7-1-2016

Transfuse: A Compile-Time Metaprogramming Solution for Reducing Boilerplate on Google's Android

John Ericksen

John C. Ericksen

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Darko Stefanovic , Chairperson

Melanie Moses

Patrick Kelly

Transfuse: A Compile-Time Metaprogramming Solution for Reducing Boilerplate on Google's Android

by

John C. Ericksen

B.S., Computer Science, Western Washington University, 2004

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2016

Dedication

To my friends, family and colleagues.

Transfuse: A Compile-Time Metaprogramming Solution for Reducing Boilerplate on Google's Android

by

John C. Ericksen

B.S., Computer Science, Western Washington University, 2004

M.S., Computer Science, University of New Mexico, 2016

Abstract

Modern Java application development makes use of metaprogramming to offset and reduce application boilerplate. Unfortunately, metaprogramming techniques typically require a relatively high run-time cost, particularly at application startup. Therefore, environments with limited resources or without the luxury of a warm-up period, often lack metaprogramming as an option. This is precisely the case with applications written for Google Android. Android applications run on low resource mobile hardware and lack an offline startup period. Therefore, Android applications often suffer from a high amount of boilerplate.

Fortunately, there is an alternative to the traditional metaprogramming approach. In this thesis, we examine the approach of a metaprogramming tool named Transfuse. Transfuse targets boilerplate reduction within the constraints prescribed by the Android environment. This is accomplished through compile-time analysis and code generation. This approach is analyzed from both boilerplate reduction and run-time performance perspectives.

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

2 Background 5

2.1 Java 5

2.1.1 Reflection 6

2.1.2 Bytecode 7

2.2 Unit Testing 8

2.3 Design Patterns 9

2.4 Dependency Injection 10

2.5 Android 14

3 Methods: Transfuse 19

3.1 Introducing the Transfuse framework 19

3.2 Dependency Injection 20

3.2.1 Dependency Graph Loops 22

3.2.2 Providers 23

3.2.3 Scoping 25

3.3 Android Integration 26

3.3.1 API Injections 27

3.3.2 Lifecycle Events 28

3.3.3 Call-through Methods 29

3.3.4 Android Manifest 29

3.4 Technique 31

3.5 Limitations 34

3.5.1 Static Dependency Injection Graph 34

3.5.2 Android Abstraction 34

4 Results **35**

4.1 Compile-time vs Run-time Execution 35

4.2 Boilerplate Reduction 38

4.2.1 Object Graph Boilerplate 38

4.2.2 Example Application Boilerplate 39

4.3 Comparison to Other Libraries 40

5 Conclusion **44**

Appendices **47**

A Data collected comparing run-time and compile-time execution **48**

B Data collected comparing libraries **49**

C Run-time and compile-time experiment code **51**

References **57**

List of Figures

1.1 A simple Android example compared with an equivalent application outfitted with transfuse. 3

2.1 An example of Java Annotations. 6

2.2 Proxy Design Pattern 10

2.3 Factory Design Pattern 11

2.4 Example DAO injection and usage 12

2.5 Example DAO Factory 12

2.6 Constructor Injection Example 13

2.7 Field Injection Example 13

2.8 Method Injection Example 13

2.9 Dependency Injection Graph 15

2.10 Android Manifest Example 16

3.1 Transfuse Dependency Injection Graph Code 21

3.2 Virtual Proxy Example 22

3.3 Transfuse Generated Graph Code 24

3.4 Transfuse Provider Code 25

3.5 Transfuse Activity with lifecycle events. 29

3.6 Transfuse Generated Activity Graph. 30

3.7 Transfuse annotated Activity 32

3.8 Transfuse annotated Module 32

4.1 Mean execution time 36

4.2 Ratio of execution times 37

4.3 JSR330 Injection Graph 39

4.4 Mean startup time 42

4.5 Startup time ratios 43

C.1 Experiment code to test the run-time of instantiating an instance via a cached class using reflection. 51

C.2 Experiment code to test the run-time of instantiating an instance via a cached constructor using reflection. 52

C.3 Experiment code to test the run-time of invoking a cached method using reflection. 52

C.4 Experiment code to test the run-time of getting a field value using reflection via a cached Field value. 53

C.5 Experiment code to test the run-time of setting a field value using reflection via a cached Field value. 53

C.6 Experiment code to test the run-time of instantiating a class directly using the new keyword. 53

C.7 Experiment code to test the run-time of directly calling a method. 54

C.8 Experiment code to test the run-time of directly getting a field value. . . 54

C.9 Experiment code to test the run-time of directly setting a field value. . . 54

C.10 Experiment code to test the run-time of directly instantiating a class. . . 54

C.11 Experiment code to test the run-time of directly instantiating a class. . . 55

C.12 Experiment code to test the run-time of instantiating an instance via a constructor using reflection. 55

C.13 Experiment code to test the run-time of invoking a method using reflection. 55

C.14 Experiment code to test the run-time of getting a field value using reflection. 55

C.15 Experiment code to test the run-time of setting a field value using reflection. 56

C.16 Bash script to count the number of lines of Java code 56

List of Tables

4.1 Line count of hand-written and generated source code 40

A.1 Mean execution time 48

A.2 Ratio of execution times 48

B.1 Execution Run-time to Start Application 49

B.2 Execution Run-time to Start Application Statistics 49

B.3 Ratio of execution times 50

Chapter 1

Introduction

Android applications often suffer from a key deficiency: application startup is very apparent to the end-user. This is in direct contrast with application startup on a web server, where startup could take minutes, and is not apparent to the user. In a server environment, startup time is inconsequential as long as the application is ready to respond (warm) to user requests by access time. An Android application's startup time begins as soon as the application icon is touched. Therefore, the time required by tasks in an Android's application startup critical path must be reduced to a minimum.

Definition 1 (Metaprogramming)**.** *Software that translates one input high-level computer program into another output low-level program.*

Definition 2 (Boilerplate)**.** *Code that serves little purpose other than satisfying system (platform, API, language) integration requirements. Often this code is seen as redundant, repetitive and unnecessary even though it may be critical to the operation of an application.*

Developers have come to expect the functionality metaprogramming libraries offer to speed application development. Unfortunately, the cornerstone techniques used by

metaprogramming libraries (reflection and bytecode generation) in the traditional server environment are relatively slow. As a result, application developers must make the decision between using metaprogramming libraries and quick startup and execution in the Android environment. The decision here is clear: while metaprogramming libraries are a convenience to the developer, minimal startup time is a necessity. Therefore, metaprogramming libraries that require a high startup time are unacceptable in an Android application. However, shedding metaprogramming techniques has a direct side effect; the developer is forced to introduce high amounts of boilerplate.

The goal of this thesis is to present and analyze a new technique for reducing boilerplate code through a metaprogramming tool named Transfuse[1, 2] on Google's Android platform[3]. Transfuse implements common metaprogramming techniques found in reflection-based libraries while preserving the critical run-time performance required of resource limited systems. Specifically, Transfuse performs metaprogramming analysis and code generation at compile-time. This is in contrast to the traditional methods of run-time analysis and run-time code generation. The result is minimal overhead during run-time startup and execution, and a significant reduction in boilerplate code. Previous techniques[4, 5, 6] focused on domain specific languages to facilitate metaprogramming. With Transfuse, the goal is to stay within the boundaries of the Java language.

Metaprogramming is an effective tool used to reduce boilerplate in source code, facilitating the translation of one high-level program description into another low-level implementation. Through this mechanism, a developer can avoid the low-level boilerplate by relying on a metaprogramming tool, like Transfuse, to make the necessary additions and changes. In essence, the metaprogramming tool manages the boilerplate, freeing the developer from the associated maintenance and readability problems. However, metaprogramming comes with a cost. Namely, metaprogramming tools require computation time to analyze the input high-level program and time to generate or configure the output low-level implementation code. This overhead is traditionally incurred during the startup run-

```
public class ExampleActivity❶ extends Activity {

  @Override protected void onCreate(Bundle state ) {
❷ setContentView(R.layout . display );
❸ Button button = (Button)findViewById(R.id. button );
   TextView textView = (TextView)findViewById(R.id. displayText );
❹ Service service = Service . getInstance ();

    button . setOnClickListener (new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        textView . setText ( service . getText () );
      }
    });
  }
}

public final class Service {
  private static Service SINGLETON;
  private Service () { /* private Singleton constructor */}

❺ public static synchronized Service getService () {
    if (SINGLETON == null) {
      SINGLETON = new Service();
    }
    return SINGLETON;
  }

  public String getText () {
    // ...
  }
}
```

```
❻ <manifest package="org.example" android:versionCode="1"
      xmlns:android="http ://.../ android">
    <application>
      <activity android: label="Home" android:name=".ExampleActivity"/>
        <intent− filter >
          <action android:name=".... MAIN"/>
          <category android:name=".... LAUNCHER"/>
        </intent− filter >
      </activity >
    </application >
</manifest>
```

(a) Native Android Example

```
❼ @Activity(name = "Home")
  @Layout(R.layout.display )
  public class ExampleActivity {
❽ @Inject @View(R.id.displayText) TextView textView;
  @Inject Service service

❾ @RegisterListener(R.id . button )
  OnClickListener listener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      textView . setText ( service . getText () );
    }
  });
}

❿ @Singleton
  public class Service {
    public String getText () {
      // ...
    }
  }
```

(b) Equivalent Code using Transfuse Metaprogramming

Figure 1.1: A simple Android example compared with an equivalent application outfitted with transfuse. This example shows a 33:18 reduction in the amount of hand written code.

❶ Android components extend a base class. This gives the developer access to the Android lifecycle and services but means the Activity component inherits a lot of baggage from the super class.

❷ Layouts are associated with the Activity within the onCreate lifecycle method.

❸ Widgets within the view are referenced by calling findViewById with the given View identifier. Unfortunately, this also requires a cast to the given View type in order to use the specific type's functionality.

❹ Singletons are manually looked up using a getInstance method call.

❺ To enforce a singleton is unique within the VM, this boilerplate is commonplace to only instantiate a singleton once.

❻ Android applications define an AndroidManifest.xml file to register components in the application. Notice this file is not required with a Transfuse enhanced application as Transfuse writes this file referencing the classes annotated with @Activity.

❼ Transfuse Activity components are declared using the @Activity annotation and assigned a layout using the @Layout annotation. Each Transfuse component may be a Plain-Old-Java-Object (POJO) meaning there is no requirement to extend a base class. This removes the boilerplate baggage inherited from extending a base class.

❽ Views may be injected using @Inject and the @View qualifier. Behind the scenes, a call is made to findById with the given qualifier value with the result cast to the assumed type. This saves a lookup method call and a cast of the return value to the desired View type.

❾ Assigning handlers using the @RegisterListener avoids a handful of calls including referencing the view directly and choosing to which method to assign the handler object.

❿ Singletons, and other scoped types, are defined using the @Singleton annotation. Transfuse manages the instance's state, injecting only one instance illuminating the boilerplate typically present in Singleton implementations.

time of an application. For applications where startup time is critical, as with Google's Android, this overhead is unacceptable. Therefore, applications with resource or performance constraints often are riddled with boilerplate to avoid the problem of lengthy start-up times.

Modern Object Oriented Design encourages the use of Design Patterns.[7] Design Patterns help developers partition and organize code into commonly found structures that communicate the intent of a class to other developers. Design Patterns can be useful in organizing code, but often they increase the amount of plumbing code necessary in an application. Plumbing code is code that organizes, brings together and manages the state of an application. Plumbing code is considered boilerplate as it exists only for integration of the parts of an application. Metaprogramming can assist with the plumbing of an application via Dependency Injection by managing the structure and state of an application.

Applied to Android, this metaprogramming solution allows the developer to focus less time on system integration and more time on business use cases resulting in a syntactically simple and semantically readable application.[8] By shedding the boilerplate of an Android application, the true intent of an application becomes clear and the maintainability is significantly increased.

Chapter 2

Background

Explaining why metaprogramming on Android is valuable requires an understanding of various supporting subjects. To fill in this background, the following section provides details on the larger topics that support Android programming. This starts with a dive into the high-level structure of Java constructs; in particular Java classes, reflection, and bytecode. These constructs are critical to understanding how modern Java libraries and frameworks may interact with an Android application. Second, the software engineering topics of unit testing, design patterns, and dependency injection give a foundation on the overall structure and approach to an Android application. Finally, the Android specification is outlined from the perspective of an application developer. This gives some footing within the Android framework.

2.1 Java

The Java programming language[9] is a popular, modern, object-oriented language. As such, it supports many features common in object-oriented languages. This includes the class construct, class inheritance (single extends, multiple implements), polymorphism,

and strong typing, among others. Classes may contain constructors, fields, methods, and other class definitions. A class may be instantiated in order to create an instance. An instance takes on the characteristics of the class including the constructors, fields, and methods. Each instance holds its own variables, enabling data encapsulation within the class.

To complement the class, Java also supports annotations. Annotations are a way to add arbitrary metadata to a class and the elements contained within the class. For example, one may annotate a class at the type level by annotating the class itself or a method at the method declaration. Annotations alone would be useless without a way to discover them as they do not directly specify behavior. Reflection is one way to probe the class definition for annotations. See Figure 2.1 for an example of a custom annotation applied to a class, field, and method.

```
public @interface Metadata {
  String value
}

@Metadata("Class level")
public class AnnotationExample {
  @Metadata("Field level")
  private int value;
  @Metadata("Method level")
  public void foo() {}
}
```

Figure 2.1: An example of Java Annotations. In this example, a custom annotation Metadata is defined with a single string parameter. In the AnnotationExample class Metadata is used to annotate the class, field and method.

2.1.1 Reflection

Reflection is the technique used in Java to perform introspection. Within Java, code the programmer writes may analyze the structure of a class at run-time. This technique allows

developers to determine information about a class, access data contained in a class, and execute methods on an instance of the class without knowing before run-time the structure of said class. This feature is commonly used by library developers to examine classes dynamically and configure the library based on the analyzed class structure.

2.1.2 Bytecode

The Java specification encompasses two languages, the Java language[9], and Java bytecode language.[10] Java is compiled to run on a Virtual Machine (VM) and this compiled version of Java is known as bytecode. Bytecode is managed and loaded into the VM by classloaders. This allows the VM to load files containing bytecode from the file system, as well as data streams from memory. Thus, bytecode can be generated during run-time and loaded into the run-time environment to be executed. This bytecode generation complements reflection, as new functionality can be generated for classes that were unknown before run-time.

One example of bytecode generated at run-time is code to speed up class instantiation. It is well known in Java that repeatedly using reflection can incur a large performance cost. Often developers will cache Reflection results, but another approach is to generate bytecode that performs the necessary action. In the case of class instantiation, one may want to create an instance of a class using a handful of dependencies. The bytecode generated may be a method call with the required dependencies as parameters and the appropriate constructor call made on the analyzed class. This class, also known as a factory, can then be loaded into a classloader at run-time, and executed when a new instance of the class is requested. The resulting code executes as fast as possible, as it is no different than a class compiled before run-time.

2.2 Unit Testing

Unit Testing is a common technique[11] used by developers to ensure code executes as expected. Unit Tests are written in such a way as to isolate functionality across an application into small units. These units are small, manageable pieces of the application with a limited scope of functionality. Each assertion made in a suite of Unit Tests should embody a case of the unit's input and the resulting output. Unit Testing exercises the individual parts of an application, ensuring they all function as expected. If each smaller part functions as expected, then the larger application also should function as expected.

Code Coverage[12] is used as a metric to highlight the amount of code that is tested by Unit Tests. Developers aim for one hundred percent code coverage, but often this goal is difficult to achieve. Code coverage statistics can be gathered through a number of tools that monitor the execution of Unit Tests. Code Coverage mirrors the cyclomatic complexity of a function or unit of code. Each line of code should be tested, therefore, lines of code with multiple execution branches must be tested separately and independently. This means code with a high branching factor or cyclomatic complexity requires more tests to be written.

Unit Testing relates to the subject of boilerplate through the idea of Code Coverage. Boilerplate code, if it exists in an application, should be unit tested if a high Code Coverage statistic is a goal. Often times, because of the nature of boilerplate, the tests are tedious to write and test monotonous behavior. This is the case with plumbing code. Testing plumbing code requires the developer to ensure that the creation of an application structures the created instances properly. The Unit Tests for this code test the structure of the resulting code or the execution patterns of the creation of instances in an application. These tests are critical, as they ensure the creation of an application is performed properly, but they do not execute the core business logic. The end results are tests that have little impact on the core functionality of the application.

2.3 Design Patterns

Design patterns serve a number of purposes in modern Object Oriented design. First, design patterns are used as a guideline of how to solve a problem, without explicitly defining the solution.[7] This gives the developer guidance on how other developers have solved problems in the past and serves as a foundation for development practices. Second, design patterns communicate the intention of a class.[7] It is paramount that code is human readable, and design pattern naming conventions communicate the intent of a class aiding in understanding.

Design patterns are categorized into several groups including Structural and Creational Patterns among others. [7] Structural design patterns facilitate how classes are and should be used. This includes such patterns as the Adapter[7] and Proxy[7] patterns. The Adapter pattern converts one class to another, effectively adapting the class API so it may be used in place of another class. Typically this is written in Java using two somewhat similar Interfaces and mapping equivalent functionality from one to another through an Adapter class. The Proxy Pattern is another example of a Structural Pattern as seen in Figure 2.2. A Proxy typically implements the same interface as the class instance it proxies (the delegate) and performs actions before and/or after the method executions in the class. This may facilitate simple operations, such as logging when a method is executed, to more complex operations such as executing the given method in a different thread.

Creational Design Patterns handle building and instantiating classes. This includes building a single object, building a graph of related objects, and composing objects together. The classic example of the Creational Design Pattern is the Factory.[7] See Figure 2.3 for an example of the Factory pattern. A Factory's job is to build an instance of the requested type, using either the state of the system or inputs to the method, or by creating new related objects. Factories can delegate to other factories, layering them together. Factories also may call upon cached values to leverage scopes or the state of the system. For

```
interface Example {
  void execute();
}
class ConcreteExample implements Example {
  void execute() {
    //do something
  }
}
class ProxyExample implements Example {
  Example delegate;
  ProxyExample(Example delegate){ this.delegate = delegate; }
  void execute() {
    //do something before
    delegate.execute();
    //do something after
  }
}

Example example = new ProxyExample(new ConcreteExample());
example.execute();
```

Figure 2.2: An example of the Proxy Design Pattern. This example demonstrates how a proxy can delegate to another object that implements the same interface while adding code before and/or after the method being executed.

instance, a class may be considered a Singleton if a Factory uses a static member variable to hold the instance created by the Factory. This instance could be instantiated along with the reference to the Factory class, thus, all instances returned by the Factory would be the same instance during a single run of the application.

2.4 Dependency Injection

Dependency injection[13, 14, 15, 16, 17] is a technique in Object Oriented design used for building object graphs in a decoupled way. If written with simplicity and modularity in mind, classes should have a narrow focus and should delegate to other classes for specific

```
class ExampleFactory {
  Example build() {
    Example example = new Example(...);
    example.setValue(...);
    return example;
  }
}
```

Figure 2.3: An example of the Factory Design Pattern. This example demonstrates how a factory can consolidate and encapsulate the complexity of constructing an object. In this case, Example may take a variety of parameters to fully build. The factory handles the instantiation of the object and ensures that the necessary class parameters are given.

behavior. As seen in Figure 2.4, a class that needs to execute a query against a database should not handle setup of the database connection, management of database passwords, pooling database connections, or possibly what database is being used. Instead, the example class just needs an instance of a Data Access Object (DAO)[18] to execute the query. This is the idea behind dependency injection; related instances (Dependencies) are issued (Injected) to the instance of the class via a dependency injection engine. The dependency injection engine is simply code that manages the creation and scope of instances within the system. The end result is the example class avoids knowing irrelevant information about the DAO, including how the DAO is created. Therefore, the DAO may be created via a Factory (See Figure 2.5), instantiated directly (via the new keyword) or already exist as a cached Singleton. This concept is known as loose-coupling and is critical to developing modular applications.[19, 20, 17] Loose-coupling aids in Unit Testing classes as mock or dummy objects can be injected in place of the given dependencies and execution against the injected classes can be monitored and validated.

Using a dependency injection engine, writing code associated with creating the objects and their dependencies, and managing the scope of those instances can be effectively avoided. This cuts down on the boilerplate required by an application significantly, increasing maintainability.[21] Instead, the dependency injection engine manages the state

```
class Example {
  @Inject UserDAO userDao;

  public void updateUser(long id) {
    User user = userDao.find(id);
    user.setUpdated(new Date());
    userDao.persist(user);
  }
}
```

Figure 2.4: Usage of an injected Data Access Object. In this case, the Example class' updateUser method is used to assign the current date to the user's updated field. Dependency injection allows for the UserDAO to abstract the database configuration details away from the Example class.

```
class UserDAOFactory implments Provider<UserDAO> {
  public UserDAO get(){
    return new UserDAO(...);
  }
}
```

Figure 2.5: Example of a UserDAO factory. This class, using the Factory design pattern and the JSR330 Provider interface, constructs a new UserDAO when the get method is called. Although the details are omitted, configuration details could be provided to the UserDAO class including database configuration or other injected objects. The UserDAOFactory class would be specified to the dependency injection engine as the source when injecting a UserDAO.

and creation of instances in a system through a number of mechanisms.

There are three ways to inject a dependency into a class instance: through the constructor, method or field. Constructor injection (See Figure 2.6) happens when the constructor is called and the instance is instantiated. Dependencies, in this case, show up as parameters in the constructor. Field injection (See Figure 2.7) happens once the instance is instantiated. Fields of the class are assigned dependencies directly by using the assignment operator. Finally, Method injection (See Figure 2.8) happens by calling a series of setter methods on a class. Setter methods may have one or more parameters, all representing injected dependencies of the class.

```
class InjectionExample {
  DependencyExample example;

  @Inject InjectionExample(DependencyExample example) {
    this.example = example;
  }
}
```

Figure 2.6: An example of constructor injection. If an instance of the InjectionExample class is requested, the dependency injection engine will provide an instance of DependencyExample to the constructor when the constructor is called during class instantiation.

```
class InjectionExample {
 @Inject DependencyExample example;
}
```

Figure 2.7: An example of Field Injection. If an instance of the InjectionExample class is requested, the dependency injection engine will provide an instance of DependencyExample after instantiating the class by setting the field directly with a DependencyExample instance.

```
class InjectionExample {
 DependencyExample example;

 @Inject void setDependency(DependencyExample example) {
  this.example = example;
 }
}
```

Figure 2.8: An example of Method Injection. If an instance of the InjectionExample class is requested, the dependency injection engine will provide an instance of DependencyExample after instantiating the class by calling the setDependency method with an instance of DependencyExample.

The dependency injection engine's job is to associate classes with their dependencies. To do this, ambiguous injections must be qualified. If an injection of a non-specific type is made (say an Interface) then the dependency injection engine must know what concrete injection to make. This could take the form of calling a constructor or another class, calling a factory or pulling an instance from a scope cache. There are a number of ways to

determine a dependency. Most dependency injection engines follow the approach of using the type to identify the dependency. If a dependency of type Foo is called for, and it is unambiguous, then an instance of Foo is built and provided to the dependent instance.

Classes and their dependencies can be represented as a directed graph, where the classes and dependencies make up the nodes and edges of the graph. See Figure 2.9 for an example. In this graph, each node is a dependency, and each directed edge points from the dependent to its dependency. Of particular interest, this collection of nodes and edges is not a tree, as there may be more than one path to get to a given dependency and cyclic dependencies are a possibility. Different dependency injection engines handle these conditions in different ways, ranging from throwing an Exception when difficult cases arise to handling the cases with specific techniques.

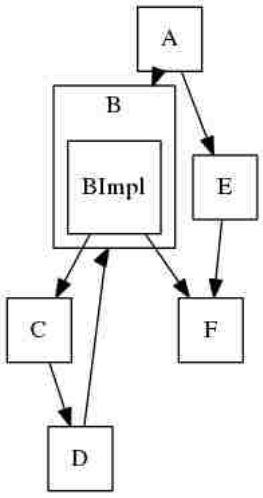
Dependency injection has become popular on the Java platform; thus, a standard has been developed in an effort to unify the semantics of dependency injection engines. Specifically, Java Specification Request 330 (JSR330)[22] was developed with a series of annotations and related unit tests on using those annotations. This includes annotations for injecting (Inject), scoping (Scope), specifying singleton scope (Singleton), qualifying injections (Qualifier and Named) and an interface for writing general factories (Provider).

2.5 Android

Google's Android platform has become a dominant programming environment for mobile phones and tablets. An Android application is built using a series of Component classes written in Java. Each Component represents an element of the application, for instance, a page, a background task or an entry point into the application. The major components of an Android application are the Activity, Service, Broadcast Receiver, and Application. Activities represent the different visible pages used in an application and

```
class A {
  @Inject B b;
  @Inject E e;
}
@ImplementedBy(BImpl.class)
interface B {}
class BImpl {
  @Inject C c;
  @Inject F f;
}
class C {
  @Inject D d;
}
class D {
  @Inject B b;
}
class E {
  @Inject F f;
}
class F {}
```

(a) Dependency Injection Graph



(b) Graph Visualization

Figure 2.9: Diagram of a dependency injection graph. In this example, when an instance of A is injected, the related instances of B, C, D, E and F are also injected. The dependency injection engine takes care of the interconnections (edges) between classes (nodes), dictated by the annotated fields. The injection of B into D closes a loop in the injection graph and represents a cyclic dependency.

the associated behavior, such as a dashboard, search results, or preference screen. Service Components represent long-running background activities, such as playing music or polling location information. Broadcast Receivers are programmatic entry points into the application that respond to system-level actions or events (Intents). Finally, the Application Component represents the high-level application and the associated functionality. Each of these components is built by extending an associated base class. For Activity the base class is android.app.Activity, for Service the base class is android.app.Service and so on. There are special types of base classes, especially for the Activity Component, which perform slightly different tasks. For example, the ListActivity class handles displaying

and manipulating an ordered list of elements.

Components in Android must not only implement the given base class, but they must also be registered in the application's AndroidManifest.xml file. This file configures the given Android application on the platform and registers the Components accordingly. To register a component, the component's class name is specified in the appropriate xml tag. A variety of Metadata accompanies the components as a means of specifying behavior on the Android platform. The manifest file is also where the Android application registers privileges it requires and system-level services it expects to use. This technique serves as a means to limit the elevated permissions that an application can use and allows the end-user of the application to know what parts of the phone environment (location information, phone book, etc.) the application can access. See Figure 2.10 for an example of the AndroidManifest.xml file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<manifest package="org.example" android:versionCode="1"
    android:versionName="1.0"
    xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="11"/>
  <application android:name="android.app.Application">
    <activity android:label="@string/app_name"
        android:name=".HomeActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:label="@string/app_name"
        android:name=".ExampleActivity">
    </activity>
  </application>
</manifest>
```

Figure 2.10: An example AndroidManifest.xml file including two Activities: HomeActivity and ExampleActivity. HomeActivity is set up with intent filters to activate the Activity as the landing page of the application (MAIN) and to appear in the application list (LAUNCHER).

Each Component has its own lifecycle within the application. This lifecycle is handled by a series of callbacks made on the implemented component via overridden methods. For instance, when an Android Activity is initialized and about to show on screen, Android will call the onCreate method. This allows the implemented component to prepare any related variables, setup the screen layout and perform a variety of other startup tasks. When an Activity is being closed by the system, either by the user closing the application or by navigating away from the screen, the onPause() method is called as an indication that any necessary actions related to shutting down the Activity should be executed. Each lifecycle callback is made essentially the same way as the examples given, only differing in under what condition they are called and what parameters are accepted by each.

Android Applications follow a common paradigm used in Graphical User Interface (GUI) based applications. Specifically, the GUI framework for Android uses a single-threaded UI model. This model enforces that all actions that happen to or from the GUI execute in the same thread. For instance, redrawing the screen is performed in the same thread as button click listeners that are responding to button events. Historically, this model has proven effective at avoiding complex problematic situations like deadlock in GUI code across many different frameworks. This does not mean that Android Applications cannot be multi-threaded; in fact, one of Android's strengths is that it is a multi-threaded platform. However, special care should be taken around the UI to make sure the UI threading model is respected.

System Services in Android are classes that allow the application to access some of the hardware features and extended operating system features on the devices on which it is running. For instance, the LocationManager service is used to access location-based information, from a GPS receiver, cell tower locations, or wifi location information. These System Service classes can be accessed from the Context base class, a super class of both Activity and Service, by calling the getSystemService(String name) method. Android will resolve the appropriate Service and return it accordingly. Again, many System Services

require elevated privileges and require these privileges to be registered in the AndroidManifest.xml file.

One strength of the Android platform is the ability to install multiple applications to handle common tasks, such as sharing photographs. If multiple applications are installed that handle these actions, then the Android system will prompt the user to choose which application should be used to handle the request. This system is known as Intents within Android. Intents are named events coupled with lightweight data that are used to call into applications. In fact, every application is started using an Intent. These Intents may be explicit or implicit. Explicit Intents are Intents that specify the exact Component within an application to run. Typically explicit Intents are used within an application to navigate from Activity to Activity. Implicit Intents are used to define and handle general activities on the phone, such as sharing photographs, or opening a web browser on the phone. Implicit Intents are defined by Intent name and handling of Intents is configured within the AndroidManifest.xml file.

Chapter 3

Methods: Transfuse

3.1 Introducing the Transfuse framework

The goal of Transfuse[1] is to give Android developers the metaprogramming tools common in traditional Java applications while supporting high performance as a critical requirement. This includes dependency injection, Aspect Oriented Programming (AOP), Plain Old Java Object (POJO) Components, Event Systems, and others. Transfuse uses different techniques to support metaprogramming, namely compile-time class analysis and code generation. As a result, applications that use Transfuse avoid the performance penalty of run-time code analysis and code generation. All the code that supports the features offered by Transfuse is generated at compile-time and is ready to execute as soon as the application starts. By using Transfuse, the developer can take advantage of the benefits of metaprogramming on Android, without sacrificing startup time and responsiveness. Moreover, Transfuse helps minimize application boilerplate while maintaining high performance.

3.2 Dependency Injection

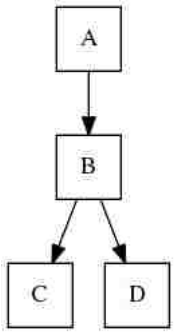
Dependency injection is a solution to reduce boilerplate code in object graph construction and state management. Transfuse implements dependency injection in Java following the JSR330 standard. This standard prescribes a set of annotations and unit tests that enforce how a dependency injection engine should behave.

The most important annotation in dependency injection, the Inject annotation, is defined in JSR330. This annotation can be placed on a constructor, field, or method, defining where a dependency should be injected. If a constructor is annotated with Inject, each of the constructor parameters should be given to the instance of the class by Transfuse. This requires the dependency injection engine to have control over the instantiation of the class to be injected. Namely, to handle constructor dependency injection, Transfuse must instantiate the class. Likewise, if a Field is annotated with Inject, the given field instance will be set by Transfuse. Unlike constructor injection, the instance to be injected does not need to be constructed by Transfuse. However, Transfuse must, during injection time, hold a reference to the instance to be injected. A method annotated with Inject dictates to Transfuse to call the method matching dependencies to the method parameters. Like Field injection, Transfuse must retain a reference to the instance to be injected during injection time. Each dependency, in turn, must be either available to Transfuse or instantiated by Transfuse prior to injection. See Figure 3.1 for an example of code generated by Transfuse.

Transfuse uses a combination of the injection type and additional annotations to identify which instance to inject. These additional annotations are also known as qualifiers and can be defined by the user, or using the Named annotation provided via JSR330. Also, the injection types are given by the constructor parameter types, field types or method parameter types, depending on the injection type. Qualifier and type combinations in Transfuse represent a unique key for each injection. This is known as a ScopeKey (Scopes are discussed later). These simple rules define the base behavior of Transfuse's dependency

```
class A {
  @Inject B b;
}
class B {
  C c;
  D d;
  @Inject B(C c){
    this.c = c;
  }
  @Inject void setD(D d){
    this.d = d;
  }
}
```

(a) DI Code



(b) Graph Visualization

```
C c$$0 = new C();
D d$$0 = new D();
B b$$0 = new B(c$$0);
b$$0.setD(d$$0);
A a$$0 = new A();
a$$0.b = b$$0;
```

(c) Generated Graph Code

Figure 3.1: Simple example of Transfuse dependency injection graph JSR330 based code. The code in Figure 3.3c is generated from the dependency injection Graph code in Figure 3.3a. Notice how Transfuse builds the graph from the leaves (C and D) first, then works its way into the root object (A).

injection engine. Each style of injection performs roughly the same action: providing an instance to the injection target.

Dependency injection can be viewed as a directed graph.[15] A node in the graph represents an instance to be injected, referenced or constructed by Transfuse. An edge in the graph represents an injection. Each edge is directional, pointing from the instance to be injected to the instance to inject. In Transfuse, the dependency injection graph may be cyclic because any class may reference any other class.

Transfuse performs analysis of classes during compile-time and builds a graph model in memory representing classes and dependencies. Using this graph, Transfuse writes a series of constructor calls, field sets, and method calls to build an instance of the graph model. These operations are Java commands, to be executed at run-time when an instance of the graph is needed. This simple technique moves the time associated with dependency injection analysis from run-time to compile-time, leaving only the minimum graph

construction operations to be executed on demand.

3.2.1 Dependency Graph Loops

```
interface Worker {
  void execute();
}
class ConcreteWorker implements Worker {
  void work() {}
}
class VirtualProxyWorker implements Worker {
  Worker delegate = null;
  void execute() {
    if(delegate == null) throw new RuntimeException("Delegate not
        created");
    delegate.execute();
  }
  void setWorker(Worker worker) { this.worker = worker; }
}
VirtualProxyWorker virtualProxy = new VirtualProxyWorker();
// Virtual proxy may be used at this point as a reference.
virtualProxy.setWorker(new ConcreteWorker());
// Once a worker has been given, then the execute method may be called.
virtualProxy.execute();
```

Figure 3.2: An example of a Virtual Proxy. This example demonstrates a proxy may be used to allow a Virtual Proxy placeholder to be constructed. After the Virtual Proxy is constructed, an implementation of Worker is given to allow calls to execute() to be fulfilled without a RuntimeException being thrown.

Constructor injection poses a challenge. A constructor both instantiates a class and provides dependencies in the same operation. This means all dependencies of a class must be instantiated prior to a constructor call. However, dependency injection graphs are cyclic[15], presenting a problem when a dependency graph loops onto itself with only constructor injections as edges in the graph. Transfuse offers the following solution to this problem: One of the nodes in the graph cycle is replaced by a Virtual Proxy instance (See

Figure 3.2). A Virtual Proxy[7] is a skeleton implementation of an interface that delegates to another instance implementing the same interface. The delegate is then passed to the Virtual Proxy after the Virtual Proxy is instantiated. In the case of the cyclic graph, this Virtual proxy breaks the constructor dependency loop. The Virtual Proxy is instantiated in place of the given dependency and provided to the dependent constructor. Next, the delegate is instantiated with its dependencies (including the prior part of the dependency loop) and provided to the Virtual Proxy. Effectively, this instantiates the graph in two parts, breaking the loop. See Figures 3.3 for an Example of a cyclic graph broken by Transfuse's use of a Virtual Proxy.

As a contrast to the constructor injection loop problem, field and method injection is performed after an instance is created. With field and Method injection, each node in the graph is instantiated before the edges are put in place. Therefore, the cyclic graph problem is not an issue as it is with constructor injection. In fact, any field and method injection may be used as an alternative technique to using a Virtual Proxy to break a cyclic graph into parts.

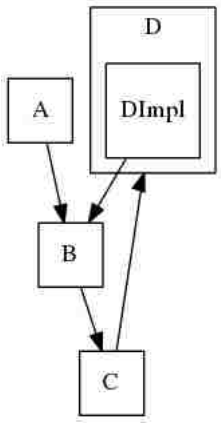
However, breaking the cyclic graph with a Virtual Proxy, field, or method injections do not cover all possible dependency graphs. Cyclic graphs that are comprised of only constructor injections with concrete classes (not compatible with the Virtual Proxy approach) allow no way of breaking the graph into parts. This means that Transfuse is unable to construct these graphs. In this case, Transfuse gives an error directing the programmer to use one of the mentioned techniques to allow Transfuse to break the cyclic graph.

3.2.2 Providers

Often, an object isn't needed until after the object graph is instantiated, or multiple objects are needed on-demand. This is the common case of the Factory design pattern. JSR330 defines a simple interface to handle this use case, called the Provider. The Provider defines

```
class A {
  @Inject B b;
}
class B {
  C c;
  @Inject B(C c) {this.c = c;}
}
class C {
  D d;
  @Inject C(D d) {this.d = d;}
}
@ImplementedBy(DImpl.class)
interface D {}
class DImpl implements D{
  B b;
  @Inject DImpl(B b){this.b = b;}
}
```

(a) Dependency Injection Graph



(b) Graph Visualization

```
DImplVProxy dImpl$$0 = new DImplVProxy();
C c$$1 = new C(dImpl$$0);
B b$$1 = new B(c$$1);
A a$$1 = new A();
a$$1.b = b$$1;
DImpl dImpl$$1 = new DImpl(b$$1);
dImpl$$0.load(dImpl$$1);
```

(c) Transfuse Generated Graph Code

Figure 3.3: Transfuse generated cyclic graph code. The cyclic graph built by Transfuse is able to be built due to the Virtual Proxy (DImplVProxy) acting as a placeholder during construction. Notice that after the target instance of A is built, the delegate implementation of D is set into the Virtual Proxy via the load method.

a single method, get(), which returns a template type.

When a Provider interface is injected, Transfuse will inject a generated implementation of the Provider based on the template type defined. Transfuse specifies the contents of the Provider get() method which is simply another object graph instantiation. Effectively, Transfuse generates the implementation to be used as the Provider node in the dependency

injection graph. See Figure 3.4.

```
class A {
  @Inject B b;
}
class B {}
class BProvider implements
    Provider<B> {
  @Override B get() {
    return new B();
  }
}
```

(a) Source

```
BProvider bProvider$$0 = new
    BProvider();
A a$$1 = new A();
a$$1.b = bProvider$$0.get();
```

(b) Generated

Figure 3.4: This example shows Transfuse calling the Provider designated for B in order to inject an instance of B.

3.2.3 Scoping

By managing the instances of an application, dependency injection engines perform the critical task of managing state in an application. This opens up the possibility of caching instances to be used more than once throughout a dependency graph or multiple dependency graphs. This practice is commonly called scoping. Scoping defines which object instances are used in what contexts. Transfuse handles scoping through a series of configured lookup caches or Scopes. These Scope objects are Singletons within an application. Each Scope defines a method getScopedObject() that returns a reference to the requested object. The getScopedObject() method takes a ScopeKey identifier, used to uniquely identify the given object and a Provider object used as a factory if the given instance does not exist. An object is associated with a scope through Transfuse's configuration, mapping the ScopeKey to the Scope, defining which object is injected when requested.

One of the most common and simplest Scopes is the Singleton Scope. The Singleton Scope caches instances for the entire lifetime of the application. When a Singleton is first

requested via the getScopedObject() method and it does not yet exist in the Scope, it is created via the Provider get() method. This instance is then cached in a Map collection and returned to the application. When another instance of the same type is requested, Transfuse calls the same getScopedObject() method with the same ScopeKey and Provider. After the call, the instance exists in the lookup map and is returned.

The most common scope, the Prototype Scope, has already been discussed, but not named. Unlike other scopes, this scope does not cache instances when they are created. Instances created in the Prototype Scope are instantiated on demand. Transfuse shortcuts using a scope object in this case and simply calls the given classes' constructor. This results in a direct constructor call being issued during the dependency graph creation.

3.3 Android Integration

Along with being a dependency injection engine, Transfuse also maps the Android API to the dependency injection graph. This mapping consists of three features applied to the dependency injection graph. First, a portion of the Android API consists of objects accessible via getters or lookup methods. Transfuse allows these objects to be injected as nodes in the dependency injection graph. This includes any Context or Component class. Second, Android components are executed through lifecycle methods. Transfuse maps these lifecycle methods to an event system, allowing any node in the dependency injection graph to execute during a phase of the lifecycle. Third, certain overridden methods on Android Components require return values. In Transfuse, these method calls are implemented through a technique called call-through methods.

In addition to transforming the Android API, Transfuse eliminates much of the work of defining the Android application by writing the AndroidManifest.xml file. This removes one of the key duplications within an Android application.

3.3.1 API Injections

The Android system offers its API through calls to the Context base classes. This includes the Activity, Application, Service, and other specific Contexts. Developers are encouraged to develop applications by extending the Context base classes. These Context extensions have access to API calls such as getSystemService() that returns a manager based on the provided key. As an example, the LocationManager (the class responsible for interacting with the GPS and other location hardware) is available via the getSystemService() call.

Transfuse facilitates the injection of the Android API into the dependency graph. This is achieved in two parts. First, Transfuse components are not extensions of Android Context base classes. They are simple Plain Old Java Objects (POJOs) annotated with a Transfuse component annotation (Activity for instance). This class is treated as the root element in the dependency injection graph. Second, Transfuse implements an Android Context that delegates to the Transfuse component. This gives Transfuse the ability to write code to access the Android API via the Context base class and use those objects as objects in the injection graph. This includes System Services, like the LocationManager, view classes via the findById() method call, and the Application object, among others. Also, for completeness, the Context itself may also be injected throughout the dependency graph. Transfuse maps each API element to a ScopeKey by Type and Qualifier.

Traditionally, Android Views are referenced through a call to the findById() method on the Activity base class. This method returns Object, which is then cast to a specific View type. Typically the returned View is assigned to a field to be used during the lifecycle of the Activity. In the spirit of boilerplate reduction, Transfuse collapses these three aspects of referencing Views into one step, treating the View lookup as a node in the dependency injection graph. Transfuse allows Views to be injected by using the View qualifier. The view takes a view id or tag name to specify the View to inject. The result is a single statement: an annotated injection.

Android System Services are looked up via the getSystemService() which takes a String name parameter. This method returns an Object that is to then cast as the specific type of System Service. Transfuse combines these steps into a single injection as a node in the dependency injection graph. Due to the one-to-one mapping of System Service type to the lookup String name, Transfuse is able to inject the System Service by type alone.

The Android Context object may also be injected into the dependency injection graph. This allows the developer to call any method on the Context object, in case they are not mapped directly by the Transfuse API.

3.3.2 Lifecycle Events

Android Context classes rely on lifecycle events to execute a variety of operations. Each lifecycle call is handled by overriding the Context base class on-Event method. These lifecycle events typically require a super call to the base class and return void.

During the dependency injection graph analysis, Transfuse also looks for On-Event annotations included in the API. On-Event annotations are only applicable to Java methods. Each method annotated with an On-Event method is earmarked to be called during the corresponding lifecycle event method. By generating the code for the context base class, Transfuse is able to write the lifecycle event methods. In turn, Transfuse is able to write method calls within the overridden event methods. This includes calls to the earmarked On-Event annotated methods throughout the dependency injection graph. The end result is that any object in the dependency injection graph can respond to Lifecycle Events through methods annotated with On-Event annotations. See Figure 3.5 for an example.

```
@Activity(label = "Transfuse Example")
public class Example {
  @OnCreate
  public void logCreate(){
    Log.i("Example Info", "OnCreate called");
  }

  @OnPause
  public void logPause(){
    Log.i("Example Info", "OnPause called");
  }
}
```

Figure 3.5: An example of a Transfuse Activity component with OnCreate and OnPause lifecycle events. Annotating the Module class with TransfuseModule configures Transfuse to perform specific actions. In this case, a Bind annotation is added, directing Transfuse to use the ExampleImpl class whenever an Example class is injected. the logCreate and logPause methods will be called by the generated Activity class within the onCreate and onPause methods.

3.3.3 Call-through Methods

In addition to the Lifecycle Event methods, Android Context classes offer a set of methods that are called expecting a return value. Transfuse delegates these calls to one, and only one, instance in the injection graph via a call-through method mapping. This enforces the single return value requirement of Java methods and allows a delegate to interact with the Android system. One class in the dependency injection graph must implement the call-through method interface. This interface mirrors the original calling Android Context class method, including method name, parameters, and return type.

3.3.4 Android Manifest

Each Android Context implementation must be defined in the AndroidManifest.xml file to be recognized as part of the application. This file contains metadata related to each Context implementation in the application, as well as global configuration details, includ-

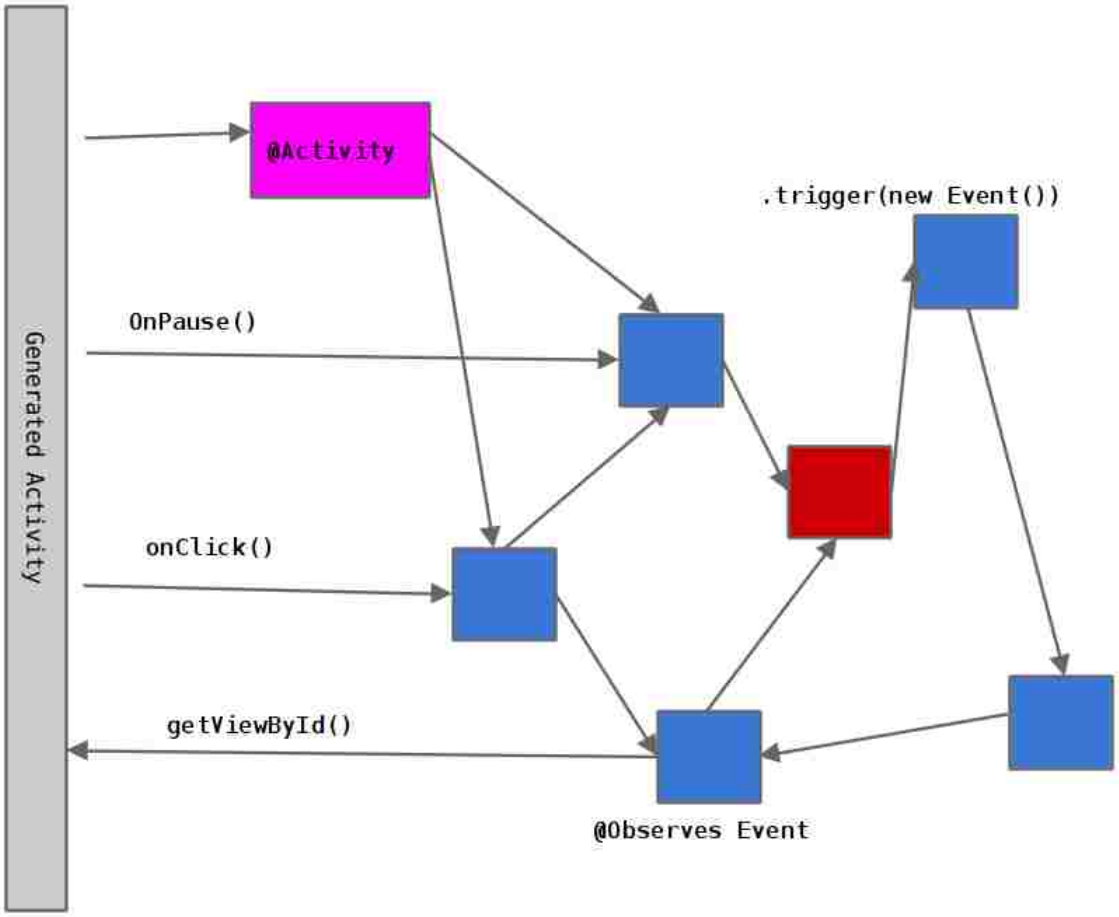


Figure 3.6: Visualization of the generated Transfuse Activity component and graph. This figure shows a dependency graph represented by the colored rectangles associated with the original Android Activity through dependency injection, API injections and lifecycle events.

ing permissions, api versions, etc. The drawback of the AndroidManifest.xml file is it is too easy to misspell or miss a reference to a new Context class. It violates the Don't Repeat Yourself principle[23], as it requires the developer to redefine Android Contexts in a separate location.

Given that Transfuse generates the Context classes for the developer, Transfuse has a set of all Context classes defined in the application. With this set of Context classes, Transfuse defines the Android Manifest metadata via annotations and annotation proper-

ties on Transfuse Components. In memory, Transfuse merges the previous Manifest file with the internal representation of the Manifest. This merge operation occurs on a property by property basis, taking the previous Manifest file's properties and either replacing them or preserving them.

This replacement or preservation is indicated by a tag written out by Transfuse to the Manifest file. If the XML property is originated by Transfuse, a single character representation is written to the "t:tag" property. This tells Transfuse that it is managing the given property and it is free to replace it during subsequent rounds of processing. Otherwise, Transfuse preserves the XML properties it does not manage. In essence this allows developers to gracefully add Transfuse to their existing application without completely replacing the Manifest file. The end result is Transfuse is able to define the Android Manifest entries that it manages, alleviating the need to write the Manifest file by

3.4 Technique

Transfuse leverages the Java Annotation Processor API JSR269[24] for a majority of its functionality. JSR269 is an API that allows a custom processor to analyze and generate code during the Java compile.[25] An annotation processor is configured to listen for Elements under compilation that are annotated with a specified set of annotations. These Elements represent the atomic parts of Java classes. They include the class, method, field, and constructor, among others. Within the Annotation Processor API, elements are similar to the reflection API which also represents the structure of Java classes.

Annotation Processing is performed in rounds in the compiler. During each round, the processor has the opportunity to analyze a given element and write files as output, either resource files or Java source to be compiled. Subsequent processing rounds will include the to-be-compiled output of previous processing rounds.

During a round, Transfuse scans for the component API annotations: Activity, Service, BroadcastReceiver, Application, Fragment, Factory, TransfuseModule, and ImplementedBy. Transfuse then passes these annotated Elements into a series of sub-processors. Each sub-processor is configured to analyze and process an Element. Annotations that represent Transfuse components, such as Activity, trigger Transfuse to generate code (Figure 3.7). Annotations that represent configuration in Transfuse, such as TransfuseModule, configure how Transfuse operates (Figure 3.8). Sub-processors effectively scope each unit of work in Transfuse, keeping the system modular and focused.

```
@Activity
class ExampleActivity {}
```

Figure 3.7: An example of a Transfuse annotated Activity. The Activity annotation on the ExampleActivity class triggers Transfuse to generate a new class that extends Android Activity and adds the generated class to the AndroidManifest.xml file. In addition, this activates the class to be injectable and take part in the event lifecycle system.

```
@TransfuseModule
@Bindings({
  @Bind(type=Example.class, to=ExampleImpl.class)
})
public class Module{}
```

Figure 3.8: An example of a Transfuse Module. Annotating the Module class with TransfuseModule configures Transfuse to perform specific actions. In this face, a Bind annotation is added, directing Transfuse to use the ExampleImpl class whenever an Example class is injected.

Sub-processors working on Transfuse components and generating code follow a two-phase pattern. The first phase is code analysis. During this phase, the sub-processor builds an InjectionNode for the input Element. The InjectionNode is the concrete representation of a node in the dependency injection graph. Each InjectionNode is matched with an InjectionNodeBuilder, a piece of code that references how to build the given node. This matching occurs based on the Element type and any associated qualifier annotations against the default and input configurations. A set of analyzers are run on the InjectionN-

ode and Element, specifically analyzing the type, methods and fields across the inheritance hierarchy of the type. Each analyzer has the ability to add metadata to the InjectionNode to be acted upon in the code generation phase. Part of analysis is building the dependency injection graph by performing a depth-first-search of the tree rooted at the input Element. Concretely, this happens by stepping through related constructor parameters, fields, and method parameters recursively, building an InjectionNode for each instance to be injected. Each InjectionNode is then associated with the parent InjectionNode to be constructed as a dependency.

Additionally, the analysis phase allows for validation to occur in the code. Any requirements above the standard Java compiler rules may issue a line-specific error if violated. Violations in the analysis phase halt execution of the sub-processor after analysis has finished.

The second phase is code generation. If code analysis succeeded, code generation occurs using the analysis phase output. This includes the InjectionNode graph and related metadata. At a high level, the InjectionNode graph is generated by performing a depth-first-search of the graph and executing the code generation associated with each node bottom-up. This means the dependencies of each InjectionNode in the graph are constructed prior to the construction of the dependent InjectionNode.

Finally, a series of code generators are run. These code generators respond to the metadata associated with each InjectionNode and emit code to the output Java file. One example of these code generators is the lifecycle event system. For each InjectionNode in the graph, if the lifecycle event metadata is found, a call to the given instance is made to the specified method in the appropriate Android lifecycle method.

3.5 Limitations

3.5.1 Static Dependency Injection Graph

The compile-time code generation technique Transfuse uses is remarkably effective at dependency injection and integrating with the Android API. This technique does have a key limitation, however. In order to generate a dependency injection graph, Transfuse must have all information about the graph prior to compile. This means that the graph cannot be reconfigured at run-time based on run-time input.

3.5.2 Android Abstraction

Transfuse's integration with Android requires that it generates the Context class for each Transfuse component. This gives Transfuse the ability to write code to issue lifecycle events, inject the Android API and provide other features. The side effect of this is that the developer is forced away from the familiar techniques encouraged by the Android documentation. Therefore, the developer needs to understand both Android and Transfuse well to successfully write an application.

In addition, Android is a changing and evolving framework on its own. It takes effort to keep Transfuse in sync with new APIs, updates and differences in versions. Therefore it falls upon developers, both in the core team and in the open source community to make these updates.

Chapter 4

Results

Metaprogramming with Transfuse benefits the developer in two ways. First, run-time performance is optimal. Transfuse is supported by compile-time generated code resulting in fast execution compared with reflection-based code. Second, boilerplate associated with the Android system is minimized. Transfuse's goal is to augment Android development with metaprogramming techniques that reduce or remove the boilerplate related to wiring together applications. Techniques like dependency injection, Lifecycle Event callbacks, and Manifest Management reduce the code required to be hand written by the developer. In this section, Transfuse's benefits will be highlighted through experiments focusing on the various aspects of these Techniques.

4.1 Compile-time vs Run-time Execution

Transfuse's compile-time code generation produces code that is directly executable. To highlight the performance benefits of this approach, five execution aspects are analyzed. These execution aspects are class referencing, constructor call, field get, field set and method execution. These low-level Java language features are used extensively throughout

Object Oriented programming and metaprogramming.

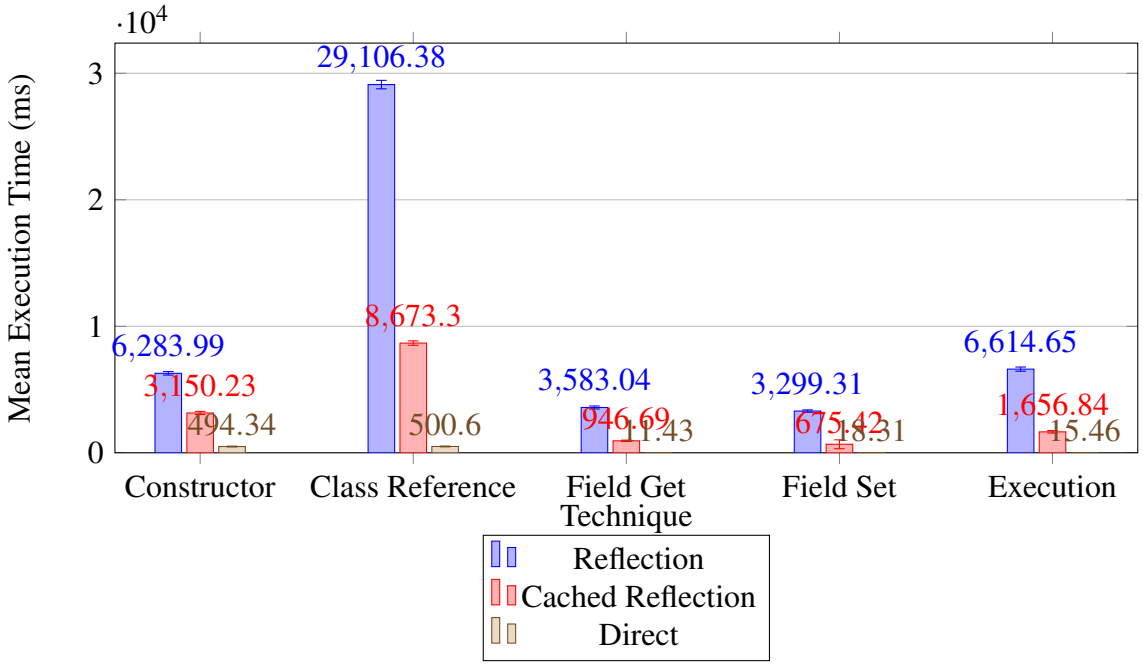


Figure 4.1: Mean execution time for 10 trials of 100,000 iterations of each sample low-level execution. Error bars indicate 1 standard deviation. The reflection based calls require a significant amount of time to execute in comparison to direct execution. Caching does reduce the time required to execute, but still is not as fast as direct execution. This data is found in tabular form in Appendix A.1.

To facilitate a comparison, these aspects are compared across three execution styles. First, reflection is used exclusively. Each experiment call is made using reflection including any supporting calls (looking up classes, Method, etc.). Second, reflection is used with caching. Often, when reflection's performance becomes an issue, caching is utilized to reduce the overhead caused by class, method, and field lookups. Each experiment call is made on cached supporting calls. The result is fewer reflection calls, but some cannot be avoided. Third, direct execution is analyzed. Each call is executed directly via the corresponding Java language constructs.

Each experiment is run for 10 trials with 1,000,000 iterations on a Motorola Moto X running Android 5.1. The run-time of each call, including reflection, is relatively fast,

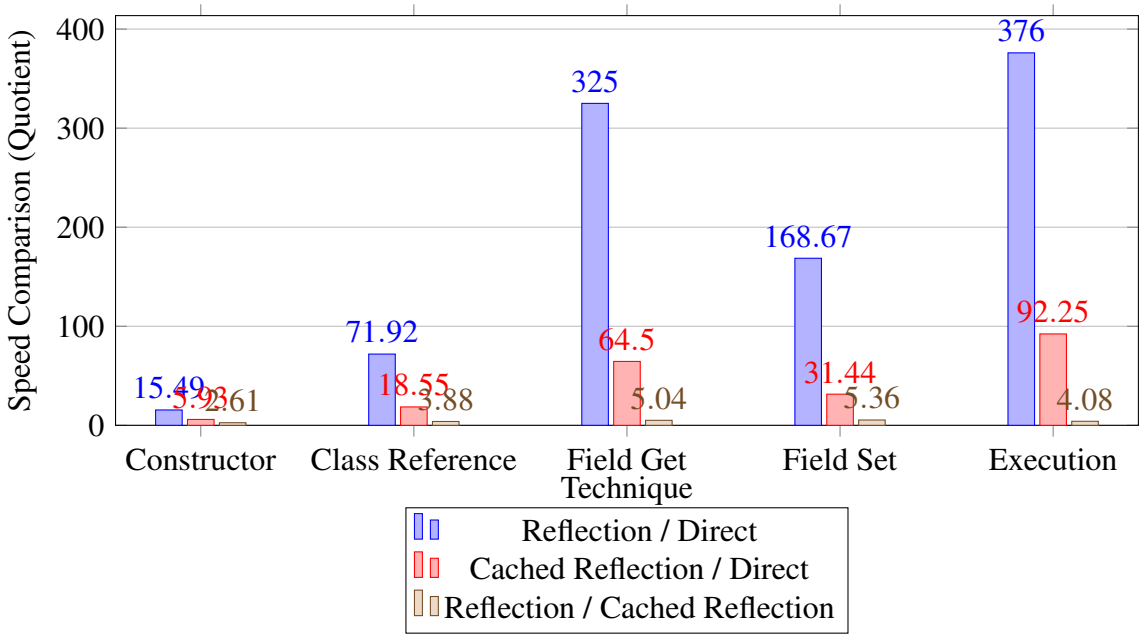


Figure 4.2: Ratio of mean execution times, comparing the 3 techniques highlighted in Figure 4.1. Although Class Reference was the slowest operation in total time, the highest percentage increase in speed is the execution and field get operations between reflection and direct execution. This data is found in tabular form in Appendix A.2

therefore running each repetitively exposes the run-time effects of a large system using each call. Repeating each call avoids any discrepancies due to Java garbage collection and reduces the variance. Additionally, running the experiments on a device exercises a real world scenario using non-emulated hardware. See Figure 4.1 for a graph displaying the mean execution time of each reflection call and Figure 4.2 for a graph displaying the ratio of execution time.

The results in Figure 4.2 clearly show direct code execution is approximately times to over 300 times faster than using reflection or cached reflection. Reflection Class Reference is one of the most expensive calls, taking more than 30 seconds to execute. Caching the Class Reference helps, as the Cached Reflection Class Reference experiment takes less than 10 seconds. Although this is faster, the Direct Class Reference experiment takes less than half a second to execute. This indicates that class referencing is a key deficiency

when using Reflection.

The largest percentage difference between techniques is not Class Referencing, however. Method Execution shows the largest percentage gains compared between Reflection and Direct execution as well as Cached Reflection and Direct Execution. Between Reflection and Direct Execution, Direct Execution is 376 times faster. Even after caching the method lookup, Direct Execution is 92.25 times faster than Caching Reflection. As can be expected, method execution is key to any system and directly executing these calls increases the speed of a system significantly.

4.2 Boilerplate Reduction

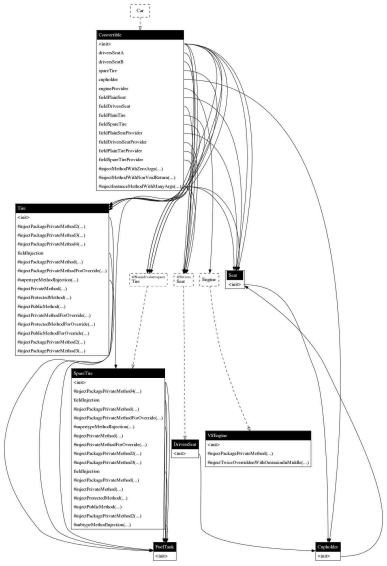
To showcase the boilerplate reduction achieved by using Transfuse, two experiments are performed. First, an object graph is created by Transfuse in order to examine the number of lines of code generated to support the direct instantiate of the graph. Second, an example Transfuse application is analyzed to count the number of lines of code generated by Transfuse. In general, these lines of generated code represents a comparable number of lines of code, not hand written by the developer, saving overall boilerplate.

4.2.1 Object Graph Boilerplate

Each edge in the dependency injection object graph requires, at least, one line of code to define the connection. Each node in the object graph requires another line of code to define the constructor call. To highlight this, the JSR330 library unit tests, which includes a feature-validating dependency injection graph, are analyzed. Transfuse is executed against this object graph in order to generate the dependency injection code. The number of lines is counted in the generated source to compare against the number of nodes and edges in the object graph. Additionally, the generated source is analyzed to produce an equation

for the minimum number of lines of boilerplate saved by using Transfuse's dependency injection.

The JSR330 unit test dependency injection graph contains 66 injections of 8 classes. For this object graph Transfuse generates 382 lines of terse injection graph building code (See Figure 4.3). Relating this back to boilerplate, this object graph construction is code that a developer can ignore and avoid maintaining. This is a direct benefit of boilerplate reduction.



(a) Graph visualization

```
public Car buildCar() {
    Convertible$$VProxy$$0 convertible$$0 = new Convertible$$VProxy$$0();
    org.atinject.tck.auto.Seat seat$$2 = scope$$2 .getScopedObject(ScopeKey.get(org.atinjec
    org.androidtransfuse.JSR330TckTest.JSR330TckModule jSR330TckModule$$1 = scope$$2 .getSc
    org.atinject.tck.auto.accessories.Cupholder cupholder$$2 = scope$$2 .getScopedObject(Sc
    org.atinject.tck.auto.DriversSeat driversSeat$$0 = new org.atinject.tck.auto.DriversSea
    org.atinject.tck.auto.FuelTank fuelTank$$0 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$1 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$2 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$3 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$4 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.Tire tire$$0 = new org.atinject.tck.auto.Tire(fuelTank$$0);
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.accessories.RoundThing.class
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.accessories.RoundThing.class
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.accessories.RoundThing.class
    InjectionUtil.setField(org.atinject.tck.auto.Tire.class, tire$$0, "staticFieldInjection
    InjectionUtil.setField(org.atinject.tck.auto.Tire.class, tire$$0, "fieldInjection", fue
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    tire$$0 .injectPublicMethodForOverride();
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    tire$$0 .injectPublicMethod();
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "inject
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "superI
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.Tire.class, tire$$0, "superI
    org.androidtransfuse.JSR330TckTest.JSR330TckModule jSR330TckModule$$2 = scope$$2 .getSc
    org.atinject.tck.auto.FuelTank fuelTank$$5 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$6 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$7 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$8 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$9 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$10 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$11 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$12 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$13 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.FuelTank fuelTank$$14 = new org.atinject.tck.auto.FuelTank();
    org.atinject.tck.auto.accessories.SpareTire spareTire$$0 = new org.atinject.tck.auto.ac
    InjectionUtil.callMethod(void.class, org.atinject.tck.auto.accessories.RoundThing.class
```

(b) Sample Transfuse Generated Code

Figure 4.3: Example JSR330 injection graph. This graph contains 8 classes and 66 injection points. Transfuse builds this graph through 382 lines of injection graph code.

4.2.2 Example Application Boilerplate

During the development of Transfuse, an example project was used as an integration test. Due to its complete feature set usage, this project is a perfect candidate to examine the amount of boilerplate augmented by Transfuse. For this experiment, each Java file is normalized in order to best represent the number of lines of code. First, each file is formatted

using the default Java formatting style in Jetbrains Intellij IDEA. Second, empty lines, single line comments and multi-line comments are removed from each Java file. Third, package and include headers are removed at the top of each class file. Finally, the number of lines of code is counted and compared between the handwritten code and the code generated by Transfuse.

Type	Lines of Code
Hand-written source	2125
Transfuse generated source	3418

Table 4.1: Count of the number of lines of hand-written and Transfuse generated source code. This represents the number of executable lines of source, ignoring empty lines, comments and include and package headers.

Interestingly, the number of generated lines of code is greater than the number of hand-written lines of code. This is a significant savings, especially when considering the number of generated lines of code were derived from the hand-written source. In essence, the application's maintainable code was less than 40 percent of the total code required for the application to run.

4.3 Comparison to Other Libraries

The metaprogramming techniques implemented by Transfuse are not exclusive to the framework. In fact, presently there exist many other similar libraries and frameworks with comparable feature sets. Writing Transfuse, inspiration was taken largely from Spring[26], Android Annotations[27], Seam[28], Guice[29], Roboguice[30], and Dagger[31] among others. Both Spring and Seam are JEE[32] specific frameworks, with a core dependency injection engine. Android Annotations includes some dependency injection, but only at a shallow level. Guice, Roboguice and Dagger are full dependency injection engines that work reasonably well on the Android platform. As a comparison, the run-time execution is

compared between Transfuse and the two most similar projects with the highest popularity, Roboguice and Dagger.

Roboguice is an adaptation of the popular Google Guice[29, 33] library that implements JSR330 dependency injection. Roboguice adds to the Guice library a handful of Android specific extensions including view injection and lifecycle event handling. At its core, Guice provides Roboguice's dependency injection implementation. Guice and Roboguice both are implemented using reflection and provide a comparison against reflection based techniques.

In comparison to Transfuse, Roboguice has a similar feature set. In fact, Roboguice is a major inspiration for Transfuse. Roboguice and Transfuse are both implemented with a dependency injection core. Around that core, Roboguice and Transfuse offer Android extensions through injections, including the mentioned view injections as well as Android API injections. Transfuse, however, takes a different approach to augmenting the component API by providing POJO components, where Roboguice offers a base class that extends the given Android component. Additionally, Transfuse uses compile time code generation techniques to implement metaprogramming where Roboguice uses reflection techniques.

Dagger is a recently launched dependency injection library. It is not a complete implementation of JSR330, but it does implement a subset. Dagger's mission statement is that it uses compile-time code generation to support dependency injection. Dagger was design with a focused mindset on just dependency injection. Therefore, in comparison to Transfuse, Dagger offers a significant subset of features. Despite this difference, Dagger and Transfuse are implemented with similar strategies, implementing dependency injection through compile-time code generation. This makes Dagger a close competitor to Transfuse and a library to compare against.

Although Dagger, Roboguice and Transfuse are similar, they do differ in many fea-

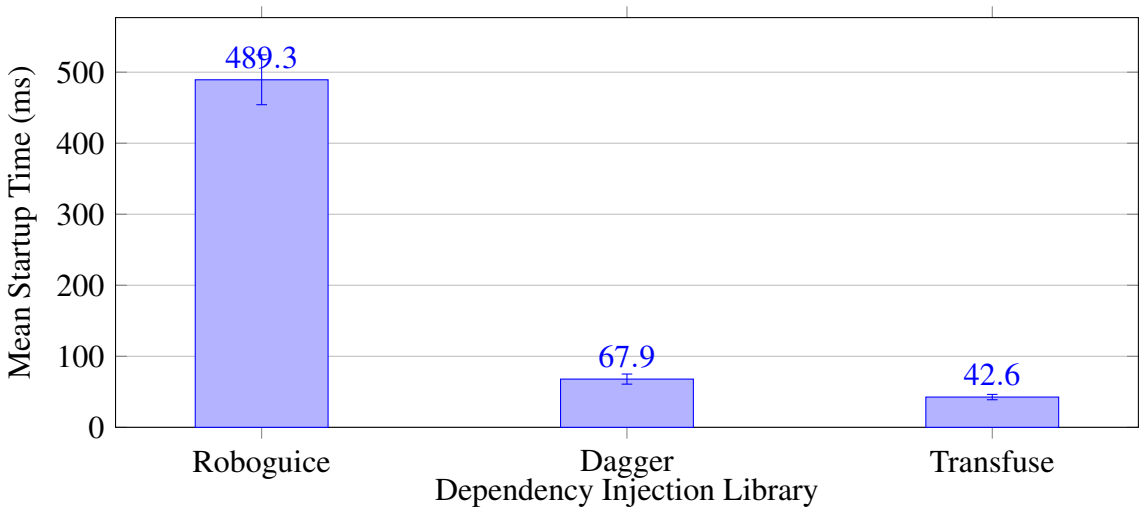


Figure 4.4: Mean execution time for 10 trial startup time of applications written with the given library. Error bars indicate 1 standard deviation. These statistics were derived from the trial data in Appendix B.1 and B.2.

tures. Therefore, the experiment focused on the common shared feature, dependency injection. An application was written[34] using each library that injects an object graph comprising 30 objects, each with unique classes. The startup run-time is measured to determine the overhead introduced by each library. Each experiment is repeated 10 times on a Motorola Moto X running Android 5.1.

The results displayed in Figures 4.4 and 4.5 show that Roboguice's performance is lacking, presumably from the reflection based injection technique implemented by Guice. In fact, Roboguice's performance, on average, is over 11 times slower than Transfuse's startup time, and over 7 times slower than Dagger's startup time. Dagger and Transfuse's startup time are comparable, but Transfuse was shown to be on average 1.5 times faster than Dagger's startup time.

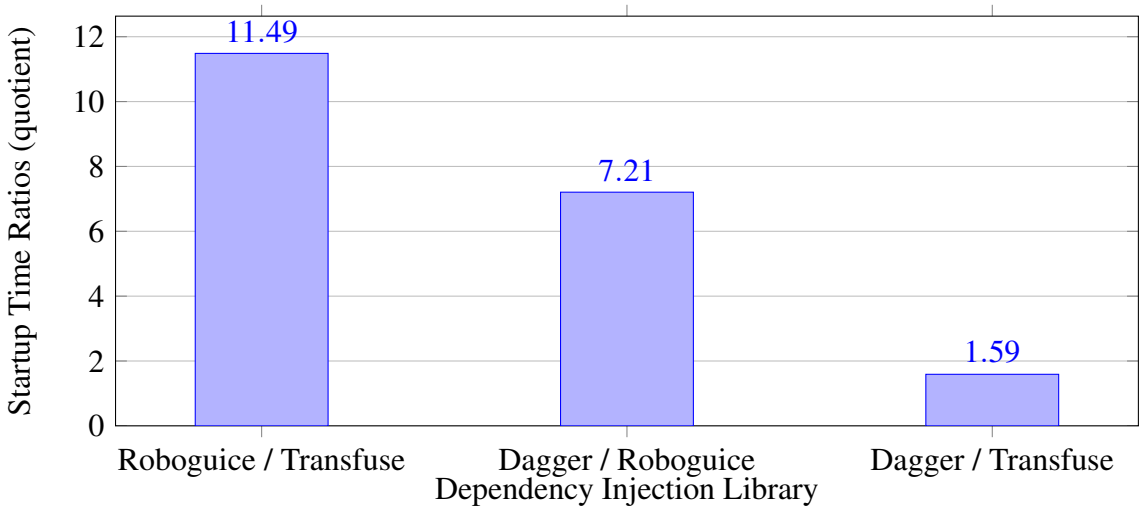


Figure 4.5: Ratio of execution times comparing application startup time ratio. This data is found in tabular form in Appendix B.3.

Chapter 5

Conclusion

This thesis has investigated metaprogramming as a tool for reducing boilerplate in Android while respecting performance. Transfuse provides a host of useful metaprogramming tools commonly found in the server environment, but without the run-time startup cost. Metaprogramming with Transfuse offers a unique position for a metaprogramming library, significantly reducing boilerplate while keeping performance at a maximum.

Transfuse takes advantage of compile-time annotation processing to accomplish the aforementioned goals. Leveraging JSR269, this tool moves the associated metaprogramming cost from run-time to compile-time. This allows metaprogramming to be used in the domain of resource starved environments, in particular, Android. Transfuse accomplishes this through a combination of dependency injection via JSR330 and API specific accommodations.

Transfuse's compile-time analysis and code generation technique showed an advantage over traditional run-time reflection based techniques. This was highlighted in the experiment focused on comparing compile-time vs run-time code execution. Compile-time generated code was significantly faster due to avoiding reflection based costs. The ratio of reflection execution time to direct execution time showed the highest increase in

performance, giving a 15x to 376x speed increase. Interestingly, even caching reflection proved to be 5x to 92x slower than direct execution. Clearly there is no substitute for directly written and executed code on the JVM or Android environments.

Application startup time benefited directly from Transfuse's approach. Applications written in Transfuse were shown to start over 11 times faster than applications written with the reflection based library Roboguice. From a user's perspective the difference is exceedingly noticeable. The experimental Transfuse application responded as soon as the icon was touched, whereas the experimental Roboguice application had a noticeable half-second lag. Similarly, an experimental application written with Dagger was compared to Transfuse. Even though Dagger uses some of the same techniques, the experimental Transfuse application showed a 1.5x faster start-up time.

Boilerplate in the Android environment was shown to be reduced significantly when using Transfuse for metaprogramming. Both the unit testing and integration testing examples highlighted in this thesis showed a significant amount of boilerplate offset by Transfuse. The integration test example gave specific reduction results, reducing lines of code by 60%. This code was absolutely critical for the example code to be functional as it represented plumbing code in the integration test application. Most importantly, this represents code that the developer can avoid writing and maintaining.

Despite the listed benefits, Transfuse suffers from two key deficiencies. First, the graph generated by Transfuse is largely static and cannot be reconfigured. Transfuse trades this flexibility for performance as the graphs are not analyzed at run-time. Second, due to the fact that Transfuse writes the context classes for the developer, this forces the developer to work in a different a more abstract way than traditional Android. Arguably this has hurt adoption as developers fear functionality will be missing or misrepresented. Even with this limitation, Transfuse does expose most of the Android API as-is, trying to stay out of the way of the developer. In addition, it has not become overwhelming to follow changes to the Android API.

In conclusion, it has been shown that applying metaprogramming can significantly reduce the amount of boilerplate associated with writing an Android application. Using Transfuse, not only is overall boilerplate reduced, but run-time application startup is respected and minimized.

Appendices

A Data collected comparing run-time and compile-time execution 4

B Data collected comparing libraries 5

C Run-time and compile-time experiment code 6

Appendix A

Data collected comparing run-time and compile-time execution

Name	Reflection	Cached Reflection	Direct
Constructor	6283.99ms	3150.23ms	494.33ms
Class Reference	29106.38ms	8673.29ms	500.59ms
Field Get	3583.03ms	946.69ms	11.43ms
Field Set	3299.31ms	567.00ms	18.31ms
Execution	6614.64ms	1656.84ms	15.46ms

Table A.1: Mean execution time

Name	Reflection / Direct	Cached Refl. / Direct	Refl. / Cached Refl.
Constructor	15.494	5.934	2.611
Class Reference	71.917	18.550	3.877
Field Get	325.000	64.500	5.039
Field Set	168.666	31.444	5.364
Execution	376.000	92.250	4.076

Table A.2: Ratio of execution times

Appendix B

Data collected comparing libraries

Trial	Robguice 2.0	Dagger 1.2.2	Transfuse 0.3.0-beta7
Trial 1	435ms	60ms	40ms
Trial 2	497ms	82ms	41ms
Trial 3	528ms	64ms	41ms
Trial 4	484ms	65ms	38ms
Trial 5	463ms	74ms	42ms
Trial 6	458ms	68ms	39ms
Trial 7	546ms	70ms	49ms
Trial 8	490ms	68ms	46ms
Trial 9	468ms	57ms	48ms
Trial 10	524ms	71ms	42ms

Table B.1: Execution Run-time to Start Application

	Roboguice 2.0	Dagger 1.2.2	Transfuse 0.3.0-beta-7
Low	435ms	57ms	38ms
High	546ms	82ms	49ms
Mean	489.3ms	67.9ms	42.6ms
Std. Dev	35.1	7.1	3.8

Table B.2: Execution Run-time to Start Application Statistics

	Dagger 1.2.2	Transfuse 0.3.0-beta-7
Roboguice 2.0	7.206	11.486
Dagger 1.2.2		1.59

Table B.3: Ratio of execution times

Appendix C

Run-time and compile-time experiment code

```
public class CachedReflectionClassReference implements Experiment {
  private Class<?> clazz;
  public CachedReflectionClassReference() {
    this.clazz = Target.class;
  }
  public String getName() {return "Cached Reflection Class Reference"; }
  public void execute() throws Exception {
    clazz.newInstance();
  }
}
```

Figure C.1: Experiment code to test the run-time of instantiating an instance via a cached class using reflection.

```
public class CachedReflectionConstructorExperiments implements Experiment
    {
  Constructor<Target> constructor;
  public CachedReflectionConstructorExperiments() throws
      NoSuchMethodException {
    constructor = Target.class.getConstructor();
  }
  public String getName() {return "Cached Reflection Constructor"; }
  public void execute() throws Exception {
    constructor.newInstance();
  }
}
```

Figure C.2: Experiment code to test the run-time of instantiating an instance via a cached constructor using reflection.

```
public class CachedReflectionExecution implements Experiment {
  private Target target = new Target();
  private Method method;
  public CachedReflectionExecution() throws NoSuchMethodException {
    this.method = Target.class.getMethod("run");
  }
  public String getName() {return "Cached Reflection Execution"; }
  public void execute() throws Exception {
    method.invoke(target);
  }
}
```

Figure C.3: Experiment code to test the run-time of invoking a cached method using reflection.

```
public class CachedReflectionFieldGet implements Experiment {
  private Target target = new Target();
  private Field field;
  public CachedReflectionFieldGet() throws NoSuchFieldException {
    field = Target.class.getField("value");
  }
  public String getName() {return "Cached Reflection Field Get";}
  public void execute() throws Exception {
    Object value = field.get(target);
  }
}
```

Figure C.4: Experiment code to test the run-time of getting a field value using reflection via a cached Field value.

```
public class CachedReflectionFieldSet implements Experiment {
  private Target target = new Target();
  private Field field;
  public CachedReflectionFieldSet() throws NoSuchFieldException {
    field = Target.class.getField("value");
  }
  public String getName() {return "Cached Reflection Field Set";}
  public void execute() throws Exception {
    field.set(target, "1");
  }
}
```

Figure C.5: Experiment code to test the run-time of setting a field value using reflection via a cached Field value.

```
public class DirectClassReference implements Experiment {
  public String getName() {return "Direct Class Reference";}
  public void execute() throws Exception {
    Class<?> clazz = Target.class;
    new Target();
  }
}
```

Figure C.6: Experiment code to test the run-time of instantiating a class directly using the new keyword.

```
public class DirectExecution implements Experiment {
  private Target target = new Target();
  public String getName() {return "Direct Execution";}
  public void execute() throws Exception {
    target.run();
  }
}
```

Figure C.7: Experiment code to test the run-time of directly calling a method.

```
public class DirectFieldGet implements Experiment {
  private Target target = new Target();
  public String getName() {return "Direct Field Get";}
  public void execute() throws Exception {
    String value = target.value;
  }
}
```

Figure C.8: Experiment code to test the run-time of directly getting a field value.

```
public class DirectFieldSet implements Experiment {
  private Target target = new Target();
  public String getName() {return "Direct Field Set";}
  public void execute() throws Exception {
    target.value = "1";
  }
}
```

Figure C.9: Experiment code to test the run-time of directly setting a field value.

```
public class DirectInstantiation implements Experiment {
  public String getName() {return "Direct Instantiation";}
  public void execute() throws Exception {
    new Target();
  }
}
```

Figure C.10: Experiment code to test the run-time of directly instantiating a class.

```
public class ReflectionClassReference implements Experiment {
  public String getName() {return "Reflection Class Reference";}
  public void execute() throws Exception {
    Class<?> aClass = Class.forName("org.thesisPerformance.Target");
    aClass.newInstance();
  }
}
```

Figure C.11: Experiment code to test the run-time of directly instantiating a class.

```
public class ReflectionConstructorExperiments implements Experiment {
  public String getName() {return "Reflection Constructor";}
  public void execute() throws Exception {
    Constructor<Target> constructor = Target.class.getConstructor();
    constructor.newInstance();
  }
}
```

Figure C.12: Experiment code to test the run-time of instantiating an instance via a constructor using reflection.

```
public class ReflectionExecution implements Experiment {
  private Target target = new Target();
  public String getName() {return "Reflection Execution";}
  public void execute() throws Exception {
    Target.class.getMethod("run").invoke(target);
  }
}
```

Figure C.13: Experiment code to test the run-time of invoking a method using reflection.

```
public class ReflectionFieldGet implements Experiment {
  private Target target = new Target();
  public String getName() {return "Reflection Field Get";}
  public void execute() throws Exception {
    Object value = Target.class.getField("value").get(target);
  }
}
```

Figure C.14: Experiment code to test the run-time of getting a field value using reflection.

```
public class ReflectionFieldSet implements Experiment {
  private Target target = new Target();
  public String getName() {return "Reflection Field Set";}
  public void execute() throws Exception {
    Target.class.getField("value").set(target, "1");
  }
}
```

Figure C.15: Experiment code to test the run-time of setting a field value using reflection.

```
#!/bin/bash

countLines(){
  totalCount=0;
  for file in `find . -name *.java`; do
     numLines=`cat $file | sed -r ':a; s%(.*)/\*.*\*/%\1%; ta; /\/\*/ !b;
         N; ba' | sed '/^\s*$/d' | sed '/^import/ d' | sed '/^package/ d'
         | wc -l`;
     totalCount=$(( $totalCount + $numLines ));
  done
  echo $totalCount
}

echo Source code files:
countLines
```

Figure C.16: Bash script to count the number of lines of Java code. This script counts the number of lines of a file, ignoring single and multi-line comments, import, and package declarations.

References

[1] "Transfuse for google android." `http://androidtransfuse.org`. Retrieved 2015-11-01.

[2] "Transfuse github repository." `https://github.com/johncarl81/transfuse`. Retrieved 2015-11-01.

[3] "Google android." `http://developer.android.com/index.html`. Retrieved 2015-11-01.

[4] R. Lämmel and S. P. Jones, *Scrap your boilerplate: a practical design pattern for generic programming*, vol. 38. ACM, 2003.

[5] S. P. Jones and R. Lämmel, "Scrap your boilerplate," in *Programming Languages and Systems*, pp. 357–357, Springer, 2003.

[6] G. Kovesdan, M. Asztalos, and L. Lengyel, "Fast android application development with component modeling," in *Cognitive Infocommunications (CogInfoCom), 2014 5th IEEE Conference on*, pp. 515–520, IEEE, 2014.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[8] S. Barnett, R. Vasa, and J. Grundy, "Bootstrapping mobile app development," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pp. 657–660, IEEE Press, 2015.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*. Pearson Education, 2014.

[10] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[11] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 201–211, IEEE, 2014.

[12] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[13] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004.

[14] H. Y. Yang, E. Tempero, and H. Melton, "An empirical study into use of dependency injection in java," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pp. 239–247, IEEE, 2008.

[15] D. R. Prasanna, *Dependency injection*. Manning Publications Co., 2009.

[16] M. D. Ekstrand and M. Ludwig, "Dependency injection with static analysis and context-aware policy," *Journal of Object Technology*, vol. 15, no. 1, 2016.

[17] S. R. Hudli and R. V. Hudli, "A verification strategy for dependency injection," *Lecture Notes on Software Engineering*, vol. 1, no. 1, p. 71, 2013.

[18] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler, *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., 2003.

[19] E. Yourdon and L. L. Constantine, *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.

[20] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[21] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, p. 7, 2007.

[22] J. E. Group, "Jsr330: Dependency injection for java." `https://jcp.org/en/jsr/detail?id=330`, 2009. Retrieved 2015-11-01.

[23] "Don't repeat yourself." `http://c2.com/cgi/wiki?DontRepeatYourself`. Retrieved 2015-11-01.

[24] J. E. Group, "Jsr 269: Pluggable annotation processing api." `https://www.jcp.org/en/jsr/detail?id=269`, 2006. Retrieved 2015-11-01.

[25] G. Kahlout, "Implementing patterns with annotations," in *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, p. 6, ACM, 2011.

[26] "Spring." http://spring.io/. Retrieved 2015-11-01.

[27] "Android annotations." http://androidannotations.org/. Retrieved 2015-11-01.

[28] "Spring." http://seamframework.org/. Retrieved 2015-11-01.

[29] "Google guice." https://github.com/google/guice. Retrieved 2015-11-01.

[30] "Roboguice." https://github.com/roboguice/roboguice. Retrieved 2015-11-01.

[31] "Dagger." http://square.github.io/dagger/. Retrieved 2015-11-01.

[32] "Java enterprise edition." http://www.oracle.com/technetwork/java/javaee/overview/index.html. Retrieved 2015-11-01.

[33] R. Vanbrabant, *Google Guice: agile lightweight dependency injection framework*. APress, 2008.

[34] "Dagger roboguice transfuse comparison." https://github.com/johncarl81/DaggerRoboguiceTransfuseComparison. Retrieved 2015-11-01.