Bridging AOP and SMP: turning GCC into a metalanguage preprocessor

1 Introduction

The management of computer program's complexity has always been one of the main challenges for software engineers. Methods and techniques to manage the complexity of software, and perhaps even more important, to manage the complexity of the software development process, have been proposed as integral parts of software engineering strategies such as *Family-Based Design* [9], *Object-Orientation* [16], *Generative Programming* [2], *Application-Oriented System Design* [3], and many others.

In order to be effective as regards of complexity management, software engineering strategies usually rely on language constructs and tools. Indeed, the lack of proper language support was probably the reason why *Family-Based Design* could not achieve in the 70's what its niece, *Software Product Lines* [17], is achieving nowadays. For this paper, the language and tool support behind the main techniques of *Generative Programming*, i.e. *Aspect-Oriented Programming* (AOP) [7] and *Static Metaprogramming* (SMP) [10], are those of greater interest. These techniques make it possible for software engineers to act upon the complexity of component-based software by the factorization of non-functional aspects from software components and also by supporting the enforcement of system-wide properties. Using these techniques, a software engineering equipped with a domain engineering strategy such as *Application-Oriented System Design* should be able to decompose a domain in software components that are less complex and in smaller number than otherwise.

Nonetheless, both techniques, AOP and SMP, rely on complex language and tool artifacts to achieve their goals. Static meta-programming implies in a multi-level programming language [6] and is better known from its implementation in the C++ programming language [12] via the `template`[1] mechanism [14, 15]. AOP, besides implying in an aspect-language such as ASPECT-J [1] or ASPECT-C++ [5], also requires a tool, usually called *weaver*, to interweave component and aspect code.

As of today, there is no single tool capable of supporting AOP and SMP techniques at the same time: ASPECT-J is reaching maturity, but JAVA does not support SMP; C++ has full support to SMP, but ASPECT-C++ is rather restrict when the subject is weaving `templates`. For this reason, both techniques are seldom deployed together in the same project, thus restricting the degree of adaptability and configurability that the software can reach.

An additional complication regarding the deployment of SMP and AOP techniques pertains the realm of embedded system. Many of the μ-controllers available today are distributed without any object-oriented language support, usually restricting developers to the features of the C programming language and consequently preventing the use of advanced software engineering techniques. This is a major constraint in a field of every-growing complexity.

This paper describes an approach to combine SMP and AOP techniques in the context of the C++ programming language and to extend their benefits to platforms equipped solely with a C compiler. The approach consists basically in modifying the GCC C++ compiler [**?**] to transform it into a *static meta-programming preprocessor* that parses the C++ input program, executes eventual metaprograms, and outputs the resulting program as an equivalent C program. The C program can subsequently be submitted to ASPECT-C [1] or ASPECT-C++ for weaving or to any standard C compiler for compilation.

[1] The support for static meta-programming in C++ extends beyond the `template` mechanism, including, among others, function inlining, class constant attributes elimination, and short-circuit evaluation of constant expressions with elimination of unused code.

2 A few words on Aspects and Metaprograms

AOP is often taken by its tools (i.e. *aspect weavers* and *aspect languages*) instead of its principles. Indeed, its tools are often deployed to "patch" defect systems instead of being a support to handle non-functional properties of component-based software. In case of properly designed software components, AOP can be practiced even without its traditional tools. For instance, *scenario adapters* are static metaprogrammed constructs that apply aspects to components much in the AOP way, but without requiring a weaver [4]. Nonetheless, the powerfulness of modern aspect languages in what concerns the specification of joint points[2] makes the weaver approach very sound.

On the other hand, SMP is often used in a mechanical way, with programmers being mostly unaware that their class and function `templates` will be executed during program's compilation. Nonetheless, concrete deployment approaches were proved effective: *Generic Programming* [8], the C++ *Standard Template Library* [11], and, more recently, *Generative Programming* [2],

It is nothing but natural that versed software engineers combine AOP and SMP techniques in their projects, and discourses aimed at convincing adepts of one technique to migrate to the other, because allegedly everything that can be done with one can also be done with the other, play minor role here. Indeed, as regards programming languages, C++ and ASPECT-C++ seem to have already answered the significant questions. In what concerns domain engineering, *Application-Oriented System Design* [3] proposes a strategy that is capable of bridging both realms. This leads us to the conclusion that what is really missing to bring SMP and AOP together is mainly tool support.

3 Alternatives to combine AOP and SMP

There are basically two alternatives to deliver the tool support needed to combine AOP and SMP in the context of C++: including a metaprogram execution environment inside the aspect weaver, or delivering a metaprogram preprocessor to execute eventual metaprograms in advance of the weaving process.

The first alternative, embedding a metaprogram execution environment into the aspect weaver, seems to be of great complexity, for it would require the aspect weaver to handle template resolution and instantiation in order to properly manage join points. Template resolution and instantiation are some of the most defying tasks for a C++ compiler [**?**]: template instantiation requires arguments to be checked against the language's type system, while template resolution depends on sophisticated inference techniques specially in the case of function templates where the template parameter must be infered directly or inderectly from the function's parameter. Furthermore, these would be nothing but fundamentals for any metaprogram execution.

The second alternative, executing the metaprogram in advance, releases the weaver from most of these tasks, since template resolution and instantiation, type checking as well as all other tasks concerning metaprogram execution are done by the *metaprogram preprocessor*. Nevertheless, this approach only makes sense if we take on a preexisting C++ compilation environment that is flexible enough to parse the input program, execute eventual metaprograms, and export the resulting single-level program before it is translated into intermediate or machine-level language. The pioneer CFRONT C++ compiler [13] operated much in this way and would be a powerful metaprogram preprocessor had it continued its evolution along the language.

One remaining question to make the case in favor of an external metaprogram preprocessor is the handling of join points targeting static metaprograms themselves. Since the preprocessor would be executed in advance of the weaver, metaprograms are no longer part of a program as it reaches the weaver. Moreover, fragments of the original program affected by metaprograms will look rather different in the preprocessed output. All this could mislead the weaver to produce a corrupted final program. However, as Veldhuizen observed [**?**, **?**], two level language introduced by C++ templates comes with the cost of a awkward syntax and coding style. This syntax and coding style comprise the potential benefits of applying aspects to static metaprograms since the direct application of aspects into a metaprogram can produce undesired side effects. The simple introduction of any code that cannot be handled at compile time inside of a metaprogram would invalidate it, so most aspects developed to be used in conventional programs are not compatible with staticaly solved metaprograms. From this point of view, restricting

[2] A joint point is a certain defined point in program execution where aspect code, usually known as advices, should be weaved. The join point mechanism of an aspect oriented programming language also include concepts as advices and pointcuts, allowing that the description of the aspect can indicate if it should be executed before the execution of the join point, after it, or other possible ways.

```
// Program code

template<class T>
class X{
  T _member;
public:
  T member(){return _member;}
};

int main()
{
 X<int> x;
 cout<< x.member() <<endl;
};

// Aspect code

aspect SimpleAction {
  advice call (" int %::member (...)") : before ()  {
      cout << "Simple Action" ;
  }
  advice call (" int X<int>::member(...)") : after ()  {
      cout << "Another Simple Action" ;
  }
};

// Pre-processed code

class X_int_{
  int _member;
public:
  int member(){return _member;}
};

int main()
{
  X_int_ x;
  cout<< x.member(); //the join points would match at this function call
}
```

Figure 1: Example of parametric classes used along with aspect C++

the application of aspects only to metaprogram results is perfectly reasonable.

In the case of parametric classes the instanciation of the templates will create new classes that can match with the aspect's join points. Although, the correct handling of these join points may be influenced by the support provided by the weaver to the proposed prepreocessor. This support includes the capability of recognizing the newly created classes as matchs for the join points targeting template instances. Aiming to reduce the complexity of the implementation of such support the instantiataion of templates by the preprocessor follows naming rules as simple as possible. The name of a template's instance is exact original instace declaration with the angle brackets exchanged by the underscore character. The figure **??** ilustrate a case where a simple parametric class is used along with a simple ASPECTC++ aspect. At the stated example the *joinpoints* of the aspect are defined as the call of any member function named *foo* from any class that returns an integer, and also the call of any member function named *foo* from a X<int> class and that also returns an integer. Both cases the match should be made with the function *foo()* from class X_INT_. In the first case this match has no special chracteristics since the advice will be applied to any class, but the second example is not of a so direct resolution. The advice targeted a member function of an instance of class X<CLASS T> obtained from X<INT>. In the code where the aspect will be weaved there will be no class named X<INT>, but instead there will be a X_INT_ class. The aspect weaver must be able to translate the X<INT> into X_INT_ and apply the aspect to the correct target.

Figure 2: An overview of the metaprogram preprocessing scheme.

4 The metaprogram preprocessor

In the previous sections we signaled our preference for a *metaprogram preprocessor* as a tool to support the combination of AOP and SMP. In summary, deploying a preexisting compilation environment to implement a metaprogram preprocessor seemed to be more practical then embedding a metaprogram execution environment into an aspect weaver. The decision was also encouraged by the fact that the C++ compiler in the GNU COMPILER COLLECTION [**?**], besides being an open-source compiler, also proved very effective in handling the complex static metaprograms in EPOS [3], the project that motivated this study.

Indeed, the current releases of G++[3] can be used, unmodified, to parse a C++ compilation unit (input program), execute eventual metaprograms, and dump the resulting parse-tree for further processing by our tools. By invoking G++ with the `-fdump-tree-inlined`, we instruct the compiler to do exactly that. Certainly, a parse-tree of this kind is not a C++ program and cannot be used as input to an aspect weaver, but it contains enough information to sustain the reconstruction of a valid C++ program that is equivalent to the input program after the execution of eventual metaprograms.

Indeed, in order to properly support AOP tools, the reconstructed program must be more than logically equivalent to the original program: it must preserve the overall structure of join points, otherwise aspects defined to be used with the original code will not produce the same results when applied to the reconstructed one (see discussion in section 3). That is the main reason why the preprocessor cannot be implemented simply as a compiler back-end.

An schematic of the tool chain behind the metaprogram preprocessor is depicted in figure 2. The parse-tree dumped by G++ is subsequently fed into the *reassembler* to reconstruct the C++ program that will be submitted to the aspect weaver. In this way, SMP duties are processed first, followed by AOP ones. An overview of the tool chain is depicted in figure 2.

[3]The results presented in this paper are based on a metaprogram preprocessor built over G++ version 3.3.

```
template<int n>
struct Factorial { enum{ RET = Factorial<n − 1>::RET ∗ n }; };

template<>
struct Factorial<0> { enum { RET = 1 }; };

int main() { return Factorial<4>::RET; }
```

Figure 3: A statically metaprogrammed factorial calculator.

4.1 The reassembler

In order to sustain the subsequent discussion about the reassembling tool, it is important to understand the basics of the parse-tree dumped by G++. Therefore, let us take the classical factorial metaprogram as an example (figure 3). This metaprogram is able to calculate the factorial of any constant number for which the function is defined during the compilation of the associated C++ unit.

The compilation of the program in figure 3 causes the `Factorial` metaprogram to be executed, producing an output function `main` that barely returns `24` (the factorial of 4). The corresponding G++ dump is presented in figure 4. It consists of a set of parse trees, one for each function defined in the program. The nodes of each tree represent the symbols and types used in the function body, return statement and parameter list. Additional nodes express the contents of the function body as commands, expressions, scope delimiters, and flow control statements of the language.

While processing the parse-tree in figure 4, the reassembler would output a single declaration (node @1) for a function (node @3), whose name is `main` (nome @2), that returns an integer (node @6) and takes no parameters (node @15, type `void`). The body of the function (node @4) consists of a compound statement whose scope is defined by nodes @8 and @24 and whose contents are defined by node @16: a return statement (node @23). The returned expression (node @28) is of integer type (node @6) and has the constant value 24 (node @29). The reassembler would thus produce the following output:

```
int main(void) { return 24; }
```

which is exactly the result of the execution of the metaprogram in figure 3.

Note that the nodes representing types (e.g. node @6) are inserted in the trees as they get used by functions. In this way, a type may be described several times in the dump of a single compilation unit, one for each referring function. Likewise, types that are not referred by any functions do not appear in the dump. Note also that even built-in types such as `integer` and `void` are listed. Consequently, the reassembler must keep track of the types included in the dump to avoid declaring a built-in type or declaring a type twice in the corresponding output file.

One relevant concern in the pre-processing of static metaprogramming is the positioning of each structure in the output code. Though not shown in figure **??**, the parse tree dumped by G++ also includes information about the original files in which a type was declared, either in the compilation unit file or in one of the header files. The reassembler preserves this information in order to include declarations and definitions at their original places while producing the output. This makes the treatment of classic code a simple task since the *classes* and *structs* are rebuilt in their original header files, or wherever they were described originaly. By using the file and line informations retrieved from GCC this case has a straight solution. The functions and expressions are also handled in a direct way during the transverse so that the attainment of the mathematical and logical expressions code is achieved by direct output of the transversed nodes of the function's body.

The first peculiarities arise when we start dealing with template instantiation. Since the original code does not present us with a *class* but with the *template* that represents a *family* of *classes*, we have to output code that was not present at original source in order to represent the instantiate templates. When new code is introduced at the program, there is a special concern on where to introduce this code so that no side-effects are created and the outputted code really represents the desired program and still valid as a C++ program. The problem arises from the fact that since templates are up to be solved classes or metaprograms they can be instantiated with actual parameters from types that were not declared at time of the template declaration, a property that is not shared with common classes and structs. An example of that situation is a simple parametric class named A, with a formal type

```
;; Function int main() (main)
;; enabled by -dump-tree-inlined

@1     function_decl   name: @2      type: @3      srcp: factorial.cc:7
                       C             extern        body: @4
@2     identifier_node strg: main    lngt: 4
@3     function_type   size: @5      algn: 64      retn: @6
                       prms: @7
@4     compound_stmt   line: 7       body: @8      next: @9
@5     integer_cst     type: @10     low: 64
@6     integer_type    name: @11     size: @12     algn: 32
                       prec: 32      min: @13      max: @14
@7     tree_list       valu: @15
@8     scope_stmt      line: 7       begn          clnp
                       next: @16
@9     return_stmt     line: 7       expr: @17
@10    integer_type    name: @18     size: @5      algn: 64
                       prec: 36      unsigned      min: @19
                       max: @20
@11    type_decl       name: @21     type: @6      srcp: <internal>:0
@12    integer_cst     type: @10     low: 32
@13    integer_cst     type: @6      high: -1      low: -2147483648
@14    integer_cst     type: @6      low: 2147483647
@15    void_type       name: @22     algn: 8
@16    compound_stmt   line: 7       body: @23     next: @24
@17    init_expr       type: @6      op 0: @25     op 1: @26
@18    identifier_node strg: bit_size_type         lngt: 13
@19    integer_cst     type: @10     low: 0
@20    integer_cst     type: @10     high: 15      low: -1
@21    identifier_node strg: int     lngt: 3
@22    type_decl       name: @27     type: @15     srcp: <internal>:0
@23    return_stmt     line: 7       expr: @28
@24    scope_stmt      line: 7       end           clnp
@25    result_decl     type: @6      scpe: @1      srcp: factorial.cc:7
                       size: @12     algn: 32
@26    integer_cst     type: @6      low: 0
@27    identifier_node strg: void    lngt: 4
@28    init_expr       type: @6      op 0: @25     op 1: @29
@29    integer_cst     type: @6      low: 24
```

Figure 4: The parse-tree produced by G++ for the factorial program above.

```
// Original Code
template<class T>
class X{
  public:
    T* b;
    void foo()
    {
        b->foo2();
    }
};

class C {
  public:
          void foo2(){};
};

int function()
{
  X<C> xc;
        xc.foo();
        return 1;
}

// pre processed code

class C{
  public:
          void foo2(){};
};

class X_C_{
  public:
          C *b;
                  void foo()
                  {
                          b->foo2();
                  }
};

int function()
{
  X_C_ xc;
        xc.foo();
        return 1;
}
```

Figure 5: Simple parametric class transformation.

parameter T and containing a member of type T. After the declaration of the template at any point where a type B has been declared the template A<B> can be instanciated. If we used the concept of declaring our new A_B_ type at the same point where the original template was declared, a compilation error would be generated since we cannot have any class with members of type B before type B is declared.

To solve that problem we must introduce the new classes just prior to the scope where the templates were instantiated. At this position, the types possibly used in the parameters of the template have already been declared. The original code and resulting code of a simple example can be seen at figure 5.

Although this approach may seem at first glance enough to translate simple parametric classes, it doesn't contemplate to an exception in the rule of using a type only after its declaration. That is the case when we have two classes A and B, being the second nested in the first one. In this case CLASS A may have among its local functions an inline one that makes use of type B before this type is declared. This situation is demonstrated by fig 6. In this situation if our new classes were declared just before the function that uses the template the compiler would emit en error, because class X_A_ would declare a member of type A, while this type is yet undeclared.

In order to solve that possible instatiation problem, the implementation of inline functions originaly imple-

```
// Original code
template<class T>
struct X{
  T I;
        void foo()
        {
          I.g();
        }

};

struct Z{
  void f()
        {
          X<A> a;
        }

        struct A{
          void g(){};
        };

};

// pre-processed code
struct Z{
  class X_A_;
  void f();
  struct A{
          void g();
  };
  struct X_A_{
    A I;
    void foo();
  };
};

inline void Z::X_A_::foo()
{
  I.g();
}

inline void Z::f()
{
  X_A_ a;
}

inline void Z::A::g()
{
};
```

Figure 6: Nested classes with posponed instantiation

mented inside the class declaration are moved to outside the class declaration. Also in the case where a parameter of a template is not yet declared, the instantiation of the template itself is postponed until this type is declared. Looking for keep the pre-processed code equivalent to the original one, the postponed declarations have a foward declaration at their original location in the code, so that for example a function may have a return of the template instance's type.

4.2 Integration with ASPECT-C++

The effective use of the preprocessing technique to bridge *AOP* and *SMP* is bound to the support that *AspectC++* weaver can provide to the combined use with this preprocessor. The most obvious concern on that is the transparency of this process to anyone writing aspects. In other words this mean that the aspect developer ideally should not need to know how the preprocessed code will look like in order to decide about the aspect's join points . This can only be fully acomplished if *aspectC++* weaver is capable of translating names of template instances present at aspects join points to the names that will be replacing those in the preprocessed code.

While *AspectC++* still does not support this kind of preprocessing, the use of simple naming rules may allow that, even without direct support of AspectC++ for this tool, join points targeting the pre-processed code may be easily described by aspect developers. That means that if the aspect weaving tool does not offer the possibility of direct translation from the original template join points, like A<INT>, into the new naming, A_INT_, the developer can easily code its aspects with the translated naming. This same simplicity is expected to make easier the implementation of direct support from *AspectC++*.

5 Extending the use of the technique

The above exposed technique of *template* pre-processing was originally envisioned to fulfill the requirements of *aspect-oriented programming* and *static metarprogramming* combined use. Without modifications it can also be used to replace template code in any circumstance where it can by any reason be undesirable. This kind of situation may arise whenever a tool must operate on C++ code, while metaprogramming creates difficulties for that tool.

One importance example of when the replacement of *template* code may be a gift, is the development of basic software or application software while targeting an architecture or platform that does not contemplate the user with a good *static meta-programming* support on its compilers. That means it can prepare complex C++ code to be compiled by less featured compilers such as those available to some μ-controllers and other architectures. A practical example of this situation would be the LanAI architecture that presents as latest C++ support a GCC compiler at version 2.95. This version of the compiler is not capable of handling complex template features, making impossible to produce a port of operating systems like EPOS [3] that really on th use of *static meta-programming*. By the use of a template pre-processor the *static meta-programmed* code and *parametric classes* could be solved with basis on a recent GCC compiler (version 3.2 by example) and after that have the final stage of compilation and code generation performed by any compiler with suitable port to the given architecture. A deeper extension of this idea could be used to assist the development of a C++ into C language converter. This converter would be a possible solution for the application of *object-oriented programming* when the targeting architecture is supported only by C language compilers. That situation is quite common and represents a restriction to the use of modern techniques in a large field of computer science.

6 Related Work

The works on compatibility between *static metaprogramming* and *aspect oriented programming* are few, and most of it if focused in the application of *aspect oriented programming* by the use of *static metaprogramming*, like *generative programming* [2], *application-oriented system design* [3].

The bridging between *aspect oriented programming* and *static metaprogramming* is deeply bounded to the small set of programing languages that support *static metaprogramming* and the languahes that already have an extension for *aspect weaving*. The main efforts on using *aspect oriented programming* in combination with languages

that support *metaprogramming* were made by ASPECTC++, since is the most advanced atempt on providing *aspect oriented programming* support to a *metaprogramming* capable language. Currently AspectC++ is not yet able to apply templates on join points defined in parametric classes and template functions, although it is capable of parsing correctly template code and apply aspects to non template code used in same compilation units as template code. Recently ASPECTC++ team has made a work in direction of supporting the use of templates in the aspects themselves, an improvement that already bring us closer to a integration between the two technologies. Although this is already an improvement, it is still not enough to provide totally free combination of these technologies.

7 Conclusion

This work focused on the combined use of static metaprogramming with weaving based aspect oriented programming tool. We defended that the combined use of both techniques, not its mutual exclusion, is the key for better modularity and configurability.

The best current opportunity on integrating both techniques is in C++ programming language along with ASPECTC++ aspect weaving mechanism. In order to eliminate the complexities of static metaprogramming tha have been delaying the use of ASPECTC++ on template code, we present a pre-processor approach capable of transforming C++ code in to equivalent C++ code without templates or metaprograms.

More than a way for integrating an aspect oriented programming tool along with metaprogramming, we could show that this same approach may be extended to any technology currently incompatible with templates, including older or less featured compilers that are common on μ-controllers.

Concerning the evolution of this concept and future work, we aim complete C++ features support, as well the use of the technique here described to help in the problem of C++ to C conversion that is most present in the μ-controllers world.

References

[1] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *International Symposium on Foundations of Software Engineering*, pages 88–98, Vienna, Austria, September 2001. ACM SIGSOFT.

[2] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[4] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[5] Andreas Gal, Wolfgan Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *Proceeding of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, U.S.A., October 2001.

[6] Robert Glück and Jesper Jørgensen. An Automatic Program Generator for Multi-Level Specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[8] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the First International Joint Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[9] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[10] Carlo Pescio. Template Metaprogramming: Make Parameterized Integers Portable with this Novel Technique. *C++ Report*, 9(7):23–26, 1997.

[11] P. J. Plauger. The Standard Template Library. *C/C++ Users Journal*, 13(12):20–24, December 1995.

[12] Bjarne Stroustrup. C++ Programming Language. *IEEE Software (special issue on Multiparadigm Languages and Environments)*, 3(1):71–72, January 1986.

[13] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[15] Todd L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[16] Peter Wegner. Classification in Object-oriented Systems. *ACM SIGPLAN Notices*, 21(10):173–182, October 1986.

[17] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.