Achieving Seamless Component Composition Through Scenario-Based Deep Adaptation And Generation

Dr. Xiaodong Liu*, Beihu Wang and Prof. Jon Kerridge
Napier University, Edinburgh EH10 5DT, UK
{x.liu, t.wang, j.kerridge}@napier.ac.uk

Abstract

Mismatches between pre-qualified existing components and the particular reuse context in applications have been a major factor hindering component reusability and successful composition. Although component adaptation has acted as a key solution of eliminating these mismatches, deep adaptation is often either impossible or incurring heavy overheads in the components. This paper proposes an approach, namely Scenario-based dynamic component Adaptation and GenerAtion (SAGA), to achieve deep adaptation with little code overhead through XML–based component specification, interrelated adaptation scenarios and corresponding component adaptation and generation.

Keywords—Component Reuse, Scenario-Based Adaptation, Component Generation, Component Composition, Component Definition Language, XML.

1. Introduction

Component–Based Development (CBD) has shown very successful improvement on efficient, high quality and low cost software design and development. One obstacle of reuse and smooth composition is the high variability of reuse context, such as constant changes in data model, architectural mismatch, and application specialty.

The rationale of CBD is that a component is an independent reuse unit to construct an application. The current state of the art is that not many components can actually be reused without adaptation. The problem is partially caused by incomplete component specifications, and the mismatches between components and the reuse context, including the application architecture, state and other collaborating components. To tackle this problem, complete component specification and deep component adaptation technology are required [6][12][15][18].

In this paper, to achieve smooth and seamless component composition in system development, a Scenario based dynamic component Adaptation and GenerAtion technology (SAGA) is developed within a generative and component-based reuse framework as the core technique for deep adaptation with little overheads in the target component code. It is often the case that provisionally qualified components still have some architectural and behavioral inconsistency with the requirements of a specific application system. To eliminate these inconsistencies, adequate adaptation technology must be applied to the components. For adaptation in depth, a new derivative of the component may need to be created with generation technology. In this paper, we propose to use a set of scenarios defined in XML to record the design configuration of components reused in specific applications. Scenarios may be adjusted, composed or associated interactively to cope with complex reuse cases.

An XML formatted Component Definition Language (CDL) is designed to represent component specification, which facilitates component understanding, use and implementation.

Database-oriented client server applications have been selected as the sample application domain of this project due to their popularity in reuse.

The remainder of this paper is organized as follows: section 2 summarizes the related work; section 3 describes the proposed approach, including the framework, scenario-based adaptation and generation; section 4 demonstrates the approach and prototype tool with an example of the adaptation and generation of an *invoice_manager* component; section 5 presents the conclusion.

2. Related works

2.1. Genesis and Avoca

Genesis and Avoca are successful examples of software component technologies and domain modeling. Genesis is the first building blocks technology for a database management system [3][4]. Using a graphical layout editor, a customized DBMS can be specified by composing prefabricated software components. Avoca is a system for constructing efficient and modular network software suites with a combination of pre-existing and newly created communication protocols.

Although Genesis and Avoca are steps in a component based approach to improve reusability, they are highly dependent on domain model, and the modification of components is restricted on options, preset by component developers. Component developers are supposed to have tremendous knowledge of a certain domain, and need to consider every possible modification to the component, and give a solution to each modification. These obstacles make component development extremely difficult, and restrict reuse.

2.2. Superimposition

A number of component adaptation technologies [1][2][6][12][13] have been proposed. Superimposition [2] is a successful adaptation technology based on blackbox adaptation. It allows the software engineer to adapt a component with a number of predefined adaptation behaviors that can be configured for a specific component.

The notion of superimposition has been implemented in the layered object model (**LayOM**), an extensible component object language model. The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customizability of adaptation behavior.

Superimposition uses nested component adaptation types to compose multiple adaptation behaviors for a single component. However, due to lack of component information, modification is limited at simple level, such as conversion of parameters, refinement of operations, modification of access mechanism, and extension of functionality. Moreover, with more layers of code imposed on original code, the overhead of the adapted component increases heavily, which will degrade system efficiency.

2.3. Customizable component

As a part of COMPOSE project [6], an environment for building customizable software components is developed. It is an approach to expressing customization properties of components. The declarations enable the developer to focus on what to customize in a component, as opposed to how to customize it. Customization transformations are automatically determined by compiling both the declarations and the component code; this process produces a customizable component. Such a component is then ready to be custom-fitted to any application.

In this work, the customized components generated for various usage contexts have exhibited performance comparable to, or better than manually customized code. However, component adaptation is only limited to pre-defined optional customization, deeper adaptation is not support in this work.

3. The approach framework

The general process of our approach is given in figure 1. Our research concentrates on the reuse of components and presumes that component mining part has already been done, i.e., reusable components, including their Component Specification (CS) in CDL, default adaptation scenarios and primitive component code have already been developed with appropriate component mining approaches and stored in the component repository.

The start point of component reuse is requirement analysis. In this stage requirements will be decomposed and represented with extended use cases, which is an extension of UML use cases with more rigorous descriptions. In the architectural design stage, based on the requirements expressed in extended use cases, the architecture of the application system will be developed. Suitable components will be selected from the repository based on the system architecture in parallel with architectural design to ensure that the pre-qualified components comply with the system architecture.

To achieve highly and flexible reuse, the mismatches between pre-qualified components and current reuse context need eliminated. In component adaptation stage, adaptation requirements are collected interactively with the application developer, and recorded in the format of adaptation scenarios (applicable scenarios). The default adaptation scenarios are defined by the component developer to cover some typical adaptations and may be used as templates to create applicable scenarios. A scenario collects series of component adaptation actions to satisfy particular adaptation requirement. From the applicable scenarios, the original component specification (CS) will be changed by the component adaptor and finally a specification of the adapted component, namely Component Derivative Instance Specification (CDIS) will be generated. The final products of architectural design and component adaptation include the adapted CDIS and the architectural model.

The next stage is component generation and integration. The target code of the adapted component will be generated based on its CDIS and its primitive component code (fetched from the repository). Then the target component code needs to be verified against the current reuse context. If the target component code does not fit, the adaptation scenarios will be refined further to fit, and more suitable components may be adapted and generated until the application developer is satisfied.

Finally, the components (implementation) will be integrated into the application system.

The application system will be tested, refined if necessary and released to the user.

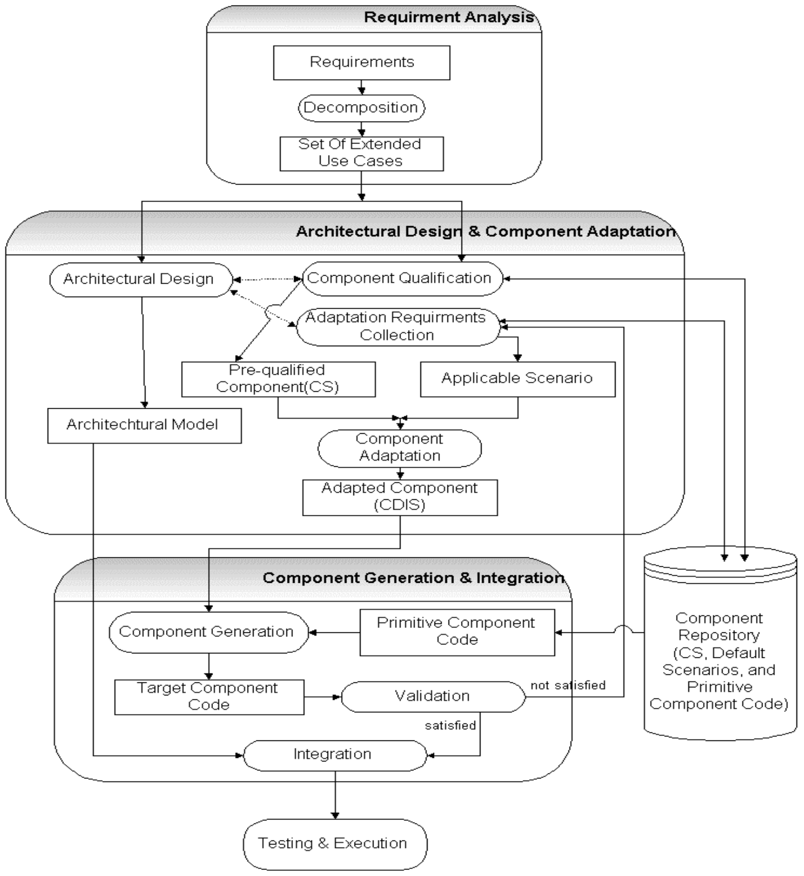


Fig. 1. The Process of SAGA Approach

4. Scenario-based component adaptation and generation

4.1. Component Definition Language (CDL)

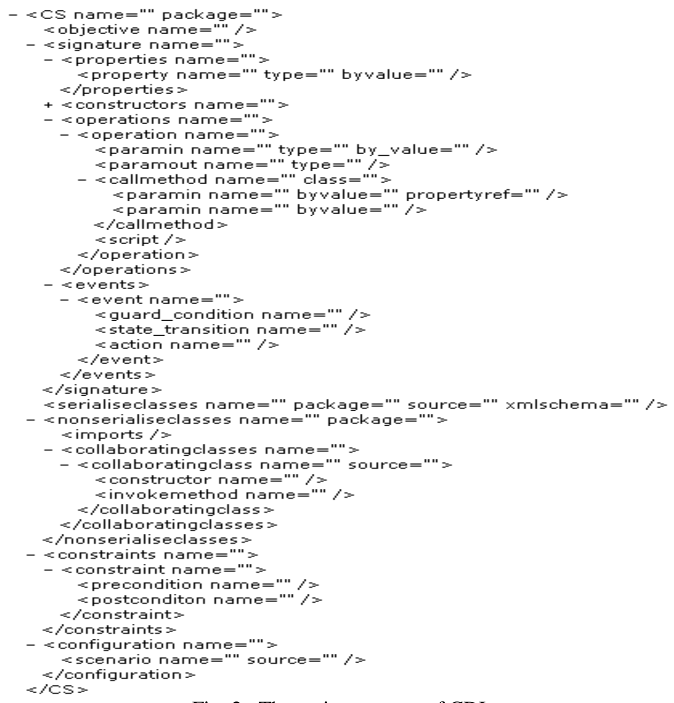
A reusable component can be described in many ways. An ideal description is the 3C model, i.e., concept, content and context [7][10]. A comprehensive, easy to understand component specification is the prerequisite of successful qualification, adaptation and integration [11][14].

We advocate that a specification of a component should consist of views at various abstraction levels. In our approach, a user view represents the definition of the public interface of the component at specification level, and a comprehensive view covers both the component interface and internal implementation details, such as XML formatted meta-code of methods. The user view is designed for software engineers to understand the structural and services of the component, and then based on this view to select pre-qualified components and to describe the adaptation requirement. The comprehensive view is machine-oriented; it is designed to facilitate deep component adaptation and generation.

Based on the collected scenarios, the comprehensive specification of a component (CS) will be adapted and the specification of a new derivative suitable for use in current application (CDIS) will be generated. Therefore, the statements of CDL are classified into two categories: statements of CS and statements of CDIS. CDIS is an adaptation and possibly also an extension of CS based on features of the current application.

In CDL, components are characterized as independent entities and in terms of their usage and interactions with other components. The overall structure of a component definition includes the following six aspects. A basic structure of CDL is given in figure 2.

- The signature of the component, which is the basis for the component's interaction with the outside world. The signature includes all the necessary mechanisms for such interactions, i.e., properties, operations and events.
- The constraints on the component signature in terms of their proper use.
- Design configurations. This section depicts the typical reuse contexts and corresponding adaptation actions in the form of scenarios.
- Non-functional properties. The non-functional properties occupy a special place in the component interface structure, and may interact with the interface signature and configuration.
- Serializeable classes. All the sub-component classes which involve in the definition of the data model of the component and possible mapping to background database will be listed in this section.



```
- <CS name="" package="">
    <objective name="" />
  - <signature name="">
    - <properties name="">
        <property name="" type="" byvalue="" />
      </properties>
    + <constructors name="">
    - <operations name="">
      - <operation name="">
          <paramin name="" type="" by_value="" />
          <paramout name="" type="" />
        - <callmethod name="" class="">
            <paramin name="" byvalue="" propertyref="" />
            <paramin name="" byvalue="" />
          </callmethod>
          <script />
        </operation>
      </operations>
    - <events>
      - <event name="">
          <guard_condition name="" />
          <state_transition name="" />
          <action name="" />
        </event>
      </events>
    </signature>
    <serialiseclasses name="" package="" source="" xmlschema="" />
  - <nonserialiseclasses name="" package="">
      <imports />
    - <collaboratingclasses name="">
      - <collaboratingclass name="" source="">
          <constructor name="" />
          <invokemethod name="" />
        </collaboratingclass>
      </collaboratingclasses>
    </nonserialiseclasses>
  - <constraints name="">
    - <constraint name="">
        <precondition name="" />
        <postconditon name="" />
      </constraint>
    </constraints>
  - <configuration name="">
      <scenario name="" source="" />
    </configuration>
  </CS>
```

Fig. 2. The main structure of CDL

- Non-serializeable classes. Sub-component classes other than serializeable classes will be considered as non-serializeable classes. The relationship among these classes and sub-components may be architectural, such as aggregation and association, or collaboration, i.e., interaction or service requests.

4.2. Scenario

The conflicts between a component and specific reuse requirements need to be eliminated with component adaptation. Certain types of adaptation actions may affect multiple parts of the component. The component-user needs to assure the consistency of the component by propagating the actions to all the affected parts of the component. A scenario of adaptation captures a series of adaptation actions to satisfy certain application requirements. The adaptation actions are classified into various types, namely adaptation types, because the same type of actions will impose common affects on component structures. An adaptation type can be used as a template to generate similar adaptation actions.

Scenario-based adaptation is a suitable technique to adapt components in component based application development. The rationale underlying is that adaptation actions in a scenario are designed for specific reuse contexts of particular applications, and this also makes the adaptation types reusable in other reuse contexts. Therefore, the activity of constructing an application with reusable components becomes one of selecting or building correct scenarios of pre-qualified components, then to do the adaptation actions based on the scenarios, and then to generate the target code of the components based on the adapted CDIS.

Based on the above observation, we have identified that component-based development, in addition to a set of reusable components, requires a set of scenarios. A scenario is composed of a series of adaptation actions, which are defined in adaptation types. With scenarios, and generation deep component adaptation becomes feasible.

Scenarios may be required in three aspects: 1) the component may be used in different reuse contexts; 2) the component may be used under different constraints; 3) the component may act in different roles in given reuse contexts. Scenarios can be composed, associated and adjusted to tackle various and complex reuse cases. The role of scenarios can be defined as the design configuration of components in specific kinds of applications.

The pre-qualified component is defined in a Component Specification (CS). Scenarios aim to perform the adaptation actions on CS instead of component code. The result will be the specification of the adapted component derivative, namely CDIS.

4.3. Component adaptation types

4.3.1 A taxonomy of component adaptation types

We have identified the following five typical categories of component adaptation types:

- Coordination of Component Interface. A typical situation in the development of a component-based system is that a component has been provisionally qualified for the reuse in the system, but its interface has some mismatch with the interface expected by the system. This may be because of differences in operation signatures, or extra functions, or unsuitable constraints. In such situations, the interface of the component needs to be adapted to match the expected interface. We call this category of adaptation

Coordination of Component Interface, which may include the following types: provided service adaptation, required service adaptation, property adaptation, and constraint coordination.

- Component Composition. In component-based development, the functionality that a component should provide may not be fulfilled by a single available component. However, a combination of two or more components is able to provide the required functionality. In such cases, the components have to be composed into a single component. This adaptation category may include component aggregation, and component association.
- Component Interoperability. A set of interacting components will be selected to construct specific application. The relationships between the components are specific in the current application [15], hence, component interoperability is a crucial part of the component adaptation. There are many aspects related to component interoperability, such as message passing, implicit invocation, and state and event invocation.
- Performance Adaptation. This kind of adaptation is motivated by performance correction reasons or environmental changes, which will result in modifications to the implementation of the component, for example, replacing with a new version of a component library, adopting a new programming language, upgrading to a new operating system, or upgrading to a new database system.

4.3.2 Definition of component adaptation types

Adaptation actions are grouped into relevant component adaptation types. An adaptation action is defined with the following attributes and sub-statements:

- *name*, which is the identity of the action;
- *type,* which indicates the type the action;
- *requirement*, which shows the reuse context the adaptation action suits for;
- *position*, which shows a position in CS where the actions to be done;
- *extra detail*, which covers other details needed by the action.

A typical definition of component adaptation types is given in figure 3. Sample definitions can be found in the example section.

```
<adapt_action  name="xxxxx " type="xx_xxx">
   <requirement>xxxxxx</requirement>
   <position>xxx.xxx.xx</position>
    ……    extra detail   ……
</ adapt_action>
```

Fig. 3. A structure of component adaptation type

4.3.3 Propagation relationship

During adaptation, to keep the consistency of the component and the application system, necessary changes must be propagated to all the affected parts of the component and other interrelated components. In SAGA, this is achieved with a set of *propagation relationships*, which are classified into the following types:

- *Triggering.* If the change imposed by an adaptation action causes more related changes to a component, the adaptation action will trigger other adaptation actions or scenarios to make the required change. If the original change and the incurred change apply to the same component, more adaptation actions will be triggered, and the relationship is called *intra-component triggering*. Otherwise, if the original change and the incurred change apply to different components, other scenarios will be triggered, and the relationship is called *inter-component triggering*.

- *Requiring*. An adaptation action may require changes to be done to a component as prerequisite, i.e., to be carried out before the execution of the action. In such cases, the adaptation action will require the execution of other relevant adaptation actions or scenarios beforehand. If the required change happens to the same component, relevant adaptation actions will be required to execute, and the relationship is called *intra-component requiring*. Otherwise, if the required change happens to different components, scenarios will be triggered, and the relationship is called *inter-component requiring*.

An adaptation action is also given an *execution priority* of the following three levels in order to make adaptation more sensible. 1) High level priority. If an adaptation action involves changes to the properties of a component, it will be assigned a high level priority. 2) Medium level priority. Adaptation types that involve changes to the operation or events of a component will have a medium level priority. 3) Low level priority. For adaptation types that make changes to constraints of a component, a low level priority will be assigned.

With propagation relationships, complex adaptation scenarios can be composed to implement deep or large-scale component adaptation. An example is given in section 5 to show how to use priorities and propagation relationships to copy with adaptation in the prototype tool.

4.3.4 Component generation

With component generation technology [9][16][17], the target components will have less overheads in functionality and more efficient performance. This technology is particularly suitable for those components that are not standalone, and designed to be plug-compatible and interoperable with other components. Component generation requires information not only from a component developer but also from a component user. Component developers provide primitive component code, which is the unchanged part of the code of the pre-qualified component. The CDIS of the adapted component, which includes the configuration of changes to the pre-qualified component, is created by the component adaptor through

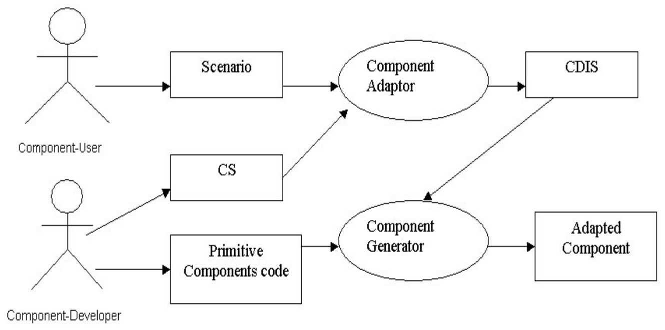


Fig. 4. The process of component generation

scenarios selected or created by component users. With those inputs, the component generator will generate new target code of the adapted component. The component generator will generate the target industry-standard components automatically since the necessary information has been collected interactively in the adaptation stage.

The process of component generation is given in figure 4. Firstly the CDIS of the pre-qualified component is parsed. As the CDIS includes implementation and modification detail, target code of the adapted classes will be generated by the component generator. Then the component generator integrates the adapted classes code with the primitive component code to create the complete code of the adapted component. Finally, for smooth deployment of the generated component into a target application, the generator will provide as much as possible the information of environment modification, such as database schema and database connection update.

Java-based client-server database applications have become very popular nowadays. Our component generator aims to generate the component in the form of JavaBeans and EJBs.

5. Example

Invoice_manger is a popular component used in business applications. In this paper, it is used as an example to explain how the proposed approach and prototype tool work. The prototype tool consists of the following parts: 1) a CS viewer for component-users to view a component interactively, 2) a scenario editor for component-users to make and edit adaptation scenarios, 3) a component adaptor to perform adaptation actions and then create a CDIS of the component based on scenarios, 4) a component generator to automatically

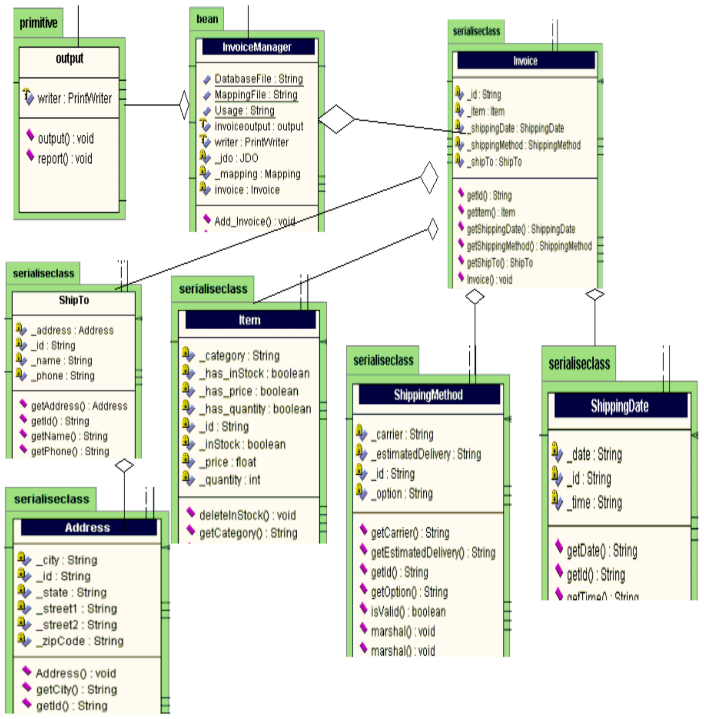


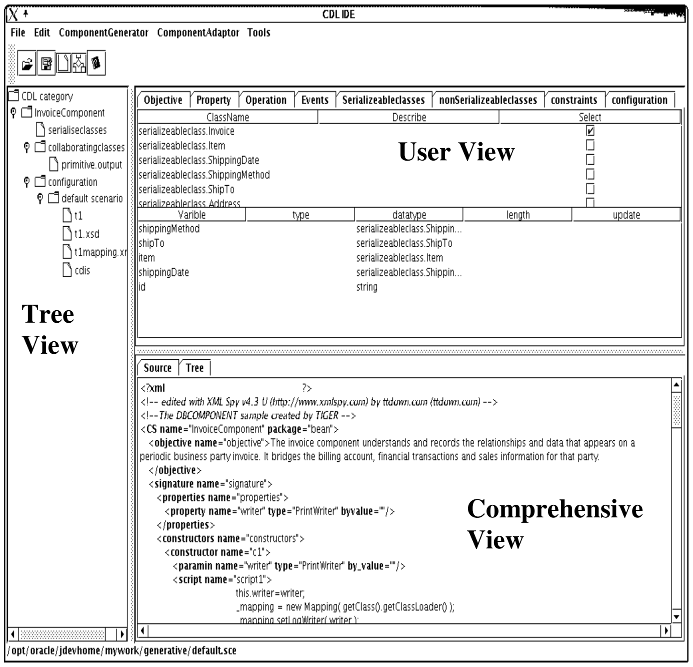
Fig. 5. The class diagram of *invoice_manager* component

generate the code of an adapted component based on the CDIS and primitive code of the component.

The *invoice_manager* component provides the services needed in common business invoice management. Presume that the *invoice_manger* component is a typical database-oriented client server component. It consists of a serial of classes or subcomponents, such as the component interface class, data structure classes, and collaborating classes. The component provides three services in the form of class methods, i.e., *Print_report* to print a simple invoice report, *Add_invoice* to save an invoice object into database, and *Import_XML* to import an XML formatted invoice and save it to the database. The class diagram of *invoice_manager* component is given in figure 5.

5.1 Invoice_ manager component CS

In the prototype tool, the CS can be read by component-users in different views, i.e., user view, tree view and comprehensive view. User view gives the component-user an interactive view of the component CS. With it, the component user can easily understand the component, such as objective, required services, provided services and data structure. Component-users may need more detailed CS information, so they can use the comprehensive view to see the complete component CS. Figure 6 shows the three CS views of



ClassName	Describe	Select
serializeableclass.Invoice		✓
serializeableclass.Item		☐
serializeableclass.ShippingDate		☐
serializeableclass.ShippingMethod		☐
serializeableclass.ShipTo		☐
serializeableclass.Address		☐

Varible	type	datatype	length	update
shippingMethod		serializeableclass.Shippin...		
shipTo		serializeableclass.ShipTo		
item		serializeableclass.Item		
shippingDate		serializeableclass.Shippin...		
id		string		

```
<?xml                                    ?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by ttdown.com (ttdown.com) -->
<!--The DBCOMPONENT sample created by TIGER -->
<CS name="InvoiceComponent" package="bean">
  <objective name="objective">The invoice component understands and records the relationships and data that appears on a
periodic business party invoice. It bridges the billing account, financial transactions and sales information for that party.
  </objective>
  <signature name="signature">
    <properties name="properties">
      <property name="writer" type="PrintWriter" byvalue=""/>
    </properties>
    <constructors name="constructors">
      <constructor name="c1">
        <paramin name="writer" type="PrintWriter" by_value=""/>
        <script name="script1">
                        this.writer=writer;
                        _mapping = new Mapping( getClass().getClassLoader() );
                        mapping setLogWriter( writer );
```

Fig. 6 The multiple views of component specification

invoice_manager component.

Component-users may want to modify a component's context and content to fit-in certain applications, so CDIS will be generated, which is a derivative instance of CS with the change made by component-user. As CDIS and CS have similar structure. Users can use the same viewer to visualize CDIS.

5.2 Component Adaptation

5.2.1 Scenario

Let us assume there are the following adaptation requirements to the *invoice_manager* component when reused in an application. A scenario is developed to tackle the adaptation. Adaptation actions will be added into the scenario to reflect the adaptation requirement.

In the original data structure of *invoice_manager*, the invoice class and item class has a one to one aggregation relationship, however, in current application, the relationship needs to be changed to a one to many relationship. It means the Invoice class needs a collection to hold a group of Item objects. A new operation, namely “add_Items” will be introduced to Invoice class to allow Item objects be added into the collection. Since “add_Items” operation is a consequence or incurred change of one to many association, a intra-component triggering relationship is set up between action “Association_modify” and action “new operation”. The full definition of the scenario is given in figure 7.

As shown in the example, the adaptation actions in the scenario give the information needed for the adaptation change, i.e., the position of the change, and how to make the change. The component adaptor will follow the guide to make the change. For each adaptation type, component adaptor not only makes the changes but also propagates the change effects to other parts of the component being affected through the rules built-in. The final product from the component adaptor is a CDIS of the *invoice_manager* component, with all changes done.

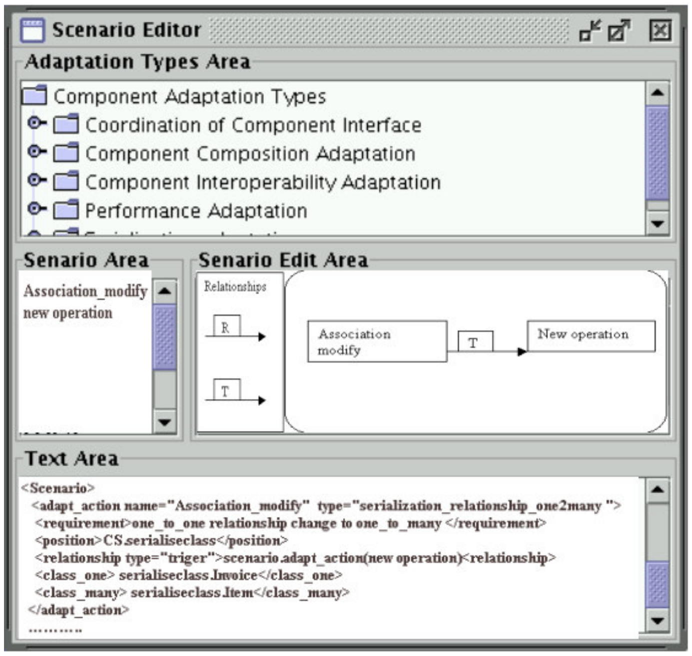
```
<Scenario Demo1>
  <adapt_action name="Association_modify"  type="serialization_relationship_one2many ">
   <requirement>one_to_one relationship change to one_to_many </requirement>
   <position>CS.serialiseclass</position>
   <relationship type="triger">Demo1.adapt_action(new operation)<relationship>
   <class_one> serialiseclass.Invoice</class_one>
   <class_many> serialiseclass.Item</class_many>
 </adapt_action>
 ………..
 </adapt_action>
 ……….
  <adapt_action name="new operation" type="interface_operation_new ">
   <requirement>a new operation is added </requirement>
   <position>CS.signature.operations</position>
   <new>
     <operation name="add_Items">add new items into database
       <paramin name=“method" type="Vector“   by_valuse="" />
       <paramout type="void" />
       <add object=“method" />
     </operation>
   </new>
   <adapt>
 ……….
</Scenario>
```

Fig. 7. An example of scenario definition

5.2.2 Scenario Editor

As shown in figure 8, a user-friendly scenario editor is developed to manage component adaptation scenarios. It consists of four areas, namely, 1) adaptation type pane, to list currently available adaptation types in a tree structure; 2) scenario pane, to list adaptation actions contained in current scenario; 3) scenario editor pane, to edit propagation relationships graphically; 4) text pane, to display the full XML formatted scenario definition.

With the scenario editor, users can easily create a scenario in three steps: 1) drag an adaptation type from adaptation types pane and drop it into scenario pane; 2) an adaptation action will be automatically created in the scenario, and user can see the change in scenario pane, scenario editor pane and text pane; 3) graphically modify the propagation relationships



```
<Scenario>
  <adapt_action name="Association_modify"  type="serialization_relationship_one2many ">
   <requirement>one_to_one relationship change to one_to_many </requirement>
   <position>CS.serialiseclass</position>
   <relationship type="triger">scenario.adapt_action(new operation)<relationship>
   <class_one> serialiseclass.Invoice</class_one>
   <class_many> serialiseclass.Item</class_many>
  </adapt_action>
  ..........
```

Fig. 8 The Scenario Editor

in scenario edit area.

5.3 Component generation

An adapted *invoice_manger* component is finally generated by the component generator. Most parts of the code of the original *invoice_manager* component are copied into the target code of the adapted *invoice_manager* component. Only the code of the classes with changes will be generated and then replaced by the component generator. The change is shown in the figure 9. The classes on the left are original classes, those on the right are the classes generated.

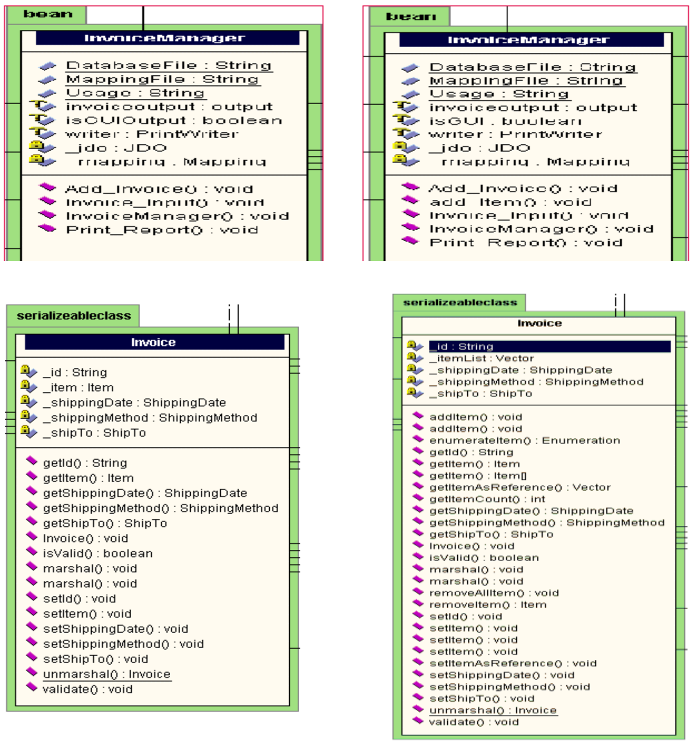


Fig. 9 The changes in InvoiceManager and Invoice class

6. Conclusion

Based on the observation that similar design circumstances recur frequently in the development of software application systems, we concluded that raising reusability in applications would improve the efficiency of development greatly. The fact that existing reuse approaches and tools are weak in providing a mechanism to adapt components with high flexibility and low overheads to fit into various reuse contexts triggered the research in this paper.

Components in our approach aim to be highly adaptable in order to fit smoothly into more flexible reuse context. This is achieved with three key techniques: 1) XML-based comprehensive and multiple-viewed component specification, 2) scenarios to capture adaptation requirements in particular applications, and 3) component generation through the derivative specifications of the adapted component and primitive component code. These components are the reusable blocks to build a new application system. Existing expertise and idioms of software design can be recorded in these components. Components are described in the XML-based Component Definition Language and stored in the component repository. The CDL is a structural and semantic specification language. The features inherited from XML make the specification of components easy to understand, to exchange and to propagate over software tools and communities.

The SAGA technology gives a great potential for coping with component incompatibilities, It meanwhile reduces the overheads in components. A scenario gives component-users understandable and interactive component adaptation information, and components generated based on the scenarios will enjoy high suitability and efficiency in particular applications. As a project, the work presented in this paper is ongoing. Our case studies have shown the work is promising.

7. References

[1] Antônio Augusto Fröhlich, "Scenario Adapters: Efficiently Adapting Components", *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Florida, USA, July 23-26, 2000.

[2] Bosch. J, "Superimposition: A Component Adaptation Technique", Information and Software Technology, Volume 41, No. 5, March 1999.

[3] Batory. D., "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering (Special Issue on Software Reuse)*, pp. 67-82,Feb. 1997.

[4] Batory. D., "Intelligent components and software generators", *Technical Report 97-06*, University of Texas at Austin, 1997.

[5] Beihu Wang, Xiaodong Liu and Jon Kerridge, "A Pattern-Based Framework for Database Reusability", *International Conference on Software Engineering: Research and Practice (SERP'02),* Las Vegas, USA, June 2002.

[6] B.Kucuk, M.N.Alpdemir, "Customizable Adapters for Blackbox Components", *Proceedings of the 3rd International Workshop on Component Oriented Programming*, 1998, pp. 53-59

[7] Biggerstaff. T.J., "A characterization of generator and component reuse technologies", *presented at the GCSE 2001*, Germany, 2001.

[8] C.J.EGYHAZY, "From software reuse to database reuse", *International journal of software engineering and knowledge engineering*, 10: pp. 227-249, 1998.

[9] Hans de Bruin, H.v.V., "The future of component-based development is generation", *Proceedings of ECBS'02 Workshop on CBSE -- Composing Systems from Components*, Lund, Sweden, 2002.

[10] Han, J., "A comprehensive interface definition framework for software components", *Proceedings of 1998 Asia-Pacific Software Engineering Conference,* Taipei, Taiwan, 1998,pp. 110-117.

[11] Han. J., "Characterization of componts", *Proceedings of International Workshop on Component-Based Software Engineering,* Kyoto, Japan, Apr. 1998.

[12] Geoge T. Heineman, H.M.O., "An evaluation of component adaptation techniques", *the 2nd Annual Workshop on Component-Based Software Engineering*, Los Angeles, CA, May 17-18, 1999.

[13] George T. Heineman, "A Model for Designing Adaptable Software Components", *Proceedings of the 22nd Annual International Computer Science and Application Conference (COMPSAC-98)*, Vienna, Austria, 1998.

[14] Motta, E., Fensel, D, "Specifications of Knowledge Components for Reuse", *Proceeding of the 11th International Conference on Software Engineering and Knowledge Engineering*, 1999.

[15] Ralph keller, U.H., "Binary Component Adaptation", *Proceedings of ECOOP'98*, Belgium, 1998.

[16] Ulrich Breymann, K.C., "Generative components: one step beyond generic programming", *presented at Generic Programming*, SchloB Dagstohl, April 27-30, 1998.

[17] V. Singhal, D. Batory, "P++: A Language for Large-Scale Reusable Software Components", *Proceedings of the Sixth Annual Workshop on Software Reuse*, Owego, New York, Nov 1993.

[18] W. Lowe, M. Noga, "Component Communication and Data Adaptation", *Proceeding of the 6th World Conference on Integrated Design and Process Technology*, 2002.