Tagged Programming paradigm

A paradigm to enable the implementation of crosscutting concerns and the use of programmable generic constructs in C-like programming languages

Ramin Zaghi
rzaghi@mosaic3dx.com
Feb 2015

Note to the reader: This is a "work in progress" proposal for a research in programming languages.
A prototype implementation codenamed Tagged-Clang based on the Clang C++ frontend[3] is in progress.
Disclaimer: This work does not represent my employer's views and is solely a result of personal interest in the subject.

Introduction

Aspect-oriented programming was invented as a solution for the "many programming problems for which neither procedural nor object-oriented programming techniques are sufficient"*[Gre97]*. There have been many attempts in providing a solution to popularise ideas behind the Aspect-oriented paradigm including *AspectJ*[1] and IBM's *Hyperspaces* approach *[Per00]*.

However, these approaches either have been abandoned or have not gained widespread acceptance among the developer communities and as a result did not make their way into mainstream compilers (more specifically popular C++ compilers such as GCC or Clang/LLVM).

Tagged programming provides a new and different approach for solving the fundamental programming issues, which Aspect-oriented paradigm has attempted to address. Compared with Aspect-orientation, the methods of the Tagged paradigm for addressing these issues are more resembling of the mainstream programming practices used today[2]. In addition, economically speaking, although they may be more complicated than Preprocessor Macros to learn, they are not only more powerful and safer to use but also applicable wherever Preprocessor Macros or C++ Templates are; and so it provides a better chance of standardisation and adoption by mainstream compilers.

As part of this text, we will also explore a core language feature required by this paradigm: '*Programmable' generic coding*. In addition to being a core requirement for making the Tagged paradigm possible, this capability gives rise to a new way of extending the compiler itself from within the very same source code that is being compiled; a practice which up to now has only ever been explored in the context of developing plug-ins for a specific compiler; and a driving factor in the creation of Clang/LLVM compiler tool-chain[3].

Problem statement

To understand the most fundamental components of Tagged programming and why they are required, let's walk through a real world scenario where separation of concerns is advantageous in the following ways:

A. Improved programmer productivity.
B. Improved clarity and quality of code.
C. Reduced learning curve for those programmers unfamiliar with the code who want to modify it.

A real world scenario: Extending Android™'s "ApplicationInfo" system class

The Android operating system makes use of an object oriented Java class called "ApplicationInfo" to maintain information about Apps that are currently installed on the system. The ApplicationInfo class provides a means of storing, retrieving and communicating the state of an installed App among various internal parts

[1]http://www.eclipse.org/aspectj/ – accessed on 29 Jan 2015

[2]For example, instead of using Extended Backus–Naur Form (EBNF) or Regular Expression-like statements to locate parts of source code for refactoring (as is used in Aspect-oriented tools such as AspectJ), the proposed approach is possible using only common constructs in a traditional language like C.

[3]*http://clang.llvm.org/ – accessed on 29 Jan 2015*

of the operating system. In other words, an interdependent set of classes (tangled classes) use the ApplicationInfo class to achieve this goal.

With the introduction of 64-bit computer Instruction Set Architectures (ISA) to the Android operating system, this class was extended to include a new member variable "requiredCpuAbi" which is responsible for keeping track of which hardware Application Binary Interface (ABI) an App requires to run on a handset – e.g. ARMv8, x86_64, etc. This modification is publicly available in the form of a "Git patch" [4].

To make this possible, it was not enough to add this new member variable to the ApplicationInfo class but it was also required to modify various other classes and files in other parts of the system.

Take the following modifications from the patch as a few examples, which we will address later in the text:

CASE-1: Extend the "PackageSettingBase" class with a new member variable which is declared according to this general pattern:

```
String  <new-member-id>String;
```

CASE-2: Change the class constructor to initialise the new member variable by following the general pattern:

```
<new-member-id>  =  orig.<new-member-id>;
```

CASE-3: Change the "writeDisabledSysPackageLPr" method of the "Settings" class to serialise the new member variable following this general pattern:

```
serializer.attribute(
   null,
   "<new-member-id>",
   pkg.<new-member-id>String );
```

and so on and so forth...

What's the problem?

Learning which parts of such a complex and large codebase to modify to achieve a goal such as the above is a daunting task which can take anywhere from days to weeks.

[4]https://android.googlesource.com/platform/frameworks/base/+/ff0c470%5E!/ – accessed on 3 Feb 2015

In addition, one can see that these are patterns, which one must follow when extending the ApplicationInfo class to define a new system property for Android Apps. You can argue that this particular class, which is designed for this particular purpose, can only be correctly extended if and only if all those modifications were applied to all those other parts of the system. The fact that Java's classes allow for modularity does not help encapsulate this concept of an "App's system property". For example, for every serialised member variable in the ApplicationInfo class, the undocumented design behind AOSP demands that a member variable of type String be also defined in the PackageSettingBase class.

Generic meta-programming using C++ Templates cannot help here since they only provide a limited feature to generalise on types and not the higher level design of a subsystem.

C-style Preprocessor Macros could be used but with many disadvantages including:

A. Decoding what a complex macro does is usually time-consuming especially because such macros hide the final identifiers (e.g. doing a simple search such as "grep in Linux" is not helpful).

B. Such macros do not provide type safety checks even at compile time. For example, you could end up with code that passed a "String pointer" where it is meant to pass an integer. In this case, not passing the right command line arguments to the compiler can simply result in an unintended type conversion.

C. Macros do not intermix with the code at the same level as the language grammar defines. For example, there is no scope rules; a macro can be expanded almost anywhere in the code no matter how wrong the point of expansion is. They are merely copy paste operations in most cases which take place before the parser starts (this is why they are called ***Pre***processor Macros).

Aspect-orientation is also not suitable for several reasons the most important of which is that it is designed to suit cases where the programmer needs to apply a general rule to the entire codebase without exception. For example, it suits scenarios where one needs to say to the compiler, "Insert this wherever in the code there is a call to

that" rather than scenarios such as "Define a new system property".

However, for the purposes of this text we are mostly interested in comparisons with features that are integrated into mainstream programming languages (such as Preprocessors and Generics) rather than features that are provided as secondary tools (such as AspectJ).

The "Tagged" approach

The issues mentioned here can be summarised as follows:

1. First, we would like the code to [naturally] highlight undocumented design decisions. We would like to be able to express to the compiler (and therefore to other programmers reading the source code) that a new member variable in class ApplicationInfo, implies (or must be followed by) a corresponding new member variable in class PackageSettingBase; or that a new "App system property" implies such new member variables in such classes. Doing this requires a facility to define design decisions. In other words, imperatively what ***Action*** a compiler is to take where/when it knows a design decision is to be applied.

2. Second, we would like to encapsulate all design decisions related to the same concept in one place even though they may have implications across the entire codebase. In other words, we would like to be able to collect all design decisions that ***Concern*** a particular aspect of the system (such as "an App's system property") in one place.

3. Third, once we have imperatively defined our design decisions by telling the compiler what actions it is to take, we would like to be able to tell the compiler "where" or "when" to take those actions. For example, if a new "App system property" is defined, exactly where and in which classes the compiler is to insert the new member variables? In other words, we would like to ***Tag*** the code where the compiler is to take action.

4. Last, as a consequence of 1, 2 and 3 above, we are essentially allowing the manipulation of code. We are looking at code as data on which we are operating. This leads us to the idea of a new category of "types" that refer to the elements in the language grammar. A ***Code Type***.

So we would like our language to support the following fundamental features: *Action*, *Concern*, *Tag* and *Code Type*.

An *'Action'* defines an operation on a given node of the program's Abstract Syntax Tree (AST) (e.g. remove it, move it, insert this new sub-AST under it, search it for something, etc.). An *Action* can be thought of as a script within the same source code as being compiled that tells the compiler what to do. *Actions* can be compared to *Advice*s in Aspect-oriented programming.

The *'Concern'* is a parameterized generic type. It is similar to the C++ Template which is a parameterized generic type. *Concern*s can be compared to *Aspect*s in Aspect-oriented programming.

A *'Tag'* is simply a parameter of a *Concern*. It parameterizes the *Concern* just the way a Typename parameterizes a C++ Template. *Tag*s can be compared to *Pointcut*s in Aspect-oriented programming.

'Code Type' is a type (or rather meta-type) that matches the types of AST nodes on which *Actions* operate. Examples of Code Types include "expressions", "statements" or "*integer* variable definitions". Although Aspect-oriented programming implicitly uses this concept, it does not have an explicit way of using it.

The example rewritten using Tagged paradigm

The following example highlights how Tagged programming can be used for separating the concept of an "App system property" with all the benefits discussed above.

In the first part, a *Concern* called "*AppSystemProperty*" is defined that captures such design decisions as "specialise and insert this snippet in this particular location in code".

In the second part, locations of interest in the code are tagged. These indicate, for example, "Where a new variable is to be declared".

```
// In the first part, let's define a Concern called "AppSystemProperty"
concern AppSystemProperty {

  // Some "Tags" will be used as a placeholder that will be replaced.
  // "static" below means that 'all code that follows' will use the same value.
  static tag Serializable serializerObject;

  // Some other "Tags" are used to locate (or tag) certain points/locations in the code.
  // In this case "static" means there's only one such location in the whole codebase.
  static tag  __variable_declaration_statement_t   declareLocation;
  static tag __variable_declaration_statement_t   serializationDeclareLocation; // CASE-1
  static tag  __statement_t                        initializationLocation;       // CASE-2
  static tag __statement_t                        serializeLocation;            // CASE-3

  // Not all placeholders need to be tagged/located in the code. They are only parameters.
  static typename TPropertyType; // What kind of values can this App system property hold?

  private __variable_id my_id; // This will hold the id of each declared Property.

  // The constructor of this "Concern" always emits the required declarations.
  action AppSystemProperty(__variable_id id) { // "id" resembles a macro's parameter in C/C++

      my_id = id;

      __variable_declaration_statement_t  declare_var = { public TPropertyType id; };

       __variable_id idString = id.Append("String"); // Same as "id##String" in C/C++

      __variable_declaration_statement_t  declare_var_str = { public String idString; };

      declareLocation.insertStatement( declare_var );
      serializationDeclareLocation.insertStatement( declare_var_str );          // CASE-1

      // "orig" below must be defined where "initializationLocation" is located.
      // It's similar to textually writing "orig" in a C Preprocessor Macro.
      ApplicationInfo orig; // This is only a declaration about "orig" and not a new object.
      initializationLocation.insertStatement( { id = orig.id; } );              // CASE-2

      final PackageSetting pkg; // This is only a declaration about "pkg"; not a new object.
      __statement_t stmt = { serializerObject.attribute(null, id.Stringify(), pkg.idString); }
      serializeLocation.insertStatement( stmt );                                // CASE-3

  }

}; // This is the end of the Concern

// The second part is listed on the following page...
```

```
// For the second part, let's use the Concern that we defined in the previous section

// == ApplicationInfo source file ==
// The code that follows replaces "TPropertyType" with "Strign"
...
AppSystemProperty::TPropertyType = String;
...
class ApplicationInfo ... {
...
    // This is the location of "declareLocation" where declaration statements are inserted.
    <AppSystemProperty::declareLocation>;
...
    // This is the ApplicationInfo constructor.
    public ApplicationInfo(ApplicationInfo orig) {
    ...
        // We initialise all our Properties here.
        <AppSystemProperty::initializationLocation>;                    // CASE-2
    ...
    }
}

// At the end, declare a new "App system property"
AppSystemProperty("requiredCpuAbi");

// == PackageSettingBase source file ==
// The code that follows replaces "TPropertyType" with "Strign"
class PackageSettingBase ... {
...
    // This is the location of "serializationDeclareLocation".
    <AppSystemProperty::serializationDeclareLocation>;                  // CASE-1
...
}

// == Settings source file ==
...
final class Settings {
...

  void writeDisabledSysPackageLPr(XmlSerializer the_serializer, final PackageSetting pkg) {
  ...

    // "serializerObject" will be replaced with "the_serializer" during the expansion.
    AppSystemProperty::serializerObject = the_serializer;

    // This is where we serialise all our App system properties.
    <AppSystemProperty::serializeLocation>;                             // CASE-3

  ...
  }

...
}
```

Here we declare a new App system property to be implemented by the compiler.

How is this better?

So now declaring a new system property for Apps is as easy as a single line of code as shown above: `AppSystemProperty("requiredCpuAbi");`

Going back to our problem statement, this implies improved programmer productivity.

Furthermore, each of the involved classes is smaller and the design decisions are encapsulated in one place, the "AppSystemProperty" *Concern*. This implies improved clarity of the code.

Also, a new programmer can instantly see the design decisions by only reviewing the *Concern*; design decisions that were previously hidden. This implies a smaller learning curve.

Programmable Generics

The above example demonstrates the concept of a *programmable* Generic type. We can program the compiler to say how to specialise a Generic type. It highlighted one possible use case where *Tag*s help identify points of specialisation and expansion.

There are other possible use cases where programmability of Generics can prove beneficial.

As a hypothetical example, take a C++ Template function `A()` which takes two distinct objects as arguments and fills them with some information:

```
template <typename T1, typename T2>
A(T1& arg1, T2& arg2) {
 ...
}
```

This function is used regularly throughout a large codebase. Analysing the code shows that in every instance two temporary objects of types `T1` and `T2` are defined right before `A()` is called except in the following cases:

a) If they have already been defined for use in a previous call in the same scope in which case the same ones are reused.

b) If two special global objects named `SObj1` and `SObj2` are defined in the same compilation unit in which case these two are used.

A programmer who is unfamiliar with this codebase would need to learn the above rules by either figuring out the pattern by analysing the code or would need to rely on some form of documentation with such detailed information. Besides, a future change in this function could mean modifying all calls to `A()`.

Now let's see how programmable Generics can help. This function could be implemented as an *Action* that took matters into its own hands.

An *Action* can analyse the code where it is being used/called (in other words a call to an *Action* is also a way of *Tag*ging a location in the code). It can then ask the compiler to act on the code based on the context in which it is being called.

The following shows what `A()` might look like if it were written as an *Action*:

```
// Let T1 and T2 be optional parameters.
// They could be parameters of a "concern".
action A (){
 if (thisScope.defines(SObj1) &&
     thisScope.defines(SObj2))
 { thisScope.insert( { // ... some stuff
                       // to fill SObj1/2...
                     } );
 } else {
  // No special objects so we need T1 and T2
  if (!thisScope.defines(T1, _tmp1) &&
      !thisScope.defines(T2, _tmp2))
  { thisScope.insert( { T1 _tmp1;
                        T2 _tmp2; } );
  }
  // ...
  // Emit code to fill both _tmp1 and _tmp2.
  // ...
  // Another example:
  // Search for all "return" statements and
  // emit some housekeeping code before each.
  // ...
 }
}
```

Applications of Tagged Programming

Tagged programming can be used in place of C Prepropcessor Macros and C++ Templates. However, replacing well-established existing features is not the goal.

Tagged programming is meant to address a particular need. It is meant to close the gap between the original developer's thought process or design decisions and what is written as source code without introducing declarative constructs.

By this definition, one area that can benefit from this approach is the domain of software frameworks where generalisation is usually a desirable requirement that contends with ease of integration where multiple frameworks must coexist. Tagged Programming could be used to reduce such contentions by allowing independent frameworks communicate (through the compiler) without having prior knowledge of each other.

In general, any codebase large enough to require a well thought design would benefit from Tagged Programming.

Effects on compilation time

Careless use of this method can lead to an increased compilation time.

One reason is that to compile a compilation unit the compiler might need to execute instructions that are defined in the same unit. In other words, the compiler may take an extra step to interpret those instructions before being able to complete the task of compiling the unit.

This issue can be addressed by defining *Concerns* in separate units to where they are used so that the compiler may access precompiled instructions.

Another reason for a slower compilation time can be the fact that *Actions* consist of arbitrary number of instructions to be executed by the compiler. Misuse of this capability would result in longer compilation times or even infinite loops.

Preventing this issue is partly down to the programmer who uses Tagged Programming features. However, it is also possible to introduce measures such as an optional limit on the number of instructions in each *Action*. The programmer can use or misuse the features as they wish.

The implementation

I had previously implemented a partial prototype using the C/C++ grammar available for Antlr (http://www.antlr3.org/grammar/list.html).
This prototype was in the form of a source-to-source transformation tool.

Today, however, we have the luxury of an industrial strength open source tool chain, Clang/LLVM. By extending the Clang implementation of the C/C++/Objective-C programming languages we can add the linguistic support for Tagged Programming in the tool chain and allow experimenting the practicalities, advantages and disadvantages of this method on real projects such as the Android open source project.

References

[Gre97] Aspect-oriented programming (1997), Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin

[Per00] Multi-Dimensional Separation of Concerns and the Hyperspace Approach (2000), Peri Tarr, Harold Ossh