2005 IEEE International
Symposium on Signal Processing
and Information Technology

Design Patterns Automation with Template Library

Sergiu Dascalu[1], Ning Hao[1], Narayan Debnath[2]

[1]Department of Computer Science and Engineering
University of Nevada, Reno
Reno, NV, 89523, USA
{dascalus, haoning}@cse.unr.edu

[2]Department of Computer Science
Winona State University
Winona, MN, 55987, USA
ndebnath@winona.edu

Abstract - Design patterns offer reusable solutions to particular software design problems. Design Patterns Automation is an approach that applies design patterns at the implementation stage of the software development life cycle. Inspired by two commonly used template libraries, Active Template Library and Standard Template Library, and one of the most popular generic programming technologies, C++ templates, this paper introduces a new method for achieving design patterns automation. This method differs from the currently available UML-based and wizard-based design patterns automation techniques and provides support for increased flexibility, expandability and compatibility in developing software using design patterns. Seven of the patterns proposed by Gamma et al. have been implemented using C++ templates, namely *singleton, factory method, visitor, memento, strategy, iterator,* and *decorator*. To illustrate the method proposed, details of *singleton* and *decorator* implementations are provided and a larger "Check" example developed using the decorator template is presented. The paper also includes a comparison with similar approaches and presents several directions of future work.

Keywords - Design patterns, Design patterns automation, Template library, Software design, Implementation.

I. INTRODUCTION

A *design pattern* represents a reusable solution to a particular software design problem [1], which is primarily a design issue. Design Patterns Automation (DPA) is an approach that applies design patterns to software construction [2], which is primarily an implementation issue. This paper focuses on how to implement design patterns using templates. The main objective is to separate the design pattern implementation from the business logic implementation. Thus, the users can focus on developing the business logic of their interest. The answers to why one uses design patterns and what kind of benefits can be achieved by using them are not within the scope of this paper. It is assumed that the software designer has decided to use design patterns and relies on the coder to implement them. Currently, existing DPA tools are mostly wizard-based, which first ask for user input then generate code for easy implementation. The advantage is that these tools are tied with the Unified Modeling Language (UML) [3], so they have the ability to bundle design and implementation. But there are also certain disadvantages: the tools are not free and wizards lack flexibility, meaning that the users have not sufficient control over what they can do.

In this paper, inspired by the Standard Template Library (STL) [4] and the Active Template Library (ATL) [5], a new approach is introduced to achieve DPA. Furthermore, the work described in this paper aims to create an open source template code library, named Design Patterns Template Library (DPTL), which is intended to incorporate design patterns implementations using C++ templates. The users will have available the distributed source code instead of the compiled executable programs and thus could easily make changes suitable for their business needs. This paper divides the design patterns template implementation into STL style and ATL style, based on whether the involving pattern class can operate alone or needs interaction with other classes.

Since the patterns proposed by the "Gang of Four" (GoF) are the cornerstone of design patterns technology, this paper focuses on "templating" patterns presented in the GoF book [1]. In total, there are 23 patterns presented in that book; the work on which this paper is based fully covered seven [6]. The paper provides details of two template-based pattern implementations and illustrates the proposed approach with a larger example (a "Check" program). In essence, we propose a certain convention for templating patterns, so patterns can be added to a library without significant effort. Once this approach is completed, it could be bring a useful contribution to the design patterns technology and the software engineering community.

In its remaining part, this paper is organized as follows: Section II provides background information, Section III gives an overview of the problem tackled and the proposed solution, Section IV details the automation of the singleton and decorator patterns, Section V shows the application of the decorator pattern template-based automation using a Check program example, Section VI includes a comparison with related work, and Section VII presents several directions of future work as well as the paper's conclusions.

II. BACKGROUND

A. Design Patterns

A design pattern, as defined by the GoF, is a description of "communicating objects and classes that are customized to solve a general design problem in a particular context" [1]. In other words, a design pattern is a reusable solution for a particular design problem. Gamma *et al* introduced 23 patterns in their book (Table 1), and these patterns have been considered the fundamental building blocks for more complex patterns. Every design pattern is characterized by four essential elements: *pattern name, problem, solution,* and *consequence*. In this sense, a design pattern is a higher level of abstraction, factored out from a common kind of successful software design. Notably, a design pattern can be implemented in almost any programming language.

Patterns usually exist in software that has already been developed. To discover and publish a pattern is called *pattern mining* [7]. This activity has accounted for a significant segment of pattern research. For a pattern to be useful, it is critical how it is published, because this influences its usage. In essence, a pattern's life and utility depend on how often it can be reused. There are two forms to follow for pattern publishing: one is the GammaForm [1], the other is the CoplienForm [8]. The *singleton* pattern is presented in Table 2 using the GammaForm.

Table 1. Pattern categories based on functions [1]

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter	Interpreter Template method
	Object	Abstract method Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

The *singleton* pattern has been used widely, for example in the open source software Java log4j [9], the Microsoft .NET remoting [10], and many other software applications.

Another area of pattern-related research is concerned with how to use patterns. After all, *reusing patterns* in new designs is the ultimate goal of the approach. This includes finding the relationships between patterns and theoretical ways to reuse patterns, like Pattern Oriented Analysis and Design (POAD)[11].

B. Design Patterns Automation

Not only researchers are interested in reusing patterns, but also some of the major software developing companies [12]. The difference is that the latter are more interested in practical ways of reusing patterns, not in their theory. This interest focuses on how to use patterns in real life situations. This also encompasses Design Patterns Automation (DPA), because a design pattern itself is mostly an idea at the design level, meaning that it can be implemented in many programming languages to fulfill a particular design issue. DPA generally works with one programming language and provides an easy way for people to integrate their business design with design patterns. The overall benefit is that once the design has involved patterns, it can be easily implemented with DPA. The users do not have to start from scratch by writing customized design pattern code.

Table 2. *Singleton* Pattern in GammaForm [1]

Name	Singleton
Problem	Ensure a class has only one existing instance, and provide a global point of access to it.

Solution	The Singleton class is declared as:
	`class Singleton {`
	`public:`
	`   static Singleton* Instance();`
	`protected:`
	`   Singleton();`
	`private:`
	`   static Singleton* _instance;`
	`};`
	The corresponding implementation is
	`Singleton* Singleton::instance=0;`
	`Singleton* Singleton::Instance () {`
	`   if (_instance == 0) {`
	`     _instance = new Singleton;`
	`   }`
	`   return _instance;`
	`}`

Conse-quence	The Singleton pattern has several benefits: (i) this solution provides controlled access to a unique instance; (ii) it is a much better solution compared with a global variable; (iii) it can be fine-tuned into a configurable singleton.

For each DPA to work, it has to address two sensitive aspects of design patterns, namely *uniqueness* (each pattern needs to address only one particular design problem), and *generalizibility* (that is, a particular pattern should be popular enough to be worthy of the automation effort). Generalizibility makes DPA possible, but uniqueness makes

automation work relatively difficult. Thus, an approach must be developed to cover most design patterns with minimum of developing work.

Bulka describes DPA at two levels, static and dynamic [2]. Most of current DPA tools have a tight involvement with UML tools, especially with those UML tools for automatic code generation. Examples of such tools include Borland Together [13], UMLStudio [14], and ModelMaker [15]. A *static DPA* such as UMLStudio stores all the necessary pattern information, and when the user wants to use a pattern, the pattern can be created, but the generated code is limited to the pattern itself. Thus, it cannot interact with other existing UML classes or interfaces. *Dynamic DPA* tools such as Together or ModelMaker are different, they not only can create pattern UML models, but can also allow new created pattern UML models to interact with other class models. This is a significant advantage, because most design patterns defined by GoF are considered building blocks for new kind of patterns, the *composite patterns*, which are composed by two or more basic patterns. Some of these UML tools also have the ability to generate code based on UML diagrams.

All these UML-based DPA tools use GUI to fulfill each pattern's uniqueness. Together uses dialog boxes, ModelMaker and UMLStudio use wizards. For each different pattern, a dialog or wizard panel has to be implemented. For example, the steps of pattern automation in Borland Together [13] are as follows:

Step1: Choose a pattern to use, and then input all the necessary parameters (pattern properties) in the dialog panel provided.

Step2: The Together tool can then create the corresponding UML class diagram for the pattern. For example, in the case of *Singleton*, a *SingletonFoo* class and a *SingletonFooFac(tory)* class can be created (the latter can also be inherited by a third class, the *SubSingletonFoo*). The Together environment can also generate code for the UML diagram shown in Fig. 1.

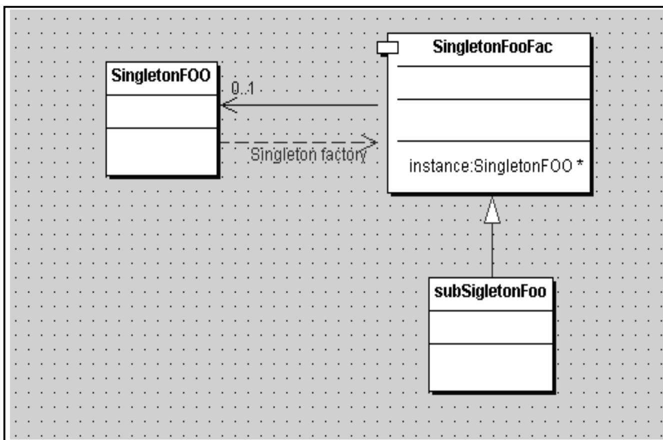

Fig. 1. UML diagram of *Singleton* in Borland Together [13]

C. Template-Based Programming

A C++ template allows using types as parameters, which results in a great deal of code reuse. This is also called Generic Programming (GP) [16], a discipline that studies the systematic organization of reusable software components. C++ templates are the most used technique for GP, but there are other, such as Microsoft .NET attribute-based programming [17]. The two most successful template GPs are STL [4] and Microsoft's ATL [5].

STL is a subset of the C++ library, which includes most useful data structures and algorithms, such as hash, vector, and linked list data structures as well as many types of sorting algorithms. Every component in STL uses a template, which takes types as parameters. Whenever a data structure or algorithm is needed, the developer needs only to simply pass suitable type(s) to get the desired results.

For example, *vector<int>* and *vector<myClass>* will result in an *int* vector and a *myClass* vector. Thus, it becomes simple and straightforward to achieve code reuse. However, this requires the user to have a good understanding of the library component before using it. For the above case, the user must know what a vector is and what it can do in terms of a specific software solution.

ATL is a subset of Microsoft VC++, which deals with COM (Component Object Model) using C++ templates and multi-inheritance. All COM libraries need to implement the *IUnkown* interface regardless of their types [18]. Figure 2 presents a code snippet of the *CFoo ATL* class definition.

```
// CFoo

class  ATL_NO_VTABLE CFoo:
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CFoo, &CLSID_Foo>,
public IDispatchImpl<IFoo, &IID_IFoo,
       &LIBID_TRAILLib>

{public:
       CFoo(){}
}
```

Fig. 2. CFoo ATL class definition

As shown above, ATL is more complicated than STL. ATL allows interaction between user classes and COM classes through multi-inheritance.

This interaction is very important for dynamic DPA tools. From this point of view, the STL method is considered more static. To use templates for DPA, both STL and ATL styles can be applied. For patterns that mostly operate alone, an STL-like template can be created. On the other hand, all patterns can be implemented in an ATL-like library. An ATL-like library also provides a way for creating composite patterns.

III. PROBLEM AND PROPOSED SOLUTION

A. What Problem the Design Patterns Template Library Addresses?

This paper proposes a new way to achieve DPA. From a software engineering point of view, design patterns provide reusable and typically successful design solutions. However, at the implementation stage the users will need to write code to fulfill their design. This requires development effort and usually involves programming errors.

DPA comes here into play, by significantly reducing both the programming workload and the likelihood of errors when implementing a design pattern from the scratch. Research shows that the majority of failed applications that tried to apply design patterns were due to incorrect implementation [19]. This proves that DPA can be an attractive and useful approach in the software development processes that rely on design patterns.

A simple analogy to describe DPA is provided by a model make, which can have different shapes. These shapes are analog to different design patterns. The model make can be created using various kinds of material, such as clay, cement and metal. In a way, this can be analogous to different business software requirements.

In this paper, a new approach for achieving DPA is introduced. The approach uses C++ templates, a powerful modern generic programming technology. The design patterns implementations proposed by this approach are bundled into a template library, similar to STL and ATL. The template library we propose is described next.

B. The DPTL Project

DPTL is a template library that uses C++ templates to achieve DPA. A software development project itself, it went through typical software lifecycle stages, namely requirements specification, design, implementation, testing, and release. In summary, the main requirement specifications of DPTL are as follow:

- This project should be implemented using the open-source approach, such that the end users can have unrestricted access to the source code;
- Each DPA implementation should be correct and thoroughly tested;
- Each DPA implementation should follow as much as possible either the STL or the ATL style, such that it will be easier to use and it will lead to less errors and misuses.

The reason for the open source solution is that it would also be possible to gain more audience for thorough testing before release. Notably, this is a project designed to contribute to the software engineering community, and not for commercial gain. The main design elements of DPTL are succinct, since this is a code library development type of project, hence there are not that many significant issues for its specific design. Each DPA should be developed in a very compact format and should strictly follow its design patterns definition and STL/ATL style.

IV. TEMPLATE-BASED PATTERNS AUTOMATION

In this section, detailed template-based automation (implementation) for two design patterns are presented. They are *singleton*, which is provided as example of the STL style, and *decorator*, which is shown as example of the ATL style. Details for the automation of several other patterns are available in [6].

A. Singleton Pattern Template

Singleton is considered the simplest design pattern and has been used frequently in software development. What it is and how to implement was described in Section II of this paper. Its implementation consists of about 20 lines of code. Then, the question may arise, is it really worth automating it? To answer this question, the following is a real life example. We use this example to go through the singleton pattern automation procedure using a template.

If one is developing a debugging and tracing tool, one will include in the tool a listener, which will catch all debug and trace information. Also, one will include in the tool a logger, which will record information into a log file, a windows event log, or simply an email message (all based on information severity). The *singleton* pattern would be the perfect design choice here, because there is no reason to have more than one listener and one logger in this application. Thus, if the designer decides to use the singleton pattern, the next question is, how to implement it? The first choice is to write two classes, *singletonListener* and *singleton Logger*, that use almost the same source code (Figure 3).

```
// Class Singleton Listener Definition

class SingletonListener {
   public:
      static SingletonListener * Instance();
   protected:
      SingletonListener ();
   private:
      static SingletonListener * _instance;
   // more Listener code
}

// Class Singleton Logger Definition

class SingletonLogger {
   public:
      static SingletonLogger * Instance();
   protected:
      SingletonLogger ();
   private:
      static SingletonLogger * _instance;
   // more Logger code
}
```

Fig. 3. Plain non-template implementation of singleton classes *Listener* and *Logger*

This type of code duplication often indicates that a template solution is possible. Next, what if the designer decides to have one of *fileLogger*, *emailLogger* and *windowseventLogger*? Three more singleton classes would need to be implemented. In other DPA tools like Together this will generate all the duplicated code that will look alike and will differ only through different names. However, a more efficient solution can be provided by templates used as in our approach.

Now let us see a template-based solution. First, a Singleton template class is presented in Figure 4.

```
Template <class T>
class Singleton{
public:
   Singleton(){getInstance();}
   T* getInstance(){
      static T instance;
      return &instance;
   }
   ~Singleton(){
   }
   T* operator ->(){
      return getInstance();
   }
};
```

Fig. 4. Template solution for the *Singleton* pattern

Then, a simple test using the above template is shown in Figure 5.

```
class foo { // A simple class which would mind
            // its own business
public:
   int a;
};

int main(){
   Singleton<foo> aa; // Create first
                      // singleton object
   foo->a=10;         // Assign value
   Singleton<foo> bb; // Now try to create
                      // another object,
                      // but it would be same
                      // as the first
   cout<< "aa->a: " <<aa->a<<endl; // 10
   cout<< "bb->a: " <<bb->a<<endl; // also 10
}
```

Fig. 5. *Singleton* template implementation test

As the above test code suggests, with the template-based solution (henceforth, *template solution*) it becomes much easier to reuse the Singleton pattern (the bold font in Figure 5 gives an indication on how easy it is to create a singleton object). However, this kind of implementation may be suitable only for *singleton*, because basically it has only one class to deal with. There is no complicated relationship between classes involved. Using a template involves just adding a new layer upon the underlying class to make sure it cannot have more than one instance. Since *singleton* is considered the simplest pattern, there are certain other ways to implement it using templates, such as Bruce Eckel's, who introduced an implementation approach similar to using the ATL style [20].

B. Decorator Pattern Template

The *decorator* pattern is one of the structural patterns. It can be used when the user wants to add new properties to individual object only, but not to the whole class. This means that, by applying different decorations, a set of different behavioral objects can be created from a single central main class. All decorations need to be encapsulated into a class derived from the *decorator* class, and this decorator class is derived from a decoration component abstract class. The reader is referred to [6] for the non-template solution, as it is rather large in size. Here, the template-based decorator pattern example is shown in Figure 6.

```
Template <class Component>
class decorator : public Component {
public:
   decorator(Component* cC) { _cC= cC; };
   virtual void execute() {_cC->execute();};
private:
   circleComponent* _cC;
};
```

Fig. 6. *Decorator* pattern template

The above *decorator* template was used to create a "Check" real-world application example, which is described next.

V. APPLICATION: "CHECK" PROGRAM DEVELOPED USING THE DECORATOR PATTERN TEMPLATE

The check program is a demonstration of applying a template-based pattern automation solution for program implementation. The program allows the user to create an electronic check that looks like a real check and print it using a color printer. The program presented here has been implemented using Microsoft Visual C++ 6.0 as an MFC windows application. All texts and graphs are drawn using GDI (Graphics Device Interface) [21], which allows for high resolution displaying and printing. As a matter of fact, if a high enough resolution printer and security-featured picture are available, this program can be used to print real business checks. The *decorator* pattern has been used in this application to add decoration features for a better looking check. Specifically, a decoration feature is for a color frame and another is for a background picture. Even though there are many ways to implement these decorations, the templated decorator pattern solution is given in this paper for demonstration purposes. As mentioned before, answering the questions of why to use this design pattern and what benefits can be gained from using it are not within the scope of this paper. This paper concentrates solely on how to implement

patterns using templates. The following are several snapshots of the check program (Figures 7, 8, and 9).

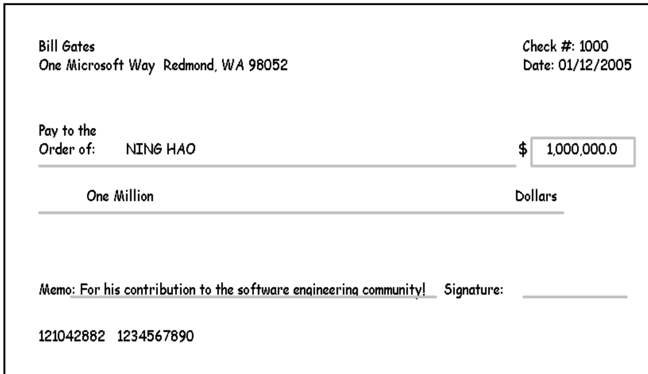

Fig. 7. Plain check with no decoration

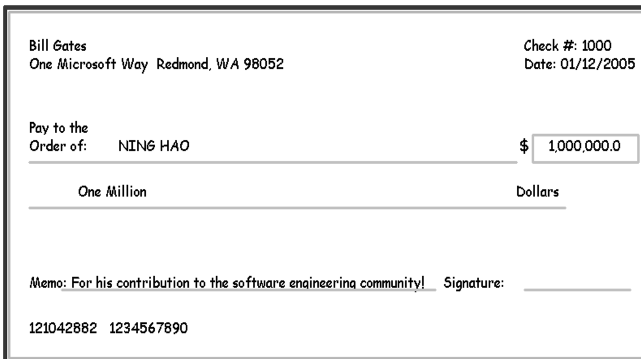

Fig. 8. Check with frame decoration

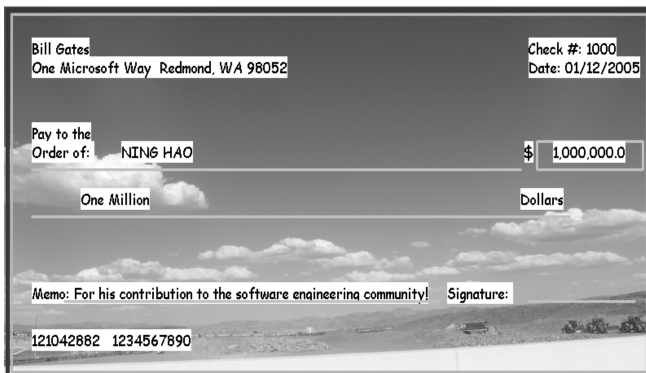

Fig. 9. Check with both frame and background decorations

Figure 10 presents the code that uses the decorator template. The decorator template is included in the *decorator.h* head file, and two corresponding decoration classes are created using the template ATL style. The detailed drawing code has been skipped due to space limitations, but can be found in [6], along with other relevant program files that make up the "Check" program.

```
#include <string>
#include "afxwin.h"
#include "MaskedBitmap.h"
#include "decorator.h" // This head file contains
                       // Decorator Template.

using namespace std;

class checkDecorationComponent {
public:
   checkDecorationComponent() {};
   ~checkDecorationComponent() {}
   virtual void draw(CDC* pDC) {};
   virtual void draw() {};
   virtual void execute() { draw();};
};

// Decorator class for background decoration

class bkgGraphDecoration :
public decorator<checkDecorationComponent>{
public:
   bkgGraphDecoration
   (checkDecorationComponent* cC):
   decorator<checkDecorationComponent>(cC){};
   virtual void draw(CDC* pDC) {
   // do drawing of background picture here.
   };
};

// Decorator class for frame decoration

class frameDecoration : public
decorator<checkDecorationComponent>{
public:
   frameDecoration (checkDecorationComponent*
   cC):
   decorator<checkDecorationComponent>(cC){};
   virtual void draw(CDC* pDC) {
      // Do drawing of frame here.
   };
};
```

Fig. 10. *CheckDecoration.h* file that implements decorations using the *Decorator* pattern template

VI. COMPARISON WITH RELATED WORK

Compared with existing wizard-based design patterns automation tools, DPTL allows the users to develop their own type classes and pass them to templates. The solution proposed in this paper is different from other existing tools, including wizard-based DPA tools, and provides the following:

- Greater flexibility. Since the users have unrestricted access to the source code, they can have the overall control of the library. Hence, they will be able to use design patterns in better and more flexible ways than when relying on wizard-based tools;
- Expandability. By using either the STL or the ATL style, new patterns can be templated and integrated into DPTL. This will support easier understanding by the users (developers), because learning to use one template can result in easier learning of all templates and patterns. Also

to the benefit of developers, new patterns can be developed and integrated into DPTL using the same programming convention and style;

- Pattern compatibility. Because most newly mined patterns are just composite GoF patterns or variations of them, if a common interface can be introduced to support GoF patterns communicate seamlessly, it will be much easier to have the DPTL itself grow straightforward and at a fast pace. In fact, a "codefarm" project named Cross-Platform DPTL has already been developed elsewhere for dealing with this issue [22].

The downside of the proposed approach is that, as compared with wizard-based tools, DPTL has a demanding learning curve, which means that the users must learn DPTL before using it. Also, the users must possess a solid understanding of design patterns.

VII. FUTURE WORK AND CONCLUSIONS

Given this is the initial version of the DPTL project, there is still work ahead to shape it into a very attractive and useful library. The following are several directions of future work considered:

- Develop the complete DPTL library, which means to cover all patterns from the Gamma *et al* book [1]. This involves a significant amount of work, because it requires thorough understanding of all the patterns presented in the book as well as extensive pattern implementation experience;
- Verify and guarantee the correctness of all DPTL implementations. Because design patterns are abstractions of special design issues, certain implementations will probably not be able to satisfy all possible programming scenarios. To solve this problem, comprehensive testing is needed. Furthermore, because from a software engineering point of view this library's development is not considered a very large project, it represents a good candidate for the application of the eXtreme Programming (XP) approach [22]. Also, it would be highly beneficial to the project to invest additional effort for having a CPPUNIT module [23] integrated into the library.
- Make DPTL available as an open source tool to the public. Thus, all its code has to be adjusted to follow open source code conventions.

In conclusion, this paper has described the foundations of the DPTL project, which in nutshell is a solution aimed at achieving design patterns automation. The paper's contributions can be summarized as follows.

First, differing from other wizard-based tools, DPTL offers novel source code implementations based on the modern technique of template-based generic programming. DPTL templates separate pattern implementations from real world business implementations. Compared with non-template implementations, this not only provides programmatic and effective automation of patterns, but also reduces the likelihood of programming errors in pattern-based software development.

Second, based on the complexity of pattern's class structure, the proposed pattern automation solution can be divided into two common template library styles: the STL style and the ATL style. Consequently, the developers and the users of the library can save significant time when learning and/or incorporating new patterns.

Third, seven patterns, namely *singleton, factory method, visitor, decorator, memento, strategy,* and *iterator*, have been investigated and presented in [6] as examples of the proposed template-based DPA solution. Among these, the *singleton* and the *iterator* patterns follow the STL style while all other patterns follow the ATL style. Details of *singleton* and *decorator* template-based automation have been presented in this paper.

REFERENCES

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

[2] Bulka, A., "Design Pattern Automation," *Proceedings of the 3rd Asia-Pacific Conference on Pattern Languages of Programs*, John Noble (editor), 2002.

[3] OMG's UML Resource Page, http://www.uml.org (accessed July 30, 2005).

[4] Josuttis, N., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999.

[5] Shepherd, G. and King, B., *Inside ATL*, Microsoft Press, 1999.

[6] Ning, H. *Design Patterns Automation with Template Library*, Master of Comp. Sc. Professional Paper, Department of Computer Science and Engineering, University of Nevada, Reno, USA, February 2005.

[7] Coplien, J., *Object World Briefing on Design Patterns*, AT&T Bell Labs Conf. Tutorial, San Francisco, 1994.

[8] Coplien Form, http://c2.com/ppr/wiki/WikiPagesAboutWhatArePatterns/CoplienForm.hhtml (accessed February 2, 2005).

[9] Log4j Open Source Resource Page (accessed July 30, 2005), http://logging.apache.org/log4j/docs/

[10] Rammer, I., *Advanced .NET Remoting in VB.NET*, Apress, 2002.

[11] Yacoub, S. and Ammar, H., *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems,* Addison-Wesley, 2003.

[12] Budinsky, F., Finnie, M., Vlissides, J., and Yu, S. "Cogent: Automatic Code Generation from Design Patterns," *IBM Technical Journal*, vol. 35, no. 2, 1996.

[13] Borland: Together Technologies, http://www.borland.com/together/ (accessed February 2, 2005)

[14] PragSoft Corp.'s UMLStudio Resource Page, http://www.pragsoft.com/products.html (accessed July 30, 2005)

[15] ModelMaker Tools, http://www.modelmakertools.com/ (accessed July 30, 2005)

[16] Vandevoorde, D. and Josuttis, N., *C++ Templates: The Complete Guide,* Addison-Wesley, 2002.

[17] Troelsen, A., *C# and the .NET Platform*, 2nd Edition, Apress, 2003.

[18] Gordon, A., *The COM and COM+ Programming Primer*, Prentice Hall PTR, 2000.

[19] Brown, J., Malveau, R., McCormick H. and Mowbray T., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* John Wiley & Sons, 1998.

[20] Eckel, B., *Thinking in C++, Volume 2,* 2nd Edition. Practical Programming, Prentice Hall, 2003.

[21] Prosise, J., *Programming Windows With MFC*, Microsoft Press, 1999.

[22] Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.

[23] Cross Platform Design Pattern Template Library, http://www.codeproject.com/library/dptl.asp (accessed Feb. 2, 2005).