodology and principles we are using.

And finally we gave an overview of what parts will be coming up and we also listed the terminology for this report.

Part II

Analysis

The following part will present an analysis of the OOP programming paradigm in order to get a problem statement that can be explored further in the next parts. This part will also examine related work to understand advancements and techniques made by others on how to implement the OOP paradigm.

6

Object-Oriented Programming (OOP) in C#

In this chapter we will define the set of features in C#, that the three OOP pillars (inheritance, encapsulation and polymorphism) contain. The OOP pillars and their features have been visualized in figure 6.1. This will also provide a foundation for the problem statement and requirement specification.

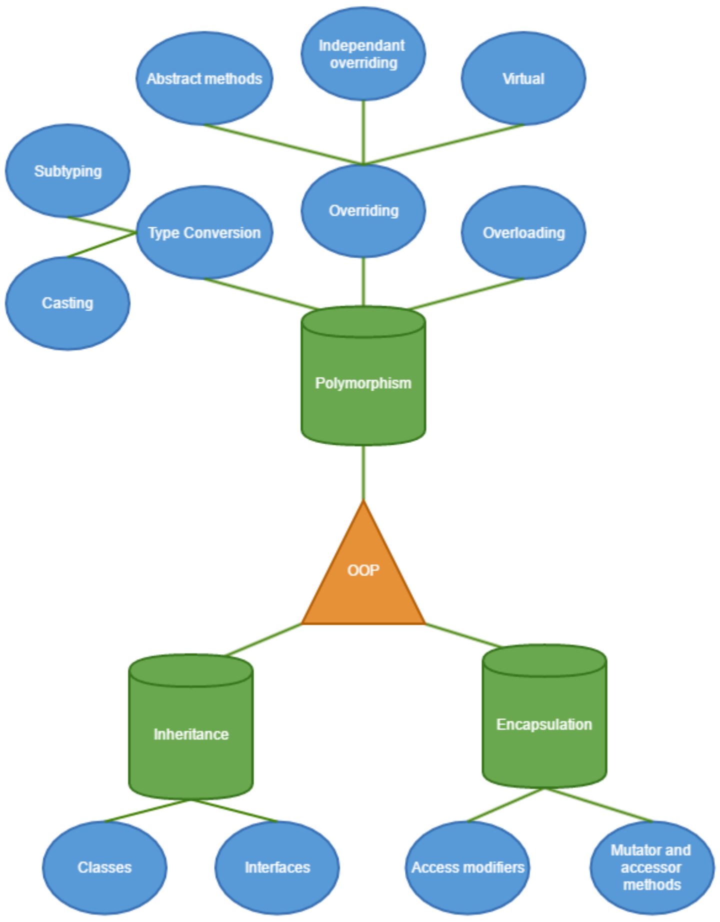


Figure 6.1: The three pillars of object-oriented programming. [12]

6.1 Inheritance

"Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics (or pillars) of object-oriented programming." [13]

With inheritance, a *class* can use the behaviours from other classes, which is useful for when multiple classes share behavior. As an example, we have a *Dog* and *Cat* class, they share behaviours such as weight, age and gender. For this we do not want redundant code, therefore we create a new class *Animal* that has those behaviours, and let the *Dog* and *Cat* classes inherit from the *Animal* class. In this case *Dog* and *Cat* are the *sub* classes and *Animal* is the *super* class. In listing 6.1 and figure 6.2, we demonstrate how inheritance works.

```
class Animal
{
    int Age { get; set; }
    float Weight { get; set; }
    bool Gender { get; set; }
}

class Dog : Animal {}
class Cat : Animal {}

class Pug : Dog {}
```

Listing 6.1: A simple C# example of inheritance with the animals

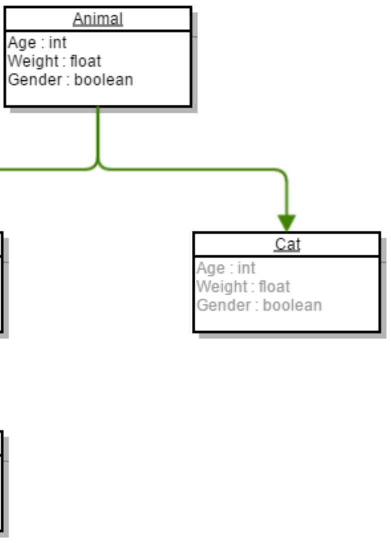


Figure 6.2: UML diagram of the animal classes

Inheritance is one of the ground pillars of OOP, and it allows classes to share behavior. C# does not have multiple class inheritance, but inheritance in C# is transitive, meaning that if a *Pug* class inherits from *Dog*, it will still have the same behaviour as *Animal*. [13] Classes in C# can inherit from the following:

- Classes

- Interfaces
- Abstract Classes

OpenCLS 1 supports classes, but not interfaces and abstract classes. These are encapsulation features and therefore we will examine how interfaces and abstract classes function and how they are used, in the next section.

6.2 Encapsulation

Encapsulation is the second pillar of OOP and the definition of encapsulation is *"the process of enclosing one or more items within a physical or logical package"* [14]. More specifically for OOP languages, it is done through the use of classes to define objects and it is also about limiting the access to implementation details. This is known as information hiding, which is advantageous by allowing programmers to only showcase implementation that enhances the abstraction of the implementation.

In some languages, such as Java or C# this is done by using access specifiers, commonly through keywords such as *Public* or *Private*, and applying them to either classes or methods. In C#, there are five different access specifiers: [14]

- Public - all public members of the class can be accessed by other classes.
- Private - only members of the same class can access private members.
- Protected - only a subclass and the class itself can access protected members of a class.
- Internal - only members declared in the same assembly as the internal members can access them.
- Protected Internal - a mix of the previous two, only a subclass and the class itself defined in the same assembly can access protected internal members of a class.

Access specifiers are enforced during the compiler's code analysis process. They are not compiled nor utilized during program execution, which means that OpenCLS 2 does not need to translate access specifiers.

C# also supports properties (setters and getters), which are method declarations that are accessed as variables. Setter properties can only accept a single parameter (a new value) and can not have a return value. Getter properties can only have a return value, and no parameters. Properties are used alongside private or protected variables, to give limited access to these variables through the property [15].

As mentioned in the previous section , a class can also inherit from interfaces and abstract classes. An interface is a class declaration without implementation, and all classes that inherit from an interface are required to implement all members declared in the interface. Effectively all members declared in an interface are implicitly abstract. Interfaces support multiple inheritance, and as a result a class can inherit from multiple interfaces.

An abstract class is a class that cannot be instantiated. Despite the name, abstract classes allow both abstract and non-abstract methods [16]. Declaring an abstract class with only abstract members is similar to declaring an interface, with the only exception that interfaces support multiple inheritance, while abstract classes do not.

Encapsulation is mainly a help to programmers, either through enclosing data into a class or hiding information to enhance abstraction. OpenCLS 1 already supports access specifiers, since these are only utilized during compile-time and not during program execution. Support for properties was added in OpenCLS 1, with both setters and getters being supported. Properties are compiled into methods, and all calls to the properties are redirected to the methods.

OpenCLS 1 meets the requirements for the encapsulation definition: enclose data (through class declarations) and information hiding (through access specifiers). However the complete set of encapsulation features in C# were not fully supported in OpenCLS 1. The encapsulation features of C# that are not supported by OpenCLS 1 are:

- Interfaces
- Abstract classes

Additionally, once inheritance and polymorphism is implemented, the existing supported encapsulation features (properties, overloading, access specifiers and class definitions) will need to be expanded to implement support for class hierarchies.

6.3 Polymorphism

Polymorphism is the third pillar of OOP. For polymorphism, this project will be using the definition provided by Microsoft in the official C# programming guide [17]:

- Polymorphism allows objects to behave differently at runtime depending on context (overloading and overriding).
- Polymorphism allows subclass objects to be treated as superclass objects in variables, parameters, etc. (Type conversion, more specifically subtyping).

This definition does not include C# features such as generics, generic methods, LINQ or exceptions, as these are not required for C# to be an object-oriented programming language [18].

For method overriding, a subclass declaration supersedes the functionality of a method declared in a superclass. This can be done using *virtual methods* which gives subclasses the option to override, or using *abstract methods* which makes overriding mandatory. Languages like C++ and C# implement method overriding using a *vtable* (virtual function table), which contains a list of pointers for methods that an object supports [19]. *Independent overriding* allows a method to be overridden only for the target class and its subclasses, while the superclass implementation remains accessible to any calls made on the superclass.

In method overloading, a method can be declared multiple times with different parameter types. This allows a method to change the execution depending on input types. In C++ and C#, the compiler considers the parameter types as part of the method name. During runtime, the parameter types are considered and the method with the name matching the parameters is invoked. [19]

Type conversion allows variables to change type either with type casting or subtyping. Type casting is converting a type to another type, either through implicit casting or explicit casting. Subtyping allows variables and methods to accept a subclass of their target type. For instance, a variable storing objects of class A, will also accept objects of class B, if B is a subclass of A. The polymorphism OOP pillar contains the following programming features [17, 20]:

- Type Conversion
    - Subtyping

- Overriding
  - Virtual Methods
  - Abstract Methods
  - Independent Overriding
- Overloading

OpenCLS 1 supports overloading, so in order for OpenCLS 2 to support the polymorphism pillar, we need to implement subtyping and at least one override feature.

6.4 Summary

Analysing the three pillars of OOP has provided an understanding of a set of features. These include class- and interface inheritance, the difference between method overriding and method overloading, and subtyping.

- Inheritance makes the code more reusable as multiple subclasses can inherit from a single superclass and the code size will therefore also be reduced.
- Encapsulation allows developers to enclose behavior in classes, and hide functionality to improve abstraction. This includes access specifiers, properties, interfaces and abstract classes.
- Method overriding and method overloading allows the developer to change the behavior of methods at runtime based on context without making significant changes to the implementation structure.
- Subtyping allows subclass objects to be accepted in variables and as parameters that accept the superclass.

Polymorphism can only be implemented when inheritance is present, and as a result the inheritance OOP pillar will need to be implemented before the polymorphism OOP pillar.

Supporting these features allows the developer to implement different design patterns for object-oriented software. In chapter 10, we will examine various design patterns for OOP.

7 Related Work

In order to get an understanding of how OOP designs can be implemented in OpenCL C and on the GPU, it is advantageous to examine existing projects or solutions that accomplish similar goals. This chapter will present a set of related papers and articles, and highlight the details of the individual implementations.

7.1 Firepile

Firepile is a library for GPU programming in Scala. [21] The library supports OOP design patterns (inheritance, polymorphism and encapsulation) on the GPU. As can be seen in figure 7.1, Firepile implements classes and objects using structs, and assigns each struct a unique ID. Virtual method calls are performed using a dispatcher method, which calls a different method depending on the incoming classID. The performance for the output code that Firepile generates is comparable to native C code (within 15% deviation) [22]. The designs presented in Firepile could also be implemented in other languages, as the designs are not specific to Scala.

7.2 Inheritance and Polymorphism in C

Inheritance and Polymorphism in C is an article published by *Prashant Gotarne* on *CodeProject* [23]. The article details how C++ classes can be implemented as C structs that support

```
class A(val x: Int) { ... }
class B extends A(val y: Int) { ... }
```

(a) Scala class definitions.

```
struct Object {
 int __id;
};

struct A {
 struct Object _Object;
 int x;
};

struct B {
 struct A _A;
 int y;
};
```

(b) Generated OpenCL C code.

Figure 7.1: Firepile class compile from Scala to OpenCL C.

inheritance and polymorphism. Inheritance is implemented by allocating the superclass as a member variable. This makes all variables in the superclass accessible to subclasses. Polymorphism is implemented using function pointers. As C does not support classes and class methods, all class methods are implemented as independent functions that take an instance of the class as parameter. The class is given a function pointer as member variable that points to the independent function. When a method on the class is overridden, this pointer is set to point to the new overridden method. The C implementation presented in this article has significantly more code that needs to be written and maintained (compared to the C++ counterpart). This would make the C implementation a poor substitute for C++, if it had to be written manually. However as our project is developing a compiler that auto-generates code, this becomes a non-issue, as C code maintenance is not a concern (all code gets updated through auto-generation on compile).

7.3 Applying Object Oriented Design Patterns to CUDA

Applying Object Oriented Design Patterns to CUDA based Pyramidal Image Blending - An Experience is a paper that introduces an object oriented framework for image processing of CUDA [24]. The paper details how to illustrate programming advantages of an OOP language by utilizing design patterns. Design patterns are a description or template that can be applied to solve a problem in different situations. The patterns used for the CUDA based pyramidal image blending implementation are *Abstract Factory*, *Factory Method*, *Adaptor* and *Template Method* pattern. These design patterns presents programming advantages such as encapsulation, inheritance, information hiding, code reuseability, complexity hiding and complexity extending from an object oriented language. As our project is extending the OpenCLS 1 compiler with OOP features, implementing design patterns can be an option to test the programming advantages of the OOP extension in OpenCLS 2. [24]

7.4 Cudafy

Cudafy is a C# to OpenCL C and CUDA compiler, and API [4]. The project uses the Cecil .NET binary analysis tool [25] to read bytecode from compiled C# blocks, and compiles the bytecode into OpenCL C and CUDA. The API then executes the compiled GPU code using the CLOO [26] library. The Cudafy project was also examined in our previous semester project [1, p. 26]. Cudafy supports all primitive types (int, byte, double, etc.), arrays and structs, however it does not support classes. OpenCLS 1 supports the subset of types that Cudafy supports, along with classes. Additionally OpenCLS 1 achieves similar performance to Cudafy (figure 1.2 on page 10).

7.5 Aparapi

Aparapi is an API for Java that allows execution of Java code on the GPU [27]. During runtime, the API reads select Java bytecode and compiles it to OpenCL C, which is then executed on the GPU. This procedure is in concept identical to Cudafy. Aparapi supports primitive types, arrays and structs, but does not support classes. Aparapi achieves better performance than sequential and threaded Java implementations for large data sets, as seen in figure 7.2.

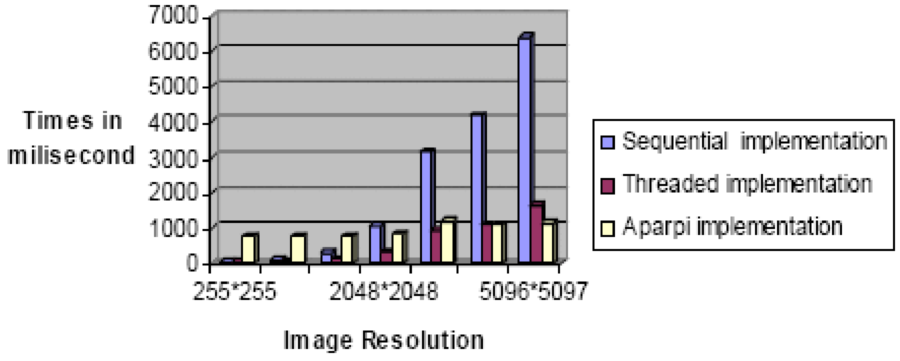


Figure 7.2: Aparapi execution time in milliseconds on the GPU for image manipulation compared with a sequential Java implementation and a threaded Java implementation. Aparapi has an initial startup period due to being required to initialize OpenCL. [28].

7.6 FSCL

FSCL is an F# to OpenCL compiler and runtime execution [29]. The compiler reads F# source code and outputs OpenCL source code while the runtime executes the OpenCL C source code. This process is similar to OpenCLS 1' source-to-source compilation and runtime execution. The FSCL compiler supports primitive types and structs, but does not support classes. FSCL achieves slightly worse performance compared to C++ OpenCL, which is similar to how OpenCLS 1 also achieves slightly worse performance compared with C++ OpenCL.

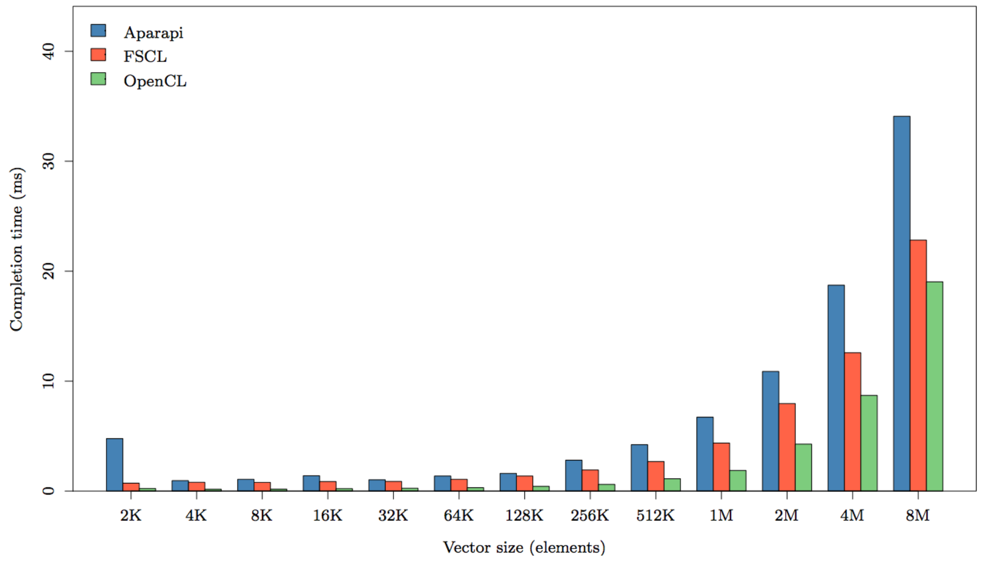


Figure 7.3: FSCL execution time in milliseconds on the GPU for vector addition compared with Aparapi and C++ OpenCL. This performance test is one of many tests available in the FSCL report.

7.7 Summary

The papers *Firepile* and *Inheritance and Polymorphism in C* both present methods of achieving inheritance and polymorphism that could be applied to OpenCL C. These papers will work as inspiration for the creation of our own OOP implementation designs. The paper *Applying Object Oriented Design Patterns to CUDA* points out that programming design patterns can be used to test OOP features in a programming language. We will utilize this, and implement programming design patterns as test cases.

The documentation on Cudafy, Aparapi and FSCL provides us with related work that we can compare our solution to. This will be useful when comparing performance of OpenCLS 2.

8 Problem Statement

This project will continue development of the OpenCLS 1 project: A C# to OpenCL C compiler that is capable of compiling and executing C# code on the GPU. The OpenCLS 1 project enables C# code to take advantage of the performance benefits of GPUs.

This is motivated by the advances made in CPU-to-GPU compilation made by OpenCLS 1 (such as classes and objects on the GPU) [1], as well as the lack of OOP features in OpenCL. As long as OOP is inaccessible on the GPU, OOP design patterns cannot be utilized on the GPU. This results in a split codebase, as project structures utilizing common design patterns will need to be rewritten or restructured to meet the limitations of the GPU.

This project presents a solution by extending OpenCLS 1 to support the three OOP pillars: inheritance, encapsulation and polymorphism. This solution (named OpenCLS 2) will allow OOP implementations and design patterns to be utilized on the GPU.
In summary, our goal for this project is to answer these questions:

- How can we extend OpenCLS 1 to support inheritance?
- How can we extend OpenCLS 1 to support encapsulation?
- How can we extend OpenCLS 1 to support polymorphism?
- How can OOP features and the three pillars of OOP be prioritized?
- How can unit tests and design patterns be used to test the OOP features in OpenCLS 2?
- How can we test the performance of OpenCLS 2 with support for OOP?
- How do design patterns execute on the GPU, and how is the performance?

9 Requirements Specification

The project requirements presented in this chapter are based on the definitions outlined in chapter 6. As described in chapter 3:

"it is impossible to predict all potential pitfalls that may appear during development of this project, as there are only few CPU-to-GPU compiler projects that this project can learn from, and these are not all fully documented." [30]

As a result, this project is prepared to pivot the development progress if the development is halted due to a potential pitfall. By pivoting the development progress, the project will conclude with a working OOP implementation which will fulfill the requirements set for this project.

The priorities will be developed sequentially, by starting with the first priority requirements. The items listed in this chapter are based on the items discussed in chapter 6.

9.1 First Priority

The first priority is to implement inheritance and encapsulation, as these two pillars of OOP do not depend on other pillars before they can be implemented. This is unlike polymorphism, which requires inheritance to be available before it can be implemented. *Interface* and *abstract* classes are part of encapsulation. Additionally, the encapsulation features from OpenCLS 1 will need to be extended to implement support for class hierarchies.

- Inheritance
  - Class
  - Interface
  - Abstract class
- Encapsulation
  - Interface
  - Abstract class

9.2 Second Priority

The second priority includes the polymorphism OOP pillar. For this priority, only one of the three possible overriding implementations will be realized, as they are equally suitable as means to achieve polymorphism. There is no indication that any of the three overriding implementations will be more or less challenging, and as a result the choice of overriding implementation for this priority is arbitrary.

- Subtyping (Type conversion)
- Overriding, by implementing *one* of the following:
  - Virtual Methods
  - Abstract Methods
  - Independent Overriding

9.3 Third Priority

The third and final priority implements all features from polymorphism, and results in OpenCLS 2 fully supporting all three OOP pillars. Additionally, casting has been added to the list of features, as it would enable objects to be stored in variables of a different type through explicit conversions. With the addition of casting, OpenCLS supports both implicit (subtyping) and explicit (type casting) object type conversions.

- Overriding, by implementing *all* of the following:
  - Virtual Methods
  - Abstract Methods
  - Independent Overriding
- Casting (Type conversion)

All priorities will be designed in part III and implemented in part IV.

10 OOP Design Patterns

In this chapter, we will present a selection of OOP design patterns, that each highlight a subset of features presented in chapter 9 (Requirements Specification). This is inspired by the related work *Aplying Object Oriented Design Patterns to CUDA* [24] which implements design patterns to test that OOP features execute as expected. The design patterns presented in this chapter have been selected as they require the first, second or third priority of this project to be implemented before the design patterns can be implemented and executed. These design patterns are not exclusive to C#, and can be implemented in all languages that support the features required by the patterns. Collectively, these design patterns utilize the OOP features from chapter 9, and therefore implementing these design patterns will test that OpenCLS 2 is able to utilize the OOP pillars.

This set of design patterns will be included as part of testing, in part V. The solutions presented in this chapter are all OOP design patterns; these have been chosen because they could not be implemented without OOP.

10.1 Proxy Pattern

The proxy pattern allows for a class to be split into two representations: the full representation and a lightweight proxy representation. This is done using an interface, which ensures that the lightweight proxy and full representation provide the same shared functionality.

This design is utilized in three different ways [31, p. 207]:

Remote proxies When an object needs to be sent to a remote address space, and throughput is reduced by utilizing the lightweight implementation that only holds the necessary information.

Virtual proxies When an object needs to be represented without being initialized. The lightweight implementation can then invoke the full class at any point.

Protection proxies When the full class needs to be abstracted to avoid incorrect usage.

```
interface Image
{
    void Display();
}

class RealImage
```

```
{
    public void Display()
    {
        // Display the image
    }
}
class ProxyImage
{
    RealImage realImage;
    public void Display()
    {
        if (realImage == null)
        {
            // Image is not displayed yet, fetching now
            realImage = new RealImage();
        }
        realImage.Display();
    }
}

void main()
{
    Image image = new ProxyImage();
    image.Display();
}
```

Listing 10.1: Implementation of the virtual proxy pattern

Listing 10.1 shows an example of image rendering using a virtual proxy pattern. The *ProxyImage* and *RealImage* need to implement the *Display()* method, as required by the interface. An instance of *RealImage* is also stored as a member within the *ProxyImage*, allowing the real implementation to be invoked at any time, when needed.

In order to implement this design pattern, the OpenCLS 2 compiler needs to support inheritance as well as interfaces [32] and the project needs to meet the first priority in the requirements specification (chapter 9).

10.2 Visitor Pattern

The visitor design pattern is an OOP design pattern where an algorithm *visitor* can iterate over a set of objects referred to as *elements*. The visitor can perform different operations depending on the type of element, without the need to modify the element structure. This pattern is often used for compilers, where the syntax and semantic analysis iterates over a set of generated tokens [33].

Figure 10.1 shows an UML diagram of a visitor design pattern implementation. The element has an abstract accept function that takes the visitor as parameter. This function is abstract, to ensure all subclasses implement it. In the body of the accept method for any of the subclasses (such as *ConcreteElementA* and *ConcreteElementB*), the element calls the appropriate visit method on the visitor. For instance, *ConcreteElementA* calls *VisitConcreteElementA()*.

The *ObjectStructure* contains a list of all elements. The visitor pattern can then iterate over the list of elements, call visit on each element, and the element calls the appropriate *VisitConcreteElement* method on the visitor.

The visitor design pattern requires overriding from the polymorphism OOP pillar, in order for the *ConcreteElements* to override the *Accept()* function. To support overriding, the design

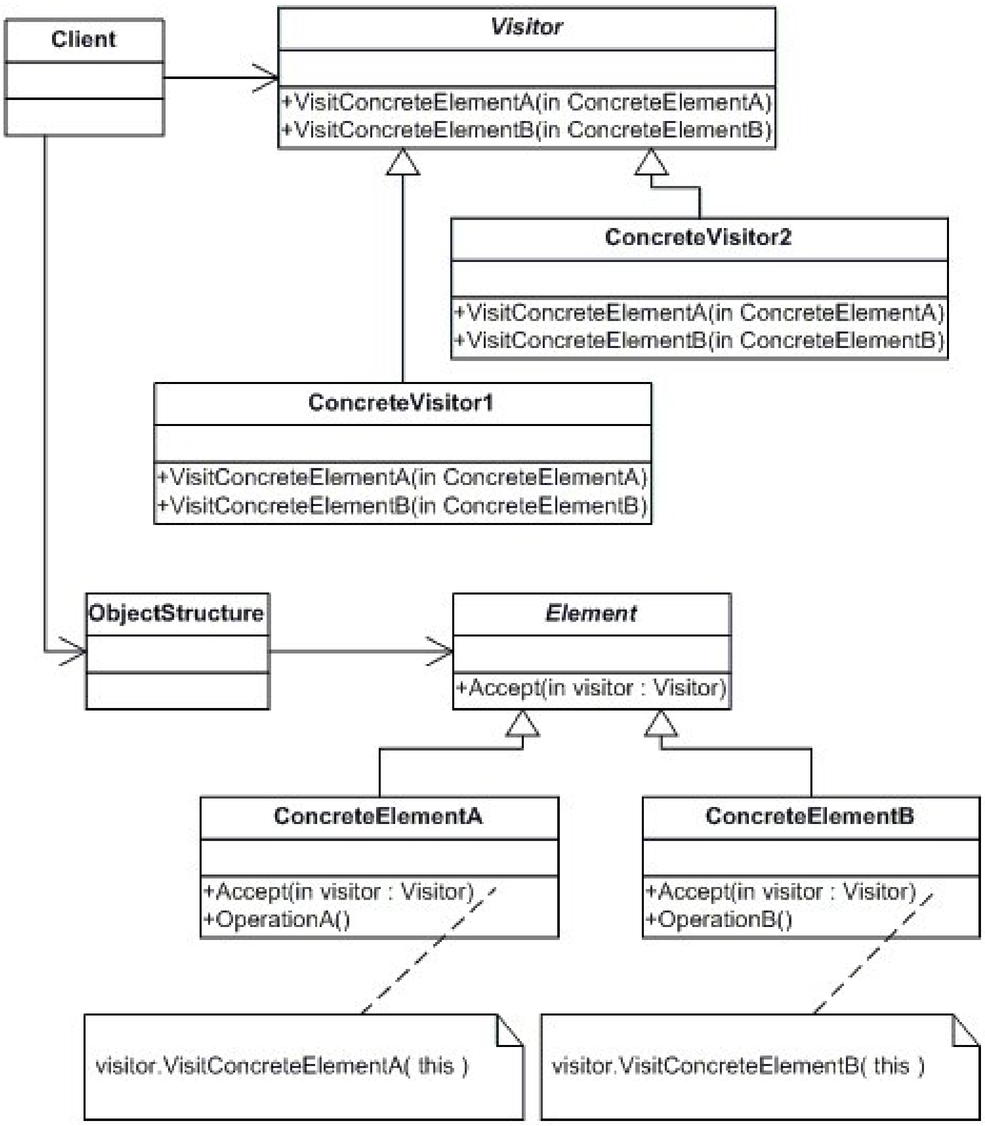

Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

Figure 10.1: UML for visitor design pattern. [34]

pattern also requires the inheritance OOP pillar. As a result, the visitor design pattern can only be implemented if the first and second priority have been fully implemented.

10.3 Observer Pattern

The observer pattern allows a one-to-many dependency between objects. These objects are split into two types of objects: *subject* (objects which are observed by other objects) and *observer* (objects which are observing or dependent on other objects). The relationship between these two key objects are that the subject can have any number of observers and the dependent observer objects are notified whenever the subject changes state. [31, p. 232]

The UML example of the observer pattern on figure 10.2 presents the following participants:

- **Subject** knows all its observers that have to be notified and the interface has methods for attaching and detaching observer objects.

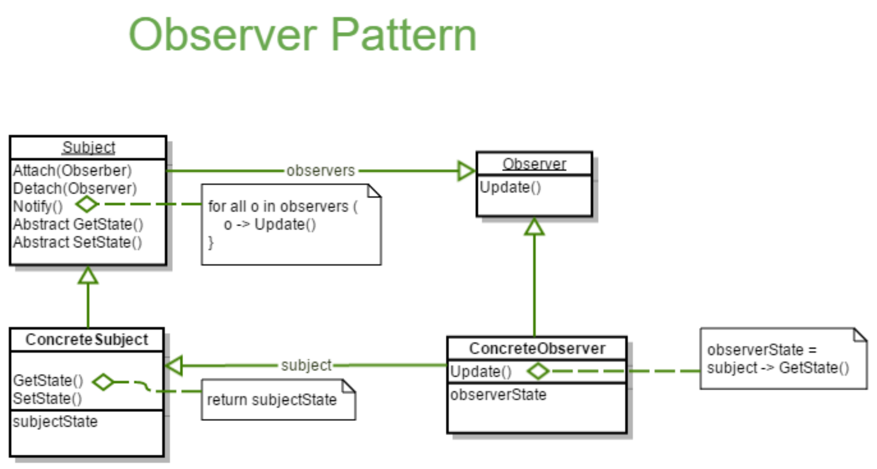


Figure 10.2: Example UML of the Observer pattern. Based on [31, p. 232]

- **Observer** contains an interface for the ConcreteObserver objects that needs to be notified of state changes in a subject.
- **ConcreteSubject** maintains the state of interest to *ConcreteObserver* objects with *GetState()* and *SetState()*. When the state changes, it sends a notification to the attached observers.
- **ConcreteObserver** contains a reference to a ConcreteSubject object and the implementation to update the interface while being state consistent with the state of the subject.

A benefit of the observer pattern is to keep the objects loosely coupled (clear distinction between objects) to reduce dependencies and maintain consistency between the related objects. Because the subject-observer relationship is loosely coupled, the pattern allows for objects to be reused. This means that the subject-observer relationship can be severed and the subject can be coupled with another observer, the observer can also be coupled with another subject. Most GUI applications, where the user interface is separated from the remainder of the application, have adopted a similar pattern to the observer pattern [35]. This gives the benefit of being able to modify and reuse either parts of the application separately without impacting the whole application. [31, p. 232]

Before we can implement the observer pattern presented in this section, subtyping, abstract methods, and virtual methods must be supported. This is because *subject* needs to hold a reference to all *observers* (subtyping), *ConcreteSubject* needs to override the abstract methods *GetState()* and *SetState()* (abstract methods), and *ConcreteObserver* needs to override the virtual method *Update()* (virtual overriding). Because of the need for two different overriding methods, priority three needs to be supported before this design pattern can be implemented.

10.4 Summary

In this chapter we examined three different OOP design patterns, we examined what OOP language features these different patterns required and how these patterns can be implemented in C#. Each design pattern requires different features from the OOP paradigm, and combined they utilize all features from the paradigm.

In summation:

- Proxy pattern requires inheritance and interfaces (Inheritance and encapsulation: priority one).
- Visitor pattern requires inheritance, overriding and subtyping (Inheritance and polymorphism: priority two).
- Observer pattern requires inheritance, subtyping, virtual methods and abstract methods (Inheritance and polymorphism: priority three).

11 Summary

In this part we have examined related work to our project and we discussed what features we could take inspiration from. We examined the OOP paradigm and describe each pillar (Inheritance, Encapsulation and Polymorphism), as well as which features are part of the individual pillars. Based on the knowledge gained from this we defined a problem statement that will influence the rest of the project and report. Based on this problem statement we created a requirement specification, which is built up of three priorities. Finally we described three OOP design patterns (Proxy, Visitor and Observer) that will help evaluate whether we have completed the requirements.

Part III

Design

In the following part we will present the design choices for the features that we will implement. Design choices for these features will be described using programming concepts and terminology. This part is a prerequisite for the implementation part, and each chapter in this part has an equivalent chapter in the implementation part, except for chapter 12 (OpenCLS 2 Prerequisite).

12 OpenCLS 2 Prerequisite

This chapter will introduce specific language and compiler features from the OpenCLS 1 project [1]. This chapter is a prerequisite for the feature designs of this part, as we will be using technical insight, terminology and details obtained from our previous OpenCLS 1 report.

12.1 Roslyn Compiler

The Roslyn compiler is the official open-source C# compiler, created by Microsoft [2]. The Roslyn compiler API can be installed by downloading the libraries from Visual Studio's Nuget Package Manager system. The following classes are used throughout this project as part of the design and implementation of the OpenCLS 1 compiler.

SyntaxTreeWalker is a class created to walk through an abstract syntax tree, it inherits this functionality from the abstract CSharpSyntaxWalker class provided by Roslyn.

SemanticModel is a class that contains the functionality to get any semantic information about the syntax tree. We use this class to get type information about syntax nodes that do not contain the information directly, such as an identifier's type.

12.2 Objects in OpenCLS 1

OpenCLS 1 implemented support for class structures, and in the following we provide a description of every class feature implemented, as well as provide a code example.

Class definitions are translated to a struct in OpenCL C, while any constructors or methods are moved out of the class definition. In C#, all objects are dynamically allocated and stored by reference, but OpenCLS 1 has no dynamic memory allocation, and as a result OpenCLS 1 stores all C# objects as value-based structs. A code example for class declarations in OpenCLS 1 is shown in listing 12.1.

```
// C#
class myClass {}

// OpenCL
struct myClass {}
```

Listing 12.1: OpenCLS 1 translation of class definitions.

Class declarations are translated to struct declarations in OpenCL C, and the constructor is called after the declaration with the struct as an argument. A code example for class declarations in OpenCLS 1 is shown in listing 12.2.

```
// C#
myClass obj = new myClass();

// OpenCL
myClass obj;
myClass_Constructor(&obj);
```

Listing 12.2: OpenCLS 1 translation of class declarations.

Class methods are moved outside of the definition, as mentioned previously. The method is modified to take an instance of the object as an extra parameter, and the method name is modified to contain the class name to ensure the name scope of the class. A code example for class methods in OpenCLS 1 is shown in listing 12.3.

```
// C#
class myClass
{
    public void myMethod() { }
}

// OpenCL
void myClass_myMethod(myClass* instance);
```

Listing 12.3: OpenCLS 1 translation of class methods.

Class constructors are moved out of the class definition, as with class methods. The constructor is changed to a method that takes an instance of the object as an extra parameter. As with class methods, the method name is also modified to contain the class name, to ensure the name scope of the class. A code example for class constructors in OpenCLS 1 is shown in listing 12.4.

```
// C#
class myClass
{
    public myClass() { }
}

// OpenCL
void myClass_Constructor(myClass* instance);
```

Listing 12.4: OpenCLS 1 translation of class constructors.

Member access are moved out of the class definition, as with class methods. Member access handles this modification by changing the method call to use the method that has been moved outside of the class definition. Member access to fields is translated directly, meaning that the syntax for member access to fields are similar in both languages. A code example for member access in OpenCLS 1 is shown in listing 12.5.

```
// C#
obj.myMethod();
obj.MyValue = 1;

// OpenCL
myClass_myMethod(&obj)
obj.MyValue = 1;
```

Listing 12.5: OpenCLS 1 translation of member access.

Properties (Setters and getters) are compiled to new methods that are placed outside the class definition. A code example for properties in OpenCLS 1 is shown in listing 12.6.

```
// C#
class myClass
{
    public int _MyValue;
    public int MyValue
    {
        get { return _MyValue; }
        set { _MyValue = value; }
    }
}

// OpenCL
int myClass_MyValue_Get(myClass* instance)
{
    return obj->_MyValue;
}
void myClass_MyValue_Set(myClass* instance, int value)
{
    obj->_MyValue = value;
}
struct myClass
{
    int _MyValue;
}
```

Listing 12.6: OpenCLS 1 translation of properties.

12.3 Summary

In this chapter we described the existing features of OpenCLS 1. We gave an insight into how OpenCLS 1 uses Microsoft's open-source compiler Roslyn, and we have described how objects are translated to OpenCL C through OpenCLS 1.

13 Inheritance

In this chapter we will design how to implement the features from the inheritance pillar (chapter 9). Since we will translate C# to OpenCL C, and these languages have a clearly defined syntax, our task is to translate the syntax from C# to OpenCL C. We take inspiration in what *FirePile* [21] and *Inheritance and Polymorphism in C* [23] have done to implement inheritance (as we examined in chapter 7), and we will take a similar approach to the implementation in our output OpenCL C code.

To create support for inheritance in OpenCLS 2, we will for every class declaration declare an instance of the superclass type as a member, called *super*. The class will then have access to all superclass members through the superclass member variable. Consider a scenario with two classes: *Dog* and *Animal*, *Dog* wants to inherit from *Animal*, to do this we will declare an instance of Animal inside Dog. In listing 13.1 we see the C# implementation for this example and in figure 13.1 we have a diagram of the class hierarchy in OpenCL C. The superclasses and subclasses must have the OpenCLS attribute in the C# implementation in order for the compiler to locate the classes and translate them to OpenCL C.

```
[OpenCLS]
class Animal { }

[OpenCLS]
class Dog : Animal { }
```

Listing 13.1: A C# implementation of a class hierachy in OpenCLS 2

13.1 Member Access

Member access will need to be modified to support inheritance through the superclass instance in class declarations. Any members declared in *Animal* will not be available in *Dog* by default. In section 12.2, we described how fields remain inside the class declaration, while methods and properties are moved out, with an instance of the class as an additional parameter.

To access fields in the superclass, all calls need to be done through the *super* variable. For instance, if *Animal* has a field *Age*, then *Dog* can access *Age* through *super.Age*.

When a class calls a superclass method, the method name is generated using the superclass name that declares the method. In figure 13.2a, if *Dog* has a method called *GetAge()*, this method will be called *Dog_GetAge()*. Then the call *Pug.GetAge()* will need to be changed to *Dog_GetAge()*. However, if *GetAge()* is declared in *Animal* (figure 13.2b), then the method

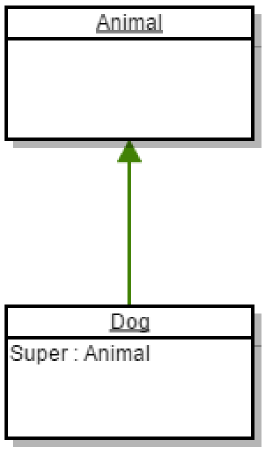
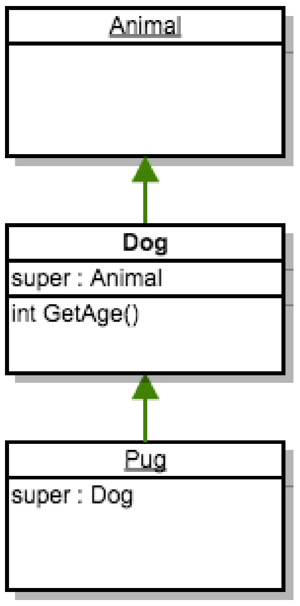
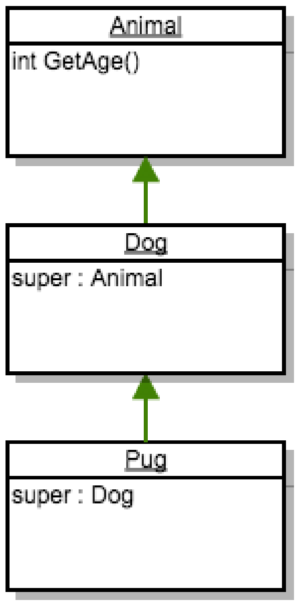


Figure 13.1: An UML diagram of the *Animal* and *Dog* hierachy in OpenCL



(a) Dog declares the *GetAge* function.



(b) Animal declares the *GetAge* function.

Figure 13.2: Two class hierarchies where the method declaration is in a superclass.

call is changed to *Animal_GetAge()*. To achieve this, we will iterate through the hierarchy that a class inherits from, in order to determine which class the method is declared in. Once the class has been found, we will use the class name to find the method name (while taking overloading and other prefixes into account). If the call is invoked by a subclass, we will iterate through the superclasses until we get an instance of the class that the method is declared in. This instance is then passed as a parameter.

Properties in OpenCLS 2 are output as methods, and as a result will be treated as methods for inheritance.

13.2 Forward Declaration

Forward declaration was introduced and implemented in OpenCLS 1 to allow class references before class declarations [1, p 48]. In C# class declarations can be written as in listing 13.1, but can also be declared in reverse order, where the subclass is declared before the superclass. This is not possible in OpenCL C, as the OpenCL C is an one-pass compiler and needs to know all information on a class once it reaches a reference to the class in code.

This means that the forward declaration order cannot be taken directly from the C# code. Instead, forward declarations will sort based on class hierarchies. In order to resolve this, we will check each class declaration if they have a superclass, and if so, we will forward declare the superclass first.

14 Encapsulation

In this section, we will design the parts of encapsulation that have to be implemented when inheritance is introduced. As described in chapter 9, these parts are: *Interfaces* and *abstract classes*.

14.1 Interface

In C#, interface members cannot have implementations in their declaration, and subclasses are forced to implement the members provided in the interface. Interfaces are unique, in that a class can inherit from multiple interfaces. Additionally, interfaces cannot be instantiated or contain fields. Because interfaces behave differently from classes in C#, we implement interfaces separately from classes.

In OpenCLS 2, we compile interfaces into a single struct (called *Interface*) which contains all interface declarations. Any classes that inherit from the interface will have an *interfaceSuper* variable (just like class-to-class inheritance has the *super* variable). OpenCLS 2 will then treat the interface like any class, including allowing casts and subtyping. However, interfaces can still not be instantiated, as that is an invalid operation in C#.

14.2 Abstract Class

An abstract class is a class where no objects of the class can be instantiated. In C# the *abstract* modifier is used to denote an abstract class. Abstract classes provide the same functionality as interfaces, with the following differences:

- Classes can only inherit from one abstract class.
- Abstract classes can declare both abstract and non-abstract methods and properties. Non-abstract methods and properties can contain a definition in an abstract class.
- Abstract classes can declare non-abstract fields.

Subclasses of an abstract class are required to override all abstract methods. In OpenCLS 2, abstract classes are compiled to non-abstract classes, in order to support subtyping.

15 Polymorphism

In this chapter we will describe the design of the different features of polymorphism described in section 6.3, in order to fulfill the requirements from chapter 9 and to allow two of the OOP design patterns to be implemented (the Visitor and Observer pattern) described in section 10.3.

15.1 Overriding

In C# overriding can be done using three different approaches. The three different approaches are as presented in the requirements specification (chapter 9):

- Virtual Methods
- Abstract Methods
- Independent Overriding

The following sections describe the design of the three approaches to overriding in C# and OpenCL C.

15.1.1 Virtual Methods

In order to support virtual methods in OpenCLS 2, first we need to understand how they work. In figure 15.1, an example of a class hierarchy is visualized. In C# a virtual method operates as non-virtual methods with the exception that the *virtual* keyword allows the method to be overridden by subclasses. When a virtual method is called, the overridden version of the method is used. For instance, if we look at figure 15.1, if *Dog* calls *MyMethod()*, the version in *Dog* is used and if *Husky* calls the same method, the version in *Dog* is also used as *Husky* does not override *MyMethod()* and it inherits from *Dog*.

However, if *Pug* calls *MyMethod()* the version in *Pug* is used, regardless of any subtyping that might take place. If a *Pug* is saved in an *Animal* variable, the *Pug* version of *MyMethod()* will still be used. In other words, which method to invoke is based on the object type that the variable references, and not on the type of the variable itself.

In order to accomplish this, a *classId* variable is introduced to each class in a class hierarchy. Figure 15.2 shows a flowchart of how the compiler will handle a virtual method. When *MyMethod()* gets invoked, we check whether it is overridden, and if so we invoke

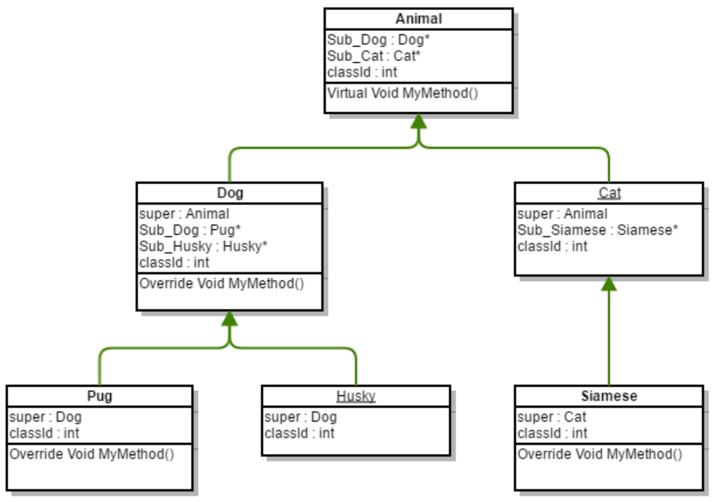


Figure 15.1: UML diagram showing a class hierarchy utilizing virtual methods.

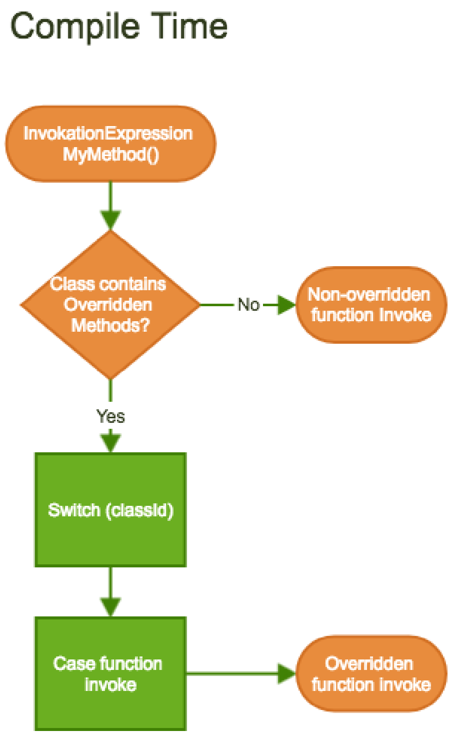

Compile Time
InvokationExpression MyMethod()
Class contains Overridden Methods?
No
Non-overridden function Invoke
Yes
Switch (classId)
Case function invoke
Overridden function invoke

Figure 15.2: Flowchart of how the compiler should handle virtual methods

any overridden methods by the class instance instead of the existing invocation. We use the *classId* to keep track of the class of each object at runtime.

A general *resolver* method is called that handles all implementations of the virtual method, and a switch case utilizing the *classId* to determine and invoke the appropriate implementation for the input object. Listing 15.1 shows this implementation in pseudo-code.

```
Virtual_Resolver_MyMethod(Animal : instance, classId)
{
    switch(classId)
    {
        case ClassId_Animal:
        case ClassId_Cat:
            invoke Animal_MyMethod(Animal : instance);
        case ClassId_Dog:
        case ClassId_Husky:
            invoke Dog_MyMethod(Dog : instance);
        case ClassId_Pug:
            invoke Pug_MyMethod(Pug : instance);
        case ClassId_Siamese:
            invoke Siamese_MyMethod(Siamese : instance);
    }
}
```

Listing 15.1: Pseudo-code for the switch case for virtual resolver

In OpenCLS 1 all class methods were implemented to pass an instance of the class as a parameter (as described in section 12.2). However, for overriding, this introduces a problem when calling a subclass' virtual method. This problem can be seen in listing 15.1 where the resolver method accepts an instance of the class that declares the virtual method, however all subclass overridden methods accept an instance of the subclass. As a result, we introduce references to the subclasses in the superclass, as can be seen in figure 15.1. These references are prefixed *Sub_* followed by the class name. This reference will allow us to iterate through the class hierarchy and convert the instance type. This process will be explained in-detail in section 15.2.2.

15.1.2 Abstract Methods

Abstract methods operate as virtual methods, with the exception that abstract methods do not have a body and require immediate subclasses to implement them. Therefore, the abstract method in itself, will not be translated to OpenCL C, but all subclass implementations of the abstract method will be translated as virtual methods.

15.1.3 Independent Overriding

In C#, independent overriding is sometimes called *method hiding*. Independent overriding can be done on any method that is in a superclass, and is accomplished using the *new* keyword. Unlike virtual or abstract methods, independent overriding does not take subtyping into account. In section 15.1.1 an example where a *Pug* is referenced by an *Animal* variable is given. In this scenario, the virtual and abstract method would invoke the *Pug* implementation of *MyMethod()*, while the independent overridden method will invoke the *Animal* implementation.

In order to translate independent overriding, we will need to make sure it is neither an abstract nor virtual method and then invoke the version of the method based on the variable type it is referenced by.

15.1.4 Conclusion

In the requirement specification chapter 9, the second priority stated that each of the three overriding methods are equally suited to achieve polymorphism and only one of the overriding methods will be implemented to fulfill the second priority. For the second priority we choose to implement virtual methods as they contain a body and does not require immediate subclasses to implement them compared to abstract methods. This means that the virtual method in itself will be translated to OpenCL C and subclasses are not required to override the methods. Independent overriding is not chosen for the second priority as it does not take subtyping into account.

15.2 Type Conversion

Type conversion is the process of converting one data type to another. There are two ways to utilize type conversion in C#: subtyping and casting [36]. Subtyping is converting a subclass type to a superclass type, while casting is converting a superclass type to a subclass type. In the following sections we describe the design of subtyping and casting.

15.2.1 Subtyping

In this section, we will detail the design for adding subtyping support to OpenCLS 2.

Subtyping can be divided into two scenarios: static and dynamic [37]. Static subtyping occurs during compile-time, while dynamic subtyping occurs during run-time. Static subtyping is only possible when the source and target types are known at compile-time, as the compiler will then be able to iterate through the class hierarchy to find the steps needed to perform the cast. Dynamic subtyping is necessary if either the source or target type is unknown at compile-time.

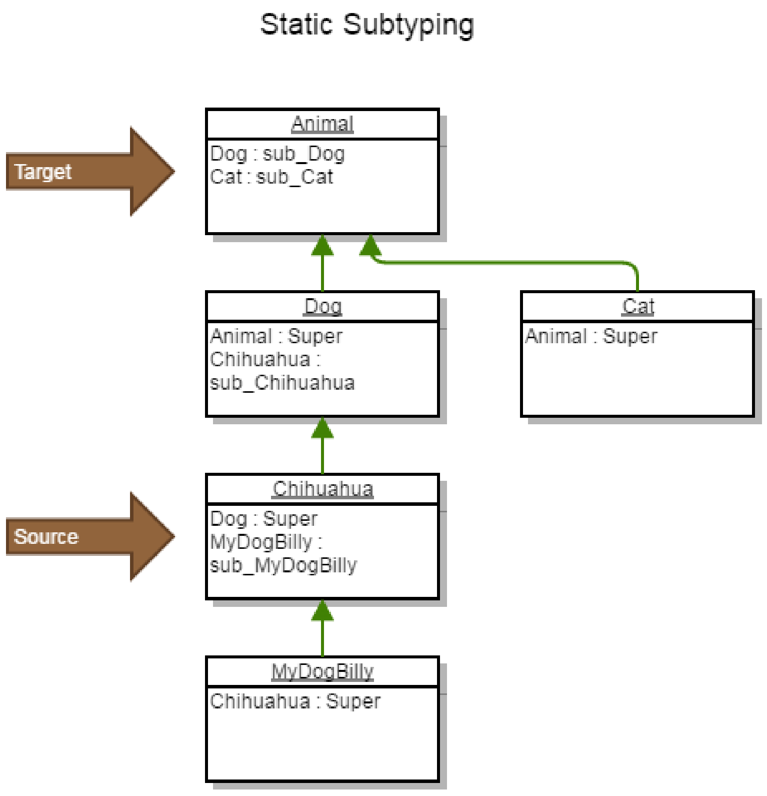


Figure 15.3: UML diagram showing a class hierarchy with a known target and source.

In OpenCLS 2, the Roslyn platform's semantic model provides the source and target type during subtyping. Because of this, we will only need to implement support for static subtyping. Figure 15.3 shows an example class hierarchy with a known target and source. It can be seen that the target type can be reached by iterating through the chain of *super* variables. Listing 15.2 shows the pseudo-code for this concept.

```
DoStaticSubtyping(InObject, SourceClass, TargetClass)
{
    while (SourceClass != TargetClass)
    {
        InObject = InObject.super;
        SourceClass = InObject.C-lass;
    }
}
```

Listing 15.2: Pseudo-code for static subtyping

The *TargetClass* will reach the *SourceClass* through the chain of *super* variables, as long as the subtyping operation is valid in C#.

15.2.2 Casting

Similar to subtyping, casting can be done either statically (during compile-time) or dynamically (during run-time). Since Roslyn provides us with a semantic model with information on the source and target type, we will only need to implement static casts.

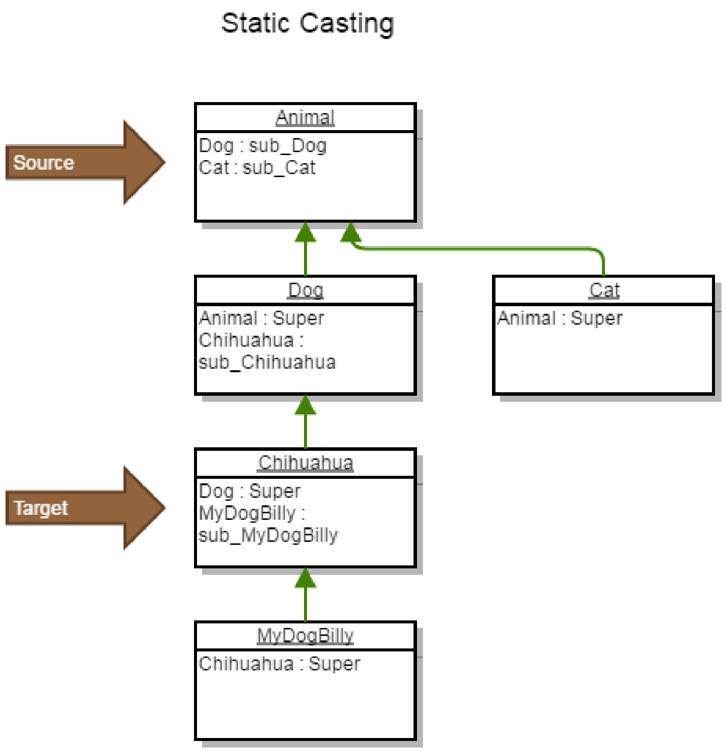


Figure 15.4: UML diagram showing a class hierarchy with a known target and source.

Figure 15.4 shows an example of a hierarchy that needs to be iterated through to cast from source to target. This procedure is different from subtyping, as the iteration goes down a tree that can branch (which was not the case in subtyping). We can see from figure 15.4 that we can iterate through the chain of *sub* member variables to reach the target type from the source type.

The pseudo-code for this iteration can be seen in listing 15.3.

```
DoStatiCasting(InObject, SourceClass, TargetClass)
{
    if (SourceClass == TargetClass) return InObject;
    {
        foreach (Sub in InObject.Sub)
        {
            Result = DoStatiCasting(Sub, SourceClass, Sub.Class);
            if (Result != NULL)
                return Result;
        }
    }
    return NULL;
}
```

Listing 15.3: Pseudo-code for static casting

The design for subtyping and the design for casting will form the basis for the implementation part.

16 Summary

In this part we described the features of the previous OpenCLS 1 project [1], which are prerequisite for the design of the features for OpenCLS 2. We examined each of the OOP pillars (Inheritance, Encapsulation and Polymorphism), and we described the design of each feature in the pillars.

For the inheritance pillar, we have designed the features of inheritance, member access and forward declaration. The features designed for encapsulation are interfaces and abstract classes. Finally we designed the overridding, subtyping and casting features of the polymorphism pillar of OpenCLS 2. In section 15.1.4, we decided to implement virtual methods for the second priority.

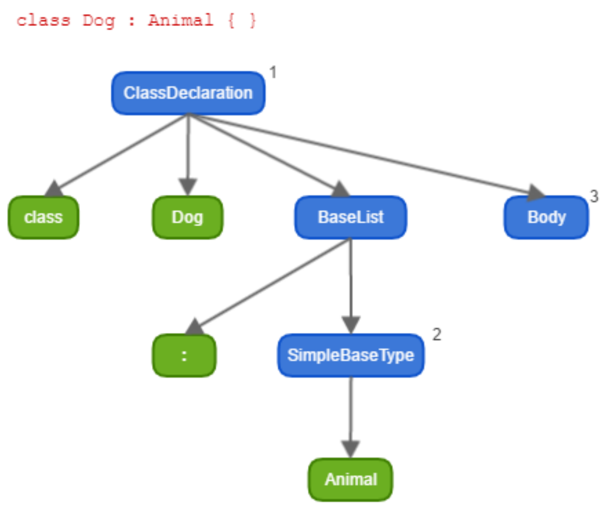
Part IV

Implementation

This part describes the implementation of the features in inheritance, encapsulation and polymorphism introduced in the design part III. The chapters will explain how C# code with inheritance, encapsulation and polymorphism is translated with OpenCLS 2. Beyond what is described in this chapter, we have also performed code refactoring, to make the OpenCLS 2 codebase more readable and reduce code duplication.

17 Inheritance

This chapter will detail the implementation of the first pillar, inheritance. The implementation includes inheritance, member access and forward declaration and is based on the designs presented in chapter 13. Any design changes to the features of the inheritance pillar during implementation will be described at the end of this chapter.



(a) C# Syntax tree.

```
struct Dog
{
    struct Animal super;
}
```

(b) Generated OpenCL C code for a class declaration with inheritance.

Figure 17.1: Sample class declaration (Dog inheriting from Animal), syntax tree and output code.

Figure 17.1 shows the C# syntax tree for a sample class declaration of the class Dog, inheriting from the class Animal. OpenCLS 2 iterates on the syntax tree depth-first, and only the numbered nodes have implemented functionality. The numbers *1*, *2* and *3* mark the nodes with functionality, and the order they are called in (lowest to highest).

1: The class declaration syntax node, which was already implemented in OpenCLS 1 [1, p 36]. This generates the struct seen in figure 17.1b.

2: The *SimpleBaseType* node, which only exists in the syntax tree if the class is a subclass. This node generates a member variable called *super*, with the type being the generated struct for the superclass of the C# class. As the superclass member variable is a value-based struct, the subclass becomes a nested structure. Nested structs require that any structs within need to be declared beforehand. In the sample class provided by Figure 17.1, the declaration for the Animal struct needs to appear before the declaration for the Dog struct. The implementation for proper forward declarations will be explained further in section 17.2.

3: This node inserts the members (fields, properties and methods), which is already documented in OpenCLS 1 [1, p 36].

Listing 17.1 shows the implementation when visiting the *SimpleBaseType* node in the syntax tree.

```
public override void VisitSimpleBaseType(SimpleBaseTypeSyntax node)
{
    var baseType = _SemanticModel.GetTypeInfo(node.Type).Type;
    if (baseType.TypeKind == TypeKind.Interface)
    {
        // Translate interfaces
    }
    else
    {
        //if the superclass does not have the OpenCLS attribute, we cannot create an instance of it, since there will be no implementation on the GPU of the class definition
        //If this is the case, GetClassDeclarationSyntax will throw a NotSupported exception
        GetClassDeclarationSyntax(node, node.Type.ToString());

        //Write an instance of the superclass in the derived class
        WriteLineOutput("struct " + baseType.Name + " super;");
    }
}
```

Listing 17.1: Implementation for visiting SimpleBaseType in the syntax tree.

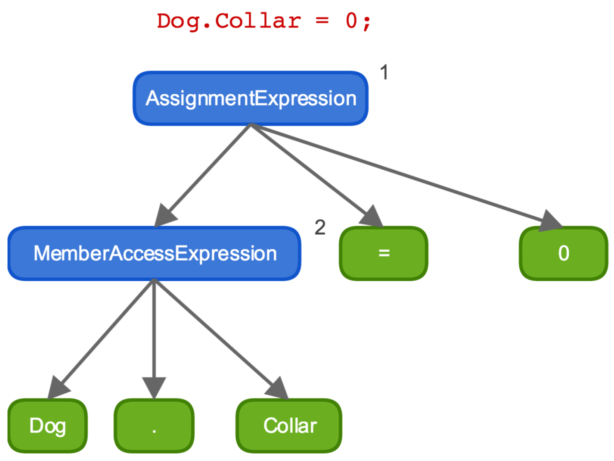
With this addition, OpenCLS 2 classes have access to an instance of their superclass. However, superclass member variables and method calls do not support inheritance yet. These two additions will be described in the following two sections.

17.1 Member Variable Access

In order for a subclass to access member variables in a superclass in OpenCL C, it needs to access them through the *super* member. As we designed in section 13.1, to enable this functionality we insert *super* before any member variables that are inherited from the superclass.

Figure 17.2 shows the C# syntax tree for member access (in the context of an assignment expression). Because the *MemberAccessExpression* node is always present when accessing member variables, we will be inserting *super* to *MemberAccessExpression*.

Listing 17.2 shows the implementation for adding *super* to *MemberAccessExpression*. Line 6 has a loop that goes through the class declaration parents recursively, until it finds the member variable that we are searching for, in order to find out how many times we need to insert *super*. Line 8 checks if the current class declaration contains the member variable, and



(a) C# Syntax tree.

```
dog.super.Collar = 0;
```

(b) Generated OpenCL C code.

Figure 17.2: Sample object member access syntax tree and output code.

whether it is of type *FieldDeclarationSyntax* or *PropertyDeclarationSyntax*, as these are the syntax nodes for member variables.

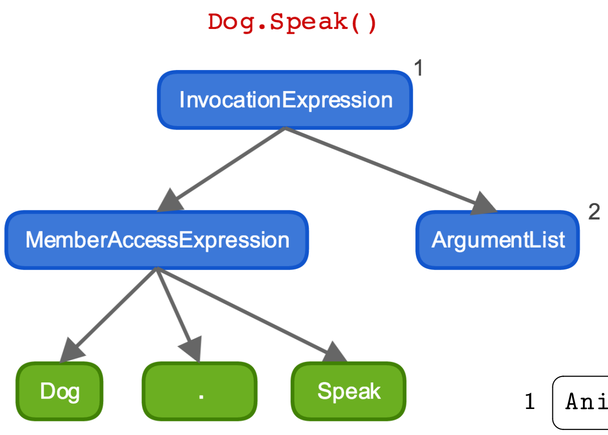
```
private void GetMembers(MemberAccessExpressionSyntax node, string ↩
    IdentifierValue, string StructTypeName)
{
    var classDeclaration = GetClassDeclarationSyntax(node, ↩
        StructTypeName);
    bool containsMember = false;

    while (classDeclaration != null && !containsMember)
    {
        containsMember = classDeclaration.Members.Any(m => (m as ↩
            FieldDeclarationSyntax)?.Declaration.Variables.Any(v => ↩
            v.Identifier.ValueText == IdentifierValue) ?? (m as ↩
            PropertyDeclarationSyntax)?.Identifier.ValueText == ↩
            IdentifierValue);
        var parent = classDeclaration.BaseList?.Types.↩
            FirstOrDefault();
        classDeclaration = GetClassDeclarationSyntax(parent, parent↩
            ?.Type.ToString());

        WriteOutput(containsMember == false ? "super." : "");
    }
}
```

Listing 17.2: Inheritance member access in OpenCLS 2.

17.1.1 Member Method Access

Similar to member variable access, method access requires some changes to access methods declared in a superclass.

In OpenCLS 2, member methods are given a prefix that corresponds to the class they are contained within. Figure 17.3b shows this using the *Speak* method declared in *Dog*'s superclass, *Animal*.



(a) C# Syntax tree.

```
Animal_Speak(&dog);
```

(b) Generated OpenCL C code.

Figure 17.3: Sample object method invocation syntax tree and output code.

In our OpenCLS 1 implementation without inheritance, the prefix was generated from the type of the calling object. However, this will need to be changed when inheritance is introduced, as figure 17.3a would result in the incorrect OpenCL C code shown in listing 17.3.

```
// class Animal { public void Speak(); }
// class Dog : Animal {}
// Dog Dog = new Dog();
// Dog.Speak();
Dog_Speak(&Dog);
```

Listing 17.3: Incorrect OpenCL C code result generated from figure 17.3a. The correct version can be seen in figure 17.3b.

To resolve this issue, we iterate through the calling object's superclasses to find the correct prefix (in this scenario, *Animal_*). The code for this can be seen in listing 17.4.

```
public override void VisitInvocationExpression(↩
    InvocationExpressionSyntax node)
{
    if (node.Expression != null && node.Expression is ↩
        IdentifierNameSyntax)
    {
        // Find ClassDeclarationSyntax for class declaring the ↩
            method being called
        ClassDeclarationSyntax className = node.SyntaxTree.↩
            DescendantNodes().OfType<ClassDeclarationSyntax>().Where↩
            (c => c.Members.OfType<MethodDeclarationSyntax>().Any(m ↩
            => m.Identifier.ValueText == node.Expression.ToString())↩
            ).FirstOrDefault();

        // Write the method call
        WriteOutput(GetClassMethodName("", className.Identifier.↩
            ValueText, node.Expression.ToString(), node.ArgumentList↩
            .Arguments.Select(a => a.ToString()).ToList(), node));
```

Listing 17.4: Obtaining the correct class for method calls to resolve the issue seen in listing 17.3.

17.2 Forward Declaration

In OpenCLS 1 we introduced forward declaration, but the implementation did not take inheritance into account, and will therefore need to be updated for OpenCLS 2. With the introduction of inheritance, we implement three checks before translating the class declaration which can be seen in listing 17.5.

Check 1: Ensure that a class only gets translated once.

Check 2: If the class has any superclasses, declare the superclass.

Check 3: Go through the members of the class, if any members are class types, these member classes are declared before we declare the class itself.

```
void DeclareClass(ClassDeclarationSyntax classDeclaration)
{
    // Check 1: Check if this class has already been declared
    if (declaredClasses.Contains(classDeclaration) || ↩
        classDeclaration == null)
        return;

    // Check 2: Declare superclass first
    if (classDeclaration.BaseList?.Types.Count > 0)
    {
        var parent = GetClassDeclarationSyntax(classDeclaration.↩
            BaseList.Types.FirstOrDefault(), classDeclaration.↩
            BaseList.Types.FirstOrDefault().Type.ToString());
        DeclareClass(parent);
    }

    // Check 3: Check if member is a class, then declare that type
    foreach (var field in classDeclaration.Members.OfType<↩
        FieldDeclarationSyntax>().Where(f => f.Declaration.Type is ↩
        IdentifierNameSyntax))
    {
        var memberType = classDeclarations.FirstOrDefault(c => c.↩
            Identifier.ValueText == (field.Declaration.Type as ↩
            IdentifierNameSyntax).Identifier.ValueText);
        if (memberType != null)
            DeclareClass(memberType);
    }

    // Declare the class
    base.Visit(classDeclaration);
    declaredClasses.Add(classDeclaration);
}
```

Listing 17.5: Class declaration rewritten to support SuperClasses.

Despite this implementation, there is one scenario that OpenCLS 2 does not support; the case where two classes reference one another in member variables. Because both classes are implemented as structs, they cannot both have a full definition, as either class will need the other class to be fully defined before it can complete its own definition. For this scenario, OpenCLS 2 will emit an error.

17.3 Design Changes

The design for inheritance has changed since its initial conception. Initially the superclass was reference-type, and subclasses were value-type.

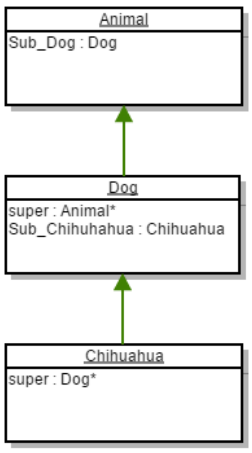


Figure 17.4: The earliest design for inheritance.

However, this design could not be implemented in OpenCL C. Consider the scenario in figure 17.4:

- An instance of the *Chihuahua* struct is declared.
- The OpenCLS-generated constructor for *Chihuahua* is called.
  - An instance of *Dog* is declared, and put in *Chihuahua*'s *super* variable as well as *Animal*'s *Sub_Dog* variable. The instance of *Chihuahua* is put in *Dog*'s *Sub_Chihuahua* variable.
  - An instance of *Animal* is declared, and put in *Dog*'s *super* variable.
  - The constructor is exited.
- Since *Dog* and *Animal* were local to the constructor, Dog and Animal go out of scope. *Chihuahua*'s *super* is now an invalid reference.
- Any operations on *Chihuahua*'s *super* will result in a segmentation fault.

This issue could be resolved with dynamic memory allocation, but this is not supported in OpenCL C and is not a planned feature for OpenCLS 2. Instead, moving the pointer reference from the *super* variables to the *Sub_* variables (as seen in the final design) resolved this issue.

18 Encapsulation

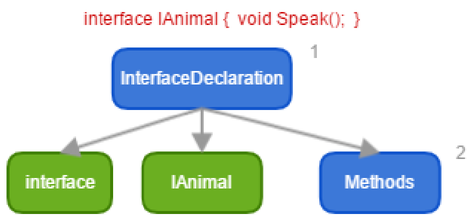
In this chapter, we will describe the implementation of the encapsulation pillar in OpenCLS 2. This implementation includes interfaces and abstract classes. Additionally, we will also describe the implementation of when interfaces are used beyond the encapsulation pillar, such as when it is being inherited from.

18.1 Interface

In this section we describe the implementation of interfaces. There are four scenarios where interfaces can be used in C#; programmers can declare interfaces, inherit from them, use them as variable types and use them as parameter types [38]. The only way interfaces can be used as variable types is through subtyping, which will be described in chapter 19.

18.1.1 Declaration

Figure 18.1 shows the C# syntax tree for a sample interface declaration of the interface IAnimal, and its corresponding OpenCL C code. As can be seen in figure 18.1a, there are two main parts to an interface declaration, labeled *1* and *2*.



(a) C# Syntax tree.

```
struct Interface
{
    struct Dog Sub_Dog;
}
void Interface_Constructor(struct↩
    Interface* _instance);
void ↩
   Interface_Speak_Virtual_Resolver↩
   (struct Interface* _instance,↩
    struct Interface dog);
```

(b) Generated OpenCL C code.

Figure 18.1: Sample interface declaration (*IAnimal*) syntax tree and output code.

1. The *InterfaceDeclaration* node, which implements all interfaces into a single *Interface* struct in OpenCL C. Because we combine all interfaces into a single struct, as described

in section 14.1, the *Interface* struct does not get a forward declaration, but is instead declared as the first struct in the OpenCL C code. The only members of the *Interface* struct are subclass pointer variables.

2. The *Methods* that the interface might have. Each method gets its own *Virtual_ Resolver* method that will determine which of its subclass overridden methods that needs to be called. Chapter 19 will describe how virtual resolver methods are implemented.

Listing 18.1 shows the code for *VisitInterfaceDeclaration*, which creates the *Interface* struct.

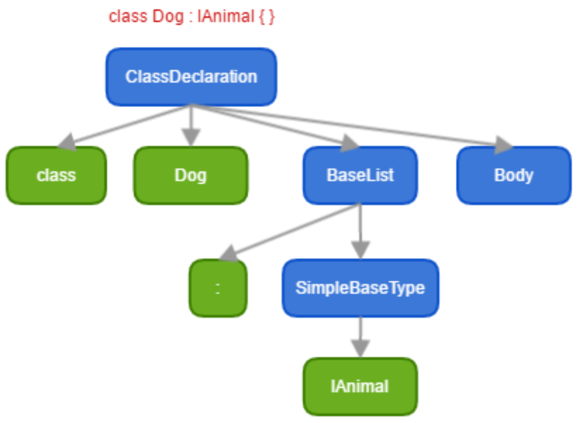
```
public override void VisitInterfaceDeclaration(InterfaceDeclarationSyntax node)
{
    // Instead of forward declaring the interface, simply declare it
    if (declarationState == DeclarationState.Forward)
    {
        WriteLineOutput("struct Interface{");
        CurrentTabs++;
        WriteLineOutput("int OpenCLS_ClassID;");
        // Finds all interfaces and the classes that inherits from them
        DeclareInterfaceSubClasses(node);
        CurrentTabs--;
        WriteLineOutput("};");
    }
    else
    {
        // implement constructor and methods
    }
}
```

Listing 18.1: Code for declaring interfaces

With this addition, OpenCLS 2 can now declare interfaces, which means we support one of the four scenarios for interfaces.

18.1.2 Inheritance

As can be seen in figure 18.2 the syntax tree for inheriting from an interface is identical to the syntax tree for inheriting from a class (see chapter 17). This means we need to look in the semantic model in order to find out if the *SimpleBaseType* is an interface or not.



(a) C# Syntax tree.

```
struct Dog
{
    struct Interface ↩
        interfaceSuper;
}
```

(b) Generated OpenCL C code.

Figure 18.2: Sample syntax tree and output code for inheriting from an interface.

If it is an interface, we add an instance of the *Interface* struct into the class struct declaration, regardless of the original interface type, as all interfaces get translated into the *Interface* struct. In order to translate this, we use *VisitSimpleBaseType* (as we did in chapter 17), as can be seen in listing 18.2.

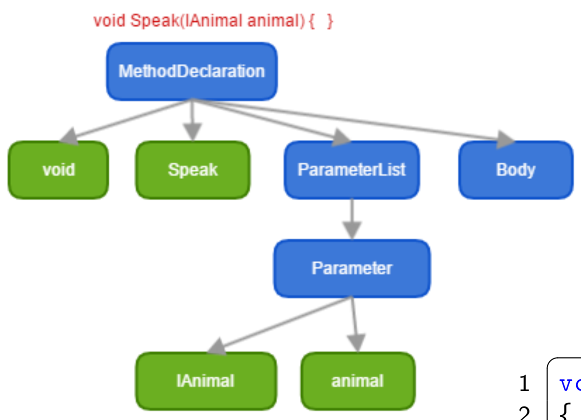
```
public override void VisitSimpleBaseType(SimpleBaseTypeSyntax node)
{
    var baseType = _SemanticModel.GetTypeInfo(node.Type).Type;
    if (baseType.TypeKind == TypeKind.Interface)
    {
        WriteLineOutput("struct Interface interfaceSuper;");
        FirstWasInterface = true;
    }
    else
    {
        // translate normal classes
    }
}
```

Listing 18.2: The *Interface* is inserted as a member variable into subclasses that inherit from an interface.

Member access through the *interfaceSuper* is added the same way as described in section 17.1.1. However, unlike normal classes, it will only be needed for subtyping.

18.1.3 Parameters

Interfaces can be used as a parameter type, and because all interfaces are translated into a single *Interface* struct, the interface as a parameter type needs to be translated as well. In figure 18.3 a C# syntax tree can be seen for a method which takes an interface type as a parameter. However from the syntax tree we can not tell the type of the parameter, and only that the parameter type is called *IAnimal*. This means that we have to look for the type of the parameter in the semantic model.



(a) C# Syntax tree.

```
void Speak(Interface animal}
{
}
```

(b) Generated OpenCL C code.

Figure 18.3: Syntax tree and output code for a sample interface as parameter type.

In listing 18.3, the code for translating the interface type can be seen. We check the semantic model, and depending on the type we translate it differently. If the type is:

Class: we add *struct* in front of the class name.

Interface: we translate it to *struct Interface*.

Not a class or an interface: we translate it directly.

```
var typeKind = _SemanticModel.GetTypeInfo(type).Type.TypeKind;
if (typeKind == TypeKind.Class)
    translatedType = "struct " + type.ToString();
else if (typeKind == TypeKind.Interface)
    translatedType = "struct Interface";
else
    translatedType = type.ToString();
```

Listing 18.3: Code for interfaces as parameter type. This gets called from the MethodDeclaration visitor.

With the changes seen in listing 18.3, OpenCLS 2 compiles C# interfaces in method declarations into the global OpenCL C *Interface* struct. This makes it possible to pass interfaces to method invocations.

With these additions, OpenCLS 2 supports interfaces in C#, by supporting declaring interfaces, inheriting from interfaces, using interfaces as variables types and using interfaces as parameter types.

18.2 Abstract Classes

As described in section 14.2, our approach to abstract classes is to compile them to non-abstract classes. To achieve this, we ignore the *abstract* keyword, and translate abstract classes as if they were non-abstract classes. This change is minor, but it achieves the goal of making abstract class available in OpenCLS 2.

18.3 Design Changes

The design for interfaces have changed since its initial conception. Figure 18.4 shows our original design and as can be seen, initially we decided to not translate interfaces to OpenCL C.

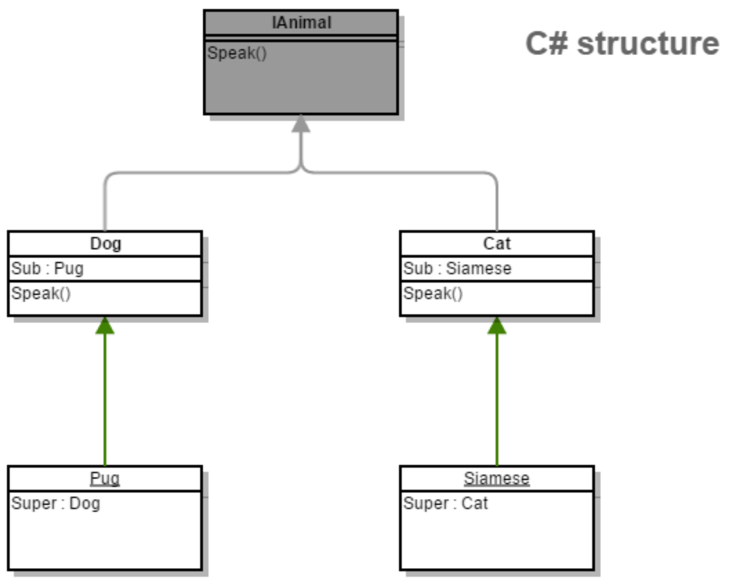


Figure 18.4: The original interface design, where the interface is not translated to OpenCL C.

However, C# allows interfaces to be used as a type for variables and parameters. Our initial idea was to replace the interface types with corresponding non-interface types. Listing 18.4 shows an example of this idea, and showcases that we would have to make multiple versions of the same method, if the method used an interface as a type for its parameters, resulting in redundant code.

```
// C#:
void Speak(IAnimal animal)
{
    Dog dog = new Dog();
    animal = new Cat();
}

// OpenCL C:
void Speak_Dog(struct Dog animal)
{
    struct Dog dog; // identical to Speak_Cat()
    Dog_Constructor(&dog); // identical to Speak_Cat()
    Cat_Constructor(&animal); //type incompatible
}
void Speak_Cat(struct Cat animal)
{
    struct Dog dog; // identical to Speak_Dog()
    Dog_Constructor(&dog); // identical to Speak_Dog()
    Cat_Constructor(&animal);
}
```

Listing 18.4: Initial interface design sample. Multiple similar methods have to be created in OpenCL C from one C# method and it creates type incompatibility when attempting to subtype the interface parameter. Additionally it creates redundant code as both OpenCL C methods would be identical except from their parameters.

In addition to creating multiple versions of the same method, this idea also made subtyping problematic, as seen in line 5 in listing 18.4. The instantiation of *animal* cannot be in the *Dog* version of the method, since *animal* would be of type *Dog* in this method. Because of this dilemma we decided to translate interfaces (as seen in the final design) to resolve the issue.

19 Polymorphism

This chapter will detail the implementation of the third pillar, polymorphism. The implementation of virtual overriding and subtyping are described and based on the designs presented in chapter 15.

19.1 Virtual Overriding

Figure 19.1 shows the flowchart for OpenCLS 2's virtual method implementation, as well as output code for a sample virtual method.

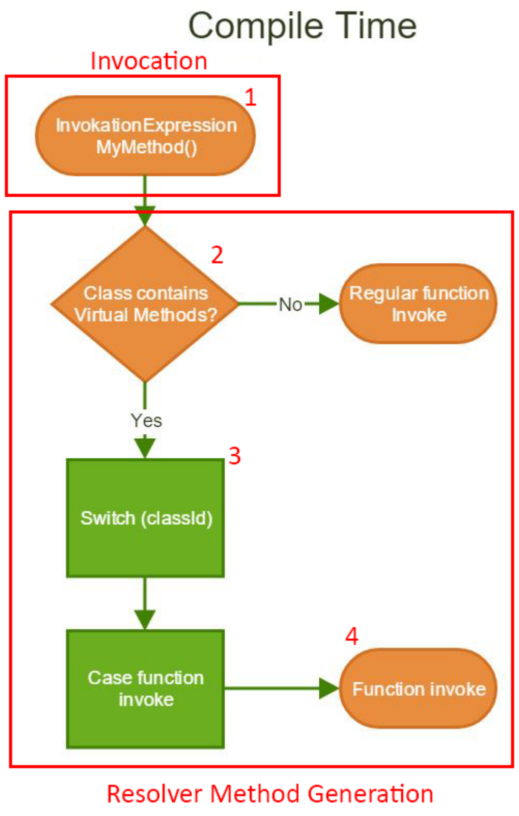
The following is a step-by-step description of figure 19.1a, showing how OpenCLS 2 handles virtual method invocations as well as how it generates the virtual resolver method.

1. When an object tries to invoke an overridden method, we replace it with a call to a virtual resolver, as seen in listing 19.1.
2. If the *virtual* keyword is used when declaring a method, we check if the corresponding class has any subclasses. For classes with no subclasses, we translate the virtual method as if it was a non-virtual method. For classes with at least one subclass, we create a resolver method, by using the virtual method's name and adding the suffix *Virtual_Resolver*, as seen in figure 19.1b.
3. The virtual resolver implements a *switch* for the instance's *OpenCLS_ClassID*, which all classes in OpenCLS 2 have (as seen in section 15.1), in order to find the subclass method matching the instance type.
4. The resolver accepts an instance of the class that implements the virtual method (as can be seen in figure 19.1b). This is done through subtyping (chapter 15.2.1).

```
// C#
dog.Sound();

// OpenCL C
Animal_Sound_Virtual_Resolver(&dog.super);
```

Listing 19.1: C# virtual method calls are replaced with a call to the virtual resolver for the virtual method.

With virtual methods implemented, OpenCLS 2 supports overriding (polymorphism pillar).



(a) Flowchart from chapter 15.1 showing the various steps in the implementation for virtual methods.

```
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 0:
            Animal_MyMethod(_instance);
            break;
        case 1:
            Dog_MyMethod(_instance->Sub_Dog);
            break;
    }
}
```

(b) Generated OpenCL C code for a virtual method.

Figure 19.1: Sample *resolver* method, syntax tree and output code.

19.2 Subtyping

OpenCLS 2 implements subtyping statically (as described in section 15.2.1). The implementation for subtyping can be seen in listing 19.2.

```
 // Only if TargetClass is a superclass of CurrentClass
private void DoStaticSubtyping(string SourceClassName, string ↩
    TargetClassName, SyntaxNode node, string AccessorToken = ".")
{
    ClassDeclarationSyntax SourceClassDecl = GetClassDeclarationSyntax(node, SourceClassName);
    ClassDeclarationSyntax TargetClassDecl = GetClassDeclarationSyntax(node, TargetClassName);

    // We need to walk CurrentClass to TargetClass using .super

    List<ClassDeclarationSyntax> Walk = GetSuperClassToSubClassPath(TargetClassDecl, SourceClassDecl);
    foreach (var ClassDecl in Walk)
        WriteOutput(AccessorToken+"super");
}
```

Listing 19.2: When both the source and target classes are known the compiler can iterate through the chain of super member variables to reach the target type.

OpenCLS 2 iterates over the set of super variables, while appending *.super* to the object, until the target class is reached.

Additionally, subtyping needs to support storing objects in superclass type variables. An example of this scenario can be seen in listing 19.3.

```
1      // C#
2      Animal MyAnimal = new Dog();
3
4      // OpenCL C
5      struct Dog MyAnimal_Sub;
6      Dog_Constructor(&MyAnimal_Sub);
7      struct Animal MyAnimal = MyAnimal_Sub.Super;
8  }
```

Listing 19.3: A *Dog* class stored in a variable with type *Animal*.

To support this kind of declaration, we create a temporary *_Sub* variable (line 5), initialize it (line 6), and then subtype it to the type of the variable (line 7). With this implementation, OpenCLS 2 supports subtyping.

20 Summary

In this part we described the implementation of the three OOP pillars: Inheritance, encapsulation and polymorphism. We implemented support for inheritance by introducing the *super* member variable, which allows subclasses to access members declared in superclasses. Additionally, we modified how members are accessed to take this change into account. Forward declaration from OpenCLS 1 was also rewritten for OpenCLS 2 to take member variables and class hierarchies into account.

Support for encapsulation was added through interfaces and abstract classes. OpenCLS 2 translates all interfaces into a single struct, and all classes that inherit from the interface have this struct as a member variable (similar to class inheritance). Abstract classes are compiled to non-abstract classes in OpenCL C.

OpenCLS 2 supports polymorphism through virtual overriding and subtyping. Virtual overriding is implemented through a *virtual resolver method*, which considers a class' *OpenCLS_ClassID* and determines which overridden method to invoke. Subtyping is implemented statically by iterating over the chain of *super* member variables to convert an object from a source class type to a target class type.

With these additions, OpenCLS 2 supports the three pillars of OOP.

Part V

Test

This part describes the testing of OpenCLS 2. We take inspiration in the IEEE 829 [39] guideline for software testing. We will utilize black-box testing through design pattern and performance tests, as well as white-box testing through unit testing.

21 Unit Test of OpenCLS 2

This chapter describes the OpenCLS 2 test of the three pillars in object-oriented programming (described in chapter 6). We will be going through the following in this chapter:

- Plan
- Design Specification
- Case Specification
- Result
- Summary

21.1 Plan

The purpose of this test is to check if the compiler translates the input code as expected. We will provide each test with an expected result, and compare it against the actual output. We conduct white-box unit test on each pillar (inheritance, encapsulation and polymorphism) of OpenCLS 2, to test the features of each pillar in the smallest testable unit. White-box testing refers to the use of program source code as a basis for testing and writing the input code of test cases [40]. White-box is necessary in order to write the smallest set of test cases to test the features of OpenCLS 2. These white-box unit test are created using the unit test framework that is present in Visual Studio 2015 as described in chapter 3, this framework also handles the execution of the test.

21.2 Design Specification

This section specifies the test design for each OOP pillar of OpenCLS 2 and describes the list of features derived from the OOP pillars that will be tested, as well as the design of test cases and the pass criteria of these features.

21.2.1 Inheritance

Testing the inheritance pillar requires us to test three parts: inheritance, member access and out of order class declarations.

Inheritance

The inheritance test scenarios are based on C# inheritance designs by *Vithal Wadje* [41] and *Gurunatha Dogi* [42]. The designs have been renamed to remain consistent with the terminology and naming conventions used in this project. The scenarios are:

- Single - tests if the superclass is translated into an instance in the subclass, in OpenCL C.
- Fork - tests if a superclass can be inherited by multiple subclasses. The full test is listed in appendix A.1.
- Chain - tests if a subclass can inherit from a subclass. The full test is listed in appendix A.2.
- Chain Fork - tests a mix of both fork and chain. The full test is listed in appendix A.3.
- River Delta - tests a mix of all the previous tests. The full test is listed in appendix A.4.
- No superclass constructor - tests if a superclass' constructor is automatically created, even when there is no instance of the superclass nor a constructor for the superclass in the C# code. The full test is listed in appendix A.5.
- No subclass constructor - tests if a subclass' constructor is automatically created, when there is no constructor in the C# code. The full test is listed in appendix A.6.
- Invoke superclass method in subclass constructor - tests if *super* is added to a method call, in a subclass' constructor, to a method from a superclass. The full test is listed in appendix A.7.
- Invoke superclass method in subclass method - tests if *super* is added to a method call, in a subclass' method, to a method from a superclass. The full test is listed in appendix A.8

Member Access

To test member access we will create unit tests that test different levels of inheritance, since member access only changes depending on how many times we have to access the superclass. We test the following scenarios:

- Access member from a superclass - tests if OpenLCS 2 translates the field access to access the superclass instead of the subclass, the full test is listed in appendix A.9.
- Access member from a superclass' superclass - tests if we correctly translate the field access to access the superclass' superclass, the full test is listed in appendix A.10.
- Method member access - tests if OpenCLS 2 translates member access to methods as expected, the full test is listed in appendix A.11.

Out of Order Declarations

C# and OpenCL C have different rules and standards for the order of declarations of classes, fields, properties, constructors and methods. When translating from C# to OpenCL C, we test if the code can build and whether the semantics are kept intact. To test out of order declarations we create unit tests for class declarations and member with class type, with different order.

- Out of order inheritance declaration - tests if OpenCLS 2 translates inheritance declaration when the subclass is declared before the superclass, the full test is listed in appendix A.12.
- Out of order class field - tests if OpenCLS 2 translates a class field, if the type of the field is declared after the class with the class field, the full test is listed in appendix A.13.

21.2.2 Encapsulation

Testing of the encapsulation pillar is divided into following parts: interface, astract and protected. For each of these test parts, test cases are required for testing if OpenCLS 2 translate C# code to OpenCL C as expected.

Interface

The interface test design is similar to the test design for inheritance in subsection 21.2.1. To test the interface implementation of OpenCLS 2, multiple test cases are added to test interface inheritance and interface utilized with classes.

- Single - tests if an interface can be inherited by a single subclass, the full test is listed in appendix A.14.
- Fork - tests if an interface can be inherited by multiple subclasses, the full test is listed in appendix A.15.
- Chain - tests if a subclass can inherit from a subclass of an interface, the full test is listed in appendix A.16.
- Chain with only interfaces - tests if a subclass can inherit from an interface that inherits from another interface, the full test is listed in appendix A.17.
- Chain Fork - tests a mix of both fork and chain interface inheritance, the full test is listed in appendix A.18.
- Chain Reverse Fork - tests if a subclass can inherit from a subclass of multiple interfaces, the full test is listed in appendix A.19.
- Multiple - test if a subclass can inherit from both a superclass and a interface, the full test is listed in appendix A.20.
- Interface class mix - tests a mix of both multiple and chain interface inheritance, the full test is listed in appendix A.21.
- Flower - tests a flower structure of interfaces, the full test is listed in appendix A.22.
- Interface method invocation - tests if the *Interface_Virtual_Resolver* is invoked instead of methods from an interface, the full test is listed in appendix A.23.

Abstract

The abstract test is designed to check for the *abstract* keyword during translation. As abstract classes should be translated as non-abstract classes, we create tests similar to our non-abstract class declaration test from OpenCLS 1 [1], but with the *abstract* keyword added to the classes. Furthermore, we test if abstract classes can be used as subclasses.

- Abstract class declaration - test if it checks for the *abstract* keyword and if abstract classes are translated as non-abstract classes, the full tests are listed in appendix A.24.
- Abstract subclass - test if an abstract subclass can be translated, the full test is listed in appendix A.39.

Protected

The *protected* keyword is enforced in C# code before translating it to OpenCL C code, the testing checks if it translates as a non-protected.

- Protected field - test if a protected field is translated as a non-protected field, the full test is listed in appendix A.40.
- Protected method - test if it translates the protected method as a non-protected method, the full test is listed in appendix A.41.
- Protected class - test if a protected class is translated as a non-protected class, the full test is listed in appendix A.42.

21.2.3 Polymorphism

Testing of the polymorphism pillar is divided into virtual override and subtyping. Multiple test cases are designed for these two parts, to test that OpenCLS 2 translates C# code to OpenCL C as expected.

Virtual Override

To test virtual override, the test for inheritance needs to work before it can be tested. These tests requires a superclass with a virtual method and subclasses that can override the virtual method.

- Virtual Override - test if a subclass can override the virtual method of the superclass, the full test is listed in appendix A.43.
- Chain virtual override - test if a chain of subclasses can override the virtual method of the superclass, the full test is listed in appendix A.44.
- Chain virtual override with a skip - test if the overriding can be skipped or not overridden for the method in a subclass, but have a subclass that overrides the virtual method, the full test is listed in appendix A.45.

Subtyping

The subtyping tests are implemented to test if OpenCLS 2 translates subtyping as expected. The subtyping designs are described in chapter 15.2.1.

- Local subtyping declaration - test if a superclass object can be declared as a subclass, the full test is listed in appendix A.46.
- Local subtyping assign object creation - test if an already declared superclass can be assigned a new instance of a subclass, the full test is listed in appendix A.47.
- Local subtyping declaration with assignment - test if a superclass object can be declared and assigned an already declared subclass, the full test is listed in appendix A.48.

- Local subtyping assignment - test if an already declared superclass can be assigned an already declared subclass, the full test is listed in appendix A.49.

- Parameter subtyping - test if a method which takes a superclass, can take a subclass, the full test is listed in appendix A.50.

21.3 Case Specification

This section describes the first test case from the set of test cases designed in section 21.2, to give an example of how test case specifications are made. We will specify the C# input code, describe the output we expect and display the actual output code in OpenCL C. Additionally a class diagram is made to give a visual representation of the code structure and its attributes. The test case specifications for the remaining set of test cases are moved to appendix chapter A and displays the input and output code.

21.3.1 Single Inheritance

Single inheritance is a test case designed in subsection 21.2.1 to test if OpenCLS 2 translates code with single inheritance as expected. In figure 21.1 we show the class diagram of the unit test. It displays a subclass inheriting from a superclass.

In listing 21.1 we show the input C# code for figure 21.1. We expect that the OpenCL C output will have a struct for the *Animal* and *Dog* class. For single inheritance between *Animal* superclass and *Dog* subclass, we expect the respective class to contain an instance of the its superclass or subclass.

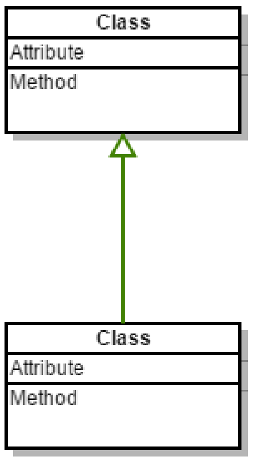


Figure 21.1: Class diagram for single inheritance test

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Dog : Animal { }
```

Listing 21.1: Input for single inheritance test

Listing 21.2 shows the OpenCL C output from translating listing 21.1. The created structs for *Animal* and *Dog* contains instances of their super and sub classes. No constuctor is made for the *Dog* class and its body in the C# code is empty. This test has passed as it has translated the single inheritance design as expected.

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
```

Listing 21.2: Output for single inheritance test. This output matches the expected result.

21.4 Result

In this section we show the result of the unit test that we have conducted on OpenCLS 2. We have separated this section into three subsections (Inheritance, Encapsulation and Polymorphism) for each OOP pillar. Each section includes a table with the result of the conducted unit test, in the table we show the number of tests conducted, how many succeeded, as well as the success rate.

These unit test are made to represent the smallest set of test cases to support a larger set of cases for each parts of the pillar, which means that whenever the code used in the unit tests of the inheritance pillar are written, the OpenCLS 2 compiler can translate it. All test tables in this section will keep this same structure.

21.4.1 Inheritance

It can be seen in table 21.1 that the unit tests for each parts of the inheritance pillar have a 100% success rate.

Test Part	Number of Tests	Successful	Success Rate
Inheritance (class)	9	9	100%
Member Access	3	3	100%
Out of Order declaration	2	2	100%

Table 21.1: Unit test results for the inheritance pillar

21.4.2 Encapsulation

It can be seen in table 21.2 that the unit tests for each parts of the encapsulation pillar have a 100% success rate.

Test Part	Number of Tests	Successful	Success Rate
Interface	10	10	100%
Abstract	16	16	100%
protected	3	3	100%

Table 21.2: Unit test results for the encapsulation pillar

21.4.3 Polymorphism

It can be seen in table 21.3 that the unit tests for each parts of the polymorphism pillar have a 100% success rate.

Test Part	Number of Tests	Successful	Success Rate
Virtual Override	3	3	100%
Subtyping	5	5	100%

Table 21.3: Unit test results for the polymorphism pillar

The result shows that the OpenCLS 2 compiler successfully translates all features from the inheritance and encapsulation pillar, and successfully translates virtual overriding from the polymorphism pillar. This means that OpenCLS 2 executes all of its OOP features as expected.

21.5 Summary

From the conducted test we conclude that OpenCLS 2 works as intended. In total we have conducted 51 unit tests, with the following separation:

- Inheritance - 14 unit tests
- Encapsulation - 29 unit tests
- Polymorphism - 8 unit tests

21.5.1 Inheritance

These tests show that OpenCLS 2 translated the C# object into the expected OpenCL code, members are accessed correctly and that the OpenCL C code is structured correctly with correct forward declaration and right order declaration of structs.

21.5.2 Encapsulation

The conducted unit tests show that OpenCLS 2 translates the C# interfaces into the expected OpenCL C code. Abstract and protected are ignored as we expected. This means that the OpenCLS 2 compiler supports the encapsulation pillar.

21.5.3 Polymorphism

These test show that OpenCLS 2 translates polymorphism features into expected OpenCL code. Virtual overriding is successfully executed in OpenCL code.

22 Design Pattern Test of OpenCLS 2

This chapter describes the design pattern test we have conducted on the OpenCLS 2 and will include three design pattern test, to test the OOP language features: Proxy and Visitor design patterns.

To describe the design pattern tests, we will be going through the following:

- Plan
- Design Specification
- Case Specification
- Result
- Summary

22.1 Plan

In this chapter, we will conduct black-box testing for each requirement, from chapter 9, by implementing programming design patterns that will use the features made available with each implemented requirement. The patterns we have implemented are the proxy and visitor design pattern, as described in chapter 10.

These black-box test are implemented and executed using Visual Studio 2015, by implementing the pattern and comparing the result when executed on the CPU and with the use of OpenCLS 2.

22.2 Design Specification

This section will describes the design of each design pattern test we have created for OpenCLS 2. The test design of each pattern is based on the analysis of the pattern in chapter 10 and the requirement specification chapter 9. For each design pattern test, we will specify what is needed to implement the design pattern and which features of OpenCLS 2 are used.

22.2.1 Proxy Pattern

To implement the proxy pattern, two classes need to inherit from a single interface as described in 10.1. The interface is created with a method, but contains no implementation for that method. The two classes that inherit from the interface must implement the method inside the interface, one class is a proxy class and the other is the class with the full implementation (FI class). The proxy class is implemented with a reference to an instance of the FI class and will instantiate the FI class when used, such as when the implemented method of the interface is called. The FI class is implemented with heavy computation and is only computed the first time it is instantiated by the proxy class. Implementing a proxy pattern will test features in the first priority requirement of OpenCLS 2 such as inheritance and interfaces.

22.2.2 Visitor Pattern

As we described in 10.2 to implement the visitor pattern, a visitor class and a root class is created. The visitor class contains a virtual *Visit* function for all types in the root class. The root class contains a virtual *Accept* function that takes an instance of the visitor as a parameter. The root class can be inherited by any subclasses and these subclasses will then override the accept function to call the appropriate visit function. Any classes in the class hierarchy is a subclass of the root class, and the visitor has a *Visit* function for all classes.

22.3 Case Specification

In this section we will describe the test cases for the design pattern test designed in 22.2. For each test we specify the expected result.

22.3.1 Proxy Pattern

The proxy pattern tests if the inheritance features of OpenCLS 2 are translated as expected.

In listing 22.1, 22.2 and 22.3 we have the implemented code for the proxy pattern in C#. In listing 22.1 we declare our *Subject* interface that will specify the common features of our two classes, a method called *GetAge()* is declared.

```
interface Subject
{
    int GetAge();
}
```

Listing 22.1: The *Subject* interface

In listing 22.2 we have declared a class called the *ProxySubject* which is the proxy implementation of *Subject*. In line 3 we store a reference to the *RealSubject* and it will only be instantiated when used. This can be seen in line 5 where the *GetAge()* method of the interface which must be implemented, instantiates the *RealSubject* if it has not already been instantiated, and then returns the *GetAge()* from the *RealSubject*.

```
1  class ProxySubject : Subject
2  {
3      RealSubject subject;
4      bool IsInitialized = false;
5      public int GetAge()
6      {
7          if (!IsInitialized)
8          {
9              subject = new RealSubject();
10             IsInitialized = true;
```

```
        }

        return subject.GetAge();
    }
}
```

Listing 22.2: The *ProxySubject* class that inherits from *Subject*

In listing 22.3 we have the *RealSubject* class, this class includes all computational heavy code, and is only instantiated when the *ProxySubject* instantiates it.

```
class RealSubject : Subject
{
    int birthYear;
    public RealSubject()
    {
        LoadBirthYearFromMemory();
    }
    public int GetAge()
    {
        // Calculate age
        return 2016 - birthYear;
    }
    void LoadBirthYearFromMemory()
    {
        birthYear = 1991;
    }
}
```

Listing 22.3: The *RealSubject* class that inherits from *Subject*

Expected Result

For this test we expect the result *25*, since it should calculate the age from a birth year (1991) and as of today we are in 2016.

22.3.2 Visitor Pattern

The visitor pattern tests if OpenCLS 2 translates inheritance, encapsulation and polymorphism as expected.

In listing 22.4 we have implemented two classes for the visitor pattern. The *VP_ Element* class is used by subclasses to override the *Accept* method which takes a *VP_ Visitor* class type as parameter. The second class is the *VP_ Visitor*, which is the superclass for all the visitor subclasses to inherit functionality from.

```
[OpenCLS]
class VP_Element
{
    public virtual void Accept(VP_Visitor InVisitor) { }
}

[OpenCLS]
class VP_Visitor
{
    public virtual void VisitConcreteElementA(VP_ConcreteElementA ↩
       InElement) { }
    public virtual void VisitConcreteElementB(VP_ConcreteElementB ↩
       InElement) { }

    public void Visit(VP_Element InElement)
    {
        InElement.Accept(this);
    }
}
```

Listing 22.4: The *VP_Element* and *VP_Visitor* classes

In listing 22.5 we have the implementation of the concrete element classes which inherits from the *VP_Element* class.

```
[OpenCLS]
class VP_ConcreteElementA : VP_Element
{
    public override void Accept(VP_Visitor InVisitor)
    {
        InVisitor.VisitConcreteElementA(this);
    }
}

// VP_ConcreteElementB has a similar implementation as above
```

Listing 22.5: The *VP_ConcreteElementA* and *VP_ConcreteElementB* classes that inherits from *VP_Element*

In listing 22.6 we have the implementation of the concrete visitor, which inherits from the *VP_Visitor* class. The class in the listing includes all the logic of the visitor.

```
class VP_ConcreteVisitor1 : VP_Visitor
{
    public int MyValue;

    public VP_ConcreteVisitor1()
    {
        MyValue = 0;
    }

    public override void VisitConcreteElementA(VP_ConcreteElementA ↩
       InElement)
    {
        MyValue += 5;
    }
    public override void VisitConcreteElementB(VP_ConcreteElementB ↩
       InElement)
    {
        MyValue *= 2;
    }
}
```

Listing 22.6: The *VP_ConcreteVisitor1* class that inherits from *VP_Visitor*

Listing 22.7 presents the visitor pattern execution class with the *Main* method which instantiates the concrete visitor and elements. The *Visit* method of the visitor is accessed with the elements as parameter.

```
class VP_Execution
{
    [OpenCLS]
    public int Main()
    {
        VP_ConcreteVisitor1 MyVisitor = new VP_ConcreteVisitor1();
        VP_ConcreteElementA ElementA = new VP_ConcreteElementA();
        VP_ConcreteElementA ElementB = new VP_ConcreteElementA();

        MyVisitor.Visit(ElementA);
        MyVisitor.Visit(ElementB);

        return MyVisitor.MyValue;
    }
}
```

Listing 22.7: The *VP_Execution* class with the *Main* method

Exppected Results

For this the expected result is *10*, as the visitor patter should take the visitors default value of *0*, plus *5* (from visiting *ElementA*), times 2 (from visiting *ElementB*).

22.4 Result

In this section we show the result of the design pattern test, common for all these tests is that we compare the results from executing on the CPU with the results of OpenCLS 2. If the results from executing the same design pattern on both the CPU and with OpenCLS 2 yields the same, the tests have passed. This means that OpenCLS 2 is able to execute the design pattern on the GPU and the required features of the OOP pillars are supported.

In table 22.1 below, we have the result from the conducted test:

Test Part	Expected Result (CPU)	OpenCLS 2 Result	Outcome
Proxy Pattern	25	25	Passed
Visitor Pattern	10	10	Passed

Table 22.1: Unit test results for the compiler

22.5 Summary

From the conducted design pattern test we conclude that OpenCLS 2 works as intended, and supports the OOP pillars (Inheritance, Encapsulation and Polymorphism).

22.5.1 Proxy Pattern

The conducted design pattern test shows that the OpenCLS 2 compiler supports inheritance from the OOP paradigm. This means that any pattern or algorithm that uses the inheritance pillar can be used with the OpenCLS 2 compiler.

22.5.2 Visitor Pattern

from the design pattern test that we have conducted on the OpenCLS 2 compiler, shows that it supports the inheritance and encapsulation pillars. As a result OpenCLS 2 supports any features or algorithms that uses either inheritance or encapsulation, or a combination of both.

23 Performance and Related Work

In this section, we will measure the performance of OpenCLS 2, and compare it to the performance of CPU execution, as well as the related work *Inheritance and Polymorphism in C*. This related works has been selected as it support OOP. The performance tests will focus on the new OOP features, as non-OOP features have already been performance tested (chapter 1). It was not possible to execute performance tests on the related work *Firepile*, as it is incompatible with current versions of Scala (2.11.8), and OpenCLS 2 does not support the performance tests from the Firepile report [21].

23.1 OpenCLS 2 GPU compared with CPU

In this section we will compare GPU and CPU performance of the two OOP design patterns described in chapter 10, as well as examine the results. We are using the official OpenCL guide *Comparing OpenCL Kernel Performance with Performance of Native Code* [43] to achieve credible OpenCL and CPU execution times. We will also be tracking host-device communication (HDC) separately, as recommended by the OpenCL performance profiling guide, using the OpenCL C function *clGetEventProfilingInfo* on the GPU.

Figure 23.1 shows that when executing the design patterns from chapter 22, the CPU has a performance advantage of factor 1600 compared to the GPU with HDC, and a factor 19 advantage compared to the GPU without HDC. We attribute this performance factor to the following elements listed by the OpenCL performance guide:

The design patterns are sequential: The design patterns can not be split up into multiple kernels and executed concurrently to achieve better performance on the GPU.

The CPU compiler optimizes CPU code before execution: During compile-time, CPU compilers remove unreachable code, calculate constants, and re-arranges code segments to optimize execution [44]. The OpenCL compiler supports optional arithmetic optimizations through compiler flags, that trade accuracy for performance [43]. However, OpenCL compiler optimizations are applied during compilation which occur during run-time, and as a result will introduce additional delay to OpenCL execution. This difference can be reduced by precomputing the OpenCL binaries.

Host-device communication: OpenCLS 2 transfers the OpenCL C code as well as the input data to OpenCL. OpenCL also provides output data that OpenCLS 2 waits to receive. This communication is limited by bandwidth and reduces performance. We can see from figure 23.1 that HDC reduces performance significantly.

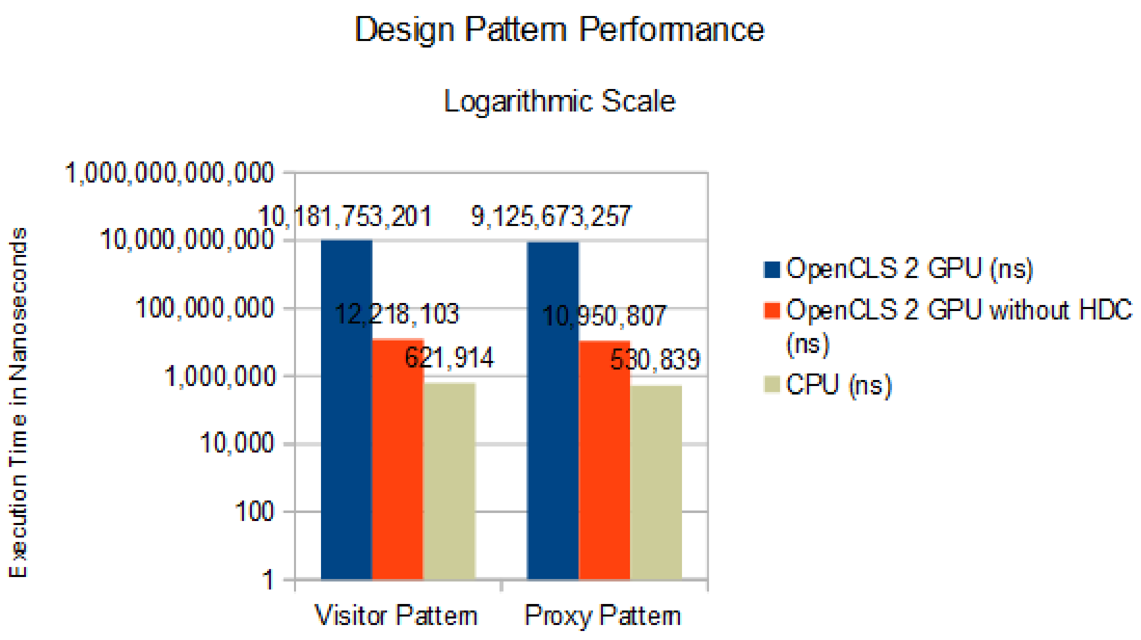


Figure 23.1: Performance comparison (in nanoseconds) between CPU and GPU execution of the non-parallel visitor and proxy design pattern. The code for the design patterns can be seen in chapter 22.

OpenCLS 2 is not memory efficient: OpenCLS 2 only utilizes global memory, which is limited by the system host, and achieves less performance than shared, local, or private memory.

CLOO wrapper: The CLOO C# OpenCL wrapper introduces additional indirections to communication between C# and OpenCL.

We can improve performance of the design patterns, by modifying the design patterns to support parallelism. This chapter is focusing on the visitor pattern as an example, as it has potential for parallelism through each visit execution. The visitor design pattern is modified to have each visit execute in a kernel, and the number of visits are increased from two to a high number that will showcase the effect of GPU parallelism. In this case, we are performing 10, 100, 1,000, 10,000, 100,000 and 1 million visit executions. In addition, the OpenCL compilation process can be optimized by utilizing the *-cl-fast-relaxed-math* and *-cl-mad-enable* OpenCL optimization flags, to trade accuracy for performance. We can also precompute the OpenCL binaries using the OpenCL C function *clCreateProgramFromBinary*, instead of using the function *clCreateProgramWithSource* which re-compiles the OpenCL C code every call. The code for this change can be seen in listing 23.1. This change will reduce delays caused by the OpenCL compilation process.

```
/* PREVIOUS CODE */
ComputeProgram program = new ComputeProgram(context, OpenCLSInput.↩
    SourceCode);

/* NEW CODE */
if (!BinariesPrecomputed())
    StorePrecomputedBinaries(new ComputeProgram(context, ↩
        OpenCLSInput.SourceCode));

```

```
ComputeProgram program = new ComputeProgram(context, ↩
    GetPrecomputedBinaries(), OpenCLS.Device);
```

Listing 23.1: Precomputed OpenCL binaries will reduce compilation times on multiple executions of the same code. CLOO's ComputeProgram executes *clCreateProgramWithSource* if given a string and *clCreateProgramFromBinary* if given precomputed binaries. This change is in *OpenCLSLauncher.cs* line 54.

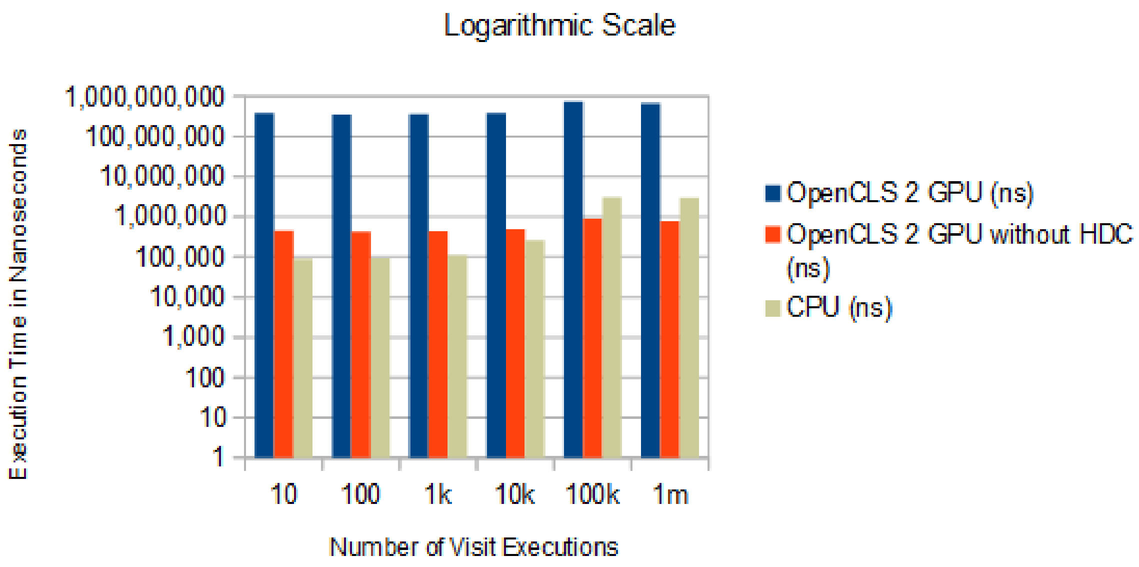


Figure 23.2: Performance comparison (in nanoseconds) between CPU and GPU execution of the optimized visitor design pattern. The code for the optimized visitor design pattern can be seen in appendix A.51.

The execution time of the optimized visitor pattern can be seen in figure 23.2. OpenCLS 2 achieves overall better performance on the optimized visitor pattern, than on the non-optimized visitor pattern. OpenCLS 2 without HDC performs better than the CPU by a factor of 3.5 for 100 thousand executions and above, which is expected as the GPU is better at performing many executions in parallel than the CPU. In addition, it can be seen that 99% of OpenCLS 2 execution time is spent on HDC, which indicates the way that either OpenCLS 2 or CLOO packs and sends data is inefficient. This is further confirmed by figure 23.3, which shows a comparison of HDC execution time between OpenCLS 2 and C++. It can be seen that OpenCLS 2 spends much more of its execution time in HDC than C++ OpenCL.

23.1.1 Summary

OpenCLS 2 without HDC achieves better performance than the CPU by a factor of 3.5 for large number of concurrent, optimized executions. For lower numbers of concurrent executions, or with HDC, OpenCLS 2 achieves worse performance than the CPU. OpenCLS 2 also achieves worse performance than the CPU for executions that are not optimized for parallelism. OpenCLS 2 could improve performance by applying the following improvements: reduced HDC execution time, improved memory efficiency, improved OpenCL code optimizations and change or improvements in the CLOO wrapper.

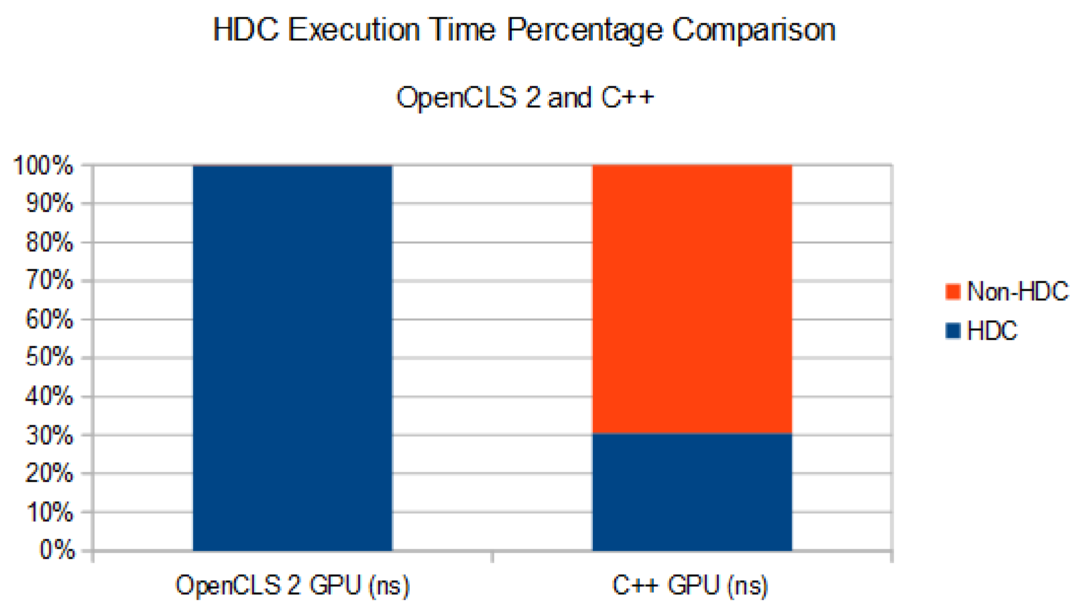


Figure 23.3: HDC execution time comparison (in percent) between OpenCLS 2 GPU execution and C++ GPU execution of the sample code *OpenCL Device Query* provided by the NVIDIA OpenCL library [45].

23.2 Inheritance and Polymorphism in C

The article '*Inheritance and Polymorphism in C*' details how virtual functions can be implemented using function pointers. This was not possible to implement for OpenCLS 2, as OpenCL C does not support function pointers. Instead, OpenCLS 2 uses class IDs and a resolver method.

Figure 23.4 shows the execution time of a virtual function in both implementations. It can be seen that both implementation achieve an execution time of 3ns, regardless of how many classes are in the class hierarchy. From this we can conclude that the indirection that both methods add to the execution flow is very minor (3ns for 10 billion virtual function calls). There is no measurable difference in performance between the two implementations. The execution code can be seen in appendix A.51.

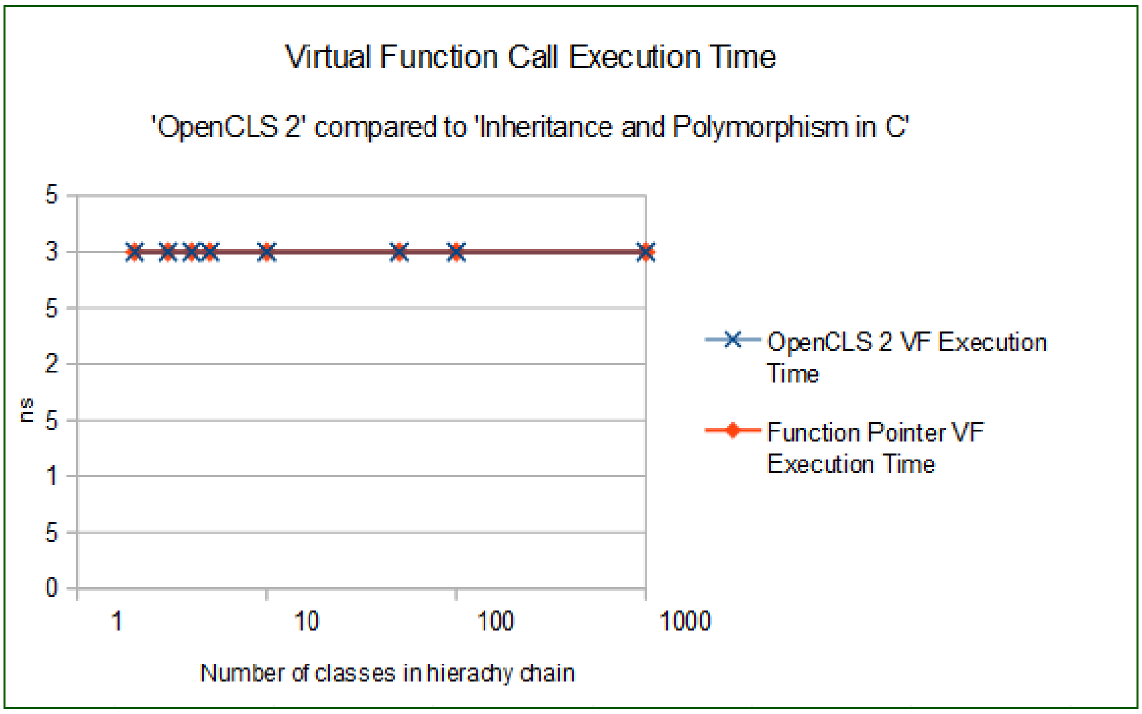


Figure 23.4: Performance comparison between virtual function calls in OpenCLS 2 and '*Inheritance and Polymorphism in C*' for 10 billion calls. OpenCL startup has been disregarded for this test.

24 Summary

We have performed unit tests each feature in OpenCLS 2, as well as each of the OOP pillars. In addition, we have performed design pattern and performance tests on OpenCLS 2 and the CPU, and compared the results. OpenCLS 2 passes all unit tests, and as a result OpenCLS 2 executes each of the OOP pillars as expected. OpenCLS 2 achieves better performance than the CPU by a factor of 3.5 for large number of executions and without host-device communication taken into consideration. For smaller number of executions or with host-device communication taken into account, OpenCLS 2 achieves worse performance than the CPU.

Part VI

Evaluation

In this part we evaluate the OpenCLS 2 project. We reflect on the status of the project, how the development method worked and finally we reflect on the related work we described in chapter 7. We conclude this project by answering our problem statement and evaluate if we have implemented the requirements specification. Additionally we discuss future work for the project, with suggestions to what we could implement if given the opportunity.

25 Reflection

In this chapter we reflect upon the choices made in part II and III, as well as give a summary of the status of OpenCLS 2.

25.1 Status

This section provides an overview of the status of OpenCLS 2 at the conclusion of this project. The section details which features are implemented, and which are excluded.

Inheritance: Class-to-class inheritance is supported, and it is possible to create chaining and branching hierarchies. Interface-to-class inheritance is also supported, and classes are able to access fields, methods and properties that are declared in superclasses.

Encapsulation: With the inclusion of interfaces and abstract classes, OpenCLS 2 supports all encapsulation features from C#.

Polymorphism: Virtual methods are supported, as well as overriding these methods in subclasses. Abstract methods and independent overriding methods are not supported. Additionally, subtyping is supported, but object casting is not.

The choice of which overriding feature we implemented would not have changed our choice of design patterns. The design patterns proposed in chapter 10 could be rewritten to utilize abstract overriding or independent overriding instead. Additionally, object casting was not implemented, as it was part of priority three.

At the conclusion of this project, OpenCLS 2 supports all features in the first and the second priority of the requirement specification. As stated in chapter 9, if the first and second priority are implemented, then OpenCLS 2 supports all three pillars of OOP: Inheritance, encapsulation and polymorphism. However, priority three is not supported, which means that OpenCLS 2 does not support the full set of polymorphism features in C#.

OpenCLS 2 is able to implement and execute the proxy and visitor design patterns. However, due to the lack of support for abstract methods, the observer design pattern proposed in section 10.3 cannot be implemented.

25.2 Development Method

For this project, we decide to overcome some of the obstacles that occurred during the OpenCLS 1 project, such as unexpected pitfalls. Unexpected pitfalls were a significant problem in the OpenCLS 1 project, as we were unprepared and unable to adjust to situations where the development did not go as planned.

To avoid pitfalls for this project, we decided to utilize a modified agile development method. We took inspiration in SCRUM features such as sprints, backlogs and weekly meetings. The weekly meetings were held with our supervisor, where we also received feedback on the project. The feedback from the supervisor made us realize where the focus should be put until the next meeting and it made us aware of when to pivot the project. Sprints and the project backlog provided all team members with an overview of the project on a daily basis, as well as giving the project a structure, which improved our ability to reason about deadlines and task durations.

We also took inspiration from extreme programming (XP) and included pair programming in our development method, to improve code quality. Pair programming provided all team members with an overview of the compiler structure, and reduced the amount of refactoring we had to perform.

Additionally we decided to use Team Foundation Server (TFS) instead of SVN. TFS includes the same features as SVN, as well as provide a backlog and automated builds for this project. The automated builds ran three times a day, and helped us identify code issues. We also created a PHP hook that extended the functionality of the TFS automated build results, by providing us with the latest check-in history.

TFS' backlog was limited, as it was not possible to delete tasks, and it did not support subtasks. This caused us to switch to using *Trello* [46] (the backlog system used in our previous project) as our sprint backlog after the implementation of priority one.

Lastly, we divided our project goals into three priorities, which helped focus our efforts.

In retrospect, our development method has helped in prioritizing our efforts, and has provided us with the tools necessary to work on an unpredictable and unexplored topic.

25.3 Related Work

In this section we reflect on the related work we described in chapter 7, and examine their differences compared to the result of OpenCLS 2.

Firepile: Firepile is a library for GPU programming in Scala that supports inheritance and polymorphism. The OpenCLS 2 implementation for inheritance and polymorphism is similar to Firepile's implementation (super / sub member variables, resolver method), however Firepile supports dynamic memory allocation, while OpenCLS 2 does not. Dynamic memory allocation allows Firepile to gradually allocate memory during algorithm execution, while OpenCLS 2 has to implement algorithms with a predetermined memory allocation. As a result, OpenCLS 2 is at risk of being inefficient (allocating more memory than is needed) or running out of memory (allocating too little memory) depending on the algorithm.

Inheritance and Polymorphism in C: This related work implements virtual functions through function pointers, while OpenCLS 2 implements virtual functions using a branching switch. This was necessary, as OpenCL C does not support function pointers. In this case, function pointers only add a single indirection, while a branching switch contains multiple cases, all of which add another set of instructions. However, our performance tests from chapter 23 show that the two implementations achieve identical performance (3ns).

Applying Object Oriented Design Pattern to CUDA: This related work presents programming advantages of OOP through the use of design patterns. For OpenCLS 2 we implemented the visitor and proxy pattern, both of which helped highlight bugs.

Cudafy: Cudafy is the related work that has the most in common with the OpenCLS projects. Both Cudafy and OpenCLS are C# compilers that translate C# code to OpenCL C and they both achieve similar performance (chapter 7). However, Cudafy does not support any of the OOP features introduced in OpenCLS 2 (chapter 9), and as a result the design patterns introduced in this project (chapter 10) cannot be implemented in Cudafy and we can only compare performance between the features from OpenCLS 1 and Cudafy, the results of which remain the same as in chapter 7.

0

26 Conclusion

General-purpose computing on graphics hardware (GPGPU) is the use of the GPU to execute tasks that are usually programmed for the CPU, to improve performance. This is advantageous because the GPU can provide a performance improvement of factor 300 compared to the CPU [1]. GPU programming is available through the C++ libraries provided by OpenCL and CUDA. These libraries are also available in other languages through OpenCL and CUDA wrappers. OpenCL and CUDA are programmed through a C-like language, which requires programmers, who wish to utilize GPU performance, to maintain a codebase for their CPU code and a codebase for their GPU code.

As part of our previous semester project, we developed OpenCLS 1, a project which aims to reduce split codebases for GPGPU programming by allowing CPU code (C#) to be compiled and executed on the GPU (OpenCL C) [1]. The implementation of OpenCLS 1 achieves performance similar to related work, *Cudafy* [4]. OpenCLS 1 is created using the Roslyn platform and CLOO OpenCL wrapper, and it supports the following C# features: classes (only as encapsulation), methods, primitive data types, recursion and C# iterators. The set of C# features that the OpenCLS projects supports is important, as more features extends the possibilities of sharing the codebase between the CPU and the GPU, as the user does not need to take the GPU language limitations into account. C# is an object-oriented programming (OOP) language [47], however OOP designs are unavailable in OpenCLS 1, and as a result, object-oriented designs can not be utilized in GPGPU programming.

This project introduces OpenCLS 2, an extension of OpenCLS 1 that implements object-oriented language features, by supporting the three OOP pillars: Inheritance, encapsulation and polymorphism. We have examined existing GPGPU solutions, and taken inspiration in the concepts that *Firepile* [21] and *Inheritance and Polymorphism in C* [23] introduce, for supporting OOP design features on the GPU. The following describes the status of OpenCLS 2, as well as the solutions for each OOP pillar.

Inheritance: We have implemented support for the inheritance OOP pillar in OpenCLS 2 by supporting class, interface and abstract class inheritance. Inheritance is supported by introducing the *super* member variable to subclasses. The *super* member variable is an instance of the superclass, which makes it possible for subclasses to access superclass variables through the *super* member variable.

Encapsulation: We have implemented support for the encapsulation OOP pillar in OpenCLS 2 by supporting classes, abstract classes, interfaces and access specifiers. Classes and abstract classes are translated into structs in OpenCL C and interfaces are translated into a single *Interface* struct. As access specifiers are a compile-time feature, OpenCLS 2 removes these during the compilation process.

Polymorphism: We have implemented support for the polymorphism OOP pillar in OpenCLS 2 by supporting virtual overriding methods and subtyping. Virtual overriding is supported by introducing a *virtual resolver method* that determines which overridden method to call depending on the caller object's *ClassID*. Subtyping is supported statically, since the source and target types are known during compile-time because of the Roslyn platform. The subtype operation is performed by iterating through the chain of *super* member variables until the target type is reached.

We have prioritised the three OOP pillars by evaluating which pillars have dependencies. The encapsulation and inheritance pillars are not dependent on any other pillars, and as a result they are the first priority for this project. The polymorphism pillar requires the inheritance pillar to be supported, which makes it the second priority. This pillar has features that achieve similar results (virtual methods, abstract methods and independent overriding), and as a result we split the features from this pillar into two priorities.

This OpenCLS 2 has been developed using the agile software methodology. Our motivation for using the agile software methodology and prioritizing the three OOP pillars, was to prepare for potential pitfalls during development, as unexpected pitfalls were an issue during the development of OpenCLS 1. For example, we ran into undocumented limitations with the 3rd party OpenCL wrapper, *CLOO* [1]. We have utilized pair programming, sprints, sprint backlogs and weekly meeting from SCRUM and XP. The choice of software development method has helped to prepare the project for potential pitfalls.

For this project we have taken inspiration in *Applying Object Oriented Design Patterns to CUDA* [24], which states that implementing design patterns can be an option to test the programming advantages of the OOP extension in OpenCLS 2. We have implemented two OOP design patterns that utilize inheritance, encapsulation and polymorphism: The proxy and visitor patterns. We have tested the OpenCLS 2 functionality using unit tests and design pattern tests. A total of 308 unit tests have been implemented, and we have created 74 unit tests and two OOP design tests that focus exclusively on testing OOP features. All OpenCLS 2 tests are successful, and OpenCLS 2 translates C# code to OpenCL C as expected (part V).

We have tested OpenCLS 2's performance against related work. We have selected *Inheritance and Polymorphism in C* and *Firepile* to test against, as these have OOP features. Our tests show that OpenCLS 2 achieves performance similar to related work (3ns, as described in section 23.2).

We have also performed design pattern performance tests for OpenCLS 2 compared against the CPU. These tests have been executed using the test structure outlined in the official OpenCL C guide *Comparing OpenCL Kernel Performance with Performance of Native Code* [43] to achieve credible OpenCL and CPU execution times. Host-device communication is disregarded from CPU testing to achieve a fair comparison, as recommended by the OpenCL C guide. CPU comparisons are performed on OOP design patterns and achieve better performance than the CPU by a factor of 3.5 for large computations without host-device communication (chapter 23).

In summary, OpenCLS 2 supports object-oriented programming, and can compile C# code to OpenCL C, as well as execute the code on the GPU. OpenCLS 2 achieves performance similar to related work, and achieves better performance than the CPU. OpenCLS 2 fulfills the requirements set for OpenCLS 1, by supporting the subset of features supported by OpenCLS

1. Based on this, we conclude that this project fulfills the requirements set in the requirement specification in chapter 9 and provide answers to the questions asked in the problem statement in chapter 8.

27 Future Work

This chapter will describe new features or modifications for OpenCLS 2 that could be implemented in an extended project.

27.1 Priority three from OpenCLS 2

Not all requirements for OpenCLS 2 were implemented. The following features are potential candidates for an extended project:

- Abstract Methods
- Independent Overriding
- Casting

With those features added, OpenCLS 2 would support all of C#'s overriding methods. Furthermore, if casting was added, OpenCLS 2 would support both implicit (subtyping) and explicit (casting) object type conversions.

27.2 Dynamic Memory Allocation

OpenCLS 2 does not support memory allocation, as this is not a feature in OpenCL C, and it was not the focus of this project. The related project Firepile circumvented this limitation by implementing a keyword which would halt GPU execution, request memory from the CPU, and then resume execution once the memory was made available. The lack of dynamic memory allocation has resulted in limitations in OpenCLS 2. For instance, all objects are stored as value-based structs. With the addition of dynamic memory allocation, these could be changed to a reference pointer, which would circumvent the scope limitations for OpenCLS 2's objects (described in chapter 12).

27.3 Static Memory Analysis

Static memory analysis performs compile-time estimates of how much memory needs to be available for code executions. Static memory analysis is a potential solution for the same problems explained in dynamic memory allocation; users currently have to estimate the amount of memory needed for an algorithm, and an incorrect estimate will reduce performance.

27.4 Automatic Parallelization

In OpenCLS 2, C# code segment are marked with the OpenCLS tag and then executed in parallel on multiple kernels. However, it is the user's responsibility to ensure that the C# code can make efficient use of parallelism. For instance, executing a '*for*' loop with no parallelism in mind will not be faster on the GPU compared to the CPU. The loop has to be modified so that each iteration of the loop can be executed on a GPU kernel, so that the GPU can execute all kernels in parallel. For an extended OpenCLS project, it would be interesting to introduce automatic parallelization, which would restructure input C# code to make it possible to run in parallel on the GPU. This extended project could take inspiration in *GPSME* [48], an automatic parallelization tool for C and C++.

27.5 Host-Device Communication

In chapter 23 it was shown that OpenCLS 2's performance is reduced by host-device communication (HDC), and that OpenCLS 2 has worse HDC performance than C++ OpenCL. An extended project could aim to reduce HDC execution time to improve the performance in OpenCLS 2.

27.6 Additional Design Patterns

In chapter 10 we described three OOP design patterns that would be used for testing the OOP features of OpenCLS 2. However there are many additional OOP design patterns that could be implemented, such as the factory, object pool and singleton pattern. These design patterns would have been interesting because they require non-OOP features that are not supported by OpenCLS, such as global and static variables, hash maps and dynamic arrays.

Bibliography

[1] S. H. C. Mads Holm, Minh Hieu Nguyen and J. Kaerlev, "Cpu to gpu compiler, with focus on c# to opencl," tech. rep., Aalborg University, 2015.

[2] D. Kean, ".net compiler platform ("roslyn") overview." https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview#introduction. Accessed on 27-10-15.

[3] P. Scobey, "The three pillars of object-oriented programming." http://cs.smu.ca/~porter/csc/common_341_342/notes/oop_3pillars.html. Accessed on 22-02-16.

[4] NickKopp and swaan79, "Cudafy.net." https://cudafy.codeplex.com/. Accessed on 10-05-16.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development." http://www.agilemanifesto.org/. Accessed on 16-02-16.

[6] S. Saravathy, "Effectuation: Elements of entrepreneurial expertise," 2008.

[7] S. W. Ambler, "The product owner role: A stakeholder proxy for agile teams." http://agilemodeling.com/essays/productOwner.htm. Accessed on 21-4-16.

[8] M. James, "Scrum methodology." http://scrummethodology.com/. Accessed on 16-02-16.

[9] D. Wells, "Extreme programming: A gentle introduction." http://www.extremeprogramming.org/. Accessed on 16-02-16.

[10] Microsoft, "Syntax visualizer overview." https://roslyn.codeplex.com/wikipage?title=Syntax%20Visualizer. Accessed on 11-04-16.

[11] Microsoft, "Team foundation server." https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx. Accessed on 08-04-16.

[12] P. Yogi, "Pillars of object oriented programming." http://www.placementyogi.com/tutorials/java/introduction-to-java/pillars-of-oops. Accessed on 24-02-16.

[13] Microsoft, “Inheritance (c# programming guide).” https://msdn.microsoft.com/en-us/library/ms173149.aspx. Accessed on 25-02-16.

[14] T. Point, “C# - encapsulation.” http://www.tutorialspoint.com/csharp/csharp_encapsulation.htm. Accessed on 25-02-16.

[15] T. Point, “C# - properties.” http://www.tutorialspoint.com/csharp/csharp_properties.htm. Accessed on 28-02-16.

[16] Jayababu, “All about abstract classes..” http://www.codeproject.com/Articles/6118/All-about-abstract-classes. Accessed on 19-05-16.

[17] Microsoft, “Polymorphism (c# programming guide).” https://msdn.microsoft.com/en-us/library/ms173152.aspx. Accessed on 25-02-16.

[18] Codeproject, “Introduction to object oriented programming concepts (oop) and more: What is polymorphism?.” http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep#Polymorphism. Accessed on 06-03-16.

[19] Microsoft, “Virtual function tables.” https://msdn.microsoft.com/en-us/library/windows/desktop/dd757710%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396. Accessed on 28-02-16.

[20] Microsoft, “Abstract and sealed classes and class members (c# programming guide).” https://msdn.microsoft.com/en-us/library/ms173150.aspx. Accessed on 10-04-16.

[21] K. D. Nathaniel Nystrom, Derek White, “Firepile: Run-time compilation for gpus in scala.” http://www.inf.usi.ch/faculty/nystrom/papers/firepile.pdf. Accessed on 03-03-16.

[22] K. D. Nathaniel Nystrom, Derek White, “Firepile: Gpu programming in scala.” http://www.program-transformation.org/pub/GPCE11/ConferenceProgram/slides-gpce11-nystrom.pdf. Accessed on 03-06-16.

[23] P. Gotarne and Pankajdoke, “Inheritance and polymorphism in c.” http://www.codeproject.com/Articles/108830/Inheritance-and-Polymorphism-in-C. Accessed on 18-4-16.

[24] A. O. O. D. P. to CUDA, “Inheritance and polymorphism in c.” http://www.sersc.org/journals/IJSEIA/vol6_no2_2012/5.pdf. Accessed on 18-4-16.

[25] Microsoft, “Mono.cecil on github.” https://github.com/jbevain/cecil. Accessed on 19-05-16.

[26] Brahma, “Cloo on sourceforge.” http://sourceforge.net/projects/cloo/. Accessed on 07-01-16.

[27] AMD, “Aparapi.” http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/. Accessed on 17-05-16.

[28] N. M. S. Gupta, K.G.; Agrawal, “Performance analysis between aparapi (a parallel api) and java by implementing sobel edge detection algorithm,” in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, PARCOMPTECH 2013, 2013.

[29] G. Cocco, "Fscl: Homogeneous programming, scheduling and execution on heterogeneous platforms," tech. rep., University of Pisa, 2014.

[30] S. H. C. Mads Holm, Minh Hieu Nguyen and J. Kaerlev, "Oop on the gpu: An opencls extension," tech. rep., Aalborg University, 2016.

[31] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[32] T. Point, "Design patterns - proxy pattern." `http://www.tutorialspoint.com/design_pattern/proxy_pattern.htm`. Accessed on 28-02-16.

[33] D. E. Parker, "Compiler techniques and principles." `http://www.eng.utah.edu/~cs5470/schedule/Lec7.pdf`. Accessed on 10-03-16.

[34] dofactory, "Visitor .net design pattern." `http://www.dofactory.com/net/visitor-design-pattern`. Accessed on 03-03-16.

[35] Microsoft, "Exploring the observer design pattern." `https://msdn.microsoft.com/en-us/library/ee817669.aspx`. Accessed on 02-03-16.

[36] M. D. Network, "Casting and type conversions (c#)." `https://msdn.microsoft.com/en-us/library/ms173105.aspx?f=255&MSPPError=-2147217396`. Accessed on 09-06-16.

[37] CareerRide.com, "Difference between dynamic and static casting." `http://www.careerride.com/C++-dynamic-vs-static-casting.aspx`. Accessed on 30-05-16.

[38] Microsoft, "Interfaces (c# programming guide)." `https://msdn.microsoft.com/en-us/library/ms173156.aspx`. Accessed on 25-02-16.

[39] C. Consulting, "Ieee 829 documentation." `http://www.coleyconsulting.co.uk/IEEE829.htm`. Accessed on 21-03-16.

[40] M. Rouse, "What is white box (white box testing)." `http://searchsoftwarequality.techtarget.com/definition/white-box`. Accessed on 05-04-16.

[41] V. Wadje, "Types of inheritance in c#." `http://www.c-sharpcorner.com/UploadFile/0c1bb2/types-of-inheritance-in-C-Sharp/`. Accessed on 05-04-16.

[42] G. Dogi, "Inheritance in c# using an example." `http://www.onlinebuff.com/article_oops-principle-inheritance-in-c-with-an-example_16.html`. Accessed on 05-04-16.

[43] K. Group, "clcompileprogram." `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCompileProgram.html`. Accessed on 08-06-16.

[44] TutorialPoint, "Compiler design - phases of compiler." `http://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm`. Accessed on 08-06-16.

[45] NVIDIA, "Nvidia opencl sdk code samples." `https://developer.nvidia.com/opencl`. Accessed on 09-06-16.

[46] trello.com, "Trello." `https://trello.com/`. Accessed on 01-06-16.

[47] Microsoft, "Introduction to the c# language and the .net framework." `https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx`. Accessed on 09-06-16.

[48] GPSME, “Project details.” http://www.gp-sme.eu/p/project-details. Accessed on 07-01-16.

Part VII

Appendix

A Unit Test

A.1 Fork Inheritance

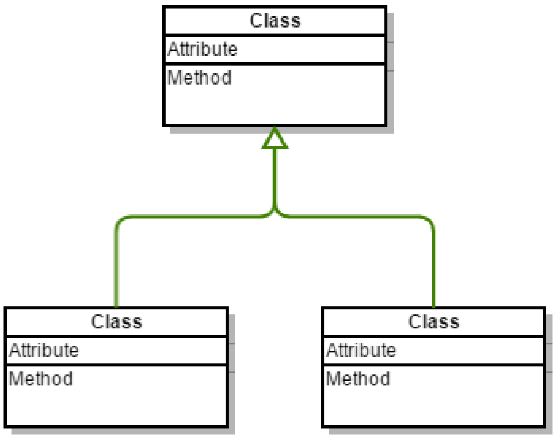


Figure A.1: Class diagram for fork inheritance test

A.1.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Dog : Animal
{
    public bool gender;
}
[OpenCLS]
public class Cat : Animal
{
    public int MeannessLevel;
    public void Speak()
    {
        Cat cat = new Cat();
        Dog dog = new Dog();
    }
}
```

Listing A.1: Input code for fork inheritance.

A.1.2 Expected Result

```
struct Animal;
struct Dog;
struct Cat;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    bool gender;
};
void Dog_Constructor(struct Dog* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
    int MeannessLevel;
};
void Cat_Constructor(struct Cat* _instance);
void Cat_Speak(struct Cat* _instance);
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
    _instance->gender = false;
}
void Cat_Constructor(struct Cat* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.Sub_Cat = _instance;
    _instance->MeannessLevel = 0;
}
```

```
void Cat_Speak(struct Cat* _instance)
{
    struct Cat cat;
    Cat_Constructor(&cat);
    struct Dog dog;
    Dog_Constructor(&dog);
}
```

Listing A.2: Expected output for fork inheritance.

A.1.3 Results

The test succeeded.

A.2 Chain Inheritance

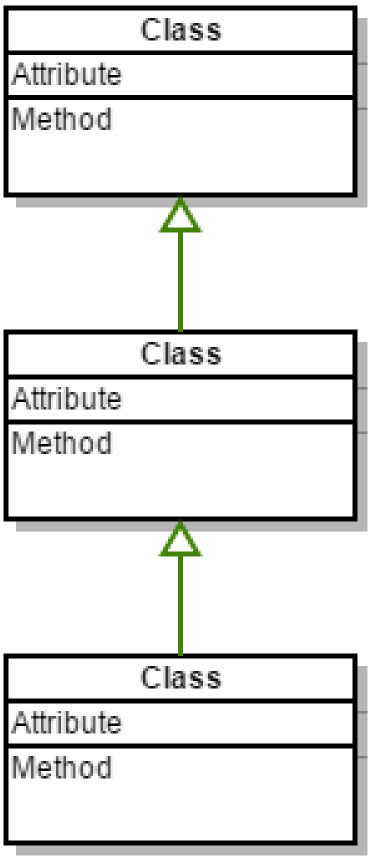


Figure A.2: Class diagram for chain inheritance test

A.2.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}

[OpenCLS]
public class Dog : Animal
{
    public float weight;
}

[OpenCLS]
public class Pug : Dog
{
    public float FrankLikeFactor;
}
```

Listing A.3: Input code for chain inheritance.

A.2.2 Expected Result

```
struct Animal;
struct Dog;
struct Pug;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Pug* Sub_Pug;
    float weight;
};
void Dog_Constructor(struct Dog* _instance);

struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
    float FrankLikeFactor;
};
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->super.Sub_Dog = _instance;
    _instance->weight = 0;
}
```

Listing A.4: Expected output for chain inheritance.

A.2.3 Result

The test succeeded.

A.3 Chain Fork Inheritance

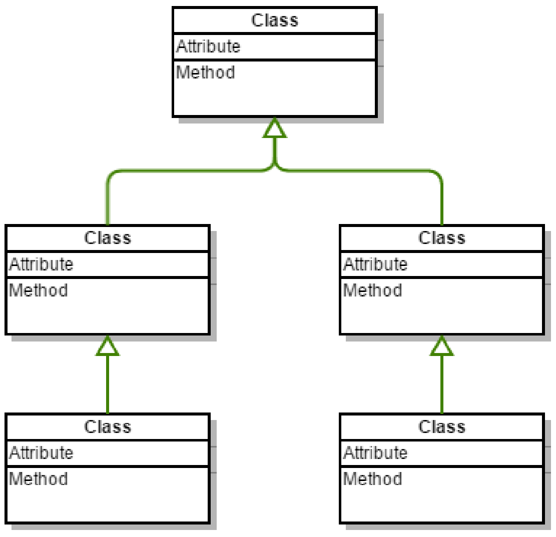


Figure A.3: Class diagram for chain fork inheritance test

A.3.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Dog : Animal
{
    public Dog()
    {
        Age = 2;
    }
}
[OpenCLS]
public class Pug : Dog
{
    public Pug()
    {
        Age = 4;
    }
}
[OpenCLS]
public class Cat : Animal
{
    public Cat()
    {
        Age = 5;
    }
}
[OpenCLS]
public class Siamese : Cat
{
    public Siamese()
    {
        Age = 10;
    }
}
class Test
{
    [OpenCLS]
    void Main()
    {
        Pug pug = new Pug();
        Siamese siamese = new Siamese();
    }
}
```

Listing A.5: Input code for chain fork inheritance.

A.3.2 Expedted Result

```
struct Animal;
struct Dog;
struct Pug;
struct Cat;
struct Siamese;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
};
void Pug_Constructor(struct Pug* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Siamese* Sub_Siamese;
};
void Cat_Constructor(struct Cat* _instance);
struct Siamese{
    int OpenCLS_ClassID;
    struct Cat super;
};
void Siamese_Constructor(struct Siamese* _instance);
__kernel void Main_Kernel();
void Test_Main();
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->super.Sub_Dog = _instance;
    _instance->super.Age = 2;
}
void Pug_Constructor(struct Pug* _instance)
{
    Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.super.OpenCLS_ClassID = 2;
    _instance->super.Sub_Pug = _instance;
    _instance->super.super.Age = 4;
}
void Cat_Constructor(struct Cat* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 3;
    _instance->super.OpenCLS_ClassID = 3;
    _instance->Sub_Siamese = NULL;
    _instance->super.Sub_Cat = _instance;
    _instance->super.Age = 5;
}
void Siamese_Constructor(struct Siamese* _instance)
{
    Cat_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 4;
    _instance->super.OpenCLS_ClassID = 4;
    _instance->super.super.OpenCLS_ClassID = 4;
    _instance->super.Sub_Siamese = _instance;
    _instance->super.super.Age = 10;
}
__kernel void Main_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Pug pug;
    Pug_Constructor(&pug);
    struct Siamese siamese;
    Siamese_Constructor(&siamese);
}
void Test_Main()
{
    struct Pug pug;
    Pug_Constructor(&pug);
    struct Siamese siamese;
    Siamese_Constructor(&siamese);
}
```

Listing A.6: Expected output code for chain fork inheritance.

A.3.3 Result

The test succeeded.

A.4 River Delta Inheritance

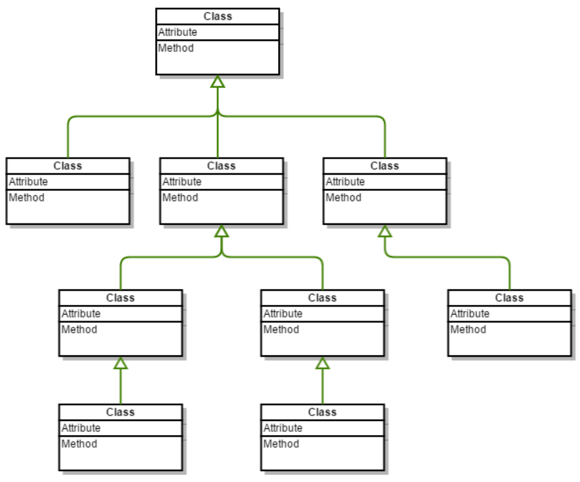


Figure A.4: Class diagram for river delta inheritance test

A.4.1 Input Code

```
[OpenCLS]
public class Pinky : Chihuahua
{

}
[OpenCLS]
public class Animal
{
    public int Age;
}
```

```
[OpenCLS]
public class Dog : Animal
{
    public float weight;
}
[OpenCLS]
public class Cat : Animal
{
    public float weight;
}
[OpenCLS]
public class Bear : Animal
{
    public float weight;
}
[OpenCLS]
public class Pug : Dog
{
    public float FrankLikeFactor;
}
[OpenCLS]
public class Chihuahua : Dog
{

}
[OpenCLS]
public class Black : Bear
{

}
[OpenCLS]
public class Frank : Pug
{

}
```

Listing A.7: Input code for river delta inheritance.

A.4.2 Expected Result

```
struct Animal;
struct Dog;
struct Chihuahua;
struct Pinky;
struct Cat;
struct Bear;
struct Pug;
struct Black;
struct Frank;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
    struct Bear* Sub_Bear;
  int Age;
```

```
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
  struct Animal super;
    struct Pug* Sub_Pug;
    struct Chihuahua* Sub_Chihuahua;
  float weight;
};
void Dog_Constructor(struct Dog* _instance);
struct Chihuahua{
    int OpenCLS_ClassID;
    struct Dog super;
    struct Pinky* Sub_Pinky;
};
void Chihuahua_Constructor(struct Chihuahua* _instance);
struct Pinky{
    int OpenCLS_ClassID;
    struct Chihuahua super;
};
struct Cat{
    int OpenCLS_ClassID;
  struct Animal super;
  float weight;
};
struct Bear{
    int OpenCLS_ClassID;
  struct Animal super;
    struct Black* Sub_Black;
  float weight;
};
void Bear_Constructor(struct Bear* _instance);
struct Pug{
    int OpenCLS_ClassID;
  struct Dog super;
    struct Frank* Sub_Frank;
  float FrankLikeFactor;
};
void Pug_Constructor(struct Pug* _instance);
struct Black{
    int OpenCLS_ClassID;
  struct Bear super;
};
struct Frank{
    int OpenCLS_ClassID;
  struct Pug super;
};
void Animal_Constructor(struct Animal* _instance)
{
  _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
    _instance->Sub_Bear = NULL;
  _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
  Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->Sub_Pug = NULL;
    _instance->Sub_Chihuahua = NULL;
    _instance->super.Sub_Dog = _instance;
  _instance->weight = 0;
}
void Bear_Constructor(struct Bear* _instance)
{
  Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 4;
    _instance->super.OpenCLS_ClassID = 4;
    _instance->Sub_Black = NULL;
    _instance->super.Sub_Bear = _instance;
  _instance->weight = 0;
}
void Pug_Constructor(struct Pug* _instance)
{
  Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 5;
    _instance->super.OpenCLS_ClassID = 5;
    _instance->super.super.OpenCLS_ClassID = 5;
    _instance->Sub_Frank = NULL;
    _instance->super.Sub_Pug = _instance;
  _instance->FrankLikeFactor = 0;
}
void Chihuahua_Constructor(struct Chihuahua* _instance)
{
  Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 6;
    _instance->super.OpenCLS_ClassID = 6;
    _instance->super.super.OpenCLS_ClassID = 6;
    _instance->Sub_Pinky = NULL;
    _instance->super.Sub_Chihuahua = _instance;
}
```

Listing A.8: Expected output for river delta inheritance.

A.4.3 Result

The test succeeded.

A.5 No Superclass Constructor

A.5.1 Input Code

```
[OpenCLS]
public class Animal
{
}
[OpenCLS]
public class Cat : Animal
{
    public int MyVar;
    public Cat()
    {
        MyVar = 5;
    }
}
class Test
{
    [OpenCLS]
    void Main()
    {
        Cat c = new Cat();
    }
}
```

Listing A.9: Input code for no superclass constructor.

A.5.2 Expedted Result

```
struct Animal;
struct Cat;
struct Animal{
    int OpenCLS_ClassID;
    struct Cat* Sub_Cat;
};
void Animal_Constructor(struct Animal* _instance);

// Full class declarations
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
    int MyVar;
};
void Cat_Constructor(struct Cat* _instance);

__kernel void Main_Kernel();
void Test_Main();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Cat = NULL;
}
void Cat_Constructor(struct Cat* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Cat = _instance;
    _instance->MyVar = 0;
    _instance->MyVar = 5;
}
__kernel void Main_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Cat c;
    Cat_Constructor(&c);
}
void Test_Main()
{
    struct Cat c;
    Cat_Constructor(&c);
}
```

Listing A.10: Expected output for no superclass constructor.

A.5.3 Results

The test succeeded.

A.6 No Subclass Constructor

A.6.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
    public Animal() { Age = 5; }
}

[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();
        if (dog.Age == 5)
        {
            Console.WriteLine(""Yahoo!"");
        }
    }
}
```

Listing A.11: Input code for no subclass constructor.

A.6.2 Expedted Result

```
struct Animal;
struct Dog;

```

```
// Full class declarations
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);

struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);

// Forward declaration of methods
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
    _instance->Age = 5;
}

void Dog_Constructor(struct Dog* _instance)
{
  Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}

__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
  Dog_Constructor(&dog);
    if(dog.super.Age == 5)
    {
        printf(""Yahoo!\n"");
    }
}
void Test_MyMethod()
{
    struct Dog dog;

    Dog_Constructor(&dog);
    if(dog.super.Age == 5)
    {
        printf(""Yahoo!\n"");
    }
}
```

Listing A.12: Expected output for no subclass constructor.

A.6.3 Results

The test succeeded.

A.7 Invoke Superclass Method in Subclass Constructor

A.7.1 Input Code

```
[OpenCLS]
public class Animal
{
    public void Bark()
    {
        Console.WriteLine(""Bark"");
    }
}

[OpenCLS]
public class Dog : Animal
{
    public Dog()
    {
        Bark();
    }
}
```

Listing A.13: Input code for invoking a superclass method in a subclass constructor.

A.7.2 Expedted Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_Bark(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Animal_Bark(struct Animal* _instance)
{
    printf(""Bark\n"");
}
```

```
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
    Animal_Bark(&_instance->super);
}
```

Listing A.14: Expected output for invoking a superclass method in a subclass constructor.

A.7.3 Results

The test succeeded.

A.8 Invoke Superclass Method in Subclass Method

A.8.1 Input Code

```
[OpenCLS]
public class Animal
{
    public void Speak()
    {
    }
}

[OpenCLS]
public class Dog : Animal
{
    public void Vuff()
    {
        Speak();
    }
}
```

Listing A.15: Input code for invoking a superclass method in a subclass method.

A.8.2 Expedted Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_Speak(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Vuff(struct Dog* _instance);
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Animal_Speak(struct Animal* _instance)
{
}
void Dog_Vuff(struct Dog* _instance)
{
    Animal_Speak(&_instance->super);
}
```

Listing A.16: Expected output for invoking a superclass method in a subclass method.

A.8.3 Results

The test succeeded.

A.9 Superclass Member Access

A.9.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Cat : Animal
{
    public Cat()
    {
        Age = 25;
    }
}
class Test
{
    [OpenCLS]
    void Main()
    {
        Cat c = new Cat();
        c.Age = 10;
    }
}
```

Listing A.17: Input code for superclass member access.

A.9.2 Expected Result

```
struct Animal;
struct Cat;
struct Animal{
    int OpenCLS_ClassID;
    struct Cat* Sub_Cat;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Cat_Constructor(struct Cat* _instance);

__kernel void Main_Kernel();
void Test_Main();
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Cat = NULL;
    _instance->Age = 0;
}
void Cat_Constructor(struct Cat* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Cat = _instance;
    _instance->super.Age = 25;
}
__kernel void Main_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Cat c;
    Cat_Constructor(&c);
    c.super.Age = 10;
}
void Test_Main()
{
    struct Cat c;
    Cat_Constructor(&c);
    c.super.Age = 10;
}
```

Listing A.18: Expected output for superclass member access.

A.9.3 Result

The test succeeded.

A.10 Superclass' superclass Member Access

A.10.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Cat : Animal
{
    public Cat()
    {
        Age = 5;
    }
}
[OpenCLS]
public class Siamese : Cat
{
    public Siamese()
    {
        Age = 10;
    }
}
class Test
{
    [OpenCLS]
    void Main()
    {
        Siamese c = new Siamese();
        c.Age = 16;
    }
}
```

Listing A.19: Input code for superclass' superclass Member Access.

A.10.2 Expected Result

```
struct Animal;
struct Cat;
struct Siamese;
struct Animal{
    int OpenCLS_ClassID;
    struct Cat* Sub_Cat;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Siamese* Sub_Siamese;
};
void Cat_Constructor(struct Cat* _instance);
struct Siamese{
    int OpenCLS_ClassID;
    struct Cat super;
};
```

```
void Siamese_Constructor(struct Siamese* _instance);
__kernel void Main_Kernel();
void Test_Main();
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Cat = NULL;
    _instance->Age = 0;
}
void Cat_Constructor(struct Cat* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Siamese = NULL;
    _instance->super.Sub_Cat = _instance;
    _instance->super.Age = 5;
}
void Siamese_Constructor(struct Siamese* _instance)
{
    Cat_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.super.OpenCLS_ClassID = 2;
    _instance->super.Sub_Siamese = _instance;
    _instance->super.super.Age = 10;
}
__kernel void Main_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Siamese c;
    Siamese_Constructor(&c);
    c.super.super.Age = 16;
}
void Test_Main()
{
    struct Siamese c;
    Siamese_Constructor(&c);
    c.super.super.Age = 16;
}
```

Listing A.20: Expected output for superclass' superclass Member Access.

A.10.3 Result

The test succeeded.

A.11 Method Member Access

```
[OpenCLS]
public class Animal
{
    public void Speak()
    {
    }
}

[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    public void Main()
    {
        Dog dog = new Dog();
        dog.Speak();
    }
}
```

Listing A.21: Input code for superclass method member access.

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_Speak(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void Main_Kernel();
void Test_Main();
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Animal_Speak(struct Animal* _instance)
{
}
void Dog_Constructor(struct Dog* _instance)
{
  Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void Main_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    Animal_Speak(&dog.super);
}
void Test_Main()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    Animal_Speak(&dog.super);
}
```

Listing A.22: Expected output code for superclass method member access.

A.12 Out of order inheritance declaration

A.12.1 Input Code

```
[OpenCLS]
public class Dog : Animal
{
    public float weight;
}
[OpenCLS]
public class Animal
{
    public int Age;
}
class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();
        dog.Age = 5;
    }
}
```

Listing A.23: Input code for out of order inheritance declaration.

A.12.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    float weight;
};
void Dog_Constructor(struct Dog* _instance);
```

```
__kernel void MyMethod_Kernel();
void Test_MyMethod();
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 0;
    _instance->super.OpenCLS_ClassID = 0;
    _instance->super.Sub_Dog = _instance;
    _instance->weight = 0;
}
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    dog.super.Age = 5;
}
void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    dog.super.Age = 5;
}
```

Listing A.24: Expected output for out of order inheritance declaration.

A.12.3 Result

The test succeeded.

A.13 Out of order class field

A.13.1 Input Code

```
[OpenCLS]
public class Animal
{
}

[OpenCLS]
public class Dog : Animal
{
    public Cat MyNemesis;
}

[OpenCLS]
public class Cat : Animal
{

}
```

Listing A.25: Input code for out of order class field.

A.13.2 Expected Result

```
struct Animal;
struct Cat;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
};
void Animal_Constructor(struct Animal* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Animal super;
};
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Cat MyNemesis;
};
void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
}
```

Listing A.26: Expected output for out of order class field.

A.13.3 Result

The test succeeded.

A.14 Single Interface Inheritance

A.14.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog : IAnimal
{
    void Speak()
    {

    }
}
```

Listing A.27: Input code for single interface inheritance.

A.14.2 Expected Output

```
struct Interface{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
struct Dog;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Dog_Speak(struct Dog* _instance);

void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Dog_Speak(_instance->Sub_Dog);
            break;
    }
}
void Dog_Speak(struct Dog* _instance)
{
}
```

Listing A.28: Expected output for single interface inheritance.

A.14.3 Result

Test succeeded.

A.15 Fork Interface Inheritance

A.15.1 Input

```
[OpenCLS]
```

```
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog : IAnimal
{
    void Speak()
    {

    }
}
[OpenCLS]
public class Cat : IAnimal
{
    void Speak()
    {

    }
}
```

Listing A.29: Input code for fork interface inheritance.

A.15.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
};
struct Dog;
struct Cat;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Dog_Speak(struct Dog* _instance);
struct Cat{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Cat_Speak(struct Cat* _instance);
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
                Dog_Speak(_instance->Sub_Dog);
                break;
            case 2:
                Cat_Speak(_instance->Sub_Cat);
                break;
        }
}
void Dog_Speak(struct Dog* _instance)
{
}
void Cat_Speak(struct Cat* _instance)
{
}
```

Listing A.30: Expected output for fork interface inheritance.

A.15.3 Result

Test succeeded.

A.16 Chain Interface Inheritance

A.16.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog : IAnimal
{
    void Speak()
    {

    }
}
[OpenCLS]
public class Pug : Dog
{
}
```

Listing A.31: Input code for chain interface only inheritance.

A.16.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
struct Dog;
struct Pug;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_Speak(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
};
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Dog_Speak(_instance->Sub_Dog);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Interface_Constructor(&_instance->interfaceSuper);
    _instance->interfaceSuper.OpenCLS_ClassID = 1;
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->interfaceSuper.Sub_Dog = _instance;
}
void Dog_Speak(struct Dog* _instance)
{
}
```

Listing A.32: Expected output for chain interface only inheritance.

A.17 Chain Interface Only Inheritance

A.17.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public interface IDog : IAnimal
{
    void Vuff();
}
[OpenCLS]
public class Pug : IDog
{
    void Speak()
    {
    }
    void Vuff()
    {
    }
}
```

Listing A.33: Input code for chain interface only inheritance.

A.17.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Pug* Sub_Pug;
};
struct Pug;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Pug_Speak(struct Pug* _instance);
void Pug_Vuff(struct Pug* _instance);
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Pug = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Pug_Speak(_instance->Sub_Pug);
            break;
    }
}
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Pug_Vuff(_instance->Sub_Pug);
            break;
    }
}
void Pug_Speak(struct Pug* _instance)
```

```
{
}
void Pug_Vuff(struct Pug* _instance)
{
}
```

Listing A.34: Expected output for chain interface only inheritance.

A.17.3 Result

Test succeeded.

A.18 Chain Fork Interface Inheritance

A.18.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog : IAnimal
{
    void Speak()
    {
    }
}
[OpenCLS]
public class Pug : Dog
{
}
[OpenCLS]
public class Cat : IAnimal
{
    void Speak()
    {
    }
}
[OpenCLS]
public class Siamese : Cat
{
}
```

Listing A.35: Input code for chain fork interface inheritance.

A.18.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    struct Cat* Sub_Cat;
};
struct Dog;
struct Pug;
struct Cat;
struct Siamese;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_Speak(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
};
struct Cat{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
    struct Siamese* Sub_Siamese;
};
void Cat_Constructor(struct Cat* _instance);
void Cat_Speak(struct Cat* _instance);
struct Siamese{
    int OpenCLS_ClassID;
    struct Cat super;
};
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Sub_Cat = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Dog_Speak(_instance->Sub_Dog);
            break;
        case 3:
            Cat_Speak(_instance->Sub_Cat);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Interface_Constructor(&_instance->interfaceSuper);
    _instance->interfaceSuper.OpenCLS_ClassID = 1;
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->interfaceSuper.Sub_Dog = _instance;
}
void Dog_Speak(struct Dog* _instance)
```

```
{
}
void Cat_Constructor(struct Cat* _instance)
{
    Interface_Constructor(&_instance->interfaceSuper);
    _instance->interfaceSuper.OpenCLS_ClassID = 3;
    _instance->OpenCLS_ClassID = 3;
    _instance->Sub_Siamese = NULL;
    _instance->interfaceSuper.Sub_Cat = _instance;
}
void Cat_Speak(struct Cat* _instance)
{
}
```

Listing A.36: Expected output for chain fork interface inheritance.

A.18.3 Result

Test succeeded.

A.19 Chain Reverse Fork Interface Inheritance

A.19.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public interface IDog
{
    void Vuff();
}
[OpenCLS]
public class Pug : IAnimal, IDog
{
    void Speak()
    {
    }
    void Vuff()
    {
    }
}
[OpenCLS]
public class Frank : Pug
{
}
```

Listing A.37: Input code for chain reverse fork interface inheritance.

A.19.2 Expected Result

```c
struct Interface{
    int OpenCLS_ClassID;
    struct Pug* Sub_Pug;
};
struct Pug;
struct Frank;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
    struct Frank* Sub_Frank;
};
void Pug_Constructor(struct Pug* _instance);
void Pug_Speak(struct Pug* _instance);
void Pug_Vuff(struct Pug* _instance);
struct Frank{
    int OpenCLS_ClassID;
    struct Pug super;
};
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Pug = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Pug_Speak(_instance->Sub_Pug);
            break;
    }
}
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Pug_Vuff(_instance->Sub_Pug);
            break;
    }
}
void Pug_Constructor(struct Pug* _instance)
{
    Interface_Constructor(&_instance->interfaceSuper);
    _instance->interfaceSuper.OpenCLS_ClassID = 1;
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Frank = NULL;
    _instance->interfaceSuper.Sub_Pug = _instance;
}
void Pug_Speak(struct Pug* _instance)
{
}
void Pug_Vuff(struct Pug* _instance)
```

```
{
}
```

Listing A.38: Expected output for chain reverse fork interface inheritance.

A.20 Multiple Interface Inheritance

A.20.1 Input

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog
{
}
[OpenCLS]
public class Pug : Dog, IAnimal
{
    void Speak()
    {

    }
}
```

Listing A.39: Input for multiple inteface inheritance.

A.20.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Pug* Sub_Pug;
};
struct Dog;
struct Pug;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
    struct Interface interfaceSuper;
};
void Pug_Speak(struct Pug* _instance);

void Interface_Constructor(struct Interface* _instance)
```

```
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Pug = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 2:
            Pug_Speak(_instance->Sub_Pug);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
}
void Pug_Speak(struct Pug* _instance)
{

}
```

Listing A.40: Expected output for multiple inteface inheritance.

A.20.3 Result

Test succeeded.

A.21 Interface Class Mix

A.21.1 Input

```
[OpenCLS]
public class MyClassA
{
    public int MyVar;
}
[OpenCLS]
public class MyClassB : MyClassA
{
    public int AnotherVar;
}

[OpenCLS]
public interface MyInterface
{
    void MyMethod();
}

[OpenCLS]
public class MyClassC : MyClassB, MyInterface
{
    void MyInterface.MyMethod()
    {
        MyVar = AnotherVar;
    }
}
```

Listing A.41: Input code for inheriting from both an interface and a class.

A.21.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct MyClassC* Sub_MyClassC;
};
struct MyClassA;
struct MyClassB;
struct MyClassC;

void Interface_Constructor(struct Interface* _instance);
void Interface_MyMethod_Virtual_Resolver(struct Interface* ↩
    _instance);
// Full class declarations
struct MyClassA{
    int OpenCLS_ClassID;
    struct MyClassB* Sub_MyClassB;
    int MyVar;
};
void MyClassA_Constructor(struct MyClassA* _instance);

struct MyClassB{
    int OpenCLS_ClassID;
    struct MyClassA super;
    struct MyClassC* Sub_MyClassC;
    int AnotherVar;
};
void MyClassB_Constructor(struct MyClassB* _instance);

struct MyClassC{
    int OpenCLS_ClassID;
    struct MyClassB super;
    struct Interface interfaceSuper;
};
void MyClassC_MyMethod(struct MyClassC* _instance);
void MyClassA_Constructor(struct MyClassA* _instance)
{
    _instance->OpenCLS_ClassID = 1;
    _instance->Sub_MyClassB = NULL;
    _instance->MyVar = 0;
}
void MyClassB_Constructor(struct MyClassB* _instance)
{
    MyClassA_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->Sub_MyClassC = NULL;
    _instance->super.Sub_MyClassB = _instance;
    _instance->AnotherVar = 0;
}
void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_MyClassC = NULL;
}
void Interface_MyMethod_Virtual_Resolver(struct Interface* ↩
   _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 3:
            MyClassC_MyMethod(_instance->Sub_MyClassC);
            break;
    }
}
void MyClassC_MyMethod(struct MyClassC* _instance)
{
    _instance->super.super.MyVar = _instance->super.AnotherVar;
}
```

Listing A.42: Expected output for inheriting from both an interface and a class.

A.22 Flower Interface Inheritance

A.22.1 Input

```
[OpenCLS]
public interface ILiving
{
    bool Breathing();
}
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public interface IMammel
{
    int WalkSpeed();
}
[OpenCLS]
public interface IEvolutionable
{
    int NumberOfEvolutionaryParents();
}
[OpenCLS]
public interface ICarnivore
{
    void TypeOfMeat();
}
[OpenCLS]
```

```
public interface IDog : ILiving, IAnimal, IMammel, IEvolutionable, ↩
    ICarnivore
{
    void Vuff();
}
[OpenCLS]
public interface IPug : IDog
{
    void Size();
}
[OpenCLS]
public interface IHusky : IDog
{
    void Adorableness();
}
[OpenCLS]
public interface IDogMix : IPug, IHusky
{
    void Name();
}
[OpenCLS]
public class Frank : IDogMix
{
    public void Adorableness()
    {
    }

    public bool Breathing()
    {
        return true;
    }

    public void Name()
    {
    }

    public int NumberOfEvolutionaryParents()
    {
        return 2;
    }

    public void Size()
    {
    }

    public void Speak()
    {
    }

    public void TypeOfMeat()
    {
    }

    public void Vuff()
    {
    }

    public int WalkSpeed()
    {
        return 30;
    }
}
```

Listing A.43: Input code for flower interface inheritance.

A.22.2 Expected Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Frank* Sub_Frank;
};
struct Frank;
void Interface_Constructor(struct Interface* _instance);
bool Interface_Breathing_Virtual_Resolver(struct Interface* ↩
    _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
int Interface_WalkSpeed_Virtual_Resolver(struct Interface* ↩
    _instance);
int Interface_NumberOfEvolutionaryParents_Virtual_Resolver(struct ↩
    Interface* _instance);
void Interface_TypeOfMeat_Virtual_Resolver(struct Interface* ↩
    _instance);
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance);
void Interface_Size_Virtual_Resolver(struct Interface* _instance);
void Interface_Adorableness_Virtual_Resolver(struct Interface* ↩
    _instance);
void Interface_Name_Virtual_Resolver(struct Interface* _instance);

struct Frank{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Frank_Adorableness(struct Frank* _instance);
bool Frank_Breathing(struct Frank* _instance);
void Frank_Name(struct Frank* _instance);
int Frank_NumberOfEvolutionaryParents(struct Frank* _instance);
void Frank_Size(struct Frank* _instance);
void Frank_Speak(struct Frank* _instance);
void Frank_TypeOfMeat(struct Frank* _instance);
void Frank_Vuff(struct Frank* _instance);
int Frank_WalkSpeed(struct Frank* _instance);

void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Frank = NULL;
}
bool Interface_Breathing_Virtual_Resolver(struct Interface* ↩
    _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            return Frank_Breathing(_instance->Sub_Frank);
    }
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_Speak(_instance->Sub_Frank);
            break;
    }
}
int Interface_WalkSpeed_Virtual_Resolver(struct Interface* ↩
   _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            return Frank_WalkSpeed(_instance->Sub_Frank);
    }
}
int Interface_NumberOfEvolutionaryParents_Virtual_Resolver(struct ↩
   Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            return Frank_NumberOfEvolutionaryParents(_instance->↩
               Sub_Frank);
    }
}
void Interface_TypeOfMeat_Virtual_Resolver(struct Interface* ↩
   _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_TypeOfMeat(_instance->Sub_Frank);
            break;
    }
}
void Interface_Vuff_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_Vuff(_instance->Sub_Frank);
            break;
    }
}
void Interface_Size_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_Size(_instance->Sub_Frank);
            break;
    }
}
void Interface_Adorableness_Virtual_Resolver(struct Interface* ↩
    _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_Adorableness(_instance->Sub_Frank);
            break;
    }
}
void Interface_Name_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Frank_Name(_instance->Sub_Frank);
            break;
    }
}


void Frank_Adorableness(struct Frank* _instance)
{
}

bool Frank_Breathing(struct Frank* _instance)
{
    return true;
}

void Frank_Name(struct Frank* _instance)
{
}

int Frank_NumberOfEvolutionaryParents(struct Frank* _instance)
{
    return 2;
}

void Frank_Size(struct Frank* _instance)
{
}

void Frank_Speak(struct Frank* _instance)
{
}

void Frank_TypeOfMeat(struct Frank* _instance)
{
}

void Frank_Vuff(struct Frank* _instance)
{
}
```

```

int Frank_WalkSpeed(struct Frank* _instance)
{
    return 30;
}
```

Listing A.44: Expected output for flower interface inheritance.

A.22.3 Result

Test succeeded.

A.23 Interface Method Invocation

A.23.1 Input Code

```
[OpenCLS]
public interface IAnimal
{
    void Speak();
}
[OpenCLS]
public class Dog : IAnimal
{
    void Speak()
    {

    }
}
[OpenCLS]
public class TestClass
{
    public void Main()
    {
        IAnimal animal = new Dog();
        animal.Speak();
    }
}
```

Listing A.45: Input code for invoking a interface method from a subtyped variable.

A.23.2 Expedted Result

```
struct Interface{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
struct Dog;
struct TestClass;
void Interface_Constructor(struct Interface* _instance);
void Interface_Speak_Virtual_Resolver(struct Interface* _instance);
```

```
struct Dog{
    int OpenCLS_ClassID;
    struct Interface interfaceSuper;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_Speak(struct Dog* _instance);

struct TestClass{
    int OpenCLS_ClassID;
};
void TestClass_Main(struct TestClass* _instance);

void Interface_Constructor(struct Interface* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Interface_Speak_Virtual_Resolver(struct Interface* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 1:
            Dog_Speak(_instance->Sub_Dog);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Interface_Constructor(&_instance->interfaceSuper);
    _instance->interfaceSuper.OpenCLS_ClassID = 1;
    _instance->OpenCLS_ClassID = 1;
    _instance->interfaceSuper.Sub_Dog = _instance;
}
void Dog_Speak(struct Dog* _instance)
{
}

void TestClass_Main(struct TestClass* _instance)
{
    struct Dog IAnimal_Sub_animal;
    Dog_Constructor(&IAnimal_Sub_animal);
    struct Interface animal = IAnimal_Sub_animal.interfaceSuper;
    Interface_Speak_Virtual_Resolver(&animal);
}
```

Listing A.46: Expected output for invoking a interface method from a subtyped variable.

A.23.3 Results

The test succeeded.

A.24 Abstract Class Empty Declaration

A.24.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
}
```

Listing A.47: Input code for abstract class with empty declaration

A.24.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct TestClass;
struct TestClass{
};

///////// FORWARD DECLARATION END /////////
```

Listing A.48: Expected output for abstract class with empty declaration

A.24.3 Result

The test succeeded.

A.25 Abstract Class Declaration Public Member Variable

A.25.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
   public int pubInt;
}
```

Listing A.49: Input code for abstract class declaration with public fields

A.25.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct TestClass;
struct TestClass{
    int pubInt;
};

///////// FORWARD DECLARATION END /////////
```

Listing A.50: Expected output for abstract class declaration with public fields

A.25.3 Result

The test succeeded.

A.26 Abstract Class Declaration Private Member Variable

A.26.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
   private int priInt;
}
```

Listing A.51: Input code for abstract class declaration with private field

A.26.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct TestClass;
struct TestClass{
    int priInt;
};

///////// FORWARD DECLARATION END /////////
```

Listing A.52: Expected output for abstract class declaration private field

A.26.3 Result

The test succeeded.

A.27 Abstract Class Declaration Int Member Variable

A.27.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
   int testInt;
}
```

Listing A.53: Input code for abstract class declaration with int field

A.27.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct TestClass;
struct TestClass{
    int testInt;
};

///////// FORWARD DECLARATION END ////////
```

Listing A.54: Expected output for abstract class declaration with int field

A.27.3 Result

The test succeeded.

A.28 Abstract Class Declaration String Member Variable

A.28.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
   string testString;
}
```

Listing A.55: Input code for abstract class declaration with string field

A.28.2 Expected Result

```
    // Forward declaration of classes when structs reference ↩
        eachother
struct TestClass;
struct TestClass{
    char testString[1000];
};

///////// FORWARD DECLARATION END /////////
```

Listing A.56: Expected output for abstract class declaration with string field

A.28.3 Result

The test succeeded.

A.29 Abstract Class Declaration Char Member Variable

A.29.1 Input Code

```
[OpenCLS]
public abstract class TestClass
{
    char testChar;
}
```

Listing A.57: Input code for abstract class declaration with char field

A.29.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct TestClass;
struct TestClass{
     char testChar;
};

///////// FORWARD DECLARATION END /////////
```

Listing A.58: Expected output for abstract class declaration with char field

A.29.3 Result

The test succeeded.

A.30 Abstract Class Declaration Int Array Field

A.30.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    int[] testIntArray;
}
```

Listing A.59: Input for abstract class declaration with an int array field.

A.30.2 Result

Test failed as expected.

A.31 Abstract Class Declaration String Array Field

A.31.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    string[] testStringArray;
}
```
Listing A.60: Input for abstract class declaration with a string array field.

A.31.2 Result

The test failed as expected.

A.32 Abstract Class Declaration Char Array Field

A.32.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    char[] testCharArray;
}
```
Listing A.61: Input for abstract class declaration with a char array field.

A.32.2 Result

The test failed as expected.

A.33 Abstract Class Declaration Empty Method With No Parameters

A.33.1 Input

```
[OpenCLS]
public abstract class TestClass
{
     public void f()
     {
     }
}
```
Listing A.62: Input for abstract class declaration with an empty method that has no parameters

A.33.2 Expected Result

```
struct TestClass;
struct TestClass{
};
void TestClass_f(struct TestClass* _instance);
void TestClass_f(struct TestClass* _instance)
```

```
{
}
```

Listing A.63: Expected output for abstract class declaration with an empty method that has no parameters

A.33.3 Result

The test succeeded.

A.34 Abstract Class Declaration Empty Method With Parameters

A.34.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    public void f(int testInt)
    {
    }
}
```

Listing A.64: Input for abstract class declaration with an empty method that has parameters.

A.34.2 Expected Result

```
struct TestClass;
struct TestClass{
};
void TestClass_f(struct TestClass* _instance, int testInt);
void TestClass_f(struct TestClass* _instance, int testInt)
{
}
```

Listing A.65: Expected output for abstract class declaration with an empty method that has parameters.

A.34.3 Result

The test succeeded.

A.35 Abstract Class Declaration Filled Method With No Parameters

A.35.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    public void f()
    {
        testInt = 1;
    }
}
```

Listing A.66: Input for abstract class declaration with a filled method that has no parameters.

A.35.2 Expected Result

```
struct TestClass;
struct TestClass{
};
void TestClass_f(struct TestClass* _instance);
void TestClass_f(struct TestClass* _instance)
{
    testInt = 1;
}
```

Listing A.67: Expected output for abstract class declaration with a filled method that has no parameters.

A.35.3 Result

The test succeeded.

A.36 Abstract Class Declaration Filled Method With Parameters

A.36.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    public void f(int testInt)
    {
        testInt = 1;
    }
}
```

Listing A.68: Input for abstract class declaration with a filled method that has parameters.

A.36.2 Expected Result

```
struct TestClass;
struct TestClass{
};
void TestClass_f(struct TestClass* _instance, int testInt);
void TestClass_f(struct TestClass* _instance, int testInt)
{
    testInt = 1;
}
```

Listing A.69: Expected output for abstract class declaration with a filled method that has parameters.

A.36.3 Result

The test succeeded.

A.37 Abstract Class Declaration Empty Constructor With Parameters

A.37.1 Input

```
[OpenCLS]
public abstract class TestClass2
{
    public TestClass2(int testInt2)
    {
    }
}
```

Listing A.70: Input for abstract class delcaration with an empty constructor with parameters.

A.37.2 Expected Result

```
struct TestClass2;
struct TestClass2{
};
void TestClass2_Constructor(struct TestClass2* _instance, int ↩
    testInt2);
void TestClass2_Constructor(struct TestClass2* _instance, int ↩
    testInt2)
{
}
```

Listing A.71: Expected output for abstract class delcaration with an empty constructor with parameters.

A.37.3 Result

The test succeeded.

A.38 Abstract Class Declaration Multiple Empty Constructors With Parameters

A.38.1 Input

```
[OpenCLS]
public abstract class TestClass
{
    public TestClass(int testInt)
    {
    }
    public TestClass(char testChar)
    {
    }
}
```

Listing A.72: Input for abstract class delcaration with multiple empty constructors with parameters.

A.38.2 Expected Result

```
struct TestClass;
struct TestClass{
};
void TestClass_Constructor_int(struct TestClass* _instance, int ↩
    testInt);
void TestClass_Constructor_char(struct TestClass* _instance, char ↩
    testChar);
void TestClass_Constructor_int(struct TestClass* _instance, int ↩
    testInt)
{
}
void TestClass_Constructor_char(struct TestClass* _instance, char ↩
    testChar)
{
}
```

Listing A.73: Expected output for abstract class delcaration with multiple empty constructors with parameters.

A.38.3 Result

The test succeeded.

A.39 Abstract Subclass

A.39.1 Input Code

```
[OpenCLS]
```

```
public class Animal
{
    public int Age;
    public virtual void Speak(int i)
    {
        Console.WriteLine(i);
    }
}

[OpenCLS]
public abstract class Dog : Animal
{
    public override void Speak(int i)
    {
    }
}

[OpenCLS]
public class Pug : Dog
{
    public override void Speak(int i)
    {
        Console.WriteLine(""Bark"");
    }
}
class Program
{
        [OpenCLS]
        void MyMethod()
        {
           Pug pug = new Pug();
           pug.Speak(1);
        }
}
```

Listing A.74: Input code for abstract class with a superclass and subclass.

A.39.2 Expected Result

```
struct Animal;
struct Dog;
struct Pug;

struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_Speak(struct Animal* _instance, int i);
void Animal_Speak_Virtual_Resolver(struct Animal* _instance, int i)↩
    ;
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_Speak(struct Dog* _instance, int i);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
};
void Pug_Constructor(struct Pug* _instance);
void Pug_Speak(struct Pug* _instance, int i);

__kernel void MyMethod_Kernel();
void Program_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Animal_Speak(struct Animal* _instance, int i)
{
    printf(""%d\n"", i);
}
void Animal_Speak_Virtual_Resolver(struct Animal* _instance, int i)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 0:
            Animal_Speak(_instance, i);
            break;
        case 1:
            Dog_Speak(_instance->Sub_Dog, i);
            break;
        case 2:
            Pug_Speak(_instance->Sub_Dog->Sub_Pug, i);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->super.Sub_Dog = _instance;
}
void Dog_Speak(struct Dog* _instance, int i)
{
}
void Pug_Constructor(struct Pug* _instance)
{
    Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.super.OpenCLS_ClassID = 2;
    _instance->super.Sub_Pug = _instance;
```

```
}
void Pug_Speak(struct Pug* _instance, int i)
{
    printf(""Bark\n"");
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Pug pug;
    Pug_Constructor(&pug);
    Animal_Speak_Virtual_Resolver(&(pug.super.super), 1);
}
void Program_MyMethod()
{
    struct Pug pug;
    Pug_Constructor(&pug);
    Animal_Speak_Virtual_Resolver(&(pug.super.super), 1);
}
```

Listing A.75: Expected output for abstract class with a superclass and subclass.

A.39.3 Result

The test succeeded.

A.40 Protected Field

A.40.1 Input

```
[OpenCLS]
public class Animal
{
    protected int Age;
}

[OpenCLS]
public class Dog : Animal
{
    public Dog()
    {
        Age = 2;
    }
}
class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();
    }
}
```

Listing A.76: Input code for protected field.

A.40.2 Expected Result

```
struct Animal;
struct Dog;

// Full class declarations
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);

struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);

// Forward declaration of methods
__kernel void MyMethod_Kernel();
void Test_MyMethod();

///////// FORWARD DECLARATION END /////////


void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}

void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
    _instance->super.Age = 2;
}

__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
}

void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
}
```

Listing A.77: Expected output for protected field.

A.40.3 Result

The test succeeded.

A.41 Protected Method

A.41.1 Input

```
[OpenCLS]
public class Animal
{
    protected void Speak(int number)
    {
        Console.WriteLine(number);
    }
}

[OpenCLS]
public class Dog : Animal
{
    public void DisplayNumber()
    {
        Speak(999);
    }
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();
        dog.DisplayNumber();
    }
}
```

Listing A.78: Input code for protected method.

A.41.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct Animal;
struct Dog;

// Full class declarations
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_Speak(struct Animal* _instance, int number);

```

```
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_DisplayNumber(struct Dog* _instance);

// Forward declaration of methods
__kernel void MyMethod_Kernel();
void Test_MyMethod();

///////// FORWARD DECLARATION END /////////


void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void Animal_Speak(struct Animal* _instance, int number)
{
    printf(""%d\n"", number);
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
void Dog_DisplayNumber(struct Dog* _instance)
{
    Animal_Speak(&_instance->super, 999);
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    Dog_DisplayNumber(&dog);
}
void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    Dog_DisplayNumber(&dog);
}
```

Listing A.79: Expected output for protected method.

A.41.3 Result

The test succeeded.

A.42 Protected Class

A.42.1 Input

```
[OpenCLS]
public class Animal
{
    [OpenCLS]
    protected class House
    {
        public int weight;
    }
}
[OpenCLS]
public class Dog : Animal
{
    public void DogHouse()
    {
        House dogHouse = new House();
        dogHouse.weight = 10;
    }
}
class Test
{
    [OpenCLS]
    public void MyMethod()
    {
        Dog dog = new Dog();
        dog.DogHouse();
    }
}
```

Listing A.80: Input code for a protected class.

A.42.2 Expected Result

```
// Forward declaration of classes when structs reference eachother
struct Animal;
struct House;
struct Dog;

// Full class declarations
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
};
void Animal_Constructor(struct Animal* _instance);

struct House{
    int OpenCLS_ClassID;
    int weight;
};
void House_Constructor(struct House* _instance);
```

```

struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_DogHouse(struct Dog* _instance);

// Forward declaration of methods
__kernel void MyMethod_Kernel();
void Test_MyMethod();

///////// FORWARD DECLARATION END /////////


void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
}
void House_Constructor(struct House* _instance)
{
    _instance->OpenCLS_ClassID = 1;
    _instance->weight = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.Sub_Dog = _instance;
}
void Dog_DogHouse(struct Dog* _instance)
{
    struct House dogHouse;
    House_Constructor(&dogHouse);
    dogHouse.weight = 10;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    Dog_DogHouse(&dog);
}
void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    Dog_DogHouse(&dog);
}
```

Listing A.81: Expected output for a protected class.

A.42.3 Result

The test succeeded.

A.43 Virtual Override

A.43.1 Input code

```
[OpenCLS]
public class Animal
{
    public int MyAge;
    public virtual void MyMethod()
    {
        MyAge = 3;
    }
}

[OpenCLS]
public class Dog : Animal
{
    public override void MyMethod()
    {
        MyAge = 6;
    }
}
class Test
{
    [OpenCLS]
    public void TestMethod()
    {
        Dog d = new Dog();
        d.MyMethod();
    }
}
```

Listing A.82: Input code for virtual overriding.

A.43.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int MyAge;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_MyMethod(struct Animal* _instance);
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_MyMethod(struct Dog* _instance);
__kernel void TestMethod_Kernel();
void Test_TestMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->MyAge = 0;
}
void Animal_MyMethod(struct Animal* _instance)
{
    _instance->MyAge = 3;
}
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 0:
            Animal_MyMethod(_instance);
            break;
        case 1:
            Dog_MyMethod(_instance->Sub_Dog);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
void Dog_MyMethod(struct Dog* _instance)
{
    _instance->super.MyAge = 6;
}
__kernel void TestMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog d;
    Dog_Constructor(&d);
    Animal_MyMethod_Virtual_Resolver(&(d.super));
}
void Test_TestMethod()
{
    struct Dog d;
    Dog_Constructor(&d);
    Animal_MyMethod_Virtual_Resolver(&(d.super));
}
```

Listing A.83: Expected output for vitual overriding.

A.43.3 Result

The test succeeded.

A.44 Chain Virtual Override

A.44.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int MyAge;
    public virtual void MyMethod()
    {
        MyAge = 3;
    }
}

[OpenCLS]
public class Dog : Animal
{
    public override void MyMethod()
    {
        MyAge = 6;
    }
}

[OpenCLS]
public class Pug : Dog
{
    public override void MyMethod()
    {
        MyAge = 8;
    }
}

[OpenCLS]
public class MyDogBuddy : Pug
{
    public override void MyMethod()
    {
        MyAge = 10;
    }
}
```

Listing A.84: Input code for chain virtual overriding.

A.44.2 Expected Result

```
struct Dog;
struct Pug;
struct MyDogBuddy;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int MyAge;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_MyMethod(struct Animal* _instance);
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_MyMethod(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
    struct MyDogBuddy* Sub_MyDogBuddy;
};
void Pug_Constructor(struct Pug* _instance);
void Pug_MyMethod(struct Pug* _instance);
struct MyDogBuddy{
    int OpenCLS_ClassID;
    struct Pug super;
};
void MyDogBuddy_MyMethod(struct MyDogBuddy* _instance);

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->MyAge = 0;
}
void Animal_MyMethod(struct Animal* _instance)
{
    _instance->MyAge = 3;
}
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 0:
            Animal_MyMethod(_instance);
            break;
        case 1:
            Dog_MyMethod(_instance->Sub_Dog);
            break;
        case 2:
            Pug_MyMethod(_instance->Sub_Dog->Sub_Pug);
            break;
        case 3:
            MyDogBuddy_MyMethod(_instance->Sub_Dog->Sub_Pug->↩
               Sub_MyDogBuddy);
            break;
    }
}
```

```
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->super.Sub_Dog = _instance;
}
void Dog_MyMethod(struct Dog* _instance)
{
    _instance->super.MyAge = 6;
}
void Pug_Constructor(struct Pug* _instance)
{
    Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.super.OpenCLS_ClassID = 2;
    _instance->Sub_MyDogBuddy = NULL;
    _instance->super.Sub_Pug = _instance;
}
void Pug_MyMethod(struct Pug* _instance)
{
    _instance->super.super.MyAge = 8;
}
void MyDogBuddy_MyMethod(struct MyDogBuddy* _instance)
{
    _instance->super.super.super.MyAge = 10;
}
```

Listing A.85: Expected output for chain virtual overriding.

A.44.3 Result

The test succeeded.

A.45 Chain Virtual Override with a skip

A.45.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int MyAge;
    public virtual void MyMethod()
    {
        MyAge = 3;
    }
}

[OpenCLS]
public class Dog : Animal
{
    public override void MyMethod()
    {
        MyAge = 6;
    }
}

[OpenCLS]
public class Pug : Dog
{

}

[OpenCLS]
public class MyDogBuddy : Pug
{
    public override void MyMethod()
    {
        MyAge = 10;
    }
}
```

Listing A.86: Input code for chain virtual overriding with class skip.

A.45.2 Expected Result

```
struct Dog;
struct Pug;
struct MyDogBuddy;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int MyAge;
};
void Animal_Constructor(struct Animal* _instance);
void Animal_MyMethod(struct Animal* _instance);
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
    struct Pug* Sub_Pug;
};
void Dog_Constructor(struct Dog* _instance);
void Dog_MyMethod(struct Dog* _instance);
struct Pug{
    int OpenCLS_ClassID;
    struct Dog super;
    struct MyDogBuddy* Sub_MyDogBuddy;
};
void Pug_Constructor(struct Pug* _instance);
struct MyDogBuddy{
    int OpenCLS_ClassID;
    struct Pug super;
};
void MyDogBuddy_MyMethod(struct MyDogBuddy* _instance);

void Animal_Constructor(struct Animal* _instance)
```

```
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->MyAge = 0;
}
void Animal_MyMethod(struct Animal* _instance)
{
    _instance->MyAge = 3;
}
void Animal_MyMethod_Virtual_Resolver(struct Animal* _instance)
{
    switch (_instance->OpenCLS_ClassID)
    {
        case 0:
            Animal_MyMethod(_instance);
            break;
        case 1:
        case 2:
            Dog_MyMethod(_instance->Sub_Dog);
            break;
        case 3:
            MyDogBuddy_MyMethod(_instance->Sub_Dog->Sub_Pug->↩
                Sub_MyDogBuddy);
            break;
    }
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->Sub_Pug = NULL;
    _instance->super.Sub_Dog = _instance;
}
void Dog_MyMethod(struct Dog* _instance)
{
    _instance->super.MyAge = 6;
}
void Pug_Constructor(struct Pug* _instance)
{
    Dog_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 2;
    _instance->super.OpenCLS_ClassID = 2;
    _instance->super.super.OpenCLS_ClassID = 2;
    _instance->Sub_MyDogBuddy = NULL;
    _instance->super.Sub_Pug = _instance;
}
void MyDogBuddy_MyMethod(struct MyDogBuddy* _instance)
{
    _instance->super.super.super.MyAge = 10;
}
```

Listing A.87: Expected output for chain virtual overriding with class skip.

A.45.3 Result

The test succeeded.

A.46 Local Subtyping Declaration

A.46.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Animal animal = new Dog();
    }
}
```

Listing A.88: Input code for local subtyping declaration.

A.46.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog Animal_Sub_animal;
    Dog_Constructor(&Animal_Sub_animal);
    struct Animal animal = Animal_Sub_animal.super;
}
void Test_MyMethod()
{
    struct Dog Animal_Sub_animal;
    Dog_Constructor(&Animal_Sub_animal);
    struct Animal animal = Animal_Sub_animal.super;
}
```

Listing A.89: Expected output for local subtyping declaration.

A.46.3 Result

The test succeeded.

A.47 Local Subtyping Assignment Object Creation

A.47.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}

[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();

        Animal animal = dog;
    }
}
```

Listing A.90: Input code for local subtyping assignment with object creation.

A.47.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    struct Animal animal = dog.super;
}
void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    struct Animal animal = dog.super;
}
```

Listing A.91: Expected output for local subtyping assignment with object creation.

A.47.3 Result

The test succeeded.

A.48 Local Subtyping Declaration with Assignemnt

A.48.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}
[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Animal animal = new Animal();

        animal = new Dog();
    }
}
```

Listing A.92: Input code for local subtyping declaration with assignment.

A.48.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Animal animal;
    Animal_Constructor(&animal);
    struct Dog Animal_Sub_animal;
    Dog_Constructor(&Animal_Sub_animal);
    animal = Animal_Sub_animal.super;
}
void Test_MyMethod()
{
    struct Animal animal;
    Animal_Constructor(&animal);
    struct Dog Animal_Sub_animal;
    Dog_Constructor(&Animal_Sub_animal);
    animal = Animal_Sub_animal.super;
}
```

Listing A.93: Expected output for local subtyping declaration with assignment.

A.48.3 Result

The test succeeded.

A.49 Local Subtyping Assignment

A.49.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}

[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    [OpenCLS]
    void MyMethod()
    {
        Animal animal = new Animal();
        Dog dog = new Dog();

        animal = dog;
    }
}
```

Listing A.94: Input code for local subtyping assignment.

A.49.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Animal animal;
    Animal_Constructor(&animal);
    struct Dog dog;
    Dog_Constructor(&dog);
    animal = dog.super;
}
void Test_MyMethod()
{
    struct Animal animal;
    Animal_Constructor(&animal);
    struct Dog dog;
    Dog_Constructor(&dog);
    animal = dog.super;
}
```

Listing A.95: Expected output for local subtyping assignment.

A.49.3 Result

The test succeeded.

A.50 Parameter subtyping

A.50.1 Input Code

```
[OpenCLS]
public class Animal
{
    public int Age;
}

[OpenCLS]
public class Dog : Animal
{
}

class Test
{
    void MyTestMethod(Animal animal)
    { }

    [OpenCLS]
    void MyMethod()
    {
        Dog dog = new Dog();
        MyTestMethod(dog);

    }
}
```

Listing A.96: Input code for parameter subtyping.

A.50.2 Expected Result

```
struct Animal;
struct Dog;
struct Animal{
    int OpenCLS_ClassID;
    struct Dog* Sub_Dog;
    int Age;
};
void Animal_Constructor(struct Animal* _instance);
struct Dog{
    int OpenCLS_ClassID;
    struct Animal super;
};
void Dog_Constructor(struct Dog* _instance);
__kernel void MyMethod_Kernel();
void Test_MyMethod();

void Animal_Constructor(struct Animal* _instance)
{
    _instance->OpenCLS_ClassID = 0;
    _instance->Sub_Dog = NULL;
    _instance->Age = 0;
}
void Dog_Constructor(struct Dog* _instance)
{
    Animal_Constructor(&_instance->super);
    _instance->OpenCLS_ClassID = 1;
    _instance->super.OpenCLS_ClassID = 1;
    _instance->super.Sub_Dog = _instance;
}
__kernel void MyMethod_Kernel()
{
    size_t OpenCLSThreadId = get_global_id(0);
    struct Dog dog;
    Dog_Constructor(&dog);
    Test_MyTestMethod(dog.super);
}
void Test_MyMethod()
{
    struct Dog dog;
    Dog_Constructor(&dog);
    Test_MyTestMethod(dog.super);
}
```

Listing A.97: Expected output for parameter subtyping.

A.50.3 Result

The test succeeded.

A.51 Related Work: OpenCLS 2 compared with Function Pointers

```
[OpenCLS]
#include <time.h>
#include <cstdio>
#include <chrono>
#include <iostream>
#include <windows.h>

class FPClassA
{
public:
  int virtual Execution(int InVal) { return InVal; }
};

class FPClassB : FPClassA
{
public:
  int virtual Execution(int InVal)
  {
    return InVal+1;
  }
};

// (...)

struct OpenCLSClass2;

struct OpenCLSClass1
{
  int ClassID;
  OpenCLSClass2* sub_Class2;
};

struct OpenCLSClass2
{
  int ClassID;
  OpenCLSClass1 super;
};

// (...)

void OpenCLSClass1_Constructor(OpenCLSClass1* _instance)
{
  _instance->ClassID = 0;
  _instance->sub_Class2 = NULL;
}

void OpenCLSClass2_Constructor(OpenCLSClass2* _instance)
{
  _instance->ClassID = 1;
  _instance->super.sub_Class2 = _instance;
}

// (...)

int OpenCLSClass1_Execution(int InVal, OpenCLSClass1* _instance)
{
  return InVal;
};

int OpenCLSClass2_Execution(int InVal, OpenCLSClass1* _instance)
{
  return (InVal+1) % 1000;
};

int OpenCLSClass3_Execution(int InVal, OpenCLSClass1* _instance)
{
  return (InVal + 2) % 1000;
};

// (...)

int OpenCLSClass999_Execution(int InVal, OpenCLSClass1* _instance)
{
  return (InVal + 999) % 1000;
};


int OpenCLSClass1_Execution_Resolver(int InVal, OpenCLSClass1* ↩
    _instance)
{
  switch (_instance->ClassID)
  {
    case 0:
      return OpenCLSClass1_Execution(InVal, _instance);
    case 1:
      return OpenCLSClass2_Execution(InVal, _instance);
    case 2:
      return OpenCLSClass3_Execution(InVal, _instance);
    //(...)
  }

}


// Test cases

void TestFP(unsigned long long int Iterations)
{
  FPClassB MyClass;

  int MyVal = 0;
  while (Iterations > 0)
  {
    MyVal = MyClass.Execution(MyVal);
    Iterations--;
  }
}

int MultiTestFP(unsigned long long int TestIterations, unsigned ↩
    long long int PerTestIterations)
{
  auto begin = std::chrono::high_resolution_clock::now();

  unsigned long long int i = TestIterations;
  while (TestIterations > 0)
  {
    TestFP(PerTestIterations);
    TestIterations--;
  }

  auto end = std::chrono::high_resolution_clock::now();
  unsigned int ns = std::chrono::duration_cast<std::chrono::↩
      nanoseconds>(end - begin).count();

  ns /= i;

  std::cout << ns << "ns" << std::endl;
  return ns;
}

```

```
void TestOpenCLS(unsigned long long int Iterations)
{
  OpenCLSClass2 MyClass;
  OpenCLSClass2_Constructor(&MyClass);

  int MyVal = 0;
  while (Iterations > 0)
  {
    MyVal = OpenCLSClass1_Execution_Resolver(MyVal, &MyClass.super)↩
        ;
    Iterations--;
  }
}

int MultiTestOpenCLS(unsigned long long int TestIterations, ↩
    unsigned long long int PerTestIterations)
{
  auto begin = std::chrono::high_resolution_clock::now();

  unsigned long long int i = TestIterations;
  while (TestIterations > 0)
  {
    TestOpenCLS(PerTestIterations);
    TestIterations--;
  }

  auto end = std::chrono::high_resolution_clock::now();
  unsigned int ns = std::chrono::duration_cast<std::chrono::↩
      nanoseconds>(end - begin).count();

  ns /= i;

  std::cout << ns << "ns" << std::endl;
  return ns;
}

int main()
{
  unsigned long long int iterationsPerTest = 10000000000ull;
  unsigned long long int iterations = 100ull; // we average the ↩
      result of this


  MultiTestOpenCLS(iterations, iterationsPerTest);
  MultiTestFP(iterations, iterationsPerTest);

  return 0;
}
```

Listing A.98: Execution time comparison between OpenCLS 2 and a function pointer implementation

```
    string s = "";
    int NumberOfExecutions = 100000;

    var watch = System.Diagnostics.Stopwatch.StartNew();
    for (int i = 0; i < NumberOfExecutions; i++)
        Exec.Main(0);
    watch.Stop();
    s += "CPU: " + watch.ElapsedNanoSeconds() + "ns, " + watch.↩
       ElapsedMilliseconds.ToString() + "ms, " + watch.Elapsed.↩
       ToString() + "s";

    int[] InArray = new int[NumberOfExecutions];
    watch = System.Diagnostics.Stopwatch.StartNew();
    OpenCLS.Run.Main(InArray);
    watch.Stop();
    s += "\nGPU: " + watch.ElapsedNanoSeconds() + "ns, " + watch.↩
       ElapsedMilliseconds.ToString() + "ms, " + watch.Elapsed.↩
       ToString() + "s";

    Assert.Report(s);
```

Listing A.99: Parallel execution of the visitor design pattern

B Team Foundation Server PHP Hook (Source Code)

```php
<?php
  $people_to_notify = array("dpt1011f16@cs.aau.dk");

  function GetAuthorShortname($in)
  {
    $org = $in;
    $in = strtolower($in);
    if ($in == "jkarle11@student.aau.dk") return "Jonas";
    if ($in == "mholm11@student.aau.dk") return "Mads";
    if ($in == "smhc11@student.aau.dk") return "Stefan";
    if ($in == "mnguye11@student.aau.dk") return "Hieu";

    return $org;
  }

  $client_data = file_get_contents("php://input");
  $client_data_json = json_decode($client_data, true);
  $event_status = $_GET["status"];

  $commit_log_file = "commit_logs.log";
  if ($event_status == "commit")
  {
    $author = $client_data_json["resource"]["author"]["uniqueName"↩
       ];
    $commit_msg = $client_data_json["resource"]["comment"];
    $commit_date = $client_data_json["resource"]["createdDate"];
    $commit_date = strtotime($commit_date);
    $commit_id = $client_data_json["resource"]["changesetId"];

    $commit_data_save = array();
    $commit_data_save["author"] = $author;
    $commit_data_save["comment"] = $commit_msg;
    $commit_data_save["createdDate"] = $commit_date;
    $commit_data_save["changesetId"] = $commit_id;

    $file_output = json_encode($commit_data_save) . "\n";

    file_put_contents($commit_log_file, $file_output, FILE_APPEND |↩
        LOCK_EX);

  }
  else if ($event_status == "build")
  {
    $build_log_file = "build_logs.log";

    $build_status = $client_data_json["eventType"];
    $build_result = $client_data_json["resource"]["result"];
    $build_start_date = $client_data_json["resource"]["startTime"];
    $build_start_date = strtotime($build_start_date);
    $build_id = $client_data_json["resource"]["id"];
    $build_log_url = "https://dpt1011f16.visualstudio.com/DefaultCollection/P10/_build?buildId=".$build_id."&_a=summary";
    // if cancelled, do nothing
    if ($build_result != "canceled")
    {
      if ($build_result != "succeeded" && $build_status == "build.complete")
      {
        if ($build_result == "failed")
          $build_status = "build.failed";
        else
          $build_status = "build.complete.but.".$build_result;
      }

      // If failed, blame!
      if ($build_status != "build.complete")
      {
        $possible_culprits = array();
        $latest_build_date = 0;

        // Find the last successful build
        $handle = fopen($build_log_file, "r");
        if ($handle)
        {
          while (($line = fgets($handle)) !== false)
          {
            $parse_build_data = json_decode($line, true);
            if ($parse_build_data["createdDate"] <= $latest_build_date) continue;
            if ($parse_build_data["createdDate"] >= $build_start_date) continue;
            if ($parse_build_data["status"] != "build.complete") continue;

            // This build is more recent
            $latest_build_date = $parse_build_data["createdDate"];
          }
          fclose($handle);
        }

        // Find all commits between then and now
        $handle = fopen($commit_log_file, "r");
        if ($handle)
        {
          while (($line = fgets($handle)) !== false)
```

```php
      {
        $parse_commit_data = json_decode($line, true);
        if ($parse_commit_data["createdDate"] > $build_start_date) continue;
        if ($parse_commit_data["createdDate"] < $latest_build_date) continue;

        $possible_culprits[] = $parse_commit_data;
      }
    }

    // Send the notification
    $mail_message = "<html><body>";
    $mail_message .= "<h1>The latest dpt build has failed (".$build_status.")</h1>";
    if ($latest_build_date > 0)
      $mail_message .= "The error occured between " . date("F j, Y, g:i a", $latest_build_date) . " and " . date("F j, Y, g:i a", $build_start_date) . ".<br>";
    else
      $mail_message .= "The error occured on " . date("F j, Y, g:i a", $build_start_date).".<br>";

    $mail_message .= "\n";
    if (Count($possible_culprits) > 0)
    {
      $author_commits = array();
      foreach ($possible_culprits as $culprit)
      {
        if (!array_key_exists($culprit["author"], $author_commits))
          $author_commits[$culprit["author"]] = 0;
        $author_commits[$culprit["author"]]++;
      }
      arsort($author_commits);

      $mail_message .= "<h2>Possible culprits are:</h2>";
      $is_first = true;
      foreach ($author_commits as $author=>$commit_num)
      {
        if (!$is_first)
          $mail_message .= "<br>";
        $mail_message .= "<b>" . GetAuthorShortname($author) . "</b> (" . $commit_num . " commit" . ($commit_num > 1 ? "s" : "") . ")";
        $is_first = false;
      }

      $mail_message .= "<br><br>Through the following commits:<br>";
      $is_first = true;
      foreach ($possible_culprits as $culprit)
      {
        $commitid = $culprit["changesetId"];
        if (!$is_first)
          $mail_message .= "<br>";
        $mail_message .= "<a href='https://dpt1011f16.visualstudio.com/DefaultCollection/P10/_versionControl/changeset/".$commitid."'><b>Commit " . $commitid . "</b></a> by <b>" . GetAuthorShortname($culprit["author"]) . "</b> (<i>" . $culprit["comment"] . "</i>)";
        $is_first = false;
      }
    }
    else
    {
      $mail_message .= "Unfortunately we were unable to determine the culprits, as we have no commits within that period on record.<br>";
    }
    $mail_message .= "<br><a target='_blank' href='".$build_log_url."'>Click here for the full build report</a><br>";
    $mail_message .= "</body></html>";


    $headers = "From: " . "dpt1011f16@cs.aau.dk" . "\r\n";
    $headers .= "Reply-To: ". "dpt1011f16@cs.aau.dk" . "\r\n";
    $headers .= "MIME-Version: 1.0\r\n";
    $headers .= "Content-Type: text/html; charset=ISO-8859-1\r\n";

    $mail_title = "DPT Build failed (".date("l M jS", $build_start_date).")";

    $debug_output = $mail_message;
    foreach ($people_to_notify as $email)
    {
      $mail=mail($email, $mail_title, $mail_message, $headers);
      if (!$mail)
      {
        // mail failed to send
        $debug_output .= "<br><br>Failed to send email to: " . $mail;
      }
      else
        $debug_output .= "<br><br>Successfully sent email to: " . $mail;
    }
    file_put_contents("output.log",$debug_output);
  }

  // Write latest build to logs
  $build_data_save = array();
  $build_data_save["status"] = $build_status;
  $build_data_save["createdDate"] = $build_start_date;

  $file_output = json_encode($build_data_save) . "\n";
  file_put_contents($build_log_file, $file_output, FILE_APPEND | LOCK_EX);

```

```
        file_put_contents("info_build_failed.log",print_r(↩
            $client_data_json,true));
      }
    }
?>
```

Listing B.1: Code for TFS PHP Hook.