JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 4, May-June 2008

Panda: a Pattern-based Programming System for Automatic Code Generation

Daniele Mazzeranghi, Softison, Italy

Abstract

This article provides an overview of a pattern-based programming system, named Panda, for automatic generation of high-level programming language code. Many code generation systems have been developed [2, 3, 4, 5, 6] that are able to generate source code by means of templates, which are defined by means of transformation languages such as XSL, ASP, etc. But these templates cannot be easily combined because they map parameters and code snippets provided by the programmer directly to the target programming language. On the contrary, the patterns used in a Panda program generate a code model that can be used as input to other patterns, thereby providing an unlimited capability of composition. Since such a composition may be split across different files or code units, a high degree of separation of concerns [15] can be achieved.

A pattern itself can be created by using other patterns, thus making it easy to develop new patterns. It is also possible to implement an entire programming paradigm, methodology or framework by means of a pattern library: design patterns [8], Design by Contract [12], Aspect-Oriented Programming [1, 11], multi-dimensional separation of concerns [13, 18], data access layer, user interface framework, class templates, etc. This way, developing a new programming paradigm does not require to extend an existing programming system (compiler, runtime support, etc.), thereby focusing on the paradigm concepts.

The Panda programming system introduces a higher abstraction level with respect to traditional programming languages: the basic elements of a program are no longer classes and methods but, for instance, design patterns and crosscutting concerns [1, 11].

1 INTRODUCTION

Software programming based on traditional programming languages (such as C, C++, Lisp, Prolog, Java, C#, etc.) is graphically represented in Figure 1 (where the C# programming language has been used).

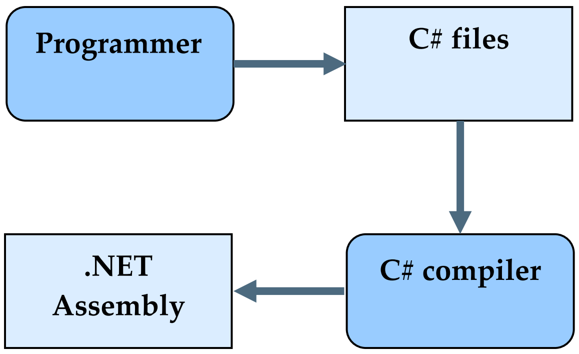

Figure 1: Manual coding process.

Such languages differ under several aspects:

- Abstraction level from the underlying hardware or operating system.
- Support for paradigms and methodologies.
- Extensive core libraries.

However, the size of equivalent real programs is not very different from language to language. In other words, traditional programming languages are not able to automate common repetitive programming tasks.

Many code generation systems have been developed [2, 3, 4, 5, 6] that are able to generate source code by applying code templates, which are defined by means of transformation languages such as XSL, ASP, etc. This new programming model is sketched in Figure 2 (where the target programming language is C#).

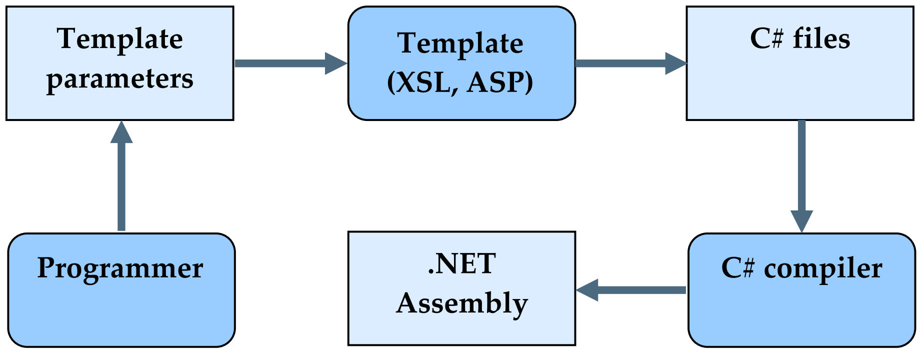

Figure 2: Traditional code generation process.

Such code generation systems are able to generate very sophisticated code snippets, including complete programs, in a variety of programming languages. Nevertheless, these code templates cannot be easily combined because they map parameters and code snippets provided by the programmer directly to the target programming language. In order to modify or extend the generated code, the programmer may edit either the template code or directly the generated code.

The former approach may require a deep understanding of the template structure; furthermore, the template source code could not be available. The latter option is suitable for small changes since the transformation engine cannot be exploited; furthermore, such changes have to be repeated if the template is applied again, due to changes in the template parameters or in the template code itself. However, some of the code generation systems are able to keep certain kinds of manual changes to the generated code if the template is applied afterwards.

Moreover, template code itself could benefit by the transformation engine, but code generation systems do not usually provide support for meta-templates which generate code templates.

Panda (**Pa**ttern **N**etwork **D**evelopment **A**ssistant) is a pattern-based programming system for automatic generation of high-level programming language code. This system attempts to inherit the capability to automate repetitive programming tasks from traditional code generation systems together with the expressive power of programming languages.

A Panda pattern may be classified as functional or procedural. A functional pattern generates an instance of a code model that can be used as input to another functional pattern or may be modified by a procedural pattern, thereby providing an unlimited capability of composition (see Figure 3).

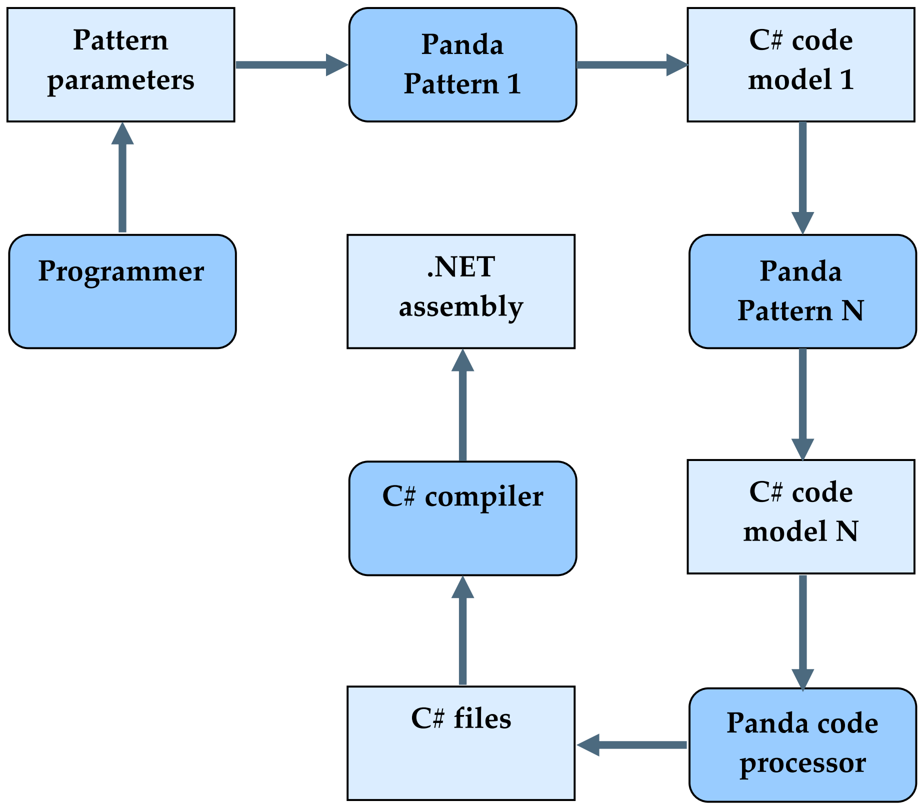

Figure 3: Panda code generation process.

Since such a composition may be split across different files or code units, a high degree of separation of concerns [15] can be achieved.

In order to extend, for example, a Java programming environment with Design by Contract [12], several approaches have been followed, but nearly everyone makes use of a different technology, such as extending preprocessor, compiler or runtime support. Such approaches are very complex and most of development time is spent on integration details instead of focusing on the paradigm concepts. Moreover, two different paradigms that make use of different technologies cannot be likely used at the same time.

On the contrary, Panda makes it possible to create a pattern itself by using other patterns, thus making it easy to develop new patterns. Moreover, an entire programming paradigm, methodology or framework can be easily implemented by developing a small library of patterns. As a consequence, Panda makes it possible to focus on the paradigm concepts instead of the implementation details. Furthermore, several paradigms may be used at the same time by simply joining the corresponding pattern libraries.

Some examples of paradigms and frameworks that can be easily implemented in Panda follow:

- Design patterns [8].
- Design by Contract [12].

- Aspect-Oriented Programming [1, 11].
- Multi-dimensional separation of concerns [13, 18].
- Data access layer.
- User interface framework.
- Class templates [7, 16].

2 SYSTEM ARCHITECTURE

The Panda programming system executes Panda programs by means of a runtime support (RTS), implemented in a given programming language or framework (RTSL). The RTS is responsible for connecting the elements of a Panda program, listed in the following:

- A set of RTS-based patterns, each of which is defined by a class implemented in the RTSL.
- A set of language-based patterns, each of which is defined by a Panda statement (where, in turn, both RTS-based and language-based patterns can be used).
- A sequence of Panda statements which are executed by the RTS by calling the referenced patterns; the result of this execution is a set of RTSL objects which represent the code to be generated.
- A code processor which generates the target code by processing the RTSL objects created by the RTS.

.NET and Java are examples of RTSL.

Some examples of patterns are listed in the following:

- Patterns for creating and modifying code elements (such as classes and methods) of an object-oriented language (such as, for instance, C#, Visual Basic .NET, Java, etc.).
- Patterns for creating and modifying SQL Server scripts.
- Design by Contract patterns [12].
- Patterns for applying design patterns [8].
- Patterns for generating a Windows user interface (by creating, for instance, C# Windows Forms).
- Patterns for generating a Web user interface (by creating, for instance, ASP.NET pages).
- Patterns for generating a data access layer (by using, for instance, the ADO.NET framework).
- Aspect-Oriented Programming patterns.

- Patterns for implementing subject-oriented programming and multi-dimensional separation of concerns.

Some examples of code represented by RTSL objects follow:

- C#, Visual Basic .NET, etc.
- ASP.NET.
- Java.
- SQL Server script.

Some examples of code processors are shown in the following:

- Generator of .NET source files (C#, Visual Basic .NET, etc.).
- Generator of Visual Studio .NET solutions.
- Generator of .NET assemblies (both libraries and applications).
- Generator of Java source files.
- Generator of SQL Server script files.

Figure 3 illustrates an example of execution of a Panda program where the following elements are involved:

- .NET RTSL.
- C# RTSL objects.
- Generator of .NET assemblies.

These elements will also be used in the examples described in the following sections. A Panda IDE (Integrated Development Environment) implementing such elements is available for download from the Panda Web site [14].

3 EXAMPLES OF PANDA PROGRAMS

Some examples of Panda programs which generate C# code are described in this section.

The first example is a very simple program which represents an empty class (`Player1`) by using the functional `EmptyClass` pattern:

```
generate EmptyClass(ClassName = "Player1")
```

Such a program generates the following C# code:

```
public class Player1
{
}
```

This simple example corresponds to the typical philosophy of traditional code generation systems.

By using the procedural `AddReadWriteField` pattern together with the `EmptyClass` pattern, the following Panda program represents the `Player2` class containing the `Name` property (the `AddReadWriteField` pattern adds a C# field together with a get/set property, given its name and type):

```
generate AddReadWriteField
(
  SourceType = EmptyClass("Player2"),
  PropertyName = "Name",
  FieldType = "String"
)
```

that corresponds to the following C# code:

```
public class Player2
{
  private String m_Name;

  public String Name
  {
    get
    {
      return this.m_Name;
    }
    set
    {
      this.m_Name = value;
    }
  }
}
```

This simple example is already sufficient to show the composition capabilities of the Panda language: the output of the `EmptyClass` pattern becomes one of the inputs of the `AddReadWriteField` pattern.

The procedural `AddCSharpMethod` pattern can be used to add the `ToString` method:

```
generate AddCSharpMethod
(
  SourceType = AddReadWriteField
  (
    SourceType = EmptyClass("Player3"),
    PropertyName = "Name",
    FieldType = "String"
  ),
  MethodCode =
  #
    public override String ToString()
    {
      return m_Name;
    }
  #
)
```

Pattern parameters can be specified either by name or position (see, respectively, the `PropertyName` parameter of the `AddReadWriteField` pattern and the `ClassName` parameter of the `EmptyClass` pattern). Primitive values (strings, numbers, source code in another programming language, etc.) are specified by using a delimiter character, which can be either the quote character (`"`) or the sharp character (`#`). The square brackets are used for specifying a list of values.

In order to improve the readability of a Panda program, the `chain` keyword can be used, that makes it possible to omit the assignment of the first parameter. For instance, the previous example can be improved as follows, where the `SourceType` parameter of the `AddReadWriteField` and `AddCSharpMethod` patterns is omitted:

```
generate chain
(
  EmptyClass("Player4")
  AddReadWriteField
  (
    PropertyName = "Name",
    FieldType = "String"
  )
  AddCSharpMethod
  (
    #
      public override String ToString()
      {
        return m_Name;
      }
    #
  )
)
```

4 EXAMPLE OF PATTERN CREATION

RTS-based patterns are mainly used for simple basic patterns (such as, for instance, the `AddReadWriteField` pattern) and performance-critical patterns (such as, for instance, the `AddCSharpMethod` pattern). A pattern may be defined by using a generic software development tool able to generate a module compatible with the RTSL. Alternatively, you can write a Panda program and use a code processor which generates modules compatible with the RTSL. In both cases, several coding conventions have to be followed (involving the base class and the public properties).

Language-based patterns are mainly used for large pattern libraries (such as, for instance, a pattern library for generating a Web user interface), since they are more intuitive with respect to RTS-based patterns:

- no coding conventions are required since the Panda language provides explicit keywords and statements;
- the definition of a language-based pattern can include previously created patterns (both RTS-based and language-based);
- the definition of a language-based pattern does not depend on a particular RTSL;
- the default value of a pattern parameter may be specified.

A typical example of code generation is described in this section: a strongly-typed C# collection [7]. The `Stack1` pattern is defined as follows, where the `functional pattern` statement has been used:

```
define StackClass =
#
  public class StackClass
  {
    private StackSlot m_TopSlot;

    public StackClass()
    {
    }

    public ElementType Pop()
    {
```

```
    ElementType l_TopElement = m_TopSlot.Element;
    m_TopSlot = m_TopSlot.NextSlot;
    return l_TopElement;
  }

  public void Push(ElementType p_NewElement)
  {
    StackSlot l_NewSlot =
      new StackSlot(p_NewElement, m_TopSlot);
    m_TopSlot = l_NewSlot;
  }

  private class StackSlot
  {
    private ElementType m_Element;
    private StackSlot m_NextSlot;

    public StackSlot(
      ElementType p_Element, StackSlot p_NextSlot)
    {
      m_Element = p_Element;
      m_NextSlot = p_NextSlot;
    }

    public ElementType Element
    {
      get
      {
        return m_Element;
      }
      set
      {
        m_Element = value;
      }
    }

    public StackSlot NextSlot
    {
      get
      {
        return m_NextSlot;
```

```
          }
          set
          {
            m_NextSlot = value;
          }
        }
      }
    }
  #

  functional pattern Stack1(StackClassName, ElementType) =
    ExecuteCSharpFunction
    (
      Code = ImportCSharpMethod
      (
      #
        TypeDefinition Stack1(
          String p_StackClassName,
          String p_ElementType,
          TypeDefinition p_StackClass)
        {
          p_StackClass.ReplaceToken(
            "ElementType", p_ElementType);
          p_StackClass.ReplaceToken(
            "StackClass", p_StackClassName);
          return p_StackClass;
        }
      #
      ),
      Parameters =
      [
        StackClassName,
        ElementType,
        ImportCSharpType(StackClass)
      ]
    )
```

The `StackClass` class, together with the `StackSlot` nested class, represents a simple implementation of a stack containing generic objects of type `ElementType`.

The definition of the `Stack1` pattern makes use of the `ExecuteCSharpFunction` pattern, which compiles and executes the C# method specified in the `Code` parameter (the method's parameters are provided by means of the `Parameters` parameter). The object returned by the method (a `TypeDefinition` object in this case) is, in turn, returned by the `ExecuteCSharpFunction` pattern and, afterwards, by the `Stack1` pattern. Therefore, the `ExecuteCSharpFunction` pattern makes it possible to achieve the same expressive power of RTS-based patterns without requiring a C# IDE, at the price of a worse performance.

The `ReplaceToken` method (available in every class of the `DotNetCode` namespace for performing text replacements) is used to insert the name of the specific stack class and the name of the specific type of stack elements.

The `Stack1` pattern can be used in the following Panda program to generate a stack of `Player` objects:

```
generate Stack1("PlayerStack", "Player")
```

Nevertheless, the implementation of the `Stack1` pattern is not very different from an RTS-based pattern, since it is based on the `ExecuteCSharpFunction` pattern. Actually, the `Stack1` pattern can also be implemented by combining basic patterns as follows:

```
functional pattern Stack2(StackClassName, ElementType) =
  chain
  (
    ImportCSharpType(StackClass)
    ReplaceToken("StackClass", StackClassName)
    ReplaceToken("ElementType", ElementType)
  )
```

You can find other and more complex examples of pattern definitions in the Panda samples available for download from the Panda Web site [14]. Such patterns are used to generate a typical three-tier C# application, which makes use of the ADO.NET and Windows Forms frameworks.

5 EXAMPLES OF PANDA LIBRARIES

The following sections will explain how some widespread programming paradigms can be implemented by means of small pattern libraries.

Multi-dimensional separation of concerns

Panda can be used to achieve multi-dimensional separation of concerns [13, 18] (an evolution of Subject-Oriented Programming [9, 17]), which will be explained by means of a class hierarchy for parsing and processing a mathematical expression language [18]. Such a hierarchy is made up of the following classes: `Expression`, `Number`, `BinaryOperator`, `Plus`, `Minus`, `UnaryOperator`, `UnaryPlus`, `UnaryMinus`. For each class, the following virtual methods are defined, as well constructors and properties: `Eval`, `SyntaxCheck`, `Display`, `Process`. If a virtual method for semantic or style checking has to be added to this hierarchy, all classes have to be modified. The Visitor design pattern [8] achieves multi-dimensional separation of concerns, but it presents the following limitations:

- The resulting code structure is rather complex.
- This design pattern must be applied before starting the development of the virtual methods (otherwise, a refactoring is required).
- If a new class representing a new mathematical expression is introduced, the classes which implement this design pattern have to be modified; therefore the separation of concerns holds for the virtual methods but not for the mathematical hierarchy.
- This design pattern does not support any composition of the virtual methods, that could be useful, for instance, for defining an overall checking which includes the syntax and semantic checkings.

Hyper/J [10] is a tool that extends the Java language with a support for multi-dimensional separation of concerns. This tool makes it possible to group Java code snippets into hyperslices which can be combined to generate the requested class hierarchy. In the mathematical expression example, the following hyperslices may be defined [18]:

- *Kernel* slice, which contains constructors and properties.
- *Display* slice, which contains the `Display` methods.
- *Evaluation* slice, which contains the `Eval` methods.
- *Syntax Check* slice, which contains the `Check` methods for syntax checking.

together with a rule for composing them. Every slice contains a separate Java code snippet for each mathematical class, thus fully achieving separation of concerns. For example, if a `Check` virtual method for semantic checking has to be introduced, a new slice is added. If a new mathematical expression is introduced, all slices have to be modified, but by adding separate code snippets.

Panda, too, is able to fully achieve separation of concerns in the following way:

- As usual, a separate `chain` statement is used for each mathematical class.
- Each method of each slice is specified in a separate `execute` statement, which makes use of the `AddCSharpMethod` pattern.
- The rule for composing the hyperslices is implemented by a specific pattern (`MergeMethods`).

The *Kernel* slice can be implemented in Panda as follows (constructors and properties have been omitted):

```
define Expression = chain
(
  EmptyClass("Expression")
  MakeTypeAbstract()
)

define Number = chain
(
  EmptyClass("Number")
  AddBaseType("Expression")
)

define BinaryOperator = chain
(
  EmptyClass("BinaryOperator")
  AddBaseType("Expression")
)

define Plus = chain
(
  EmptyClass("Plus")
  AddBaseType("BinaryOperator")
)

define Minus = chain
(
  EmptyClass("Minus")
  AddBaseType("BinaryOperator")
)

define UnaryOperator = chain
(
  EmptyClass("UnaryOperator")
  AddBaseType("Expression")
)

define UnaryPlus = chain
(
  EmptyClass("UnaryPlus")
  AddBaseType("UnaryOperator")
)

define UnaryMinus = chain
```

```
(
  EmptyClass("UnaryMinus")
  AddBaseType("UnaryOperator")
)
```

The code of the *Display* slice follows, where the implementation of the methods has been omitted (the code of the *Evaluation* slice is analogous):

```
execute AddCSharpMethod
(
  Expression,
  #
    public void Process()
    {
      Display();
    }
  #
)

execute AddCSharpMethod
(
  Expression,
  #
    public abstract void Display();
  #
)

execute AddCSharpMethod
(
  Number,
  #
    public override void Display()
    {
    }
  #
)

execute AddCSharpMethod
(
  BinaryOperator,
  #
```

```
    public override void Display()
    {
    }
  #
)

execute AddCSharpMethod
(
  Plus,
  #
    public override void Display()
    {
    }
  #
)

execute AddCSharpMethod
(
  Minus,
  #
    public override void Display()
    {
    }
  #
)

execute AddCSharpMethod
(
  UnaryOperator,
  #
    public override void Display()
    {
    }
  #
)

execute AddCSharpMethod
(
  UnaryPlus,
  #
    public override void Display()
```

```
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    UnaryMinus,
    #
      public override void Display()
      {
      }
    #
  )
```

The code of the *Syntax Check* slice follows (the implementation of the methods has been omitted):

```
  execute AddCSharpMethod
  (
    Expression,
    #
      public void Process()
      {
        Check();
      }
    #
  )

  execute AddCSharpMethod
  (
    Expression,
    #
      public abstract void Check();
    #
  )

  execute AddCSharpMethod
  (
    Number,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    BinaryOperator,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    Plus,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    Minus,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    UnaryOperator,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    UnaryPlus,
    #
      public override void Check()
      {
      }
    #
  )

  execute AddCSharpMethod
  (
    UnaryMinus,
    #
      public override void Check()
      {
      }
    #
  )
```

The names of the methods defined in the *Semantic Check* slice are identical to those defined in the *Syntax Check* slice. Therefore, the `MergeMethods` pattern renames the conflicting methods and creates a new method which calls them. Such a pattern is applied as follows:

```
execute MergeMethods
(
  TypeList =
  [
    Expression,
    Number,
    BinaryOperator, Plus, Minus,
    UnaryOperator, UnaryPlus, UnaryMinus
  ],
  Methods = ["Check", "Process"]
)
```

The resulting C# code follows (only the `Expression` and `Number` classes are shown):

```
public abstract class Expression
{
  private void Process_1()
  {
    Display();
  }

  public abstract void Display();

  private void Process_2()
  {
    Eval();
  }

  public abstract void Eval();

  private void Process_3()
  {
    Check();
  }

  public abstract void Check();

  public void Process()
  {
    Process_1();
    Process_2();
    Process_3();
  }
}

public class Number : Expression
{
  public override void Display()
  {
  }

  public override void Eval()
```

```
    {
    }

    private void Check_1()
    {
    }

    private void Check_2()
    {
    }

    public override void Check()
    {
      Check_1();
      Check_2();
    }
  }
```

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [1, 11] is a programming paradigm for achieving separation of crosscutting concerns, such as tracing of method calls. A crosscutting concern is defined by an aspect, which is made up of a pointcut and an advice. A pointcut is a collection of joint points, each of which represents a point in the execution of a program, such as method calls, field assignments, etc. An advice specifies the code to be executed when one of the joint points is reached.

AOP can be easily applied to several widespread concerns:

- Tracing.
- Profiling.
- Logging.
- Security.
- Persistence.
- Caching.
- Consistency checking.

AOP can be easily implemented by means of a library of Panda patterns. One of them will be applied to a very simple mathematical library, whose methods may be called only if the library is properly licensed. Given the following Panda code which represents the C# classes of the mathematical library:

```
define BasicMath = chain
(
  EmptyClass("BasicMath")
  AddCSharpMethod
  (
    #
      public Double Square(Double p_Number)
      {
        return Raise(p_Number, 2);
      }
    #
  )
  AddCSharpMethod
  (
    #
      public Double Cube(Double p_Number)
      {
        return Raise(p_Number, 3);
      }
    #
  )
  AddCSharpMethod
  (
    #
      private Double Raise(Double p_Base, Double p_Exponent)
      {
        return Math.Pow(p_Base, p_Exponent);
      }
    #
  )
)

define AdvancedMath = chain
(
  EmptyClass("AdvancedMath")
  AddCSharpMethod
  (
    #
      public Double Cos(Double p_Number)
      {
        return Math.Cos(p_Number);
      }
    #
  )
)

define MathLibraryLicensing = chain
(
  EmptyClass("MathLibraryLicensing")
  AddCSharpMethod
  (
    #
      public static void CheckLicense()
      {
        // ...
      }
    #
  )
)
```

the following Panda code adds license checking at the beginning of all public methods, by using the `BeforeExecutionAspect` AOP pattern:

```
  execute BeforeExecutionAspect
  (
    TargetTypes = [BasicMath, AdvancedMath],
    Pointcut = "public * *.*(..)",
    Advice = # MathLibraryLicensing.CheckLicense(); #
  )
```

as shown by the resulting C# code:

```
  public class BasicMath
  {
    public Double Square(Double p_Number)
    {
      MathLibraryLicensing.CheckLicense();
      return Raise(p_Number, 2);
    }

    public Double Cube(Double p_Number)
    {
      MathLibraryLicensing.CheckLicense();
      return Raise(p_Number, 3);
    }

    private Double Raise(Double p_Base, Double p_Exponent)
    {
      return Math.Pow(p_Base, p_Exponent);
    }
  }

  public class AdvancedMath
  {
    public Double Cos(Double p_Number)
    {
      MathLibraryLicensing.CheckLicense();
      return Math.Cos(p_Number);
    }
  }

  public class MathLibraryLicensing
  {
    public static void CheckLicense()
    {
      // ...
    }
  }
```

Design by Contract

Design by Contract [12] could be implemented by means of the AOP patterns presented in the previous section. However, creating a specific Panda pattern library simplifies the application of consistency checking.

For example, the `AddInvariantCondition` pattern could be defined that adds a given condition involving the fields of a given class to the class itself (inside a method conventionally named `Invariant`), whereas the `AddInvariantChecking` pattern is used to execute the `Invariant` method in every public or internal method or property. These two patterns can be used to check the content of the `m_Name` field of the `Player5` class:

```
  generate chain
  (
    EmptyClass("Player5")
    AddReadWriteField
    (
      PropertyName = "Name",
      FieldType = "String"
    )
    AddCSharpMethod
    (
      #
        public Player5(String p_Name)
        {
          m_Name = p_Name;
        }
      #
    )
    AddCSharpMethod
    (
      #
        public override String ToString()
        {
          return m_Name;
        }
      #
    )
    AddInvariantCondition
    (
      "m_Name != null && m_Name.Length > 0"
    )
    AddInvariantChecking()
  )
```

as shown by the resulting C# code:

```
  public class Player5
  {
    private String m_Name;

    public String Name
    {
      get
      {
        try
        {
          this.Invariant();
          return this.m_Name;
        }
        finally
        {
          this.Invariant();
        }
      }
      set
      {
        try
        {
          this.Invariant();
          this.m_Name = value;
        }
        finally
        {
          this.Invariant();
        }
      }
    }

    public Player5(String p_Name)
    {
      try
      {
        m_Name = p_Name;
      }
      finally
      {
        this.Invariant();
      }
    }

    public override String ToString()
    {
      try
      {
        this.Invariant();
        return m_Name;
      }
      finally
      {
        this.Invariant();
      }
    }

    public void Invariant()
    {
      if (!(m_Name != null && m_Name.Length > 0))
        throw new System.ApplicationException(
          @"The following invariant condition has been violated:
              m_Name != null && m_Name.Length > 0.");
    }
  }
```

Design Patterns

The design patterns presented in [8] can be applied more easily by means of a Panda pattern library. For example, the Decorator design pattern applied to a drawing library is sketched in Figure 4.

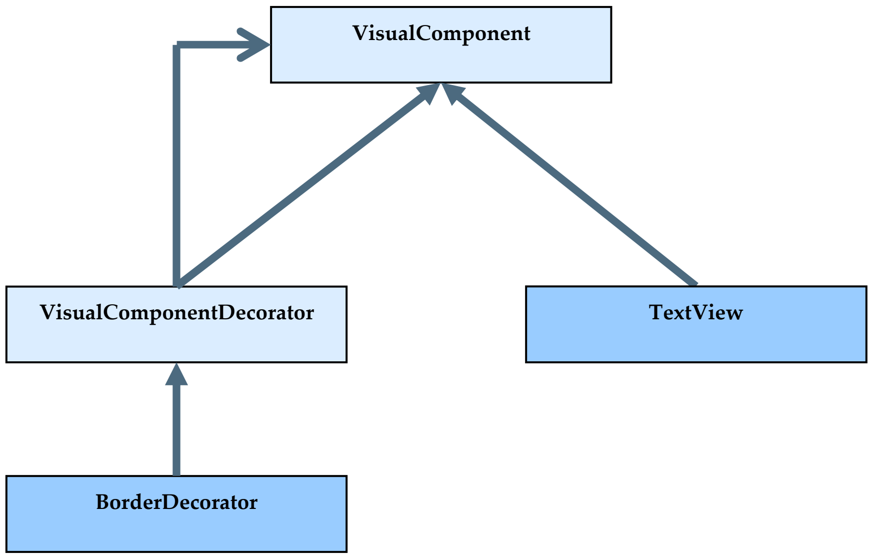

Figure 4: The Decorator design pattern.

The VisualComponentDecorator class contains a field of type VisualComponent and a virtual method for each method of the VisualComponent interface. Such virtual methods provide the default behaviour of a decorator class, by forwarding any method call to the object of type VisualComponent. Hence, the BorderDecorator class overrides only the VisualComponent methods involved in drawing a border.

The VisualComponentDecorator class can be generated automatically by means of a Panda pattern (AbstractDecorator), applied to the VisualComponent interface as follows:

```
define VisualComponent = ImportCSharpType
(
#
  public interface VisualComponent
  {
    void Draw(System.Drawing.Graphics p_Graphics);
    Boolean IsEnabled();
  }
#
)

define VisualComponentDecorator = AbstractDecorator
(
  "VisualComponentDecorator",
  VisualComponent
)
```

As a consequence, the definition of the IsEnabled method may be omitted from the definition of the BorderDecorator class:

```
define BorderDecorator = ImportCSharpType
(
#
  public class BorderDecorator : VisualComponentDecorator
  {
    private Int32 m_Width;

    public BorderDecorator(
      VisualComponent p_VisualComponent, Int32 p_Width) :
        base(p_VisualComponent)
    {
      m_Width = p_Width;
    }

    public override void Draw(
      System.Drawing.Graphics p_Graphics)
    {
      base.Draw(p_Graphics);
      DrawBorder(p_Graphics);
    }

    private void DrawBorder(
      System.Drawing.Graphics p_Graphics)
    {
      // ...
    }
  }
#
)
```

that generates the following C# code:

```
  public interface VisualComponent
  {
    void Draw(System.Drawing.Graphics p_Graphics);
    Boolean IsEnabled();
  }

  public abstract class VisualComponentDecorator :
    VisualComponent
  {
    private VisualComponent m_DecoratedObject;

    protected VisualComponentDecorator(
      VisualComponent p_DecoratedObject)
    {
      m_DecoratedObject = p_DecoratedObject;
    }

    public virtual void Draw(System.Drawing.Graphics p_Graphics)
    {
      m_DecoratedObject.Draw(p_Graphics);
    }
    public virtual Boolean IsEnabled()
    {
      return m_DecoratedObject.IsEnabled();
    }
  }

  public class BorderDecorator : VisualComponentDecorator
  {
    private Int32 m_Width;

    public BorderDecorator(
      VisualComponent p_VisualComponent, Int32 p_Width) :
        base(p_VisualComponent)
    {
      m_Width = p_Width;
    }

    public override void Draw(
      System.Drawing.Graphics p_Graphics)
    {
      base.Draw(p_Graphics);
      DrawBorder(p_Graphics);
    }

    private void DrawBorder(System.Drawing.Graphics p_Graphics)
    {
      // ...
    }
  }
```

6 CONCLUSIONS

Tha main benefit of the Panda programming system is to increase the abstraction level of software programs. Instead of talking about classes, fields, methods, etc., a Panda program is made up of decorators, tracing of method calls, data layers and user interfaces derived from a business layer, etc. Hence, adopting Panda requires a new approach to software development, that can be followed in different ways:

- Development of a new project by using only Panda code: only the Panda IDE is required and all Panda features can be exploited.

- Import, in a new Panda project, of an existing project (developed, for instance, in C#), extended by using only Panda code: only the Panda IDE is required and the legacy code can be both extended (by generating new classes) and enriched (for instance by means of Design by Contract patterns), but applying a pattern to the legacy code is more tricky.
- Extension of an existing project by using both a traditional language and Panda: handling the legacy code is tricky as before and, furthermore, a Panda add-in for a traditional IDE is required.

However, this is just the first stage of the Panda programming system, which could evolve in several directions:

- Development of pattern libraries for the programming paradigms or frameworks presented in this article: Design patterns, Design by Contract, Aspect-Oriented Programming, Multi-dimensional separation of concerns, etc.
- Development of pattern libraries for other widespread programming paradigms or frameworks: three-tier architectures, ASP.NET, multithreading, unit testing (including user interface modules), the model-view-controller pattern, etc.
- Development of statement-level patterns (such as, for instance, executing a given C# code snippet for each element of an array).
- Development of patterns for crosscutting concerns (tracing, profiling, caching, etc.).
- Evolution of the Panda language (such as pattern inheritance).
- Support for other target programming languages (Visual Basic .NET, Java, C++, etc.).
- Introduction of new code processors (such as, for instance, a generator of ASP.NET pages).
- Enhancement of the C# Panda IDE.
- Integration of Panda in other development tools (such as, for instance, Microsoft Visual Studio .NET).
- Graphic designer to simplify pattern definition, particularly for template-like patterns.

REFERENCES

[1] AspectJ Web site: http://www.eclipse.org/aspectj/.

[2] Code Generation Network Web site: http://www.codegeneration.net.

[3] CodeCharge Web site: http://www.codecharge.com.

[4] CodeSmith Web site: http://www.codesmithtools.com.

[5] Codify Web site: http://www.workstate.com/codify/.

[6] DeKlarit Web site: http://www.deklarit.com.

[7] Ecma International. "C# Language Specification". Standard ECMA-334: http://www.ecma-international.org/publications/standards/Ecma-334.htm, 2006.

[8] Gamma E., Helm R., Johnson R. and Vlissides J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[9] Harrison W. and Ossher H. "Subject-Oriented Programming (A Critique of Pure Objects)". Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications: 411-428, 1993.

[10] Hyper/J Web site: http://www.alphaworks.ibm.com/tech/hyperj/.

[11] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J. and Irwin J. "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming: 220-242, 1997.

[12] Meyer B. "Object-Oriented Software Construction (2nd Edition)". Prentice-Hall, 1997.

[13] Multi-Dimensional Separation of Concerns Web site: http://www.research.ibm.com/hyperspace/.

[14] Panda Web site: http://www.softison.com/panda/.

[15] Parnas D. L. "On the Criteria To Be Used in Decomposing Systems into Modules". Communications of the ACM, 15 (12): 1053-1058, 1972.

[16] Stroustrup B. "The C++ Programming Language (3rd Edition)". Addison-Wesley, 1997.

[17] Subject-Oriented Programming Web site: http://www.research.ibm.com/sop/.

[18] Tarr P., Ossher H., Harrison W. and Sutton S. "N degrees of separation: multi-dimensional separation of concerns". Proceedings of the 21st international conference on Software engineering: 107-119, 1999.

About the author

Daniele Mazzeranghi is a software developer with over 10 years of industrial experience. He has recently founded the Softison company to develop, patent and market the Panda programming system. His interests include object-oriented programming, code generation systems and the .NET framework. He can be reached at daniele.mazzeranghi@softison.com. See also www.softison.com.