

CGC : An Architecture to support Better and Faster Component Evolution

Naiyana Tansalarak

Kajal T. Claypool

Department of Computer Science
University of Massachusetts
Lowell, MA 01854
{ntansala | kajal}@cs.uml.edu

Abstract

Component-based technology has become the preferred way of producing high quality software systems fast and with less effort. However, component development while offering modularity, introduces a number of dependencies between the different interacting classes, making class dependencies a critical factor in the achievement and performance of component evolution and further component-based system. In this paper, we look at managing class dependencies as the first step towards achieving online, dynamic component evolution. Towards that, we propose a new architecture, *CGC*, that provides (1) the isolation of class behavior and class interaction; and (2) a hierarchical class dependency graph. For component evolution, this architecture offers: (1) fast search capability to discover all affected classes for a given change; (2) reduction of the result set of affected classes; and (3) minimization of the update process. Another key advantage of our architecture is that its base implementation requires no additional constructs to be added to current existing programming languages. In this paper, we provide an overview of the *CGC* architecture, an evaluation of its performance compared to other architectures that have been proposed in literature, and a prototype implementation using current Java technology and JML.

Keywords: core-class, gateway, contract, class-level dependency, method-level dependency, CGC Architecture

1 Introduction

Current trends show that component-based software technology has become the preferred way of producing software systems fast and with less effort, while maintaining high quality [9, 11]. However, it has been observed that after deployment, software systems typically enter a phase of *evolution* that can last many years. This period is the longest and perhaps the most expensive phase of a system's life cycle [18, 20], with its costs likely to exceed the first development costs by a factor of 3 or 4 [18]. The software evolution problem is aggravated further in component-based software systems, where internally a component, conforming to component models [2] such as JavaBean, COM, and CORBA, are often implemented by multiple classes [16, 6]. This behavior distribution can result in many implicit class dependencies, making it less likely that a class can work without the support of other classes within a component. This dependency among classes becomes a critical factor when a class internal to a component needs to be upgraded to (1) correct faults; (2) enhance functionalities; and (3) adapt to new environments [14, 18].

Currently, component evolution is performed manually by first discovering all classes affected by a given class change, and then propagating the appropriate change to all affected classes. Searching for all affected classes requires tracing *all* interactions from *all* classes in a component, to the now changed class. Here, a search typically implies searching of the actual class code. This searching is tedious, time-consuming, error-prone and hence an expensive process. For large components, this process alone can render component-based systems unavailable for long periods of times. Clearly, this is unacceptable for mission-critical software systems such as air traffic control and banking systems that cannot be taken off-line. A faster and less labor-intensive search is needed to facilitate faster online component evolution and further component-based software evolution [13, 14].

Class dependency has been recognized as a major deterrent in achieving faster searches for affected classes, and hence for achieving better performance for component evolution. Several approaches have been proposed to reduce class dependencies, and in some cases to also make class dependencies explicit. Examples of such approaches include *Interface Connection Architecture (ICA)* [4] and *Mediator Pattern (MP)* [3, 7]. ICA specifies explicit class dependencies via interfaces. Searching for affected classes in this architecture requires examination of the interfaces only (as opposed to examination of code). However, ICA requires additional constructs to be introduced in current programming languages to enable its implementation, thereby limiting its applicability in current systems. MP architecture introduces mediators respectively, to govern the interaction between classes. Mediators reduce inter-class dependencies, but do little to make class dependencies explicit. As a result, class dependencies are still embedded in code requiring time intensive and laborious examination of code to discover affected classes. Moreover, the dependency among classes in these three approaches is provided at class-level.

This paper addresses the deficiency of the previous approaches and focuses on the fine-grain level of components, providing a hierarchical class dependency graph to reduce the search space for affected classes. To achieve this, we introduce architectural concepts that provide (1) isolation of class behavior from class interaction; and (2) explicit class dependency. To isolate behavior and interaction, we split the *traditional class* into a *core-class*, and a *gateway*. The *core-class* encapsulates the class's local behavior, while the *gateway* encapsulates the interaction between the classes. The class dependency is now broken into a *core-class to gateway* dependency and a *gateway to gateway* dependency. This breakdown of a class by behavior and interaction has several benefits namely, (1) local change in the core-classes that does not affect other core-classes is contained in the core-class; and (2) dependencies for non-local changes are easily found via gateway to gateway dependencies. However, it is possible that using these dependencies alone, the class may be specified to be an *affected* class when in fact the class is not. This is especially true when the granularity of the class change is at the method level. For this, we further refine the *core-class to gateway* and *gateway to gateway* dependencies by introducing *contracts* that provide an additional level of dependency for methods. This new *method-level dependency* allows the filtering of false positives, in terms of the affected classes, found by a search using only *class-level* dependency.

To effectively parallel our new architectural concepts at the implementation level, we show in this paper how the *class* and the *interface* constructs of current Java technology [5] can be utilized to achieve the core-class and gateway concepts. In particular we show how Java can provide the (1) interfaces for core-classes; (2) interfaces for gateways; and (3) class dependencies that are embedded in these interfaces. We also show how tools such as JML (Java Modeling Language) [10] can be used to validate the conformance of the implementation with the specification supplied at the architecture level.

To summarize, our work makes the following contributions:

- The proposal of the *CGC (core-Class::Gateway::Contract)* Architecture. This architecture offers a precise plan for (1) predicting component behavior before a component is built; (2) guiding the development of a component; and (3) predicting the modification when a component is changed.
- The *implementation* of *CGC* architecture using current Java technology and JML.

The rest of paper is organized as follows. Section 2 presents our running example. The next two sections, Section 3 and 4, describe the *CGC* architecture. In particular, Section 3 introduces the *architectural concepts*, *core-class* and *gateway*, to represent the *class-level dependency*; and Section 4 introduces *contracts* for methods to represent the *method-level dependency*. Section 5 reports a performance comparison of *CGC* with *OCA*, the current class architecture, and the previously proposed approach, *MP*. Section 6 surveys related literature. Finally, Section 7 concludes the paper.

2 Working Example

In this section, we present a *Student Registration System (SRS)* [1] as an example to illustrate the key concepts of *CGC* throughout the rest of this paper. Figure 1 illustrates the class dependencies among four classes: *Student*, *Course*, *Section*, and *Transcript*, using *Object Connection Architecture (OCA)* [4]. *OCA*, implemented in many current programming languages such as Java, provides interfaces that specify the signature of all provided features and a set of modules. Class interaction is typically embedded in class source code as depicted by dashed lines in Figure 1. Consider

an enrollment component that provides the interface to allow a student to enroll for a section. This functionality is mainly responsible by the *enroll()* method of class *Section*. The *enroll()* method enrolls a student into the section of a course if the student (1) is not currently registered for this section; (2) has never registered for this course **or** has failed this course; (3) has passed all prerequisites for this course; and (4) there is seat available. To achieve the *enroll* behavior, class *Section* must interact with three classes: *Course*, *Student* and *Transcript* as shown in Figure 1. For example, to verify whether a student is currently enrolled in this section, *Section* must interact with *Student* by sending the message *isEnrolledIn()* to *Student* as shown in the sequence diagram [8] (Figure 2). Note here that all class interactions are embedded in the class code and the dependencies between classes are considered to be implicit.

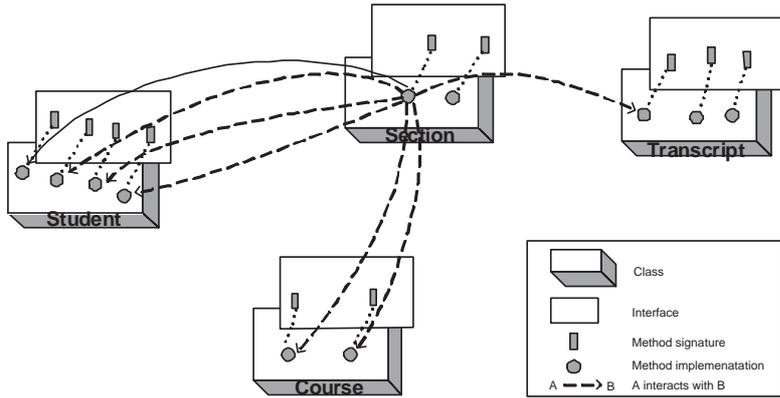


Figure 1: The Class Dependency of the *enroll()* method on OCA

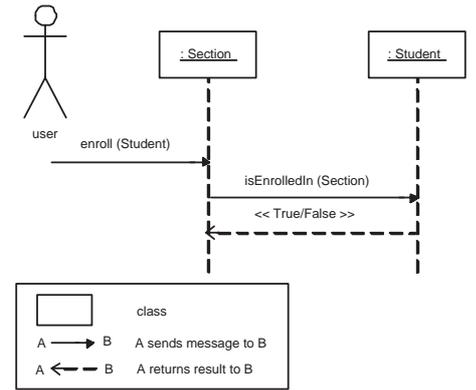


Figure 2: The Sequence Diagram on OCA

Based on this example, in the rest of the paper we show how class dependencies can be made explicit and also reduced from *class to class* dependency to *core-class to gateway* dependency and *gateway to gateway* dependency.

3 Class-Level Dependency

Classes, defined using current specifications such as the Java specification, include both behavior as well as interaction between classes as an integral part of the class's definition [9, 19]. When a class interacts with other classes, it is implicit that its behavior is not complete within itself, i.e., it is not complete without interacting with other classes. The class is thus said to be dependent on other classes. We term this the *class-level* dependency. Based on previous approaches, we recognize that reducing the tight coupling of behavior and interaction in a class is critical for reducing class dependencies. Moreover, to facilitate easy discovery of affected classes, it is important to make explicit the dependency between classes. In this section, we address these two issues by proposing an architecture that (1) isolates the class behavior from the class interaction; and (2) explicitly captures class dependencies. We then show how these concepts can be implemented in Java.

3.1 Isolation of Behavior and Interaction

Figure 3(a) illustrates the *traditional* class architecture as implemented in Java. Figure 3(b) illustrates the *core-Class::Gateway* architecture where the traditional class architecture is separated by class behavior and class interaction. We now introduce two distinct terms: *core-class* and *gateway*. *Core-class* encapsulates the class behavior while *gateway* encapsulates the class interactions. Let T be *traditional class*, C be *core-class*, and G be *gateway*. Let dm_t and dm_c denote the data members of T and C respectively. Let m_t , m_c and m_g be methods of T , C and G respectively. In addition, b denotes behavior; i interaction; and msg the message. Based on these, we present the definition of traditional class, core-class and gateway as shown in Figure 4. Here the key difference is that in the *traditional* class architecture a class T_i interacts directly with another class T_j . In the *core-Class::Gateway* architecture, there is now a layer of indirection that is introduced by the gateway. Hence, the msg_i is sent from core-class C_i to gateway G_i which then routes msg_i to the gateway G_j of the class C_j .

The core-class thus completes its behavior not by interacting directly with other core-classes, but rather by interacting through its gateway (by sending messages to its gateway). For example, instead of class *Section* sending the message

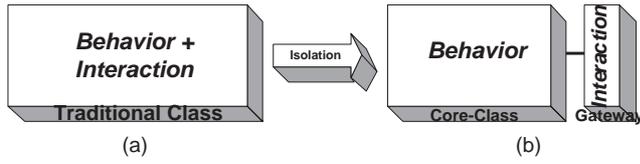


Figure 3: Isolation of class behavior and class interaction

$$\begin{aligned}
 T &= \{ dm_t \} \cup \{ \text{private and public } m_t \mid m_t = \{b, i\}; i \text{ is msg to other classes} \} \\
 C &= \{ dm_c \} \cup \{ \text{private and public } m_c \mid m_c \text{ is similar to } m_t \text{ except } i \text{ is msg to its } G \}; \text{ where } dm_c = dm_t \\
 G &= \{ \text{public static } m_g \mid m_g = \{i\}; i \text{ is msg to its } C \} \cup \{ \text{public static } m_g \mid m_g = \{i\}; i \text{ is msg to other } Gs \}
 \end{aligned}$$

Figure 4: Definition of Traditional class, Core-class, and Gateway

isEnrolledIn() to class *Student*, core-class *Section* will now send the message to its gateway, *SectionGateway*. Gateway *SectionGateway* then is in charge of routing the message to core-class *Student* via gateway *StudentGateway* as shown by the sequence diagram in Figure 5. The gateway serves two purposes, namely, it is the communicator through which other core-classes interact with its core-class, and through which its core-class communicates with other core-classes. Thus the gateway provides a list of not only the *provided* features, methods that its core-class provides, but also a list of *used* features, methods that are used by its core-class. The gateway, thus, effectively provides a wrapper for every external method, method in another core-class that is used by its core-class. Moreover, by virtue of providing a list of *used* and *provided* features, the gateway makes explicit the *class to class* dependency, eliminating the need for expensive code searches typically involved in discovering affected classes. A point to note, there is no gateway between a *superclass* and its *subclasses* as an object of a subclass can be treated as an object of its superclass [5].

Figure 6 illustrates an architectural view of the example given in Figure 1. Here, we introduce four gateways namely, *StudentGateway*, *CourseGateway*, *SectionGateway* and *TranscriptGateway* for classes *Student*, *Course*, *Section* and *Transcript* respectively. These classes now represent core-classes and hence provide only the class behavior. All interactions between core-classes are done via gateways. The class-level dependency is therefore now reduced from *class to class* dependency to *core-class to gateway* dependency and *gateway to gateway* dependency. As can be seen in the figure, all dependencies are captured at gateways.

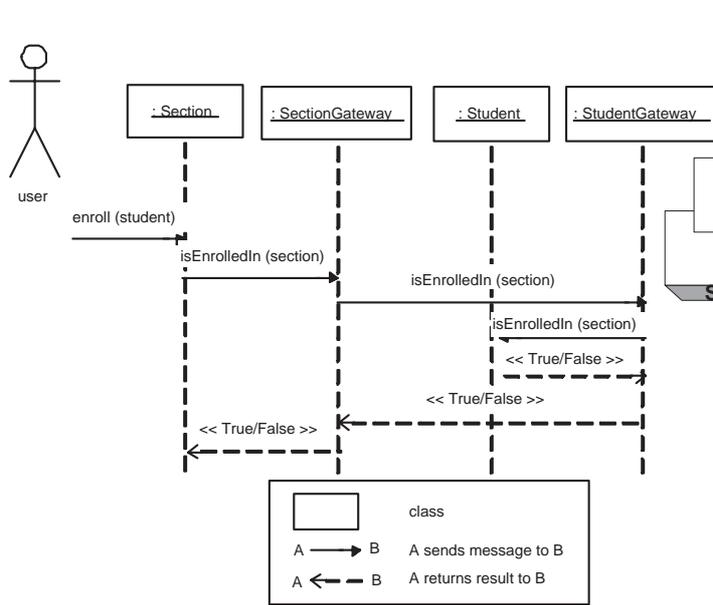


Figure 5: The Sequence Diagram on CGC

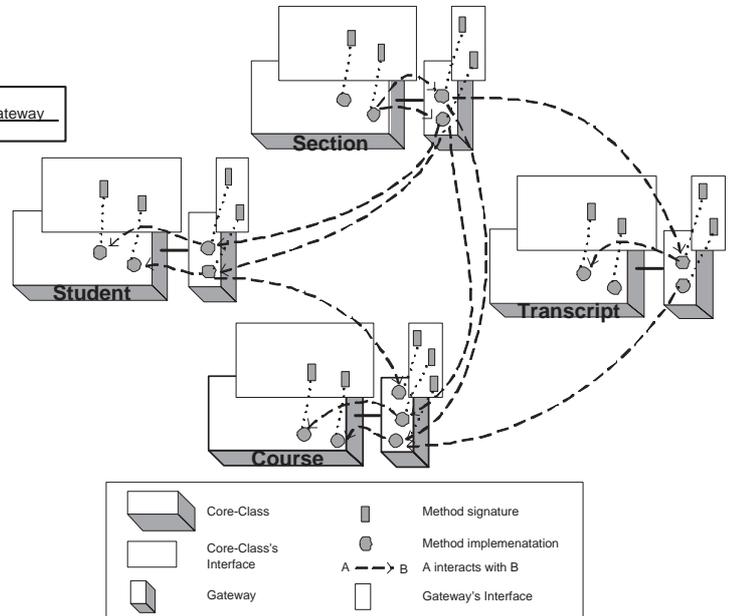


Figure 6: The Class-level Dependency of the *enroll()* method on CGC

3.2 Implementing with JAVA

One of the goals of our work is to remain within current capabilities of programming languages in the implementation phase. We next show how we use the Java programming language to implement *Core-class::Gateway Architecture* for the example introduced in Section 2. As behavior and interaction of a class are now separated into core-class and gate-

way respectively, each core-class and gateway needs to have a separate implementation. We use the *class* construct to implement both the core-class and the gateway, and the *interface* construct to specify the *provided* and the *used* method signatures for both the core-class and the gateway. To provide a complete implementation of the `Section` class, we must provide the following: (1) `SecInterface.java` to define all provided method signatures for the core-class `Section`; (2) `Section.java` to implement local behavior and all core-class interactions that are bounded by `SectionGateway`; (3) `SecGateInterface.java` to define all provided and used method signatures for the gateway `SectionGateway`; and (4) `SectionGateway.java` to implement the interaction between core-class `Section` and other core-classes via their gateways. A fragment of the code for the core-class and gateway are shown in Figure 7(a) and (b) respectively. Here, the `enroll()` method of `Section` interacts with the `isEnrolledIn()` of `Student` by sending a message `isEnrolledIn()` to `SectionGateway`. `SectionGateway` then sends a message `isEnrolledIn()` to `StudentGateway` which in-turn sends the message to its core-class `Student`. All methods in a core-class are implemented as usual with the exception that a core-class has no knowledge about other core-classes and hence all messages to other core-classes are routed via its gateway. In our implementation, we have chosen to provide one gateway for all instantiation of the core-class. This decision is based on the logic that the gateway does not capture any state information but simply serves as a router. For this reason, methods in a gateway are defined as *static* methods. A point to note, there are no real restrictions that are imposed on direct core-class to core-class interactions. A construct such as *friend* construct in C++ can be utilized to enforce and hence restrict direct interactions between core-classes. This would require an extension to the current Java specification. We do not discuss this point in this paper.

```

Class Section {
    public int enroll (Object sObj) {
        - calls SectionGateway.isEnrolledIn (sObj, this)
        - ...
    }
}
Class SectionGateway {
    public static boolean isEnrolledIn (Object sObj, Section sec) {
        - calls StudentGateway.isEnrolledIn (sObj, (Object) sec)
        - ...
    }
}
(a)

```

```

Class Student {
    public boolean isEnrolledIn (Object secObj) {
    }
}
Class StudentGateway {
    public static boolean isEnrolledIn (Object sObj, Object secObj) {
        - calls (Student) sObj.isEnrolledIn (secObj)
        - ...
    }
}
(b)

```

Figure 7: The implementation of the core-class and the gateway

4 Method-Level Dependency

The most frequent type of component evolution occurs at the fine-grained level, which is at the level of data members and methods of a class. In a class environment, often a method (class method) m_i in class c_i cannot complete its behavior without interacting with some other method(s) m_j in some other class(es) c_j . Here, we say that method m_i is dependent on method m_j . We term this the *method-level dependency*. Typically, a class c_i will interact with some but not all methods of a class. Taking only the class-level dependency into account may falsely identify classes to which an update must be propagated. Consider the class-level dependency shown in Figure 6. All three core-classes: `Student`, `Section`, and `Transcript`, interact with core-class `Course`. Now consider that the `getCourseName()` method in core-class `Course` is renamed. With only class-level dependency, all three core-classes are considered to be affected. Intuitively, we can observe that only one core-class, `Student`, should be affected as only its method `displayCourseSchedule()` interacts with the `getCourseName()` method. Thus, to further reduce the result set of affected classes, we now define a new explicit level of dependency, *method-level dependency*, that captures the dependency between the methods of two or more classes. The method-level dependency represents both *direct* and *indirect* method dependencies. Consider the scenario that a method m_i in class c_i interacts with a method m_j in class c_j which in turn interacts with a method m_k in class c_k . This implies that the method m_j directly interacts with a method m_k , while method m_i indirectly interacts with a method m_k . A modification to method m_k affects not only a class c_j but also a class c_i . Therefore, all affected classes must be discovered by not

only the direct but also the indirect method dependencies. In general, *class-level dependency* alone can be considered for evolution operations such as class addition and class removal, while *method-level dependency* must also be considered for class modification operations such as addition, deletion, and renaming of data members and methods.

In this section, we now show how our *core-Class::Gateway* architecture can be extended using *contracts* to support *method-level dependency* and thus provide a refinement to class dependencies as well as explicitly capture class dependencies at interface level. We then show how contracts can be implemented in Java and JML.

4.1 Contracts

Enhancing the syntactic information with semantic information (*specification*) is a useful way of expressing the intent of the interface provider, and enabling developers to have all information necessary for the correct and effective use of methods [15]. Moreover, specification can be used to track method dependencies and determine the impact of the respective evolution step [15, 17]. Specification can be represented as *contract* [12] between the supplier of a certain service and the client of that service. As in [12], we use *precondition* and *postcondition* to describe the contract. The *precondition* defines the condition under which a call to a method is legitimate, and the *postcondition* defines conditions that must be ensured by a method on return.

Contracts must be provided for methods of both core-class and gateway, allowing explicit specification of *intra-method* dependencies, i.e. dependencies within the class, and *inter-method* dependencies, i.e. dependencies between classes. Therefore, core-class explicitly captures dependencies within class and between superclass and subclass, while gateway explicitly captures dependencies between classes. As an example, consider Figure 8. The *enroll()* method in core-class `Section` interacts with the *getRepresentedCourse()* method also in core-class `Section`, and the *isEnrolledIn()* method in core-class `Student`. This interaction is depicted by dashed lines in Figure 8. Contracts, pre- and postconditions, for the *enroll()* method of core-class `Section` and the *isEnrolledIn()* method of gateways `SectionGateway` and `StudentGateway` are specified to provide explicit dependencies at the method level. Figure 9 presents the contracts for the sample methods. All pre- and postconditions are represented using logical statements.

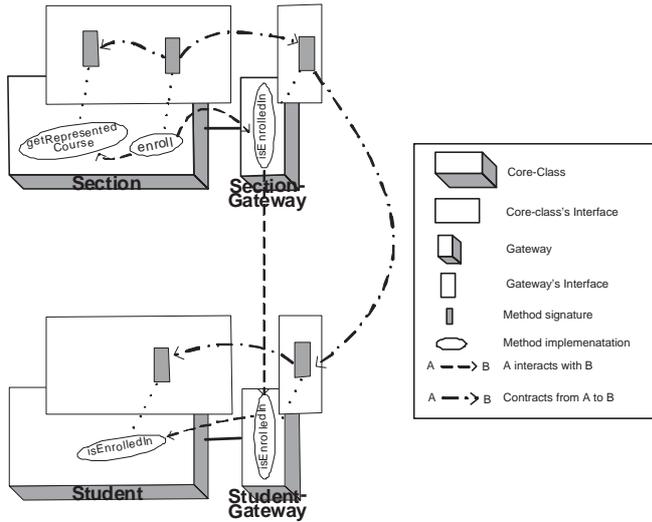
We now introduce a new **method** clause in the pre- and postconditions to explicitly capture the method interactions. For all local methods, i.e., methods of the core-class itself, the core-class instance name does not need to be specified, while a full path to the method must be specified for all other methods, such as super and the gateway’s name. For example, as indicated in Figure 9, the precondition for the *enroll()* method has two method clauses, the *getRepresentedCourse()* method and the *SectionGateway.isEnrolledIn()* method. Note that the *isEnrolledIn()* method must be preceded by the path to its location. Similarly, the postconditions indicate the results obtained from the different methods with which the given method interacts. For example, the postcondition *StudentGateway.isEnrolledIn()* for the *isEnrolledIn()* method in gateway `SectionGateway` indicates the interaction with gateway `StudentGateway`, and the postcondition (*Student*) *sObj.isEnrolledIn()* for the *isEnrolledIn()* method in gateway `StudentGateway` indicates the interaction with core-class `Student`.

4.2 Implementing with Java and JML

From an evolution standpoint, the specifications, i.e. contracts, offer another level of dependency which can now filter out the false positives in terms of the classes to which a change may have to be propagated. From a practical standpoint, however, the usefulness of contracts is limited without some form of verification of the contracts. JML¹ [10] is a tool, written in Java for Java, to verify the conformance of the implementation to the specification provided for each method signature in an interface. JML provides both static and run-time checking. JML currently supports checks for the existence of the variable type in specification, binding of variables between specifications and implementation, and can also generate run-time code to support run-time checking. Currently, however, the run-time checking is limited to parameter checking in preconditions only.

In our work, we use JML at the implementation level to verify the conformance of the implementation to the contracts embedded in the interfaces of both core-class and gateway. As can be seen in Figure 8, all dependencies are captured at

¹JML stands for "Java Modeling Language". JML [10] is a behavioral interface specification tailored to Java and developed by Iowa State University.



```

Interface Section {
    method enroll
    precondition : sObj is Object && exist getRepresentedCourse() &&
                  exist SectionGateway.isEnrolledIn()
    postcondition : result == PREVIOUSLY_ENROLLED ||
                  result == SUCCESSFULLY_ENROLLED
}
Interface SectionGateway {
    method isEnrolledIn
    precondition : sObj is Object && sec is Section &&
                  exist StudentGateway.isEnrolledIn()
    postcondition : result == StudentGateway.isEnrolledIn(sObj,(Object) sec)
}
Interface StudentGateway {
    method isEnrolledIn
    precondition : sObj is Section && sec is Object
    postcondition : result == (Student) sObj.isEnrolledIn(sObj,(Object) sec)
}

```

Figure 8: CGC Architecture showing the interface and implementation for core-classes and gateways. The upper-level represents the interfaces, and the lower-level represents the implementation.

Figure 9: Contract : pre- and postconditions

the interface level and in Figure 9, all contracts are defined via interfaces. However, only partial verification can be done as JML does not support the *method clause* as introduced here for specifying class dependencies.

5 Performance

The *CGC* architecture provides improved searching of dependent classes and methods, using the two-levels of dependencies as introduced earlier in this paper. In providing these explicit levels of dependencies, our architecture however, introduces a level of indirection. That is, rather than having direct class to class communication (current Java architecture), we now have core-class to gateway, gateway to gateway, and gateway to core-class communications. In this section, we explore (1) the performance of the introduced indirection in our architecture; and (2) the benefits of this architecture for the reduction of the search space of dependent classes.

5.1 Runtime Performance

We compare the runtime performance of the *CGC* architecture with that of the *Object Connection Architecture (OCA)* [4] - current Java technology, and *Mediator Pattern (MP)* [3, 7]. Recall that *OCA* enables direct interaction between classes, while *MP* introduces mediators to govern class interactions. Thus for n invocations from one class to another, *CGC* requires $3n$ invocations, while *MP* requires $2n$ invocations and *OCA* requires n invocations.

All experiments were done using *OCA*, *MP*, and *CGC* versions of the same Java program. The Java files can be found at <http://www.cs.uml.edu/~ntansala>. The experiments were conducted on a standalone PC Pentium IV 1.8 GHz, and 132 MB RAM in Microsoft Windows 2000 environment. The load on the machine was kept constant for all runs of the experiments. As we were primarily interested in the degradation of performance caused by the additional context switch, the behavior of the method in the experiments was also kept at a constant. No JML runtime checking was used when running the experiments for the *CGC* architecture. Figure 10 summarizes the performance results of the three architectures measured in milliseconds as the number of class invocations was increased. For a typical program where the number of class invocations is less than 5 or 10 class invocations, it can be noted that the difference in the performance of all three architectures is negligible. As the number of class invocations was increased, we did see a linear degradation in the performance of the *CGC* architecture. However, from a practical standpoint, such a case is highly unlikely.

5.2 Component Evolution Performance

We categorize component evolution into three main types; class addition, class removal, and class modification. *Class addition* involves the addition of a new class into the component. This new class must typically be integrated into the

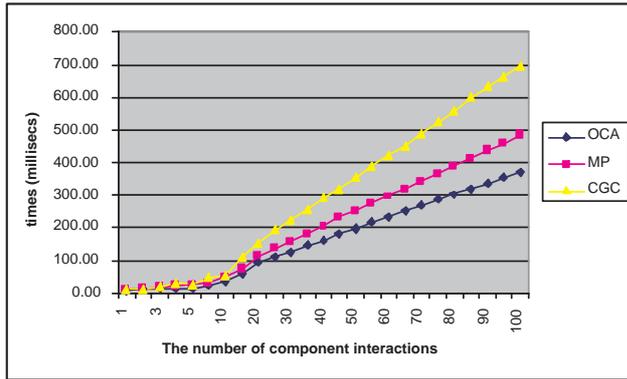


Figure 10: The Runtime Performance

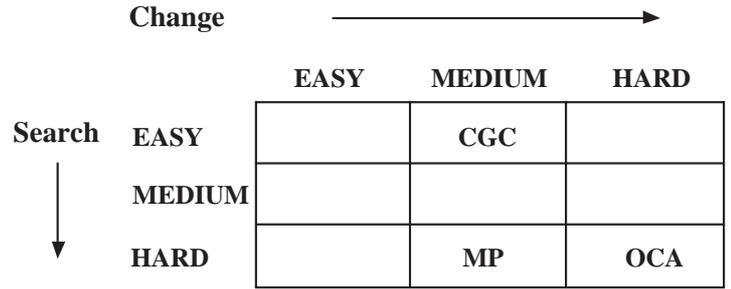


Figure 11: The System Evolution Performance

component, i.e., if it requires interaction with other existing classes or vice versa, then the appropriate actions must be taken. *Class removal* involves the deletion of an existing class from the component. This can affect existing classes that interact with it. Thus, all affected classes must be discovered and the appropriate change must be propagated to all of these classes. *Class modification* involves the addition, deletion, and modification of both data members and methods of an existing class in the system. *Method addition* and *method removal* are considered to have the same effect as class addition and class removal respectively. *Method modification* may involve a change to the method’s behavior, rename of the method name, or a change to the method’s signature. Changing the method behavior is considered a local change that can be contained within the class itself, while other changes, such as the rename and signature change, are *non-local changes* and can affect other classes requiring interaction with it. Thus, in such a case all affected classes must be searched and the appropriate change must be propagated to all affected classes.

We compare how these operations are handled in each of the three architectures that we consider here. In this comparison, we categorize the search space for affected classes, and the number of changes that must be made under each architecture. Figure 12 presents the comparison. Based on the search for all affected class, in OCA and MP, all code must be searched, while searching the interface alone is sufficient in CGC. Based on the number of changes that must be made, in OCA all affected classes must be changed, while only all affected mediators in MP and all affected gateways in CGC need to be changed. However, all affected classes in MP and all affected core-classes in CGC must be changed in a scenario where the signature of a required method is changed. Based on the search and the number of changes provided in this comparison, we thus evaluate these three architecture on the three difficulty levels: *easy*, *medium*, and *hard* as shown in Figure 11. Here, both the search and the number of changes of OCA are considered to be in the hard category. In MP, the search is considered hard, while the number of changes is at the medium level. In CGC, the search is easier and hence categorized at the easy level. The number of changes that must be made are on par with MP.

6 Related Work

Object Connection Architecture (OCA). The object connection architecture [4] provides (1) interfaces that specify the signature of all provided features; and (2) a set of classes that implement all of the provided features. In this architecture, both behavior and interaction between classes are embedded in the class code, introducing implicit dependencies between all classes. From the evolution perspective, all of the *class code* in a component must be traced to discover all affected classes for any given change in a class. This is tedious, time-consuming and therefore an expensive process. In *CGC*, all *gateway interfaces*, encapsulating only signature and specification for interaction purpose, are searched to seek all affected classes. This thus reduces the search complexity.

Interface Connector Architecture (ICA). The interface connector architecture [4] is similar to OCA with the only difference being in the provision of an interface that specifies both *required features*² and *provided features*. All class

²The provided features are similar to the used features in CGC

Evolution Operation	OCA	MP	CGC
Class Addition <i>Assumption</i> : A new class is integrated to a system Change	Class codes	Class codes Mediator codes	Core-class interfaces & codes Gateway interfaces & codes
Class Removal <i>Assumption</i> : A class is removed Search Change	Class codes Class codes	Mediator codes Mediator codes	Gateway interfaces Gateway interfaces & codes
Class Modification <i>Assumption</i> : A method of a class is renamed Search Change <i>Assumption</i> : The signature of a method of a class is changed Search Change	Class codes Class codes Class codes Class codes	Mediator codes Mediator codes Mediator codes Class codes Mediator codes Component codes	- The gateway interface & code of that changed core-class Gateway interfaces Core-class interfaces & codes Gateway interfaces & codes

Figure 12: The Comparison of Evolution Operations

interactions are specified via interfaces rather than being embedded in class code alone. From the component evolution perspective, all interfaces are searched to discover all affected classes based on a single class change, thereby reducing the time complexity of the search space. The disadvantage of this approach is the introduction of new programming constructs that do not fit in with available programming interfaces and classes. For example, new mechanisms are needed to define connections between the required features of one interface and the provided features of another interface, and to allow a class to *use* the required features of its own interface. In contrast, while *CGC* offers this capability, it can be implemented within the bounds of current programming constructs. Moreover, *CGC* provides explicit dependencies at method-level via contracts.

Mediator Pattern (MP). Mediator Pattern [3, 7] is a behavior pattern that helps define the interaction between classes in a system and the system flow. It promotes loose coupling between classes in a component by introducing mediators to govern the interaction between classes. The mediator thus has knowledge about all classes in a component, while each individual class does not. From the component evolution perspective, all paths from classes to mediators and the corresponding code in a component must be searched to discover all affected classes based on a single class change. Thus, MP does not offer any reduction in the search space, rather it increases the search space.

7 Conclusion

In this paper, we focus on class dependencies which are critical for dynamic component evolution to further support component-based software evolution. A key aspect of component evolution revolves around the search for all affected classes based on a single change. Our goal is to reduce this search space by (1) isolating class behavior and class interaction, which are currently encapsulated by a traditional class, into *core-class* and *gateway* respectively; and (2) making class dependencies explicit. In addition, we introduce *contracts* that are embedded in both the core-class and the gateway, introducing explicit *method-level dependency*. By doing so, we implicitly introduce a hierarchy of class dependencies, *class-level and method-level*, that allow for more targeted, hence faster searches for all affected classes based on a single change. Another goal of our work is to achieve this new architecture within the frameworks of *current* programming languages. In this paper we have shown how this architecture can be implemented using current Java technology. In

addition, we have shown how a tool such as JML can be used to enforce the contracts from specification to implementation. Source code for all examples and a demonstration can be found at <http://www.cs.uml.edu/~ntansala>.

This paper offers the tip of the iceberg in terms of solving the problem of software evolution. Future work needs to focus on providing the taxonomy of changes, and needs to look at how these can be accomplished, as well as a tool to generate core-classes, gateways, and contracts on the fly to support programmer works based on the component design. Transparent evolution is another area that should also be investigated as an approach to support other components and applications during the evolution process.

References

- [1] J. Barker. *Beginning Java Objects*. Wrox Press Ltd, Arden House, 1102 Warwick Road, Acocks Green, Birmingham, UK, 2000.
- [2] G. T. H. Bill Councill. Definition of a software component and its elements. In *George T. Heineman and William T. Councill, Component-based Software Engineering*. Addison-Wesley Publishing Company, 2001.
- [3] J. W. Cooper. *Java Design Patterns : A Tutorial*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.
- [4] J. V. David C. Luckham and S. Meldal. Key concepts in architecture definition languages. In *Gary T. Leavens and Murali Sitaraman, Foundations of Component-based Systems*. Cambridge University Press, 2000.
- [5] H. M. Deitel and P. J. Deitel. *Java How to Program (4th Edition)*. Prentice Hall, Upper Saddle River, New Jersey, 2001.
- [6] H. M. Deitel, P. J. Deitel, and S. E. Santry. *Advanced Java 2 Platform -How to Program-*. Prentice Hall, Upper Saddle River, New Jersey, 2002.
- [7] J. V. Erich Gamma, Richard Helm and R. Johnson. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
- [8] M. Fowler and K. Scott. *UML Distilled Second Edition A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 2000.
- [9] G. T. Heineman and W. T. Councill. *Component-based Software Engineering*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.
- [10] G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary Design of JML*. Iowa State University-<http://www.cs.iastate.edu>, 2001.
- [11] G. T. Leavens and M. Sitaraman. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
- [12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, 1997.
- [13] S. N. Michael Hicks, Jonathan T. Moore. Dynamic software updating. In *Proceedings of the ACM SIGNPLAN'01 conference on Programming language design and implementation*. SIGPLAN : ACM Special Interest Group on Programming Languages, 2001.
- [14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on software engineering*. IEEE Computer Society, 1998.
- [15] D. E. Perry. Software evolution and 'light' semantics. In *Proceedings of the 21st international conference on software engineering*. IEEE Computer Society, 1999.
- [16] J. S. Rainer Weinreich. Component models and component services: Concept and principles. In *George T. Heineman and William T. Councill, Component-based Software Engineering*. Addison-Wesley Publishing Company, 2001.
- [17] A. Rausch. Software evolution in componentware using requirements/assurances contracts. In *Proceedings of the 22th international conference on software engineering*. IEEE Computer Society, 2000.
- [18] I. Sommerville. *Software Engineering 6th Edition*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.
- [19] C. Szyperski. Component software and the way ahead. In *Gary T. Leavens and Murali Sitaraman, Foundations of Component-based Systems*. Cambridge University Press, 2000.
- [20] M. Vigder. The evolution, maintenance, and management of component-based systems. In *George T. Heineman and William T. Councill, Component-based Software Engineering*. Addison-Wesley Publishing Company, 2001.