An Approach for Adding Type-Safe Static-Context Duck Typing to an Object-Oriented Programming Language

BY

Kevin Pond

A thesis submitted in partial fulfillment of the requirements for the

Master of Science

Major in Engineering

South Dakota State University

2010

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

An Approach for Adding Type-Safe Static-Context Duck Typing to an Object-Oriented Programming Language

This thesis is approved as a creditable and independent investigation by a candidate for the Master of Science degree and is acceptable for meeting the thesis requirements for this degree. Acceptance of this thesis does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department.

Dr. George Hamer — Date
Thesis Advisor
Electrical Engineering and Computer Science

Dr. Sung Shin — Date
Graduate Program Coordinator
Electrical Engineering and Computer Science

Dedication

To,

my friend,

Andold Thunk

may he rest in peace.

Abstract

An Approach for Adding Type-Safe Static-Context Duck Typing to an Object-Oriented Programming Language

Kevin Pond

June 24, 2010

A practical approach for adding type-safe static-context duck typing to an object-oriented programming language is proposed. This approach is suitable for languages such as Java, C#, or Visual Basic.NET. Duck-typing is shown to increase testability and flexibility by reducing type coupling. This approach is implemented for .NET languages and the impact on reliability, tooling, maintainability, and performance is compared to existing alternatives.

Table of Contents

Chapter 1. Introduction 1

Chapter 2. Background 9

2.1. Wrapper Class 9

2.2. Duck Typing 11

2.2.1. Dynamic-Context Duck Typing 12

2.2.2. Static-Context Duck Typing 16

2.3. Dynamic Metaprogramming 22

Chapter 3. Design 27

3.1. Language Modification 27

3.2. Bytecode Rewriting 32

3.2.1. Novelty 36

3.3. .NET Implementation 36

3.3.1. Tools 36

3.3.2. Components 38

3.3.3. Example 40

Chapter 4. Analysis 46

4.1. Reliability 46

4.1.1. Variable Usage Error 46

4.1.2. Duck Compatibility Error 51

4.1.3. Logical Type Mismatch 57

4.1.4. Wrapper Class Implementation Error 58

4.1.5. Reliability Summary 59

4.2. Tooling 59

4.2.1. Tooling Based on Static Type Information 59

4.2.2. Segments/Breaks Existing Tooling 61

4.2.3. Increased Build Time 62

4.2.4. Tooling Summary 63

4.3. Maintainability 63

4.4. Performance 65

4.4.1. Virtual Method Calls 65

4.4.2. Call Site Interpretation 66

4.4.3. Run-Time Code Generation 66

4.4.4. Run-Time Type Checking 67

4.4.5. Empirical Results 68

4.4.6. Performance Summary 71

4.5. Summary 71

Chapter 5. Findings 73

5.1. Conclusions 73

5.1.1. Reliability 73

5.1.2. Maintainability 73

5.1.3. Tooling 73

5.1.4. Performance 74

5.1.5. Overall 76

5.2. Future Work 77

List of Tables

Table 4.1: Maintenance Cost from Lines of Code 64

Table 4.2: Virtual Method Call Requirements 66

Table 4.3: Test Machine Details 68

Table 4.4: Call Performance 70

Table 4.5: Metaprogramming Duck Cast Performance 71

Table 4.6: Analysis Summary 72

Table 5.1: Call Performance 75

Table 5.2: Metaprogramming Duck Cast Performance 75

List of Figures

Figure 1.1: IFoo Interface 1

Figure 1.2: Foo Class 2

Figure 1.3: Raz and Baz Methods 3

Figure 1.4: Saz Class 6

Figure 2.1: SazWrapper Wrapper Class 10

Figure 2.2: Incorrect SazWrapper Wrapper Class 11

Figure 2.3: Dynamic-Context Duck Typing in Python 13

Figure 2.4: Dynamic-Context Duck Typing in Objective-C 14

Figure 2.5: Dynamic-Context Duck Typing in C# 4.0 15

Figure 2.6: Static-Context Duck Typing in C++ 17

Figure 2.7: Static-Context Duck Typing in C++ Fails to Locate Methods Available only at Run-Time 19

Figure 2.8: Logical Incongruity using Static-Context Duck Typing in C++ 21

Figure 2.9: dynamic_duck_cast Method and Example Usage 23

Figure 2.10: Logical Incongruity using dynamic_duck_cast (Java or C# like language) 25

Figure 3.1: Language Modification (Java or C# like language) 28

Figure 3.2: static_duck_cast Method 33

Figure 3.3: Hello World in CIL (abridged) 37

Figure 3.4: DuckTyping.Contracts 39

Figure 3.5: Source Code of Third-Party Library Containing Saz (C# language) . 41

Figure 3.6: Example Input Program (C# language) 41

Figure 3.7: CIL Listing of Main method within Figure 3.6 before Transformation 42

Figure 3.8: Generated DuckProxy_Saz Wrapper Class (C# Language) (abridged) 43

Figure 3.9: CIL Listing of Main method within Figure 3.6 after Transformation .. 44

Figure 3.10: Listing of Main Method within Figure 3.6 after Transformation (C# Language) 44

Figure 3.11: Bytecode Rewriting Data Flow 45

Figure 4.1: Incorrect Variable Usage with Wrapper Class (C# Language) 47

Figure 4.2: Incorrect Variable Usage with Dynamic-Context Duck Typing (C# Language) 47

Figure 4.3: Incorrect Variable Usage with Static-Context Duck Typing (C++ Language) 49

Figure 4.4: Incorrect Variable Usage with Metaprogramming (C# Language) 49

Figure 4.5: Incorrect Variable Usage with Language Modification 50

Figure 4.6: Incorrect Variable Usage with Bytecode Rewriting (C# Language) .. 51

Figure 4.7: Duck Incompatibility with Wrapper Class (C# Language) 52

Figure 4.8: Duck Incompatibility with Dynamic-Context Duck Typing (C# Language) 53

Figure 4.9: Duck Incompatibility with Static-Context Duck Typing (C++ Language) 54

Figure 4.10: Duck Incompatibility with Metaprogramming (C# Language) 55

Figure 4.11: Duck Incompatibility with Language Modification 55

Figure 4.12: Duck Incompatibility with Bytecode Rewriting (C# Language)..... 56

Figure 4.13: Ineffective Static Type Information (C++ Language) 60

Figure 4.14: Ambiguous Rename (C++ Language) 62

Chapter 1. Introduction

Many object-oriented programming languages support the concept of *interfaces*. Interfaces allow the programmer to define a set of methods that correspond to some capability. An interface defines a set of methods that must be supported by classes that implement the interface, but it does not define how these methods are implemented. Interfaces can contain methods, but they cannot contain fields that store state.

Figure 1.1 defines an interface named IFoo (the I prefix is a common naming convention used to signify an interface). IFoo defines one member—a method named Bar. The signature of Bar is defined by the interface, but the implementation is not. If a class wishes to implement IFoo, it must define an implementation for all members of the interface and mark that the class implements the interface.

```
interface IFoo
{
    void Bar();
}
```

Figure 1.1: IFoo Interface

IFoo should be thought of as a capability. A class that implements IFoo supports the IFoo capability. A class that implements an interface like IComparable would support comparison operations.

Some programming languages use the term protocol instead of interface, since interfaces define an allowed set of interactions between components. In

some programming languages, such as Java, Objective-C, and C#, interfaces are supported directly with an explicitly defined syntax for the creation of interfaces. In other languages, such as C++, no special syntax exists for interfaces, but they can still be created by defining a class consisting only of pure virtual functions.

A class can be marked as an implementor of one or more interfaces. For example, consider Figure 1.2. Notice that Foo explicitly marks itself as an implementation of IFoo and IComparable. Foo also defines an implementation for all members found within IFoo (i.e. the Bar method) and IComparable (i.e. the CompareTo method). If Foo did not implement all the members defined within IFoo and IComparable, this would be an invalid program.

```
class Foo : IFoo, IComparable
{
    void Bar()
    {
        /* implementation of Bar */
    }

    int CompareTo(object other)
    {
        /* implementation of CompareTo */
    }
}
```

Figure 1.2: Foo Class

The same interface may be defined by multiple classes. Each class can define its own implementation of the interface. For example a program might define an ISerializable interface specifying methods for loading and saving an object from a stream. Several classes could implement ISerializable, but the

details of how to perform serialization could vary based on the structure of each implementor.

Interfaces are *abstract* types. Abstract types cannot be instantiated directly; instead they define the protocol that is supported by all implementors. Obtaining a reference to an interface type requires instantiating an implementor of the interface and assigning this instance to the interface reference. We can create instances of Foo and assign them to a reference variable of type IFoo, but we cannot create an instance of IFoo itself since IFoo does not specify an implementation. That requires an implementor, which in this case is Foo.

Since Foo implements IFoo, instances of Foo can be used wherever instances of a class implementing IFoo is expected without type error (1). In this way interfaces behave like base classes. In Figure 1.3, both Baz and Raz accept a parameter named foo. This parameter is known as a *collaborator* or *dependency*. A collaborator helps another unit complete its task. Likewise, since Baz and Raz cannot do their work without foo, they could be considered dependent of foo. In other words, foo is a dependency of Baz and Raz.

```
void Baz(IFoo foo) { /* implementation of Baz */ }
void Raz(Foo foo) { /* implementation of Raz */ }
```

Figure 1.3: Raz and Baz Methods

In both Baz and Raz the caller is required to specify the dependency supplied for foo. Alternatively these methods could create their own dependencies. In effect the Baz and Raz methods ask the caller for their dependencies, instead of looking for or creating their own. When dependencies

are provided by the caller, this is known as dependency injection or inversion of control. Dependency injection greatly improves the flexibility of software systems. Components that control the creation of their own dependencies fix the set of dependencies allowed to those created by the component itself. The scoping and lifetime of these dependencies are also fixed by the component. When a component uses dependency injection, the control of both what the dependency is and its scope and lifetime are controlled by the caller. This means that the same component can potentially be reused by many callers each having vastly different requirements. Each caller simply provides the dependencies it needs (2).

Baz can accept an instance of any type that implements the IFoo interface for the foo parameters. This means that the Baz method can collaborate with any implementor of IFoo. If a class that implements IFoo is later defined, Baz will be able to accept instances of this class as a parameter to foo without modification.

Raz by contrast can only collaborate with Foo or one of its subclasses. While subclassing certainly is a mechanism for extensibility, it has limitations. Many programming languages only support inheritance from a single base class, limiting extensibility to a single type hierarchy. In the example of Raz, only classes in a single hierarchy (the one that inherits from Foo) are suitable as a collaborator for the foo parameter. Even in languages that support multiple

inheritance issues such as the diamond problem may discourage its use. Sometimes a class will not allow subclassing at all.

Within a system, the degree to which components are aware of each other is known as *coupling*. Components that have a high degree of awareness of each other are *tightly coupled*, whereas components with a low degree of awareness are *loosely coupled*. Loose coupling is widely acknowledged as a characteristic of well designed systems. The Baz method above is loosely coupled to Foo. Neither Baz nor Foo is aware of the other—they share only the definition of the interface used to communicate between them.

Loose coupling enables code reuse. Existing components can gain new features and abilities by swapping out existing dependencies with new dependencies that implement the new capability. The original component can be reused. This enables programs that can support new features after they are written. A program can be extended or customized by someone other than the original author without modifying the original program.

Loose coupling also creates systems that are testable. A system is testable when a small part of the program can be ran and tested independently of the rest of the program. When a program is tightly coupled, running any component requires also running all of its dependencies. This makes testing difficult because many different components need to be tested together. Sometimes dependencies are slow, complex, difficult to setup, or have side-

effects that are undesirable for testing. With loose coupling these dependencies can easily be replaced by lightweight *mock* dependencies.

In Figure 1.4, Saz implements all the members defined on IFoo, however it does not explicitly mark itself as an implementation of IFoo. Instances of Saz therefore are not suitable for the foo parameter of Baz. Languages often require that classes explicitly mark their supported interfaces to avoid the risk that logically unrelated methods will inadvertently become correlated simply by virtue of their signatures matching. It is possible that the Bar method in Saz is logically unrelated to the Bar method defined in IFoo, even though their signatures do match and Saz implements all the members of IFoo. If the programmer that defined IFoo is also the author of Saz, there is little concern. The compiler would quickly make the programmer aware of his mistake. If Saz is logically related to IFoo, the programmer will then mark Saz as an implementor of IFoo and recompile.

```
class Saz
{
    void Bar()
    {
        /* implementation of Bar */
    }
}
```

Figure 1.4: Saz Class

A more difficult problem arises if Saz was defined within a third-party library. Software development projects often utilize libraries created by third-parties. These libraries are often presented as compiled binaries with no source

code provided. If the Saz class were defined within such a library, a third-party programmer could not mark that Saz implements his IFoo interface.

This becomes a serious issue when writing the Baz method. Although an instance of Saz could be otherwise suitable for the foo parameter, the fact that it is not and cannot be marked as an implementation of IFoo means that instances of Saz are unsuitable for the Baz method.

Despite the well known advantages of loose coupling that can be achieved by interfaces, many library authors are reluctant to use them extensively. The essential problem with interfaces for library implementors is versioning. As soon as a library using interfaces is published, third-party users may start implementing the interfaces in their own classes. When the next version of the library is released, any new methods added to an interface will break users of the library since classes implementing the interface will not support the new methods. This means that interfaces are essentially immutable when defined by authors of libraries used by third-party programmers.

Objective:

Loose coupling between types using interfaces provides flexible, testable, and extensible systems, but classes defined within third-party libraries frequently do not support interfaces because of versioning issues and cannot be modified because they are distributed in binary form. We will develop an approach that provides loose coupling to third-party libraries without reducing reliability, maintainability, tooling support, or performance.

Course of Action:

The existing approaches for solving this problem will be examined. A practical approach for adding type-safe static-context duck typing to an object-oriented programming language based on bytecode rewriting will be proposed. This approach should provide excellent reliability, tooling, maintainability, and performance.

Chapter 2. Background

Tight coupling does not allow the flexibility required for testable and extensible systems. When code is coupled directly, collaboration is restricted to a single type hierarchy, or in the case of classes that prohibit subclassing, to exactly one class. This greatly limits our ability to support new capabilities or swap out implementations with mock implementations for testing. Programmers end up writing tightly coupled code simply because it is typically the path of least resistance within a statically typed programming language.

In the following sections, well known techniques for achieving loose coupling are examined in detail. These techniques attempt to solve the problem outlined in the first chapter. We will examine the tradeoffs involved with each approach.

2.1. Wrapper Class

A *wrapper class* is a class that delegates calls to an internal object instance, possibly of a different type. A wrapper class can implement an interface by simply delegating calls to an instance of another class that does not implement the interface. For example, Figure 2.1 shows how we could define a wrapper class called SazWrapper that allows loose coupling with the Saz class defined in Figure 1.4. Since SazWrapper implements IFoo, instances can be passed as a collaborator to Baz. Internally SazWrapper holds a reference to an instance of Saz. Calls on SazWrapper simply delegate to the implementation on

Saz. This approach allows the loose coupling and extensibility of interfaces and the ability to reuse third-party classes while maintaining loose coupling.

```
class SazWrapper : IFoo
{
    Saz _saz;
    SazWrapper(Saz saz) { _saz = saz; }
    void Bar() { _saz.Bar(); }
}
```

Figure 2.1: SazWrapper Wrapper Class

Since SazWrapper implements IFoo, instances can be passed as a collaborator to Baz. Internally SazWrapper holds a reference to an instance of Saz. Calls on SazWrapper simply delegate to the implementation on Saz.

Wrapper classes eliminate the coupling problem, but they add maintainability and writability problems. The number of types within a system can dramatically increase when using wrapper classes. Large programs could reference hundreds of third-party classes, each requiring a wrapper class. Each of these wrapper classes must be written and maintained. The code within these wrapper classes does not add any new functionality to the software, it really only exists to satisfy the compiler. *The sole purpose of the SazWrapper class in Figure 2.1 is to convince the compiler that it is acceptable to use instances of Saz as an implementation of IFoo*.

Since wrapper classes must be manually created, implementation errors can lead to reliability problems. For example imagine that SazWrapper had instead been implemented as shown in Figure 2.2. The programmer forgot to

delegate the call to Bar to the implementor _saz. In general these kinds of errors cannot be detected by the compiler when wrapper classes are created manually.

```
class SazWrapper : IFoo
{
    Saz _saz;
    SazWrapper(Saz saz) { _saz = saz; }
    void Bar() { /* delegation to _saz forgotten */ }
}
```

Figure 2.2: Incorrect SazWrapper Wrapper Class

Wrapper classes allow excellent tooling support. Code completion editors and refactoring tools will work with wrapper classes. Since wrapper classes are no different than any other class defined within an application, code editors can provide helpful code completion hints and refactoring tools will correctly rename symbols defined within the wrapper classes.

Wrapper classes will have a minor impact on performance. The wrapper classes themselves will increase the code space required for an application. A small amount of extra memory usage will be required for the wrapper class instance. Calls through a wrapper class will also incur the run-time overhead of an additional virtual method call. This overhead is probably insignificant for most applications and hardware configurations.

2.2. Duck Typing

Another possibility is to consider an alternative typing strategy known as *duck typing*. Duck typing is named after the duck test (3). The duck test states: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably

is a duck." Duck typing applies this concept to typing within a programming language. With duck typing, type compatibility is determined by looking at the set of methods defined on an object rather than the classes it inherits from or the interfaces it implements.

Many programming languages have support for duck typing. Although most commonly associated with dynamic typing, duck typing actually takes one of two forms: *static-context duck typing* and *dynamic-context duck typing*.

2.2.1. Dynamic-Context Duck Typing

Dynamic-context duck typing involves determining at run-time whether a given object supports the methods that are actually called. At the point of run-time method invocation the variable is searched for a method matching the caller's signature. If the method is found it is invoked, otherwise a run-time error is thrown. Dynamically typed languages typically use dynamic-context duck typing, but some statically typed languages have special syntax causing the compiler to emit statements delaying the method binding until run-time.

Python's type system uses duck typing extensively. Figure 2.3 demonstrates Python's use of duck typing. Notice that the foo parameter to Baz has no explicit type name. The type of foo does not matter. If the parameter passed to foo has a Bar method it will be called, otherwise there will be an error.

```
class Saz:
    def Bar(self):
        print("Saz.Bar()")

def Baz(foo):
    foo.Bar()

Baz(Saz())
```

Figure 2.3: Dynamic-Context Duck Typing in Python

Objective-C supports both static and dynamic typing. Normally the type of a variable is specified and typing is static, but the programmer may use the keyword id in place of a type. This indicates that the variable uses dynamic typing, and an object of any type may be assigned to it (4). Figure 2.4 shows an example. Since we wish to employ dynamic-context duck typing within Baz, we specify id as the type for the foo parameter. When Baz calls the Bar method on foo, the run-time system determines if the object passed to Baz actually had a Bar method. If it does then Bar is invoked; otherwise Baz will raise an exception that will terminate the program unless correctly handled.

Dynamic-context duck typing is also possible with version 4.0 of C#. Figure 2.5 shows an example within C#. C# uses the keyword dynamic to indicate the object may be of any type and that it bypasses compile-time static typing (5). As with Objective-C, C# also supports static typing. Both of these languages intend for static typing to be used wherever possible, but enable switching to dynamic typing when static typing proves too difficult, inflexible, or inconvenient.

```
#import <Foundation/Foundation.h>

@interface Saz : NSObject
{
}

-(void) Bar;

@end

@implementation Saz

-(void) Bar {
    NSLog(@"[Saz Bar]");
}

@end

void Baz(id foo) {
    [foo Bar];
}

int main (int argc, const char * argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];

    Saz *saz = [[Saz alloc] init];
    Baz(saz);
    [saz release];

    [pool drain];
    return 0;
}
```

Figure 2.4: Dynamic-Context Duck Typing in Objective-C

```
using System;

class Saz
{
    public void Bar() { Console.WriteLine("Saz.Bar()"); }
}

class Program
{
    static void Baz(dynamic foo)
    {
        foo.Bar();
    }

    static void Main(string[] args)
    {
        Baz(new Saz());
    }
}
```

Figure 2.5: Dynamic-Context Duck Typing in C# 4.0

Dynamic-context duck typing is obviously extremely flexible. It is also extremely dangerous. Even small mistakes, such as a mistyped method name will not be discovered until run-time. It also means tools that depend on static type information will be ineffective. Code completion editors will be unable to offer hints, as the available methods will be unknown until run-time. Refactoring tools will be unable to identify required changes within dynamically typed variables. The loss of these features mean significant reduction in programmer productivity.

With dynamic-context duck typing the Baz method can be written to accept an object of any type. Only when Baz actually called the Bar method would the run-time system actually determine if the object passed to Baz actually

had a Bar method. If the method did not exist on the object a run-time error would occur.

Perhaps the biggest advantage of dynamic-context duck typing is the ease at which loose coupling is achieved. In a language such as Python, no extra work is required to achieve loose coupling. This makes components extremely flexible and testable. Unfortunately since checks that would normally be performed at compile-time in a statically typed language are delayed until run-time many more tests are required.

2.2.2. Static-Context Duck Typing

Static-context duck typing applies the duck test given the type information available at compile-time. C++ templates are a widely known example of static-context duck typing (6). The compiler will check that all methods called from a variable whose type is a template parameter are implemented by all types for which the template is expanded.

In Figure 2.6, Baz is a function template. The type of Baz's foo parameter is the type parameter T. Since a method named Bar is called off the foo parameter, all types passed to Baz as the type parameter T must support a Bar method. If Baz was passed a type that did not support a Bar method, a compile-time error would result. Also notice that the supported signature allowed for the Bar method is inferred by its usage and never explicitly defined. In the example a Bar method that returned a value would also be valid.

```
#include <iostream>
using namespace std;

// C++ interfaces are classes with no fields
// and only pure-virtual (abstract) methods
class IFoo
{
public:
    virtual void Bar() = 0;
};

class Saz
{
public:
    void Bar()
    {
        cout << "Saz::Bar()" << endl;
    }
};

template<typename T>
void Baz(T &foo)
{
    foo.Bar();
}

int main()
{
    Saz s;
    Baz<Saz>(s);

    return 0;
}
```

Figure 2.6: Static-Context Duck Typing in C++

Dynamic-context duck typing is slightly more flexible than the static-context duck typing within C++. With dynamic-context duck typing an object only needs to implement those methods actually called at run-time. With C++ static-context duck typing any method that might be called must be implemented, even

if that method does not get called at run-time. Additionally types unknown at compile time are unavailable.

Dynamic-context duck typing is slightly more flexible than the static-context duck typing within C++. With dynamic-context duck typing an object only needs to implement those methods actually called at run-time. With C++ static-context duck typing any method that might be called must be implemented, even if that method does not get called at run-time. Additionally types unknown at compile time are unavailable.

In Figure 2.7, an instance of Yaz is created and assigned to w. So the run-time type of w is Yaz but the compile-time type is Waz. Since the Waz class does not define a suitable implementation of the Bar method, w cannot be passed to the Baz method. If the checking was delayed until run-time, the Bar method would be found.

```
#include <iostream>
using namespace std;

class Waz
{
};

class Yaz : public Waz
{
public:
    void Bar()
    {
        cout << "Yaz::Bar()" << endl;
    }
};

template<typename T>
void Baz(T &foo)
{
    foo.Bar();
}

int main()
{
    Waz *w = new Yaz();
    Baz<Waz>(*w); // compiler-error
    delete w;

    return 0;
}
```

Figure 2.7: Static-Context Duck Typing in C++ Fails to Locate Methods Available only at Run-Time

Static-context duck typing enjoys reliability and performance advantages over dynamic-context duck typing. Since type checking is performed at compile-time, a mistyped method name can easily be corrected at compile-time. With dynamic-context duck typing, these errors might not occur until the user sees the program crash. Since static-context duck typing allows the binding to occur at

compile-time, there is less run-time performance overhead associated with the method lookup.

Since C++ static duck typing syntax infers the typed requirements based on usage it does have some limitations. Logical incongruities can happen if unrelated types happen to support the same methods. In Figure 2.8, CreditCard and Rhinoceros both have a Charge method, however they are completely unrelated. The FinalizePurchase function accepts a type parameter T. Since FinalizePurchase calls the Charge method off an instance of T, any parameter for T must have a Charge method. FinalizePurchase is meant to finalize a credit card transaction. No error will be reported if an instance of Rhinoceros is passed to FinalizePurchase, even though it will not produce the desired effect.

```
class CreditCard {
public:
    void Charge() {} // withdraws money
};

class Rhinoceros {
public:
    void Charge() {} // runs toward foe
};

// Accepts any reference that has Charge method, with
// the idea that some credit card classes do not derive
// from CreditCard
template<typename T>
void FinalizePurchase(T &card) {
    card.Charge();
    /* other side effect of charging credit card */
}

int main() {
    CreditCard visa;
    Rhinoceros rhino;

    FinalizePurchase<CreditCard>(visa);

    // compiles, but produces unintended effect
    FinalizePurchase<Rhinoceros>(rhino);

    return 0;
}
```

Figure 2.8: Logical Incongruity using Static-Context Duck Typing in C++

A related problem with the C++ static duck typing syntax is a lack of sufficient compile time type information for tooling. Since literally any call could be valid, code completion editors cannot possibly offer suggestions for variables whose type is supplied as a template parameter. Similarly automatic refactoring tools lack the type information needed to apply correct transformations. For example, if the Charge method within the CreditCard class were renamed, the

refactoring tool could not automatically infer that the call site within FinalizePurchase should also be renamed. Lack of code completion and refactoring tools significantly limit programmer productivity.

2.3. Dynamic Metaprogramming

Another way to solve the third-party library coupling problem within a statically typed programming language is via a metaprogramming (7) (8). With this approach program generates another dynamically linked program containing the required wrapper classes at run-time. Essentially this approach is another take on wrapper classes, but unlike the manually written wrapper classes from section 2.1, the wrapper class generation is done automatically by a library at run-time.

In Chapter 1, Baz required an instance of IFoo. Saz supports all the methods defined on IFoo, but is not marked as an implementation of IFoo. Since Saz is defined within a third-party for which we are provided only the binaries, we cannot simply mark Saz as an implementor of IFoo and recompile. Instead we call the dynamic_duck_cast function shown in Figure 2.9. This function dynamically generates a wrapper class like the SazWrapper shown in Figure 2.1.

```
T dynamic_duck_cast<T>(object obj)
{
 // 1. dynamically examine the methods supported on obj
 // 2. determine if obj contains all methods defined on T
 //    2a. otherwise generate error
 // 3. generate a library containing a wrapper class
 // 4. create an instance of the wrapper passing in obj
 // 5. return the instance of the wrapper
}

Baz(dynamic_duck_cast<IFoo>(new Saz()));
```

Figure 2.9: dynamic_duck_cast Method and Example Usage

Implementing dynamic_duck_cast requires several things. First we need to dynamically examine the set of methods supported by obj. This requires some kind of reflection. In other words, compile time metadata, such as the set of methods defined on a type, must be made available at run-time. This feature can be implemented manually, but is generally preferable if implemented by the programming language itself. This provides better assurances that the metadata will be available, accurate, and accessible in a standard way.

Since machine code is tied to a single machine architecture, any dynamic code generation is either tied to a specific machine architecture or requires an interpreted language. Alternatively, dynamically generated code can target a virtual machine architecture, thereby supporting any physical architecture that the virtual machine supports. Programming systems that run on a virtual machine architecture often support dynamic code generation.

Languages such as Java or .NET programming languages support all the features required to implement this metaprogramming implementation. Both

have a rich metadata systems. Since both run on virtual machines, platform-independent dynamic code generation is relatively easy. Both support dynamic code loading and linking.

The metaprogramming approach is good for productivity and maintainability. Since wrapper classes are generated automatically, programmer productivity is increased and maintenance costs are decreased. There is much less risk of errors within the wrapper class. Unlike the C++ syntax, duck types are created based on a named interface so tools requiring static type information, such as code completion editors and refactoring tools, are possible.

Unfortunately the metaprogramming approach suffers from poor reliability and performance. Since the dynamic metaprogramming approach is an example of dynamic-context duck typing, it has the reliability problems described in section 2.2.1. If the type parameter T passed to dynamic_duck_cast is incompatible with the object passed to obj, a run-time error will occur. In many cases all the information required to prevent this error is available at compile-time.

One advantage of this approach over the C++ syntax is that it suffers much less from the problem of logical incongruities resulting from types that happen to support the same methods. With the C++ approach, types could accidentally support the implicit interface required on the type parameter. With the metaprogramming approach the programmer is required to make an explicit dynamic_duck_cast. Of course the programmer could still make the same

mistake allowed by C++, but it would be much less likely because of the explicit cast. Figure 2.10 shows dynamic_duck_cast using the credit card and rhinoceros example from Figure 2.8.

```
interface ICreditCard {
    void Charge() {} // withdraws money
}

class CreditCard {
    void Charge() {} // withdraws money
};

class Rhinoceros {
    void Charge() {} // runs toward foe
};

void FinalizePurchase(ICreditCard card) {
    // full static type tooling now possible:
    //  - rename on Charge is possible
    //  - code completion for card is possible
    card.Charge();
}

int main() {
    CreditCard visa = new CreditCart();
    Rhinoceros rhino = new Rhinoceros();

    FinalizePurchase(dynamic_duck_cast<ICreditCard>(visa));

    // The following error is less likely than in C++. The
    // programmer must explicitly say that a Rhinoceros
    // is suitable as an implementor of ICreditCard.
    FinalizePurchase(
          dynamic_duck_cast<ICreditCard>(rhino));

    return 0;
}
```

Figure 2.10: Logical Incongruity using dynamic_duck_cast (Java or C# like language)

Since the metaprogramming dynamic-context duck typing approach generates wrapper classes at run-time it can incur a significant overhead. Although wrapper class types can be cached after their initial generation, each call to dynamic_duck_cast with a new type parameter will cause new code to be dynamically compiled. This approach also requires analysis of reflection metadata at run-time to guarantee type compatibility. Even with caching of wrapper class types this approach can cause noticeable delays during code generation.

Chapter 3. Design

As seen in the previous chapter there are many existing approaches that achieve loose coupling to third-party libraries within object-oriented programming languages, however these approaches suffer from reliability, tooling, maintainability, and performance problems. Having examined several approaches, we now design an approach that enables loose coupling between components while still enabling high reliability, good tooling support, high maintainability, and relatively good performance.

We describe two possible implementations for the design. The first, language modification, is most suitable for language designers and implementors and will be explained but not implemented. The second, bytecode rewriting, is suitable as an extension to an existing language and will be explained and implemented.

3.1. Language Modification

One approach for enabling loose coupling to third-party components is to extend the language.

Figure 3.1 shows an example of a language extension that allows for static-context duck typing by adding an asduck operator to the language. The asduck operator has two parameters. The first is a variable reference that precedes the asduck operator. The second parameter is an interface type name that follows the operator. At compile time, the variable's compile time type is

examined for duck compatibility. If class X implements all members defined on interface Y, but X is not necessarily marked as an implementation of Y, then X is *duck compatible* with Y. If the interface type parameter is not duck compatible with the compile-time type of the variable a compile-time error occurs, otherwise the operator simply returns an instance of the interface type parameter.

```
interface IFoo {
   void Bar();
};

class Saz {
    void Bar() {}
};

void Baz(IFoo foo) {
    foo.Bar();
}

int main() {
    Saz s = new Saz();
    Baz(s asduck IFoo); // new asduck cast operator

    return 0;
}
```

Figure 3.1: Language Modification (Java or C# like language)

This language feature could be implemented by having the compiler generate a wrapper class like SazWrapper shown in Figure 2.1. Unlike the metaprogramming approach, language modification would not require run-time reflection metadata. The wrapper classes are generated by the compiler, which will have access to compile-time metadata even if this is not available at run-time. Additionally this would not require generation of code at run-time, so a virtual machine architecture or dynamic module loading would not be required.

These reduced requirements might make this approach more feasible for languages without the required metadata and run-time code generation features.

Like the metaprogramming approach, language extension is good for maintainability. Wrapper classes are automatically generated by the compiler, so no programmer maintenance is required. The overall amount of code required to achieve loose coupling is reduced. Generally with all other things being equal, given two programs that have the same behavior the one that has less code will be more maintainable because there is simply less code to maintain. Since wrapper classes are automatically generated by the compiler the risk of programmer error within these classes is greatly reduced.

The biggest advantage of the language modification approach is probably reliability. Since duck compatibility is determined at compile-time, duck compatibility errors cannot occur at run-time. This is a huge advantage over dynamic-context duck typing. It is also a big advantage over the metaprogramming approach where accidentally using a type that was not duck compatible resulted in a run-time error. With language modification these errors are detected earlier. It is generally accepted that detecting errors earlier improves reliability.

The language modification approach is good for performance. The overhead is equal to that of manually writing wrapper classes; that is, one extra virtual method call for each duck type method invocation. Virtual method invocations are certainly slower than non-virtual calls, but virtual method calls are

much faster than the dynamic lookup typically required for dynamic-context duck typing. Additionally since the wrapper classes are generated at compile time, language modification does not incur the cost of determining duck type compatibly and dynamic code generation at run-time.

Language modification as proposed here is an example of static-context duck typing. As shown earlier static-context duck typing is slightly less flexible than dynamic-context duck typing, but has better reliability. For most cases static-context duck typing is flexible enough, and for those cases in which it is not a programmer could use an existing language feature or library for dynamic-context duck typing. Since static and dynamic context duck typing are not mutually exclusive, this language modification is not less flexible than dynamic-context duck typing; it is just more reliable whenever applicable.

Although the language modification approach enjoys the possibility of tooling such as code completion editors and refactoring tools, any existing tooling for a language would be broken causing frustration and reduced productivity for programmers accustomed to such tooling. Of course these tools could be updated, but the likelihood of this occurring depends on who makes the update.

If these language changes were made outside a controlling standards body for the language, it would cause a branch in the language—essentially creating a whole new language. This new language's evolution would then split from the original. Any improvements made to the original language, compilers, or tooling would either not be available to users of the new language or require a

difficult, and perhaps unlikely, reunification. Programmers may fear these extensions because of their potential to isolate the programmer from the larger community of the original language.

Language modification is probably most suitable if designed by the group controlling the language specification and implemented by the major compiler vendors for the language. If designed and implemented at this level, the language modifications would enjoy excellent tooling support. Future additions to the language would also include the duck typing extensions. Compilers themselves are complex pieces of software and making these language modifications could be quite difficult.

Changing a programming language with a large user base is an incredibly difficult task. Small changes in the language can have subtle and unanticipated consequences. Adding new keywords might break existing programs that happened to use that keyword for a variable name. Many compiler vendors will need to update their compiler software. Tooling infrastructure will also require modification. After release, an error in a language specification is difficult to correct. Programs will have already been written against the flawed specification, and simply correcting the flaw could break these programs.

Features added directly to a language can make the language more difficult to understand. Even if a programmer is unaware of a new feature, he or she might still encounter code written by other programmers that makes use of a feature. At this point the programmer will probably need to study and understand

this new feature. With every feature the amount of study required to master all the language's features increases. While language features can improve the ease of expressing ideas within the language, they also increase the time required by programmers to fully understand the language. Since language features are likely to be encountered by all users of a language, features only useful to a subset of the language's community should probably be implemented in libraries whenever possible.

Language extensions have some compelling advantages, but ultimately they are only practical if designed by the body in control of the language.

3.2. Bytecode Rewriting

Another approach for supporting static-context duck typing is compile-time metaprogramming or bytecode rewriting. Bytecode rewriting is a technique for implementing features that would normally require language modification within a library. Unlike the language modification implementation described in section 3.1, this approach is practical even if done outside the body controlling a language.

Many languages have a relatively high-level bytecode instruction set. For example Java and .NET define virtual machines supporting high-level instruction sets. This bytecode serves as a machine independent intermediate language. Language compilers targeting the virtual machine run-time output this bytecode rather than machine code. At run-time the virtual machine takes the bytecode instructions and compiles them to native machine code. For performance, native

code can be cached to avoid re-compilation or interpretation costs for subsequent calls. This process is known as just-in-time compilation and is implemented on the primary virtual machines for both Java and .NET.

With bytecode rewriting a specially designed tool takes another program as an input. The tool disassembles the input program, changes it in some way, then reassembles the program as output. During this manipulation phase custom features normally requiring compiler support can be implemented. Although theoretically this technique could be employed for languages outputting machine code directly, bytecode preserves just enough of the original code structure to make this process much easier. The bytecode rewriter can easily identify elements within the compiled source program that it wishes to modify.

On the surface this approach looks much like dynamic-context metaprogramming. In fact the API as used by the programmer could look exactly the same. The programmer has a library that allows him to request a duck for some compatible interface type. Unlike the dynamic metaprogramming approach, bytecode rewriting requires the the object's compile time type to be duck compatible with the target interface type. The API might look something like Figure 3.2.

```
TDestination
static_duck_cast<TDestination, TSource>(TSource source)
{
    throw error;
}
```

Figure 3.2: static_duck_cast Method

Under the surface the bytecode rewriting approach is much different. A program author that wishes to use duck typing within his project would need to include the duck typing library. This is exactly the same as what would be required with the dynamic metaprogramming approach, but the implementation of this library is dramatically different. Unlike the implementation overviewed in Figure 2.9, this library does nothing. The program built against the static duck typing library may be compiled using any compiler targeting the bytecode format supported by the bytecode rewriting tool. So a bytecode rewriting tool supporting the Common Intermediate Language (CIL) bytecode format used by .NET could rewrite programs written in C#, Visual Basic.NET, or any other .NET language. After compilation of the source program, it is sent to the bytecode rewriting tool—ideally as a post build operation.

The bytecode rewriting tool examines the input program looking for call sites where a duck type is requested. When a call site is found the following operations are performed:

1. Determine the members supported on instances of the source type (TSource).
2. Determine the members required on instances of the destination type (TDestination).
3. Determine if the source type contains compatible members for all members on the destination type.
    a. otherwise generate an error

4. Generate a wrapper class implementing all members of TDestination by delegation to the compatible member found on the source type contained within the wrapper. Add this wrapper type to the output program.
5. Modify all call sites to create a duck (e.g. static_duck_cast) to create an instance of the generated wrapper class, passing the source instance to the wrapper class.
6. Write out the modified program.

Although these steps are fairly similar to the dynamic metaprogramming approach, there are a few important differences. First, all these steps are performed as part of the build process. The steps required for dynamic metaprogramming are performed at run-time. Second, since the bytecode rewriting is performed at compile-time, the actual run-time members supported on the instance of TSource supplied for the source parameter are not examined, only the members defined on TSource are considered when checking for duck compatibility. Third, and perhaps most importantly, the errors determined in step 3a are reported at build-time. Any incorrect usage can be easily corrected by the programmer at this point before the error reaches the user. Fourth, all of the overhead associated with performing these operations occurs at build-time. With the dynamic metaprogramming approach, the overhead of determining duck compatibility and code generation was left until run-time. The only overhead still present at run-time is that of the wrapper class. While the wrapper class

certainly will add overhead, it is probably by far the lowest run-time overhead of any of the methods discussed for achieving loose coupling to third-party libraries that do not define interfaces. In fact this approach generates code that is basically indistinguishable from the manually written wrapper classes, with equally little run-time overhead.

3.2.1. Novelty

The novelty of this approach lies in its migration of wrapper class generation from run-time to build-time. This allows us to keep the maintenance advantages found with automatically generated wrapper classes without introducing any new reliability or performance issues. Duck compatibility errors that would occur at run-time with the dynamic metaprogramming approach are shown to the programmer sooner, increasing the reliability of software. Additionally the performance impact related to run-time code generation is mitigated by performing the code generation at build-time.

3.3. .NET Implementation

We now examine a .NET example implementation created using the bytecode rewriting implementation of the static-context duck typing design presented within section 3.2.

3.3.1. Tools

The following tools and libraries were utilized in the creation of this .NET implementation.

MSIL Assembler

MSIL[1] Assembler is a tool written by Microsoft designed to take a text file containing MSIL instructions and produce a compiled executable or library (9).

Figure 3.3 shows a simple hello world program written in CIL. The MSIL Assembler takes a text file like Figure 3.3, containing CIL instructions, and outputs a compiled binary executable or library containing bytecode instructions.

```
.class private auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main() cil managed
  {
    .entrypoint
    .maxstack  8
    IL_0000:  ldstr "Hello World!"
    IL_0005:  call void
            [mscorlib]System.Console::WriteLine(string)
    IL_000a:  ret
  }
}
```

Figure 3.3: Hello World in CIL (abridged)

MSIL Disassembler

MSIL Disassembler is a tool written by Microsoft designed to take a compiled .NET program as an input and produce a text file containing MSIL instructions (10). This output can be provided as an input to MSIL Assembler.

Given a compiled hello world program written in any .NET language MSIL Disassembler would produce an output similar to Figure 3.3.

[1] The bytecode format used within the .NET Framework was originally known as Microsoft Intermediate Language (MSIL). During the standardization of .NET Framework components, MSIL was renamed Common Intermediate Language (CIL), but several tools still refer to CIL by its original name.

DeftTech.DuckTyping

DeftTech.DuckTyping is a library written by David Meyer (7). This library implements dynamic-context duck typing for the .NET framework as described in section 2.2.1.

System.Text.RegularExpressions

System.Text.RegularExpressions is a regular expression processing library built into the .NET Framework. It can search input text for specific patterns using a special regular expression syntax.

Red Gate's .NET Reflector

.NET Reflector is a tool that takes compiled .NET software and reverse compiles it to several source languages (11). The tool applies heuristics based on knowledge of how compilers perform CIL code generation. This tool can take a compiled .NET program and show an approximation of the original source.

3.3.2. Components

The following components were built for the .NET implementation.

DuckTyping.Contracts

DuckTyping.Contracts is a dynamic-link library (DLL) written in C# that contains the contracts allowing a programmer to request a duck type. This DLL contains the .NET implementation of the static_duck_cast operator shown in Figure 3.2. The source listing of this component is shown in Figure 3.4.

The AsDuck method uses the C# extension method syntax. In C# an extension method is a static method that may be called using instance method syntax (12). In .NET all types derive from System.Object, which is aliased by the C# keyword object. Since the AsDuck method extends System.Object, it may be called as an instance method from any object so long as the DuckTyping.Contracts library is referenced.

```
using System;

namespace DuckTyping.Contracts
{
    public static class DuckType
    {
        public static TDestination
        AsDuck<TSource, TDestination>(this TSource source)
            where TDestination : class
        {
            throw new NotImplementedException();
        }
    }
}
```

Figure 3.4: DuckTyping.Contracts

Programs that use this .NET implementation must reference the DuckTyping.Contracts library during compilation, but the library is not needed at run-time. This library provides the hooks that are used by the bytecode rewriting tool.

ilrewrite

ilrewrite is the bytecode rewriting tool for the .NET implementation. This tool takes a compiled .NET program as its input. The program takes this input

program and calls the MSIL disassembler to get a textual representation of the input program's CIL code. Next the tool uses a regular expression to search the CIL for sites where the AsDuck extension method from the DuckTyping.Contracts DLL is called. At each call site the source and destination types are examined for duck compatibility using the DeftTech.DuckTyping library. If the destination type is not duck compatible with the source type an error is recorded, otherwise the DeftTech.DuckTyping library is used to dynamically create a DLL containing a wrapper class. The wrapper contains an instance of the source type and implements the destination interface though delegation to the source type instance. Each call site to the AsDuck method within the input program is then replaced with an instantiation of the appropriate wrapper class. The source parameter to AsDuck is passed as the input argument to the wrapper class. Next the MSIL disassembler is again called for each of the wrapper class DLLs generated by the DeftTech.DuckTyping library. The CIL code for each wrapper class is merged into the output CIL. Finally the output CIL code is passed to the MSIL assembler to create the compiled output software.

3.3.3. Example

The program from Figure 3.6 references a third party library. The source code for this library is shown in Figure 3.5, but the programmer that wrote the code in Figure 3.6 does not have access to this source. The programmer was given only the compiled binary file that contains Saz, so he cannot simply mark Saz as an implementor of IFoo.

```
using System;

public class Saz
{
    public void Bar()
    {
        Console.WriteLine("Saz.Bar()");
    }
};
```

Figure 3.5: Source Code of Third-Party Library Containing Saz (C# language)

To maintain loose coupling the programmer defined the IFoo interface and wrote Baz so it is coupled only to the IFoo interface. Since the Saz class is not marked as an implementor of IFoo, the saz reference cannot be directly passed to Baz. It is possible to call the AsDuck extension method shown in Figure 3.4.

```
using System;
using DuckTyping.Contracts;

public interface IFoo {
    void Bar();
};

static class Program {
    static void Baz(IFoo foo) {
        foo.Bar();
    }

    static void Main() {
        var saz = new Saz();
        Baz(saz.AsDuck<Saz, IFoo>());
    }
};
```

Figure 3.6: Example Input Program (C# language)

If the programmer were to build and run the program shown in Figure 3.6 it would crash at run-time upon encountering the call to the AsDuck extension

method. Recall from the definition of AsDuck shown in Figure 3.4 that it simply throws a NotImplementedException. Instead however the programmer takes the output program created from compiling Figure 3.6 and passes it as an input to ilrewrite. Ideally this is an automatic post build step in the build script.

```
.method private hidebysig static void  Main() cil managed
{
.entrypoint
.maxstack  1
.locals init ([0] class [ThirdParty]Saz saz)
IL_0000:  nop
IL_0001:  newobj instance void [ThirdParty]Saz::.ctor()
IL_0006:  stloc.0
IL_0007:  ldloc.0
IL_0008:  call !!1 [DuckTyping.Contracts]
    DuckTyping.Contracts.DuckType::
    AsDuck<class [ThirdParty]Saz,class IFoo>(!!0)
IL_000d:  call void Program::Baz(class IFoo)
IL_0012:  nop
IL_0013:  ret
}
```

Figure 3.7: CIL Listing of Main method within Figure 3.6 before Transformation

In Figure 3.7 the CIL code from the Main method in Figure 3.6 is shown as ilrewrite would see the compiled input program. The call to AsDuck as well as the type parameters are clearly visible in the assembly listing. In this case the source type is Saz (defined within the library named ThirdParty) and the destination is IFoo. ilrewrite uses a regular expression to locate this call site and extract the type parameters.

```
internal class DuckProxy_Saz : IFoo {
    private readonly Saz duck;

    internal DuckProxy_Saz(Saz A_1) {
        this.duck = A_1;
    }

    public sealed override void Bar() {
        this.duck.Bar();
    }
}
```

Figure 3.8: Generated DuckProxy_Saz Wrapper Class (C# Language) (abridged)

After locating the call sites, ilrewrite calls into the DeftTech.DuckTyping library to determine type compatibility. This library was originally written to perform dynamic-context duck typing at run-time, but was modified slightly to perform static-context duck typing at compile-time. When ilrewrite runs, any duck-compatibility problems are reported. Since Saz implements all the methods defined on the IFoo interface, there are no duck incompatibilities so the bytecode rewriting process continues.

For each unique source/destination type pair within the input program ilrewrite generates a wrapper class. The wrapper class created from the call to AsDuck within Main in Figure 3.6 is shown in Figure 3.8. The wrapper class is shown as C# code after reverse compiling the generated class from the compiled CIL using Red Gate's .NET Reflector.

The actual call site where the AsDuck extension method is called is replaced by an instantiation of the wrapper class. Since CIL is a stack-based assembly language, and since both the wrapper class constructor and the

AsDuck method both take a single argument of type TSource and return an object of type TDestination, it is possible to simply replace the call to the AsDuck method with a newobj instruction using the wrapper class constructor. This works because both statements have the same stack semantics. Figure 3.9 shows the CIL after being transformed by ilrewrite. Figure 3.10 shows Figure 3.9 as C# code, again using Red Gate's .NET Reflector to reverse compile the output program.

```
.method private hidebysig static void  Main() cil managed
{
.entrypoint
.maxstack  1
.locals init ([0] class [ThirdParty]Saz saz)
IL_0000:  nop
IL_0001:  newobj instance void [ThirdParty]Saz::.ctor()
IL_0006:  stloc.0
IL_0007:  ldloc.0
IL_0008:  newobj instance void DuckProxy__ThirdParty_Saz::
    .ctor(class [ThirdParty]Saz)
IL_000d:  call void Program::Baz(class IFoo)
IL_0012:  nop
IL_0013:  ret
}
```

Figure 3.9: CIL Listing of Main method within Figure 3.6 after Transformation

```
static void Main()
{
    Saz saz = new Saz();
    Baz(new DuckProxy_Saz(saz));
}
```

Figure 3.10: Listing of Main Method within Figure 3.6 after Transformation (C# Language)

After determining duck type compatibility, merging in generated wrapper classes, and modifying AsDuck call sites, the modified CIL listing is passed to the

MSIL Assembler and reassembled into a binary executable or library. The code can now be ran without error. All of the run-time overhead of determining duck type compatibility and generating wrapper classes has been performed at build-time. Any duck compatibility errors are detected at build-time and will not occur at run-time. Since all references to the DuckTyping.Contracts DLL have been stripped from the output code it is no longer needed at run-time.

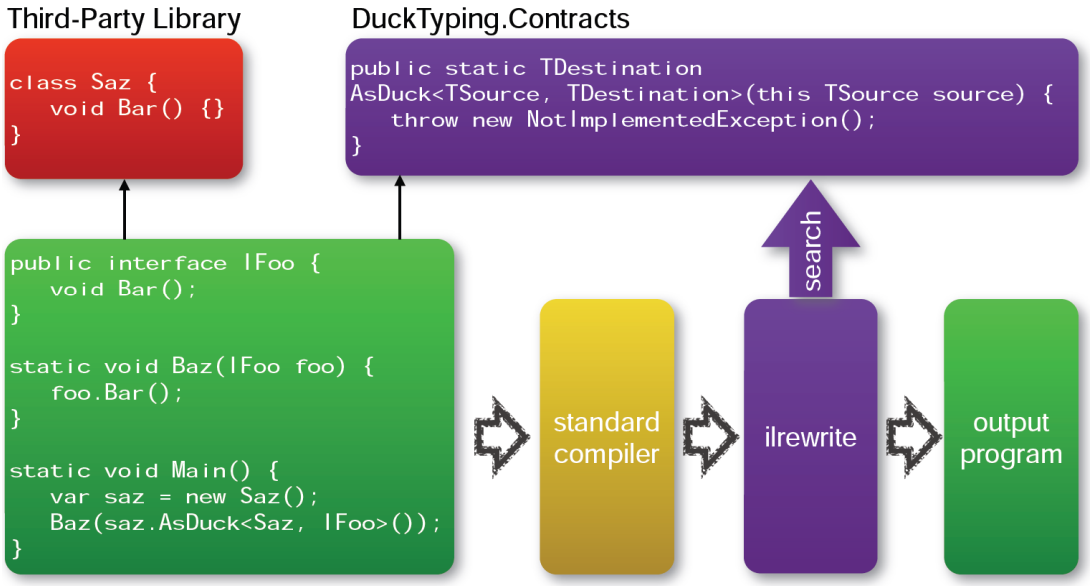

Figure 3.11: Bytecode Rewriting Data Flow

Chapter 4. Analysis

4.1. Reliability

The earlier an error or defect can be detected the more reliable a system will be (13) (14). When an error is detected at compile-time it eliminates any possibility of it occurring at run-time. An error detected at compile-time is much easier and less costly to correct than an error discovered at run-time.

In this subsection, the types of errors that may occur for each of the approaches described in the proceeding chapters are explained. For each approach, the timing (run-time versus compile-time) and possibility of the error is examined to assess the reliability of each approach.

4.1.1. Variable Usage Error

Each variable within a program supports a set of operations. If the programmer specifies an operation that is not supported by the variable an error will result. The reliability of the system is better if these errors can be detected during compile-time.

Techniques that do not detect variable usage errors at compile-time are subject to serious reliability issues. Even an extremely simple error such as a typographical error could easily result in a run-time error. In a large system it is quite possible that some of these errors will be missed during testing and affect the user.

Wrapper Class

Figure 4.1 shows a program using a wrapper class with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo, but foo does not support the Gar method. Compiling this program gives the following compile-time error:

```
'IFoo' does not contain a definition for 'Gar'
```

```
static class Program {
    static void Baz(IFoo foo) { foo.Gar(); }
    static void Main() {
        Baz(new SazWrapper(new Saz()));
    }
}
```

Figure 4.1: Incorrect Variable Usage with Wrapper Class (C# Language)

Dynamic-Context Duck Typing

Figure 4.2 shows a program using dynamic-context duck typing with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo, but foo does not support the Gar method. This program compiles successfully despite the incorrect variable usage.

```
static class Program {
    static void Baz(dynamic foo) { foo.Gar(); }
    static void Main()
    {
        Baz(new Saz());
    }
}
```

Figure 4.2: Incorrect Variable Usage with Dynamic-Context Duck Typing (C# Language)

When the program in Figure 4.2 executes, the following run-time error crashes the program:

```
'Saz' does not contain a definition for 'Gar'
```

Although this is basically the same error as in section 4.1.1, the fact that it is delayed until run-time hurts the reliability of software written using dynamic-context duck typing.

Static-Context Duck Typing

Figure 4.3 shows a program using static-context duck typing with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo. Baz is passed Saz as its type parameter, but since Saz does not have a Gar method the compiler produces the following compile-time error:

```
'Gar' : is not a member of 'Saz'
```

```
#include <iostream>
using namespace std;

class Saz {
public:
  void Bar() { cout << "Saz::Bar" << endl; }
};

template <typename T>
void Baz(T &foo) { foo.Gar(); }

int main() {
  Saz s;
  Baz(s);

  return 0;
}
```

Figure 4.3: Incorrect Variable Usage with Static-Context Duck Typing (C++ Language)

Metaprogramming

Figure 4.4 shows a program using metaprogramming with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo, but foo does not support the Gar method. Compiling this program gives the following compile-time error:

```
'IFoo' does not contain a definition for 'Gar'

static class Program {
    static void Baz(IFoo foo) { foo.Gar(); }
    static void Main() {
        Baz(DuckTyping.Cast<IFoo>(new Saz()));
    }
}
```

Figure 4.4: Incorrect Variable Usage with Metaprogramming (C# Language)

Language Modification

Figure 4.5 shows a program using the language modification described in section 3.1 with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo, but foo does not support the Gar method. Although a compiler implementing this approach has not been created, it would produce something like the following compile-time error:

```
'IFoo' does not contain a definition for 'Gar'
```

```
static class Program {
    static void Baz(IFoo foo) { foo.Gar(); }
    static void Main() {
        Baz(new Saz() asduck IFoo);
    }
}
```

Figure 4.5: Incorrect Variable Usage with Language Modification

Bytecode Rewriting

Figure 4.6 shows a program using bytecode rewriting with an incorrect usage of the variable foo. Within the Baz method the Gar method is called from foo, but IFoo does not have a Foo method causing the compiler to produce the following compile-time error:

```
'IFoo' does not contain a definition for 'Gar'
```

```
static class Program {
    static void Baz(IFoo foo) { foo.Gar(); }
    static void Main() {
        var saz = new Saz();
        Baz(saz.AsDuck<Saz, IFoo>());
    }
};
```

Figure 4.6: Incorrect Variable Usage with Bytecode Rewriting (C# Language)

4.1.2. Duck Compatibility Error

A class is duck compatible with an interface if it implements all members defined on the interface. If a variable treated as a duck does not support the interface required by the duck variable, an error will occur. For dynamic-context duck typing the interface is not explicitly defined, but the set of methods called from the variable could be considered the interface.

Duck compatibility errors may be detected at compile-time or run-time. Discovering duck incompatibilities at compile-time improves reliability because it eliminates a possible run-time error. A programmer who accidentally supplied an incompatible type would see the error during compilation when it can be fixed easily.

Wrapper Class

Figure 4.7 shows a program with a duck compatibility error. The programmer attempted to pass an instance of Paz to the foo parameter of Baz by creating the wrapper class PazWrapper that implements the IFoo interface required for foo.

```
public class Paz { }

class PazWrapper : IFoo {
    private readonly Paz paz;
    public PazWrapper(Paz paz) { this.paz = paz; }
    public void Bar() { this.paz.Bar(); }
}

static class Program {
    static void Baz(IFoo foo) { foo.Bar(); }
    static void Main() {
        Baz(new PazWrapper(new Paz()));
    }
}
```

Figure 4.7: Duck Incompatibility with Wrapper Class (C# Language)

With correctly written wrapper classes, duck incompatibilities manifest themselves as wrapper classes that will not compile. In the case of Figure 4.7, attempting to compile results in the following compile-time error:

```
'Paz' does not contain a definition for 'Bar'
```

occurring within the Bar method within the PazWrapper class. This should make sense as Paz is not duck compatible with IFoo.

Dynamic-Context Duck Typing

Figure 4.8 shows a program with a duck compatibility error. An instance of Paz is passed to the Baz method for the foo parameter. Although the foo parameter's type is dynamic and therefore has no explicit interface requirements, by looking at the body of the method we can see that Baz requires the object passed to foo to support a parameterless Bar method. Since the program in Figure 4.8 uses dynamic-context duck typing, it compiles without error.

```
class Paz { }

static class Program {
    static void Baz(dynamic foo) { foo.Bar(); }
    static void Main() {
        Baz(new Paz());
    }
}
```

Figure 4.8: Duck Incompatibility with Dynamic-Context Duck Typing (C# Language)

Since Paz is not duck compatible with the implicit 'interface' required by Baz, a duck compatibility error will occur and crash the program. The following error occurs at run-time:

```
'Paz' does not contain a definition for 'Bar'
```

Static-Context Duck Typing

Figure 4.9 shows a program using static-context duck typing containing a duck compatibility error. A reference to an instance of Paz is passed to the foo parameter of the Baz method. Although the Baz method is templated to accept a foo parameter of any type, since the Bar method is called off from foo, only types that support a Bar method are allowed. Compiling the program in Figure 4.9 results in the following error message:

```
'Bar' : is not a member of 'Paz'
```

confirming that with static-context duck typing duck compatibility errors are reported at compile-time.

```
#include <iostream>
using namespace std;

class Paz {};

template <typename T>
void Baz(T &foo) { foo.Bar(); }

int main() {
  Paz p;
  Baz(p);

  return 0;
}
```

Figure 4.9: Duck Incompatibility with Static-Context Duck Typing (C++ Language)

Metaprogramming

Figure 4.10 shows a program with a duck compatibility error. An instance of Paz is cast to the IFoo interface from Figure 1.1 using the metaprogramming library described in section 3.3.1, but Paz does not implement the members of IFoo. This creates a run-time error that crashes the program with the following error:

```
Duck type does not implement a method named "Bar"
with compatible parameters and return type.
```

```
public class Paz { }

static class Program {
    static void Baz(IFoo foo) { foo.Bar(); }
    static void Main() {
        Baz(DuckTyping.Cast<IFoo>(new Paz()));
    }
}
```

Figure 4.10: Duck Incompatibility with Metaprogramming (C# Language)

Language Modification

Figure 4.11 shows a program with a duck compatibility error. An instance of Paz is cast using the asduck operator to the IFoo interface with the language modification approach described in section 3.1. Since Paz does not implement all the members defined on IFoo it is not duck compatible. Although an actual implementation of the language modification approach was not created, the following shows the kind of compile-time error that would be reported if such a compiler existed:

```
     'Paz' is not duck compatible with 'IFoo'

public class Paz { }

static class Program {
    static void Baz(IFoo foo) { foo.Bar(); }
    static void Main() {
        Baz(new Paz() asduck IFoo);
    }
}
```

Figure 4.11: Duck Incompatibility with Language Modification

Bytecode Rewriting

Figure 4.12 shows a program written using the bytecode rewriting technique that contains a duck compatibility error. The program attempts to create a duck for an instance of Paz that implements the IFoo interface, but since Paz does not implement all the methods in IFoo (namely the Bar method), it is not duck compatible with IFoo.

```
public class Paz {}

static class Program
{
    static void Baz(IFoo foo) { foo.Bar(); }
    static void Main() {
        var paz = new Paz();
        Baz(paz.AsDuck<Paz, IFoo>());
    }
};
```

Figure 4.12: Duck Incompatibility with Bytecode Rewriting (C# Language)

Although the program in Figure 4.12 will compile, the compiled program is immediately sent to the ilrewrite program. This program determines that Paz is not duck compatible with IFoo and reports the following error:

```
cannot create duck type for Paz implementing
interface IFoo
```

Since this error occurs during the program's overall build process, bytecode rewriting detects duck compatibility errors at compile-time.

4.1.3. Logical Type Mismatch

It is possible for logically unrelated classes to be duck compatible. In Figure 2.8 one such example was presented involving CreditCard and Rhinoceros classes both having a Charge method. Programming languages typically require that classes explicitly mark themselves as interface implementors to avoid such accidental type compatibility.

Duck typing provides a mechanism that allows programmers to avoid the checks that normally would prevent accidental structural type compatibility. As such there is a potential for reliability issues if a programmer accidentally uses a type that, although being duck compatible, is logically unrelated. So long as the programmer is required to explicitly specify when duck type compatibility is desired the reliability concerns are largely mitigated. When duck casts are explicit the programmer is more likely to identify accidental compatibility, such as with the CreditCard and the Rhinoceros, and the compiler does not need to make any assumptions about the intent of the programmer.

Wrapper Class

Figure 4.1 shows how the assignment of an instance of Saz to IFoo during the call to Baz requires an explicit instantiation of the SazWrapper wrapper class.

Dynamic-Context Duck Typing

Figure 4.2 shows how the assignment of an instance of Saz to the foo parameter of type dynamic is implicit when using dynamic-context duck typing.

Static-Context Duck Typing

Figure 4.3 shows how the assignment of an instance of Saz to the foo parameter of type T is implicit when using static-context duck typing.

Metaprogramming

Figure 4.4 shows how the assignment of an instance of Saz to the foo parameter is explicit when using metaprogramming.

Language Modification

Figure 4.5 shows how the assignment of an instance of Saz to the foo parameter of type IFoo requires an explicit asduck cast when using language modification.

Bytecode Rewriting

Figure 4.6 shows how the assignment of an instance of Saz to the foo parameter of type IFoo requires an explicit call to the AsDuck extension method providing both the source and destination types.

4.1.4. Wrapper Class Implementation Error

Manually written wrapper classes have the potential to be written incorrectly, causing reliability problems. When wrapper classes are generated automatically by a library or language, implementation errors are much less likely. If they do occur, fixing them once in the library or language averts the bug for all generated wrapper classes. Figure 2.2 shows an example of a wrapper class containing an implementation error.

Wrapper class implementation errors are a reliability problem unique to wrapper classes. All other techniques either generate wrapper classes automatically or do not use wrapper classes.

4.1.5. Reliability Summary

The language modification and bytecode rewriting techniques have the best reliability since they do not suffer from any of the reliability problems described in this section.

4.2. Tooling

4.2.1. Tooling Based on Static Type Information

Many software development tools utilize static type information to assist the programmer with various activities. These tools work most reliably and predictably when based on static type information.

A code completion editor is a tool that uses static type information. It provides the programmer with a list of supported operations for a given variable. These tools analyze the static type information to determine what options should be shown. If no static type information is available for the variable in question, these tools cannot provide any suggestions as any operation could potentially be valid.

Another useful tool that uses static type information is an automatic refactoring system. These systems are able to automatically rename all occurrences of a type or operation. Instead of an error-prone textual find and

replace, these tools work symbolically based on static type information. The tools can only locate symbols deterministically when the bindings are static. When bindings are dynamic they may be undecidable or ambiguous.

Tools based on static type information are effective for all techniques except dynamic-context duck typing described in section 2.2.1 and static-context duck typing described in section 2.2.2. Tools based on static type information are ineffective for dynamic-context duck typing because no static type information exists when variables are dynamically typed.

A few examples show how tools based on static type information are ineffective for the static-context duck typing described in section 2.2.2. In Figure 4.13, the set of operations allowed on the foo parameter within the Baz method is completely undefined. Since the Baz template is never expanded within the program, literally any operation could potentially be valid on foo.

```
#include <iostream>
using namespace std;

template <typename T>
void Baz(T &foo) {
  // what operations are legal on foo?
}

int main() {
  return 0;
}
```

Figure 4.13: Ineffective Static Type Information (C++ Language)

In Figure 4.14, the Bar method in Baz is statically bound to two Bar symbols (the one defined in Saz1 and the other defined in Saz2). If attempting to

rename Bar from Baz, Saz1, or Saz2 it is unclear what effect this should have on the other Bar symbols. If they are all renamed the rename may be unintentionally broad. If only a single symbol is renamed the program becomes invalid.

4.2.2. Segments/Breaks Existing Tooling

The primary disadvantage unique to the language modification approach from section 3.1 is that it breaks existing tooling and segments the language's user community. This disadvantage is explained fully in section 3.1.

```
#include <iostream>
using namespace std;

class Saz1 {
public:
  void Bar() { cout << "Saz1::Bar" << endl; }
};

class Saz2 {
public:
  void Bar() { cout << "Saz2::Bar" << endl; }
};

template <typename T>
void Baz(T &foo) { foo.Bar(); }

int main() {
  Saz1 s1;
  Baz(s1);

  Saz2 s2;
  Baz(s2);

  return 0;
}
```

Figure 4.14: Ambiguous Rename (C++ Language)

4.2.3. Increased Build Time

Large programs can frequently take a long time to build. All things being equal, a shorter compilation time is obviously preferable. The dynamic-context duck typing and metaprogramming approaches perform the bulk of their operations at run-time instead of compile-time. These approaches are therefore preferable for decreasing build times.

All the other approaches increase build times. Wrapper classes increase the amount of code to compile, static-context duck typing increases the amount

of code to compile because of template expansions, language modifications increase build times because the amount of work performed by the compiler increases, and bytecode rewriting increases build times because the ilrewrite tool must run as a post-build operation. Of these approaches bytecode rewriting is likely to take the most time because it requires a separate process to run after the build. Instead of running inside the compiler itself bytecode rewriting must re-read the compiled software modify it and re-write the modified software.

Comparing measurements of build times for each of the various approaches would not be appropriate. The primary implementation concern for the bytecode rewriting implementation described in section 3.3 was programmer efficiency in creating the tool—not its compilation speed. If the tool was tweaked for performance such an evaluation would be meaningful.

4.2.4. Tooling Summary

The dynamic metaprogramming approach has the best tooling support. This was the only approach that worked well for all three of the tooling evaluation criteria.

4.3. Maintainability

More code within any software, especially code that provides no intrinsic functionality, is detrimental to maintainability. This follows simple logic—the more code within a system, the more code subject to change during maintenance, and the more expensive the maintenance becomes.

Additional code can also affect readability, an important consideration for maintainability. When many statements are required to express a single succinct idea a maintainer has much more code to read, increasing the cost of maintenance. When the same template or pattern is repeated because of limitations or restrictions within a language the cost of maintenance is increased.

	Per Type	Per Duck Cast
wrapper class	one line per interface member	One statement to instantiate wrapper
dynamic duck	none	none
static duck	none	none
metaprogramming	none	one statement to request duck type
language modification	none	one asduck cast operator
bytecode rewriting	none	one statement to call AsDuck method

Table 4.1: Maintenance Cost from Lines of Code

Maintenance costs are very important in software development. It has been estimated that the cost of maintenance can be four times the cost of development (15).

Table 4.1 shows the additional code required for each approach. One additional statement per duck cast is of little concern to maintainability. Although this does slightly increase the number of lines, it also makes the code more readable by explicitly specifying the desired type compatibility.

The maintenance cost of adding a new wrapper class type for every interface and then a line of code for each member defined on that interface is a much greater concern. These classes add a whole new type and many lines of code that provide no intrinsic functionality; furthermore each wrapper class

expresses the same delegation pattern. The additional maintenance cost makes wrapper classes less maintainable than other approaches.

4.4. Performance

Performance can be analyzed both theoretically, by examining the number and types of operations required, and empirically, by measuring the real world performance. Theoretical analysis provides a logical framework that should explain real world result and is less dependent on a multitude of implementation factors. Empirical performance measurements provide evidence that demonstrates the correctness of the logical model, at least under the conditions under which the measurements were performed.

Performance can be measured in terms of both speed and space. Speed is related directly to the number and type of operations that occur at run-time. Assuming that each operation takes the same amount of time, the process requiring the fewest operations will perform fastest.

4.4.1. Virtual Method Calls

Table 4.2 shows the approaches that require an extra virtual method call. Approaches that require virtual method calls will incur this run-time overhead when calling methods through a duck.

Although virtual method calls require virtual table lookups at run-time, the cost is usually minimal. In C++ for example, virtual method calls require five

more memory references than non-virtual method calls (16). The run-time overhead from an extra virtual method call is likely to be very minimal.

Approach	Required
wrapper class	X
dynamic duck	
static duck	
metaprogramming	X
language modification	X
bytecode rewriting	X

Table 4.2: Virtual Method Call Requirements

4.4.2. Call Site Interpretation

With dynamic-context duck typing each method call will require run-time interpretation. A variable using dynamic type binding using pure interpretation typically takes at least ten times longer than the equivalent machine code (17). In some implementations, such as the implementation found within .NET, rather than pure interpretation, the dynamic type binding is compiled to machine code at run-time and cached, so most of the overhead is paid when the code executes for the first time with new types rather than for each execution (18).

This performance issue is unique to dynamic-context duck typing. None of the other approaches require run-time call site interpretation.

4.4.3. Run-Time Code Generation

The metaprogramming technique described in section 2.3 requires run-time code generation. At run-time the library builds a dynamic module by emitting the bytecode instructions and types required to delegate calls off the target interface to an instance of the source type. This module is built in-

memory. Creating this module requires inspecting the methods available on the source type for compatible calls on the target interface, and then emitting the appropriate instructions for delegation. After the module is built, it is loaded into the current process and an instance of the generated wrapper class is created.

Although this run-time code generation is a fairly expensive process, the resulting wrapper classes and dynamic modules can easily be cached, so the process is only required once per source/target pair. When duck types are requested and there is a cache hit, the performance overhead is approximately that of a hash table search. This is typically quite fast, although probably slower than a virtual method call.

Run-time code generation is mostly unique to metaprogramming. The other approaches do not require run-time code generation, although the dynamic-context duck typing approach may use run-time code generation as an optimization technique to avoid the cost of call site interpretation for each execution.

4.4.4. Run-Time Type Checking

All of the approaches other than dynamic-context duck typing and metaprogramming use static type checking. Static type checking reduces the run-time overhead of performing type checking. Although dynamic-context duck typing and metaprogramming both have overhead related to run-time type checking they are slightly different.

With dynamic-context duck typing the run-time type checking is actually an aspect of the call site interpretation from section 4.4.2. One of the important tasks that must be performed during run-time call site interpretation is type checking. Any usage errors are reported during this run-time interpretation.

With metaprogramming, run-time type checking assures that the source and destination types in the duck cast are duck compatible. This was explained in section 4.1.2 with respect to its detrimental affects on reliability, but so to does it negatively affect performance. The source and destination types specified by the duck cast must be examined for duck compatibility at run-time. As described in section 4.4.3 the results of this operation can be cached to reduce the performance impact for additional duck casts involving the same types.

4.4.5. Empirical Results

All empirical measurements were obtained on a machine with the specifications shown in Table 4.3. The programs were written in C# version 4.0 using the Microsoft C# compiler version 4.0.30319.1 on a release build. The programs were compiled to machine code before execution to avoid the effects of the .NET just-in-time bytecode compiler. Timings were performed by a high-performance timer with resolution to around 279 nanoseconds.

Processor	Intel Core Duo T2500 at 2.00GHz
Operating System	Windows XP Professional SP 3
Memory	2.00 GB

Table 4.3: Test Machine Details

Call Performance

To measure the performance of method calls for each of the approaches a simple test program was written. The program compared a call to a virtual method (through an interface), a call to a non-virtual method, a virtual method call through a dynamic variable (call site interpretation), a non-virtual method call through a dynamic variable (call site interpretation), and a call to a method through a wrapper class. Measurements were made by placing each type of call within a loop executing 100 million times. The timing was done outside this loop, using the high performance timer. The timing of an empty loop was also measured so the execution time incurred from the loop itself could be removed. Measurements were taken fifty times and the min, max, and mean averages are shown in Table 4.4.

virtual method		
• mean	4.18 ns	
• min	4.15 ns	
• max	4.24 ns	
non-virtual method		
• mean	3.17 ns	
• min	3.15 ns	
• max	3.22 ns	
dynamic variable to virtual method		
• mean	53.16 ns	
• min	52.95 ns	
• max	54.92 ns	
dynamic variable to non-virtual method		
• mean	53.07 ns	
• min	52.90 ns	
• max	53.48 ns	
call through wrapper		
• mean	6.21 ns	
• min	6.18 ns	
• max	6.30 ns	

Table 4.4: Call Performance

Metaprogramming

To measure the performance from the metaprogramming approach, a test program was written. This test program contained one-thousand interface definitions similar to the IFoo interface seen in Figure 1.1. A class like Saz from Figure 1.4 was written. For each of the one-thousand interfaces, a cast from an instance of Saz to the interface was performed. The timing for all of the 1000 calls was measured using the high-performance timer. Since the metaprogramming library (7) used caching to avoid recomputing duck compatibility and regenerating wrapper class libraries for previously encountered types, the timing was measured twice. The first time each of the one-thousand types was new and not in the cache. During the second iteration the type was

found in the cache. These measurements were taken fifty times and the mean averages are shown in Table 4.5.

cache hit	0.037 ms
cache miss	1.08 ms

Table 4.5: Metaprogramming Duck Cast Performance

4.4.6. Performance Summary

The C++ static duck typing has the best theoretical performance. Language modification, bytecode rewriting, and wrapper classes all have excellent performance.

4.5. Summary

Table 4.6 provides a summary of the analysis contained within this chapter. Each column shows an evaluation criterion within its evaluation category (i.e. Reliability, Tooling, Maintainability, Performance). Each row represents one of the considered approaches. For each evaluation criterion a plus (+) symbol is assigned to a generally positive or beneficial characteristic and a negative symbol (–) is assigned to a generally negative or detrimental characteristic.

	2a				2b			2c	2d			
	i	ii	iii	iv	i	ii	iii	i	i	ii	iii	iv
1a	+	+	+	–	+	+	–	–	–	+	+	+
1b	–	–	–	+	–	+	+	+	+	–	+	–
1c	+	+	–	+	–	+	–	+	+	+	+	+
1d	+	–	+	+	+	+	+	+	–	+	–	–
1e	+	+	+	+	+	–	–	+	–	+	+	+
1f	+	+	+	+	+	+	–	+	–	+	+	+

Table 4.6: Analysis Summary

1. Approaches
   a. Wrapper Class (section 2.1)
   b. Dynamic-Context Duck Typing (section 2.2.1)
   c. Static-Context Duck Typing (section 2.2.2)
   d. Metaprogramming (section 2.3)
   e. Language Modification (section 3.1)
   f. Bytecode Rewriting (section 3.2)
2. Evaluation
   a. Reliability
      i. Variable Usage Error
         + Detected at Compile-Time
         – Detected at Run-Time
      ii. Duck Compatibility Error
         + Detected at Compile-Time
         – Detected at Run-Time
      iii. Logical Type Mismatch
         + Requires Explicit Cast or Wrapper Class
         – Happens Implicitly
      iv. Wrapper Class Implementation Error
         + Unlikely (handled by tool) or Not Applicable
         – Possible (handled by programmer)
   b. Tooling
      i. Tooling Based on Static Type Information
         + Effective
         – Ineffective
      ii. Segments/Breaks Existing Tooling
         + No
         – Yes
      iii. Increases Build Time
         + No
         – Yes
   c. Maintainability
      i. Wrapper Class Maintenance
         + Not Required
         – Required
   d. Performance
      i. Virtual Method Call
         + Not Required
         – Required
      ii. Call Site Interpretation
         + Not Required
         – Required
      iii. Run-Time Code Generation
         + Not Required
         – Required
      iv. Run-Time Type Checking
         + Not Required
         – Required

As can be seen in Table 4.6, bytecode rewriting has the most advantages and fewest disadvantages.

Chapter 5. Findings

5.1. Conclusions

5.1.1. Reliability

The analysis shows that the approaches described in the design are better for reliability. The two approaches described in the design were the only approaches able to detect all of the reliability errors described in section 4.1 at compile-time.

For most applications reliability is a very important concern. Good reliability is always a concern for software, whereas performance is primarily a concern only when the software is unacceptably slow.

5.1.2. Maintainability

The only approach that had serious maintainability issues was wrapper classes. All other approaches including those described in the design did not suffer from the maintainability problems caused by wrapper classes.

5.1.3. Tooling

The metaprogramming approach described in section 2.3 is the best approach in terms of the tooling criteria considered. Wrapper classes and bytecode rewriting both have excellent tooling support with their only downside being increased build times. The increase in build time from wrapper classes is likely to be less than the increase from the metaprogramming approach.

Although the metaprogramming approach is better than wrapper classes and bytecode rewriting in terms of tooling alone, it is extremely unlikely that build speeds are more important than reliability.

Bytecode rewriting is also compatible with a hybrid approach combining the fast build times enjoyed by metaprogramming and the excellent reliability found with bytecode rewriting. With this hybrid approach metaprogramming would serve as a fallback if the bytecode rewriting post-build step had not been performed. Programmers might choose not to perform the bytecode rewriting for every debug build. The bytecode rewriting could still be performed for release builds and on demand.

5.1.4. Performance

Theoretically the C++ static duck typing described in section 2.2.2 should have the best performance since it does not require virtual method calls, call site interpretation, run-time code generation, or run-time type checking. Since this form of static-context duck typing is not supported within the C# language where the rest of empirical results were measured, the performance of the approach was not measured directly, but it would have call performance equivalent to non-virtual method calls from Table 5.1.

Language modification, bytecode rewriting, and wrapper classes all enjoy excellent performance. All of these methods either explicitly or implicitly use wrapper classes. Wrapper classes have a slight performance hit since they require one extra virtual method call. Table 5.1 shows the actual time

measurements for virtual methods. As can be seen in the table, a virtual method call took only 4.18 nanoseconds on the test machine. Calls to non-virtual methods through a wrapper class took 3.04 nanoseconds longer, about twice as long as calling the non-virtual method directly. For most applications the time required for the virtual call itself will be an extremely tiny percentage of the overall time required by the method.

virtual method	4.18 ns	
non-virtual method	3.17 ns	
dynamic variable to virtual method	53.16 ns	
dynamic variable to non-virtual method	53.07 ns	
call through wrapper	6.21 ns	

Table 5.1: Call Performance

The metaprogramming approach has a significant performance impact during a duck cast, especially when the wrapper class does not already exist within the cache. Although the performance impact of creating a wrapper for a new type is fairly significant, the fact that this only needs to be done once per type provides a dramatic speedup. After the duck creation, each call through the duck has the same performance as language modification, bytecode rewriting, and wrapper classes. The performance measurements for duck casting is shown in Table 5.2.

cache hit	0.037 ms
cache miss	1.08 ms

Table 5.2: Metaprogramming Duck Cast Performance

With dynamic-context duck typing using dynamic variables there is no big up front performance hit required for generating a wrapper class, but there is

much more overhead for each call through a dynamic variable. Table 5.1 shows the overhead for calls through dynamic variables.

5.1.5. Overall

Our original goal was to develop an approach that provides loose coupling to third-party libraries without reducing reliability, maintainability, tooling support, or performance. The bytecode rewriting technique described in section 3.2 is the best overall technique for achieving this goal. These findings support our original thesis.

The only downsides to bytecode rewriting were build times and the performance impact from an additional virtual method call.

Although short build times are always desirable, they are usually less of a consideration than other issues. Also as described in the tooling section above, a hybrid metaprogramming and bytecode rewriting approach can easily mitigate build time issues.

As for performance the only impact caused by bytecode rewriting was an additional virtual method call. Although virtual method calls are somewhat slower than non-virtual method calls, they are still extremely fast. The performance overhead of the bytecode rewriting technique will only be an issue in extreme edge cases for which programming languages based on bytecodes are probably inappropriate anyway.

Many of the most widely used programming languages, such as Java, C#, and Visual Basic.NET, are based on bytecode systems. The bytecode rewriting

technique enables a reliable, maintainable, productive, and performant means to achieve loose coupling to third-party classes within these languages.

5.2. Future Work

While the bytecode rewriting technique is designed to be applicable to all languages based off bytecodes, it was only implemented for .NET languages. Another implementation, perhaps based on the Java Virtual Machine, would help to show the generality of this approach.

The bytecode rewriting tool built for .NET languages could be written for better performance. The implementation described in section 3.3 used existing tools based on input and output files. The rewriter's performance could be greatly improved by removing its reliance on these external tools and performing its disassembly, transformation, and reassembly in place and in memory. In addition, breaking the dependency on tools within the .NET Framework would also allow the tool to work with other .NET implementations, such as Mono. The contracts DLL could also be written to do metaprogramming if the bytecode rewriter is not run. This would greatly reduce the impact of bytecode rewriting's build time since the rewriting could be skipped for quick debug builds.

One of the implementation approaches described in the design section was language modification. As described in section 3.1, this approach is more suitable for the standards body in charge of the language then as an extension so it was not implemented. It was suggested that compiler generated classes could be the implementation used by the compiler, but perhaps an even better

implementation that avoided the run-time cost of wrapper classes could be designed using the language modification approach. Perhaps these approaches could be considered for inclusion into popular programming languages.

Bibliography

1. **Liskov, Barbara.** Keynote address - data abstraction and hierarchy. Orlando, Florida, United States : ACM, 1987, pp. 17-34.

2. **Wolter, Jonathan, Ruffer, Russ and Hevery, Misko.** Guide: Writing Testable Code. [Online] [Cited: June 14, 2010.] http://misko.hevery.com/attachments/Guide-Writing Testable Code.pdf. pp. 3-15.

3. **Davis, Robin S.** *Who's Sitting on Your Nest Egg?* s.l. : BookPros, LLC, 2007. p. 7.

4. **Apple.** Introduction to The Objective-C Programming Language. [Online] [Cited: May 25, 2010.] http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocObjectsClasses.html. p. 1.

5. **Microsoft.** *C# Language Specification.* [Online] 1999-2010. [Cited: April 24, 2010.] p.90 (1.29.3). http://www.microsoft.com/downloads/details.aspx?familyid=DFBF523C-F98C-4804-AFBD-459E846B268E&displaylang=en.

6. **Koenig, Andrew and Moo, Barbara E.** Templates and Duck Typing. *Dr. Dobb's.* [Online] June 1, 2005. [Cited: June 20, 2010.] http://www.drdobbs.com/cpp/184401971. pp. 1-4.

7. **Meyer, David.** Duck Typing Project. [Online] [Cited: May 2, 2010.] http://www.deftflux.net/blog/page/Duck-Typing-Project.aspx. pp. 1-2.

8. **Povey, Dean.** Duck Typing in Java using Dynamic Proxies. [Online] November 13, 2008. [Cited: May 30, 2010.] http://thinking-in-code.blogspot.com/2008/11/duck-typing-in-java-using-dynamic.html. pp. 2-8.

9. **Microsoft.** MSIL Assembler (Ilasm.exe). [Online] [Cited: May 1, 2010.] http://msdn.microsoft.com/en-us/library/496e4ekx(VS.80).aspx. p. 2.

10. —. MSIL Disassembler (Ildasm.exe). [Online] [Cited: May 1, 2010.] http://msdn.microsoft.com/en-us/library/f7dy01k1(VS.80).aspx. p. 2.

11. **Red Gate Software.** .NET Reflector, class browser, analyzer and decompiler for .NET. [Online] [Cited: June 5, 2010.] http://www.red-gate.com/products/reflector/. p. 1.

12. **Microsoft.** Extension Methods (C# Programming Guide). [Online] [Cited: June 1, 2010.] http://msdn.microsoft.com/en-us/library/bb383977.aspx. p. 2.

13. **Ben-Ari, M.** Understanding Programming Languages. 1996, p. 30.

14. **Sebesta, Robert W.** Concepts of Programming Languages, Seventh Edition. s.l. : Addison-Wesley, 2005, pp. 16-17.

15. **Sommerville, Ian.** Software Engineering. s.l. : Pearson Education Limited, 2004, p. 494.

16. **Stroustrup, Bjarne.** What Is Object-Oriented Programming? s.l. : IEEE Software, 1988, pp. 10-20.

17. **Sebesta, Robert W.** Concepts of Programming Languages, Seventh Edition. s.l. : Addison-Wesley, 2005, p. 215.

18. **Microsoft.** Inside C# 4.0 (video). [Online] Channel 9, November 31, 2008. [Cited: June 8, 2010.] http://channel9.msdn.com/shows/Going+Deep/Inside-C-40-dynamic-type-optional-parameters-more-COM-friendly. p. 2.