JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 269-282 (2006)

Wrapper-Based Framework for Domain-Specific Software Reuse*

HISHAM M. HADDAD AND YING XIE
Department of Computer Science and Information Systems
Kennesaw State University
Kennesaw, GA 30144, U.S.A.
E-mail: {hhaddad; yxie2}@kennesaw.edu

Component-based development is a defacto requirement for many competitive IT industries. The implementation of component reuse ranges from the more isolated solution such as creating a new tool or language to the use of current industry standards such as CORBA COM, EJB, and .NET. However, the complexity of managing and maintaining reusable artifacts increases with large systems In the case of standard reuse libraries, more overheard is associated with the management, storage, and retrieval of artifacts. To overcome this problem, this work presents a *Wrapper-Based Framework* that promotes and facilities the design and development of domain-specific reusable software components. The framework is based on the concept of *Atomic Domain* methodology that signifies a reusable subsystem, which intelligently manages a collection of highly reusable components in the application domain of interest. This methodology largely reduces cost of component maintenance and the overhead associated with standard reuse libraries. This paper highlights the concept of atomic domains and proposes a wrapper-based framework for domain-specific software reuse. Illustrative atomic domains and a proof of concept functional prototype are also presented to demonstrate the Atomic Domain methodology and the feasibility of this framework.

Keywords: software reuse methodology, component reuse, atomic domain, wrapper-based framework, reusable domains

1. INTRODUCTION

The implementation of Component-Based Software Engineering (CBSE) has in effect become a requirement in today's IT industry. The literature indicates that without an investment in CBSE, there are possibilities for longer lead times, higher costs, lower quality and possibly dropped projects, or a continued higher level in resource efforts including maintenance. The implementation of CBSE ranges from the use of more isolated solutions such as new languages [1], wrapping legacy components [2], refactoring [3], program mining [4], implementing product populations [5], product line technologies [6], and various domain analyses [7, 8] to the use of industry standards such as OMG's CORBA Component Model, Microsoft's Component Object Model, COM+, .NET, and Sun's Enterprise Java Beans [9]. Many of these industry standards are defined by a dis-

---

Received July 1, 2005; accepted November 24, 2005.
Communicated by Sung Shin.

* Preliminary partial foundation work of this work was published in the ACM Symposium on Applied Computing, held in Madrid, Spain, 2002. The conference is sponsored by the ACM Special Interest Group on Applied Computing (SIGAPP).

tributed or runtime component model [10] while some lesser-known solutions follow a more ‘local’ or embedded component model [11] such as PECT, Koala, PBO, PECOS and the Rubus Component Model [12]. Each model has its own merits, conventions, component definitions, and scope of use (application domain).

However, the complexity of managing and maintaining reusable artifacts increases with large systems. In the case of standard reuse libraries, more overheard is associated with the management, storage, and retrieval of artifacts. Leading from previous work [13] on Atomic Domains, described in section 2 below, this work focuses on implementing CBSE as a local model based on two promises. The first promise defines the component as the source code of a module, class, or function. The use of such components, which is not necessarily seen in the end product, may lead to lower development costs or shorter lead times. This is code reuse in the strict sense of the term and implies that it targets specific application domains. The second promise defines the components’ domain as atomic. Atomic means that each domain is comprised of related components that are irreducible. These two promises are significant in the approach to CBSE and together comprise a general model from which has evolved the concept of the *Wrapper-Based Framework.* Each atomic domain has a wrapper as its intelligent interface, through which the selection of the proper component for reuse among all the related ones contained in this atomic domain is automatically conducted based on the requirements of the corresponding application. Unlike the standard model for domain construction, which focuses on an application or a system [14], the construction of atomic domain incorporates related components from a heterogeneous collection of applications.

Fig. 1 shows a block diagram of the general model (section 3) in terms of multiple atomic domains and their wrappers relative to an Integrated Development Environment (IDE) and compiler. Wrappers are distinct units that communicate with the IDE and compiler through their APIs (section 4). The wrapper API is an open design allowing each wrapper manager to have access to the IDE and compiler and their resources such as help, syntax checking, parsing, etc. In effect, each Wrapper Manager acts as a particular extension of the source code’s compiler, applying rules from an application domain as to the use of a particular atomic domain.

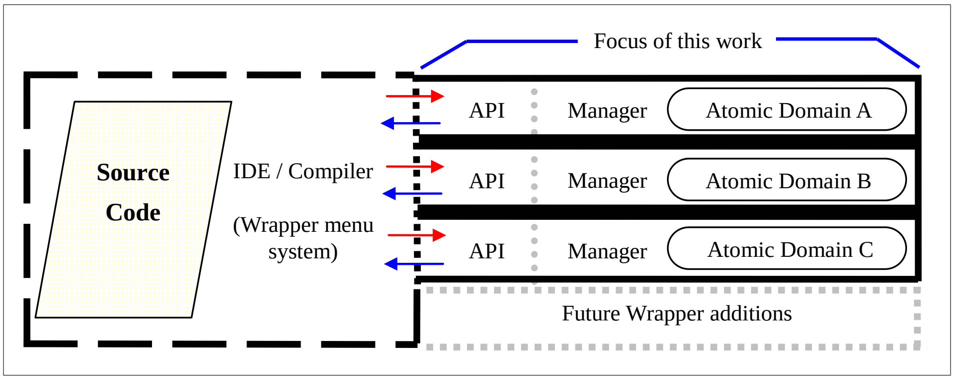

Fig. 1. The general model framework: an IDE, compiler, and three atomic domains.

2. ATOMIC DOMAIN METHODOLOGY

The current reuse practice is mainly at the component-level. Reusable software components can be found in numerous repositories. In general, the requirements for reusability revolve around concepts such as generic packaging of abstract data types of basic utilities and mathematical functions. A number of packages describing the stack, for example, are found in the literature. These packages provide the generic parameters necessary to reflect a broad range of applications of the stack. In contrast, RESOLVE components [15] are derived from complete abstract specification, which begins to address the issue of reasoning about component behavior. One of the major problems with component-based reuse is that past component-level designs may not match new system requirements. In addition, the overhead associated with the management, storage, and retrieval of components in the reuse library is quite huge. Domain-level reuse may provide solutions to these problems. However, domains differ significantly from components. The increased complexity of most domains makes it virtually impossible to formally specify the behavior of an entire domain. For example, the complexity of a real-world domain [16] would present the designer with many dimensions as to make a generic description forbidding. This problem can be illustrated by considering the vast number of application areas of robotics, communications, data mining, e-commerce, and others. In order to overcome challenges brought by both component-based and domain-level reuse, we propose a novel atomic domain methodology to promote multiple-component reuse at the domain level.

2.1 Atomic Domain

Domain analysis and design activities are intended to facilitate software component reuse. In such context, each reusable component is considered independent and self-contained item with limited relationships to other related components that share some common behavior. The Atomic-Domain methodology is centered around the idea of grouping related reusable software components into atomic domains. It is introduced to promote software reuse from its traditional component-level to the domain-level for effective reuse. At the domain-level, related artifacts (including source code, compiled objects, documentation, procedures, and policies) are grouped and treated as a single unit (atomic domain). The atomic domain is embedded within a wrapper that acts as intelligent interface between atomic domain and its clients. The wrapper handles communication and management aspects between the internal artifacts and the outside world.

Defined in previous work [13], the concept of an atomic domain is derived from the concept of *Aggregates* [17]. An aggregate is a collection of objects referenced by a single aggregate name, and addressing mechanism facilitates the cooperation between objects within the aggregate and its users.

Definition An atomic domain is an aggregate of reusable artifacts that possess properties associated with a group of related domains. Properties of the artifacts cannot be substantially decomposed without altering the primary properties of the domains. The atomic domain exhibits the following properties:

1. High-level of cohesion among components. This property distinguishes the atomic domain boundary from the standard domain boundary.
2. A common wrapper to support multiple-component reuse and management.

The atomic domain is an irreducible representation of a common subdomain of related domains, and it is amenable to formal specification. The composition of several atomic domains using glue components help realize the advantages of the atomic domain concept. One common obstacle to developing effective reusable components is the difficulty of interfacing reusable components in new applications. Current technologies provide support for defining interfaces to reusable objects. While necessary, an interface specification alone is insufficient to compose complex systems. The specification is essentially a syntactic aid. Tools that support the interpretation of behavior of the subdomain are needed along with interface specification. The atomic domain concept is tended to provide such semantic support.

2.2 The Wrapper

As an extension of the interface concept, the main role of the wrapper is to support reuse of components (or artifacts in the general sense) encapsulated in the atomic domain. Among others, the wrapper performs control and management functions to exhibit information about the atomic domain content and to provide the appropriate services to the atomic domain clients. It acts as an intelligent interface between the user applications and the atomic domain.

Definition The wrapper is a collection of mechanisms that exhibit the atomic domain content to the outside world, control the interactions between the atomic domain and its clients, and manage the use of artifacts in the atomic domain.

The wrapper consists of five components described as follows:

1. **Taxonomy:** The wrapper uses the common taxonomy of experts in the field to "describe" the atomic domain content. The taxonomy may include search keywords, terminology and definitions, patterns, algorithms, and textual descriptions. In some cases the taxonomy is expressed in a formal language; while in other cases the language is not formal but a universally accepted interpretation among the domain experts.
2. **Application Programmer Interface (API):** The API descriptor includes formal parameters and expected behavior, known limits, and constraints (pre- and post-conditions and data formats).
3. **Manager:** The wrapper manager determines the appropriate applicable components in the atomic domain, and assigns (conceptually speaking) the component to the user application. The manager relies on information presented through the API descriptor.
4. **Communication Mechanism:** The communication mechanism is used to present the domain to the outside world. For example, the atomic domain source code artifacts may be used through message passing or function calls.
5. **Control Mechanism:** The control mechanism is responsible for program flow within

the domain, especially with complex atomic domains where input may be asynchronous due to dependencies among domain components. The cohesiveness of the domain content is essential characteristic of the domain.

The wrapper components support domain analysis and design by improving the cohesiveness of the atomic domain. When a new component is designed for an existing application, the wrapper helps determine whether the new component belongs to the atomic domain. The wrapper description of the atomic domain boundary differs from the traditional description of domain boundary as follows: It improves cohesion within the atomic domain; the atomic domain artifacts communicate with the outside world through a coherent set of message structures (or data structures); and the wrapper taxonomy can be employed to support cataloguing and subsequent search, which is essential for effective component reuse.

2.3 Atomic Domain Creation Rules

During domain analysis, domain experts identify reusable artifacts that belong to the problem domain, such as specification; source code; object code; test cases; results of requirements, analysis, and design; policies, plans, guidelines, and metrics; and applications or parts of applications. A key descriptive characteristic of the atomic domain is the irreducibility of its components. The domain expert examines existing atomic domains to check if any identified reusable artifact can be incorporated into any of these domains. If not, new atomic domains will be created to accommodate those identified reusable artifacts.

The descriptive approach outlined below is not exclusive of standard methods of domain analysis, so it only address items unique to building an atomic domain. The general steps for defining new atomic domains are:

1. The domain expert group reusable artifacts into new categories and extract common characteristics for each of them, such that each category has taxonomy, a set of criteria to verify whether an artifact belongs to this category, naming convention, define sets, hierarchies, and other organization techniques. This step helps the domain expert identify groups (atomic domains) of artifacts of potential reuse across related domains.
2. Develop an abstract behavior model for each atomic domain. This behavior is expressible in the taxonomy built by domain experts. In particular, the content of the domain(s) that share the services of the atomic domain are used to construct (generic) parameters passed to/from the atomic domain.
3. Identify constraints within the each atomic domain, such as constant relationships, preconditions, and post-conditions. To enhance reusability, it is best to plan the use of the atomic domain across application environments.
4. Analyze the “dimensionality” of each atomic domain. Here, the domain expert defines the environment in which the atomic domain will exist. For example, a sort operation may take place in an environment where performance is critical, or, conversely, one in which performance is not critical but memory usage is.
5. Define the boundary of each atomic domain. This step would require three actions:

– Define the scope of the services of the atomic domain. By performing this step the designer has a criterion by which components can be included or excluded from the atomic domain.
– Define the interface of the atomic domain. The interface identifies how interaction with the domain components is achieved.
– Define a wrapper for the atomic domain.

6. Embed the atomic domain in a wrapper.

Although domain analysis and design are intended to support software reuse, each reusable component is considered independent and self-contained object with limited relation to other components. The atomic domain methodology focuses on grouping functionally related components into an atomic domain (not tightly coupled, which may lead to maintenance issues).

Knowledge of reusable artifacts of an existing domain of interest facilitates the analysis and design of new domains. Reusable atomic domains expedite the development of new applications through integration of systems that belong to different application domains. Working with reusable atomic domains may reduce the management effort due to artifact grouping, because part of the management functions is assigned (delegated) to the wrapper. The wrapper encapsulates a group of components that perform similar functionality, and provides the user with a higher level of component abstraction. Therefore, it is the wrapper rather than the developer who determines the appropriate component to be reused for a particular application.

2.4 Atomic Domains and Source Code Components

The atomic domain methodology promotes reuse, at the domain level, of various domain-specific artifacts including specification, analysis, requirements, design, source code, object code, test cases, policies, plans, guidelines, and other artifacts. The wrapper-based framework derived from atomic domains focuses on source code components. Below we highlight few issues specific to atomic domains of source code components.

The wrapper concept has been widely used in component-based development and other areas, such as embedded system development. Wrappers defined for such specific domains are tend to be more focused on working with components within that domain, making their usability limited to a single domain. In comparison to existing wrapper concepts, the wrapper defined in section 2.2 is more general. The proposed wrapper consists of five components, each of which provides a distinct functionality needed to facilitate source code reuse across related domains. With this wrapping approach, a source code component in the atomic domain can be reused in many applications. For example, an atomic domain of components for robot arm control software can be reused in various robot applications such as rescue missions, farming applications, fire fighting, and removal of hazards materials.

Previous research suggests that perhaps more time is spent on building reusable components than the benefits received from reusable components because the actual reuse is very low. Unlike the traditional approach, the atomic domain methodology promotes reuse of already existing source code components, rather than calling for building new components for reuse. Therefore, the presented framework is not an exercise in creating components that may never be reused. Rather, it is an exercise in reusing what has already been created, tested, and proven to be usable for applications within our domain of interest, an effort to make components more readily reused. From the initial illustrative atomic domains [13] and the prototype model (described in section 4), it can be seen that the benefits brought by atomic domain reuse largely outweigh the initial cost required to implement atomic domains. In addition, the ease of implementation and usage were also evident by these examples.

3. THE GENERAL MODEL DESIGN

A wrapper-based framework that facilitates the development of reusable components is implied in the general model shown in Fig. 1. The general model functions include a modified IDE and Compiler, and template(s) for atomic domain and wrapper. These two parts make up the framework and facilitate reuse of source code components as defined by the atomic domain and wrapper concepts. Fig. 1 shows the general model with three atomic domains and their wrappers as well as how those wrappers communicate with a modified IDE and compiler.

The selection of the IDE and compiler is the developer's choice so long as the necessary functions of the IDE and compiler are exposed and the compiler control can be passed externally. Although just a command line compiler could be used, an IDE and compiler combination is preferred since it is one of the current developer's tools of trade and the interaction between the developer and environment is beneficial. Aside from the choice, the modification and template(s) must address the following *design criteria,* which lists the basic functions of the general model.

Each atomic domain is based on one related context. All reusable and irreducible components related to the context are placed in the same atomic domain. Those components can be (and should be) virtually linked via the wrapper manager rather than physically contained within the same file or package. This way the components could be reused by multiple domains and therefore helps attenuate the maintenance issues.

1. Each atomic domain is associated with a single command that is referenced by the IDE and compiler. References can also be made by the wrapper such as expected syntax, help, etc.
2. The single command associated with a wrapper refers to all components within the atomic domain. The command does not act as a single library call or an overloaded function.
3. The atomic domain components are hidden from the developer and can only be referenced by the wrapper manager.
4. When the single command is used in the developer's source code at compile time, the compiler supplies all required data to the appropriate manager through the wrapper's API for assessment.
5. For a single command assessment, the wrapper's API requires: source code; current command line text; command line number; and count of command use.

6. The wrapper manager assesses the single command at the point it is used in the source code and determines which component(s) to return based on application domain rules.
7. Application domain rules are built into the wrapper manager during its construction process. The rules determine which component(s) to select as well as whether any additional code is needed.
8. If the single command assessment fails, the wrapper manager either requests additional data, or posts an error through the IDE and stopping the compiler.
9. If the command assessment passes, the wrapper manager returns the selected component(s) and other needed codes to the compiler.
10. The wrapper's API returns: returned component(s), other needed codes, and status of assessment.
11. Per single command line in the developer's source code, the compiler replaces the single command with the returned component(s) and other needed codes.
12. The developer does not make declarations or function calls to the wrapper's API. Instead, the IDE and compiler make these calls as the developer executes IDE or wrapper commands.
13. All atomic domain components are either in the compiled state, tokenized for multiple platforms, or in the form of source code. This greatly affects the design of the wrapper manager.
14. The wrapper and atomic domain exist as entities or objects within the IDE and compiler environment rather than any other form in the final compiled project.

Given an atomic domain and wrapper template, the construction process of each wrapper manager around its atomic domain is the most critical step. This step creates the application domain rules that determine the selection of the component(s) as well as other needed codes. These rules can exist as either sensible defaults or hints depending on factors like the surrounding codes placed by the developer, data type(s) used, previous command usage in the same source code, and the place where the code is put. Therefore, the wrapper manager's design determines the effectiveness of atomic domain reuse. This is clearly a minimal programming effort before reuse can begin. An understanding of how the atomic domain components were used within the application domain of interest is a key to constructing a wrapper manager that behaves as expected.

4. PROOF OF CONCEPT MODEL

A proof of concept prototype model, using Microsoft Access version 2002 automated by Visual Basic for Applications (VBA), has been developed to illustrate the framework implementation in the database application domains. This is a form of Visual Basic, which has reference to numerous database objects and, more importantly, allows access to the source code (Modules) and IDE object models. This means that the source code and IDE can be manipulated programmatically. The working model utilizes two atomic domains and their wrappers. One atomic domain contains components related to sort functions [18, 19], and the other contains related components that test attached table(s) and assist the end user in their redirections (reattachment to new location). All of the components within the two atomic domains are prime candidates for reuse within the application domain. The steps of using the framework for developing the prototype model are illustrated below.

1. **Source Code Analysis:** For the working model, the analysis is conducted by an application domain analyst, who analyzes past source code for reuse potential and returns the results to the application domain for consolidation and assignment of its application domain rules. In terms of the atomic domain and wrapper concepts, reuse potential is not simply concerned with code that repeats but with code that both relates and repeats. In a real world setting, atomic domains come from the same organization, which creates them for reuse. For the working model, the components in the two atomic domains were a given. It should be noted that this approach to reusable components has the benefit of avoiding difficulties related to COTS, distributed components and licensing, as well as the problems related to unique interfacing.
2. **Domain Formation:** The Sort atomic domain, one of the two atomic domains within the working model, is shown as a pane in the VBIDE in Fig. 2. The functions related to sorting are consolidated and installed or coded as a single stand-alone module within the VBA project. All functions related to sorting can be seen listed from the declarations pull-down menu. Theses components are assumed to be supplied by the domain analyst from the previous step. The second atomic domain within the working model has components related to attached tables. The name of each atomic domain module has a prefix of 'atd_wrp_' and a postfix of '_AtomicDomain', which is a naming convention used by the VBA project. For both atomic domains, the components are in the form of source code without being compiled or tokenized.

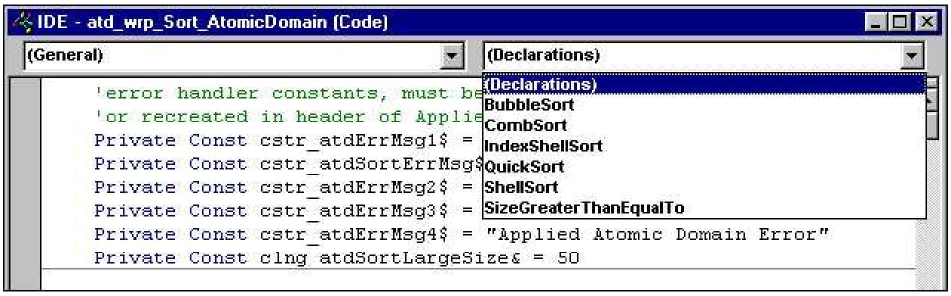

Fig. 2. The Sort atomic domain of the working model.

3. **Wrapper Definition:** To facilitate the construction of the two wrappers and atomic domains for the working model, the framework of the general model was introduced to the VBIDE by the coding of two modules. One contains a higher-level set of short functions, and another contains a lower level set of larger functions. The majority of Sort domain wrapper functions can be seen listed from the declarations pull-down menu. A subset of these functions comprises the wrapper manager that applies the application domain rules to the atomic domain and returns the results. The wrapper is

another VBA module coded and installed within the VBA project and whose naming convention calls for a prefix of 'atd_wrp_' and a postfix of '_Wrapper'. The wrappers were constructed after the VBIDE was modified as the wrapper API standards had to be established.

The two wrapper single commands are 'atdSort' and 'atdCheckLinkedTables' as a result of the two installed wrappers. Each single command is usable within the developer's source code to call the respective wrapper manager to apply the application domain rules to the respective atomic domain and effectively replace the single command with the selected component(s) and any additional code.

4. **Domain Integration:** This step installs all required wrapper and atomic domain modules into the VBA project that contains the modules for VBIDE and wrapper manipulation. In general, integration requires that the framework of the general model be introduced to the VBIDE by the coding of two modules as mentioned in the previous step. The coding required research into the programmability of such object models [9-11]. The coding essentially augments and exploits some of the VBIDE capabilities as prescribed by the general model's design and rationale.

Fig. 3 shows one of the two modules that modify the VBIDE of the working model. The module named 'basResources' contains numerous VBIDE wrapper functions including the function 'AtomicDomainApply'. Other functions can also be seen listed from the declarations pull-down menu. The other module containing the higher-level set of functions is named 'basMain'. All function names Conform to VBA project convention. Note that the names 'basMain' and 'basResources' are not by convention, unlike most other modules that must be referenced programmatically, and all functions from both could have been placed in a single module. Function 'AtomicDomainApply' is the primary function that communicates with wrapper manager, while function 'AddCommandBarControlToVBE' sets up the wrapper menu system.

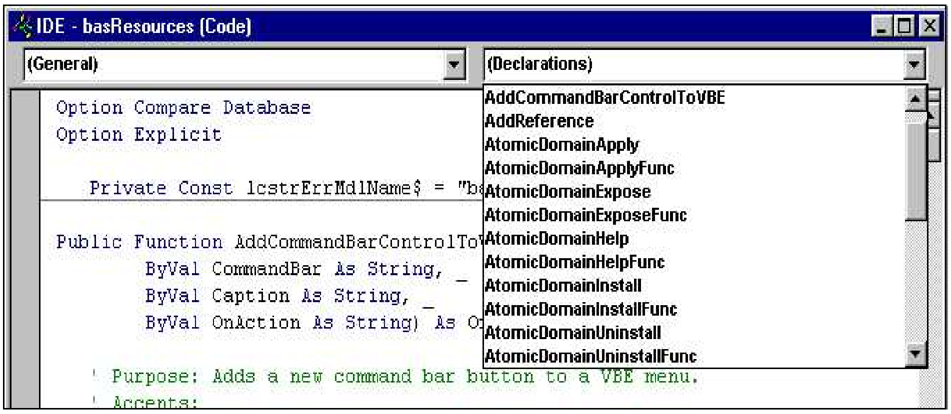

Fig. 3. Modified IDE (VBIDE) of the working model, the VBIDE lower-level function module.

Fig. 4 shows the wrapper menu system as well as the two wrappers and atomic domains installed in the VBA project in the left pane. All VBIDE wrapper functions for the developer are apparent from the figure. The main focus here is the 'Apply' menu item. It applies all atomic domains to the source code module named 'basSourceCode', which is written by the developer. The wrapper manager returns the results in a new, compilable, and stand-alone module called 'basSourceCode-atdApplied'. This compiled module is the end result of applying application domain rules to the atomic domains per single command in the source code.

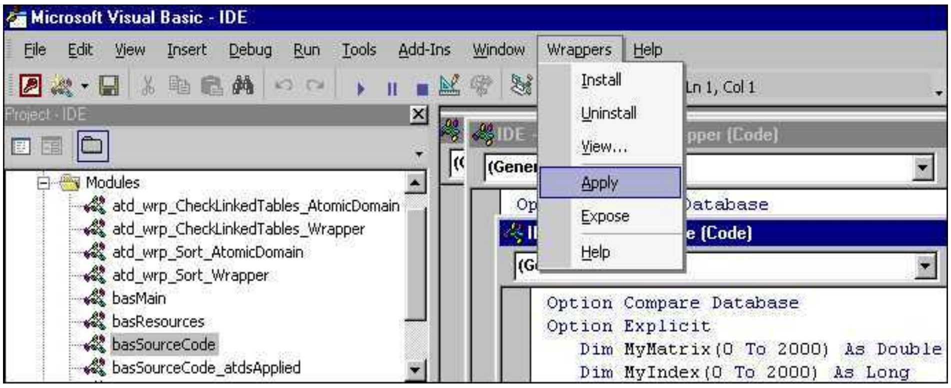

Fig. 4. Modified IDE (VBIDE) of the working model, the VBIDE with wrapper menu system and two wrappers and atomic domains installed as seen in first four entries under 'Modules' in left pane

5. **Domain Utilization:** As Fig. 4 depicts, the wrapper menu system is designed to allow the developer to access the necessary and beneficial wrapper functions available through the modified IDE (VBIDE). The first two menu items, 'Install' and 'Uninstall', respectively install and uninstall the atomic domains and their wrappers, which add or remove the associated modules. The atomic domains and their wrappers are checked for the existence of essential constants and functions as they are installed. The next menu item, 'View', allows the developer to view some of the information related to an installed atomic domain such as the component (function) name(s), related constants, etc. which may aid the developer in parameter selection.

The next menu item, 'Apply', applies all atomic domains to the source code per single command and is further discussed in prior subsections. After the atomic domains have been applied to the source code, the developer can select the next menu item, 'Expose', which shows the function(s) and any additional code that were selected per single command. This lets the developer see what was selected based on the application domain rules per single command line. The last menu item, 'Help', allows the developer to view help on usage of single commands with examples.

The application of atomic domains and their wrappers is apparent from the analysis of three source code examples that involve sorting 20,000 random numbers. Table 1 shows an obvious advantage of using atomic domains. It contrasts the LOC and NOF

Table 1. Contrast of LOC and NOF.

Source Code Sample #	Standard Code (before)	Code Using Atomic Domains (after)	Single Command Call Counts
	Lines of Code (LOC)		
1	83	33	1
2	161	31	1
3	292	61	7
	Number of Functions (NOF)		
1	4	2	
2	6	2	
3	13	3	

required for these examples before and after using atomic domains. From a developer's point of view, single commands make it simple in that the actual function(s) applied do not have to be known by the developer. That is how an atomic domain wrapper facilitates component reuse.

The first program (Sample #1) has a single static call to 'atdSort' atomic domain. Static implies always using the same function. In the second program (Sample #2), the program has a single dynamic call to 'atdSort' where the selection between two functions is based on the size of an array. Although the size is 20,000, it could be dynamic and alter the best function to use. The third program (Sample #3) has seven static calls to 'atdSort' to illustrate the time differences in sorting between different available sort components (functions). It should be noted that the development of these example programs required very few coding effort after atomic domains are defines and utilized.

5. CONCLUSIONS

The Wrapper-Based Framework for developing atomic domain is motivated by the need for new architectures and models to facilitate the design and development of component reuse in specific domains of applications. Reuse, integration, and domain management are key investigation issues in this work. The atomic domain methodology focuses on adding existing reusable components to the atomic domain (rather than creating them from scratch). Instead of physically organizing all source code components within a same file or package, the wrapper manager virtually link those components together and select the best candidate based on the specification provided by client applications. This type of virtual atomic domain method largely reduces the cost caused by the component maintenance. Another factor that can help attenuate the maintenance concern is coupling among components. When creating an atomic domain, the domain analyst aims to achieving low coupling among selected components for the domain.

The application of the general model is illustrated verbally or graphically as a mapping of the described framework to each selected case (found in the literature). These sample applications demonstrate that the proposed wrapper-based framework not only facilitates reuse practice but also encourages a higher reuse in the corresponding domain.

Future work focus on making the general model a more intimate part of the IDE and compiler by exploiting the compiler's level of control and using devices such as the preprocessor, syntax checker, and error handler among others. This would also allow wrappers to have automatic precedence in the compiling process as well as to be able to handle ideal atomic domains, which would be in a compiled or tokenized state and truly hidden from the developer.

REFERENCES

1. V. C. Sreedhar, "Mixin'Up components," in *Proceedings of 24th International Conference on Software Engineering*, 2002, pp. 198-207.
2. J. Guo and Y. Liao, "Integrating software components through wrapper technologies," in *proceedings of 7th IASTED International Conference on Software Engineering and Applications*, 2002, pp. 441-446.
3. R. E. Caballero and S. A. Demurjian, "Towards the formalization of a reusability framework for refactoring," in *Proceedings of 7th International Conference on Software Reuse*, LNCS 2319, 2002, pp. 293-308.
4. D. Xia, Y. Zhang, and L. Zhou, "A multi-agent system for program mining," in *Proceedings of 7th IASTED International Conference on Software Engineering and Applications*, 2003, pp. 95-99.
5. R. van Ommering, "Building product populations with software components," in *Proceedings of 24th International Conference on Software Engineering*, 2002, pp. 255-265.
6. C. Atkinson and D. Muthig, "Enhancing component reusability through product line technology," in *Proceedings of the International Conference on Software Reuse: Methods, Techniques, and Tools*, LNCS 2319, 2003, PP. 93-108.
7. S. Jarzabeck, "Modeling multiple domains in software reuse," in *Proceedings of the Symposium on Software Reusability*, 1997, pp. 65-74.
8. R. de A. Falbo, G. Guizzardi, and K. C. Duarte, "An ontological approach to domain engineering," in *Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering*, 2002, pp. 351-358.
9. R. L. Owens, "Development and implementation of a domain-specific reuse plan," in *Proceedings of 10th Annual Washington Ada Symposium on Ada: Ada's Role in Software Engineering*, 1993, pp. 31-42.
10. W. Emmerich, "Distributed component technologies and their software engineering applications," in *Proceedings of 24th International Conference on Software Engineering*, 2002, pp. 537-546.
11. A. J. Wang, "Device drivers as reusable components," in *Proceedings of 7th IASTED International Conference on Software Engineering and Applications*, 2003, pp. 396-401.
12. A. Möller, M. Akerholm, J. Fredriksson, and M. Nolin, "An industrial evaluation of component technologies for embedded-systems," MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE, Mälardalen University, Västerås, Sweden, 2004.
13. H. Haddad and H. Tesser, "Reusable subsystems: domain-based approach," in *Pro-*

ceedings of the Annual ACM Symposium on Applied Computing, 2002, pp. 971-975.
14. K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
15. W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben, "The RESOLVE framework and discipline: a research synopsis," *ACM SIGSOFT Software Engineering Notes*, Vol. 19, 1994, pp. 23-28.
16. M. Frankel and B. Winant, "Domain partitioning and reuse taxonomy," *Object Magazine*, Vol. 7, 1997, pp. 38-45.
17. A. A. Chien and W. J. Dally, "Concurrent aggregates (CA)," in *Proceedings of 2nd Symposium on Principle and Practice of Parallel Programming, ACM Press*, 1990, pp. 187-196.
18. F. M. Carrano and J. J. Prichard, *Data Abstraction and Problems Solving with C++*, 3rd ed., Addison Wesley, 2002.
19. G. J. Bronson, *Program Development and Design Using C++*, 2nd ed., Cole Publishing Company, Pacific Grove, CA, 2000.

Hisham M. Haddad received his Ph.D. in Computer Science from Oklahoma State University, Stillwater, Oklahoma, U.S.A., in 1992, and his M.S. degree in Computer Science from Northrop University, Los Angeles, California, U.S.A., in 1988. Since August 2001, he is Professor of Computer Science in the Department of Computer Science and Information System at Kennesaw State University, Kennesaw, Georgia, U.S.A. Before joining Kennesaw State University, he served as Associate Professor at Marshall University, in Huntington, West Virginia, for 8 years. His research interests include software engineering, software reuse, and computer science education.

Ying Xie (謝穎) received his Ph.D. and M.S. degrees in Computer Science from University of Louisiana, Lafayette, U.S.A., in 2004 and 2001 respectively, and his B.S. in Computer Science from Chongqing University, Chongqing, China, in 1995. Since June 2005, he is an Assistant Professor of Computer Science in the Department of Computer Science and Information System at Kennesaw State University, Georgia, U.S.A. Before joining Kennesaw State University, he served as a full time researcher in the Center of Advanced Computer Studies at University of Louisiana, Lafayette, since January, 2005. His research interests include data mining, bioinformatics, information retrieval, and software engineering.