This paper has been accepted at the OOPSLA 2002 workshop on Behavioral Semantics.

Integrating Behavior Protocols in Enterprise Java Beans

Andrés Farías, Yann-Gaël Guéhéneuc,[*]
and Mario Südholt[†]

École des Mines de Nantes
4, rue Alfred Kastler – 44307, Nantes, France
{`afarias|guehene|sudholt`}`@emn.fr`

September 3, 2002

Abstract

Behavioral protocols have been proposed to enhance component-based systems by including sequencing constraints on component interactions in component interfaces. However, no existing component-based models provide support for behavioral protocols. In this paper, we discuss the integration of behavioral protocol in Sun's Enterprise JavaBeans (EJB) component model in three steps. First, we introduce the notion of coherence between behavioral protocols and component source code. Second, we discuss of the relations of behavioral protocols to the different interface-related concepts in EJB components (remote interface, deployment descriptor...). Third, we describe possibilities of automatic enforcement of behavioral protocols by means of automated extraction of protocols from components and verification of the notion of coherence against expected behavioral protocols.

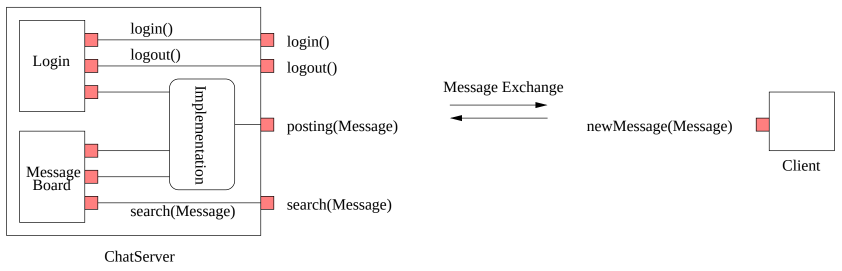
1 Introduction

Component-based programming facilitates the construction of large-scale applications through the composition of simple building blocks in complex applications. A fundamental notion of component-based programming is explicit interfaces. Interfaces impose strong restrictions on components: they make explicit all the means to use components, such as interaction and transfer of control interactions between components. In Sun's Enterprise JavaBeans (EJB) [2], for example, component (beans) interactions and how interactions are constrained are defined in form of several entities, among others a bean's remote interface and deployment descriptors. These entities essentially make explicit the types of the services, i.e., the methods, provided by a bean. More elaborate behavioral specifications are expressed using separate methodologies, such as the UML [10]. An important behavioral specification of components is sequencing constraints that components must obey when calling services of one another. Figure below presents the component-based client-server architecture of a chat server application for broadcasting messages among several clients. The `ChatServer` component offers services for clients to log in and to log out, to broadcast messages to all logged clients and to search for a posted message. The component `ChatServer` relays the services of its two collaborators `Login` and

[*]This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

[†]This work is partly funded by the EU project "EasyComp" (`www.easycomp.org`), no. IST-1999-014191

`MessageBoard` through its interface, e.g. `login()`, or uses them to implement its own services, e.g. `posting()`.

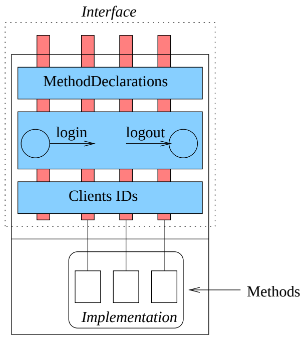


The availability of the services of the chat server component depends on its runtime state (e.g., the identities of logged in clients) as well as on the state of its clients. Messages, for example, can only be posted by clients who have previously logged in.

However, such sequencing constraints are not explicit in the interfaces of the components. Following work on object-oriented languages, the introduction of explicit protocols into component interfaces has been proposed [3, 8, 12, 13], essentially in the context of simple component models, which are rather small and simple, compared to commercial-strength industrial ones, like the EJB component model. In this paper, we investigate the feasibility and impact of the introduction of explicit protocols in interfaces of beans, where "interface" is understood as all EJB entities governing interactions of individual beans (i.e., including home and remote interfaces, deployment descriptors, and policy specifications, and at the different phase of the been life-cycle). We present three contributions. Most importantly, we discuss how sequencing constraints can be represented using beans interface entities, thus showing how protocols can be integrated in EJB interfaces. Furthermore, we develop a solution to the problem of coherence of interface-level protocols and the sequences of method calls executed at runtime by extraction of implementation-level protocols. We also sketch a prototype supporting such a notion for coherence.

The paper is structured as follows: after a brief introduction to on the notion of explicit protocols in components in Section 2, we introduce the notion of protocol coherence, in Section 3. In Section 4, we discuss the integration of explicit protocols in EJBs. Related work is discussed in Section 5. Conclusions and future work are given in Section 6.

2 Component model with explicit protocols



In this section, we present how protocols can be integrated into component models by revisiting such a model [3]. We consider components as software units providing an interface consisting of a set of method declarations, one protocol, and a set of lists of identities of collaborating components; lists that are used by the protocol to control the reception and sending of methods calls. Informally, the semantics of interfaces is the following: the method declarations define the services a component offers, the protocol defines sequences of possible interactions (receiving and sending ones) by means of transitions of a finite-state system, and the collaborator lists provide information to restrict protocol interactions based on component identities.

The figure on the right illustrates this structure for the chat server component introduced earlier. The provided services include method `login()`, the protocol includes a sequencing constraint between `login()` and `logout()`, and a collaborator list records logged-in clients.

In the component model introduced in [3], a component protocol, formalized in terms of a finite-state machine, describes sequencing constraints that specify the order in which services can be requested as well as constraints on the identity of the collaborators that interact with it. A transition is labeled with a method call, a direction specifying whether the method call comes from or goes to a collaborator, and an identity-constraint term restricting the identity of the caller or receiver. Figure 1 shows a protocol definition of a basic behavioral specification for the chat server component. The initial state *Stable* has three transitions representing services provided by the server (denoted by the direction '`-`') to add a client (`login()`), to remove a client (`logout()`), and to post a message (`post()`). The transitions labeled with identity-constraints record added and removed clients and thus ensure that messages are posted by clients that are currently logged in to all clients that have been added (and not removed). Once a client has successfully posted a message the protocol transits to the *Sending* state from which the server broadcasts a message to every client that are logged in using a `newMessage(Message)`. A transition label states the direction of interactions (a message sent: '`+`' or a message received: '`-`') and identity-constraint terms, which are lists of component identities that restricts the execution of a service only to collaborators whose identities are in the list denoted by the identity-constraint term. There are four kinds of identity-constraints terms for *adding* an identity of the component, which performs the request corresponding to the current identity-constraint term ($l+$), for *removing* an identity ($l-$), for *constraining* ($l!$) to identities in the list, and for *sending* the message to every identity in the constraint term ($l*$).

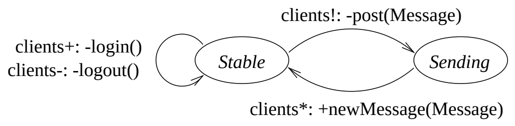


Figure 1: `ChatServer` protocol

We consider component composition as the basic relation that enables a component to use the services provided by another one. In the context of components with explicit protocols, component composition naturally involves composition of protocols. In [3] we propose several protocol-composition operators to support component composition that preserve a correctness property of substitutability by construction: the protocol resulting of a protocol operation can safely substitute its operands (see [3] for details).

3 Notion of coherence

We now direct the discussion towards the problem of coherence between a protocol and a component source code that rises when we expect that a component behaves as described by the protocol. Coherence between a component implementation and a protocol specification is important in that it allows static verification of the sequencing constraints specified in the protocol.

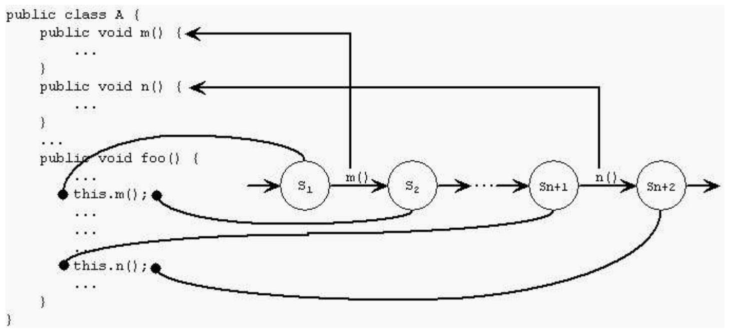
In the previous section, we showed how we associate a protocol with a component. We also showed that a component protocol specifies completely the behavior of its component. However, component protocol only represents the desired behavior of the component at the specification level. So far, nothing prevents the component implementation to behave, in facts, differently from its expected behavior, as defined by the component protocol.

We now present a model of implementation-level protocol and a notion of coherence between a protocol and a source code, which link a component protocol with a component implementation.

Then, we introduce four different operators on coherence verifications between a component protocol and a component implementation. Finally, we briefly describe a prototype tool, CwEP, that partially implements the model of equivalence and the coherence verifications.

3.1 Model of Implementation-level Protocol and Notion of Coherence

Using the model of protocols presented in Section 2, we call specification level protocol a protocol specifying the desired behavior a component, prior to the component implementation (such as the one presented in Figure 1). We call implementation level protocol a protocol extracted from a component source code and describing the *actual* behavior of the component. We extract an implementation level protocol by assimilating method declarations with transitions, and by matching states with the locations before and after method invocations. Figure 2 shows the correspondence between source code, method declarations and method invocations, and an implementation level protocol: a method invocation expression corresponds to a unique state of a protocol; a method declaration corresponds to several transitions in different protocols.



```
public class A {
    public void m() {
        ...
    }
    public void n() {
        ...
    }
    ...
    public void foo() {
        ...
        this.m();
        ...
        ...
        this.n();
        ...
    }
}
```

Figure 2: Correspondence between (a subset of) an implementation level protocol and the related method declarations and invocations.

We now introduce a notion of coherence between protocols at the specification-level and at the implementation-level: two protocols are coherent if and only if they define the same sequences of method invocations with the same collaborator identities, even if their structures differ. With this notion of coherence we can compare two protocols and verify their coherence.

3.2 On the Notion of Coherence

From the notion of coherence, it is possible to verify the coherence between a specification-level protocol and an implementation-level protocol, both dynamically and statically.

Dynamic verification of the coherence During the execution of a component-based application, the server, on which the components run, could check dynamically the coherence between the behavior of a component, given by its implementation-level protocol, and a given specification-level protocol, thus guaranteeing that the component does not violate its expected behavior, i.e., the expected sequences of method invocations. The dynamic verification of a component with a protocol allows to prevent a component from by-passing, for example, security checks, before it actually tries to do so.

Static verification of the coherence The static verification of the coherence allows to verify whether the implementation of a given component follows a desired specification-level protocol, through the implementation-level protocol of the component. The static verification of the coherence of a component with a protocol is especially important with COTS, because software engineers in charge of the integration must be certain that the COTS integrate with the rest of the application, i.e., that the COTS have the expected behavior and thus comply with the specifications-level protocols of the application.

From the notion of coherence, it is also possible to modify the source code of a component according to its associated protocol, either dynamically or statically.

Dynamic Modification of a Component Implementation. We could also dynamically adapt collaborating components to work together by generating automatically an adaptor required for a component to satisfy the protocols of its other collaborators. However, the automatic generation of adaptors would require precise theorems and proofs on our model of protocols and would have limitations, as [13].

Static Modification of a Component Implementation We also consider to modify the source code of a component in two steps, to add tests that check the identities defined by the design-level protocol, and to ensure that the method respect the required sequence of method invocations.

First, we assume that the component source code is already compatible with the sequence of methods invocations specified by the protocol; that is, the component source code is coherent *w.r.t.* the sequences of method invocations but (possibly) *not w.r.t.* the identities of the collaborators. From a transition labeled $x : +m()$ starting from a state s_i and leading to a state s_f, we can univocally determine the statement in the component source code that invokes method `m()`. Thus, we can insert a test for checking that x is effectively the receiver of the method invocation, just before the method invocation, at the point corresponding to state s_i. This check is required because the real value of a variable, which represents the receiver, can be dynamically reassigned. For example, in the following code:

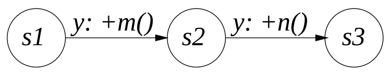
```
(new Point(1, 1)).moveTo(2, 3);
```

It is not possible to determine dynamically whether the receiver expression is the object corresponding to the one specified in the protocol. We introduce the check for verifying the identity of the receiver as follows:

```
if ((new Point(1, 1)).equals(collaborator)) {
    collaborator.moveTo(2, 3);
}
else {
    throw new UnknownReceiverException();
}
```

Second, we could modify a source code to make it compliant with a given design-level protocol *w.r.t.* the sequence of method invocations. The following example describe such a source code modification, based on the protocol shown on the right figure below and based on the following source code:

```
{
    y.m();
    z.a();
}
```



We assume that state s_1 is associated to the point before the evaluation of the `y.m();` statement, we can easily check that an extra method invocation is being performed instead of the second transition supposed to be executed at the state s_2. A straightforward solution consists in modifying the source code to remove the extra `z.a();` statement and to add the required `y.n();` statement. With a deeper analysis, we could anticipate the fact that statement `z.a();` corresponds to a transition forward in the given protocol and then must be kept.

3.3 Prototype-tool

We are currently developing a framework, the Component with Explicit Protocol[1] detector, generator, and validator (CwEP), that can extract implementation-level protocols from component source code and check their equivalence with given design-level protocols. Also, given a design-level protocol, our framework will generate the corresponding source code, and modify existing source code so that the it complies with the protocol.

CwEP is based on the compiler provided with the Eclipse framework [6]. Using the parser, we analyze source files and we build their corresponding abstract syntax tree (AST). We visit the different nodes of the AST and build the corresponding implementation-level protocol. Appendix A presents the source code implementing the `ChatServer` component. When analyzing this source code with CwEP, we obtain the implementation-level protocols of the `ChatServer` methods, depicted in Figure 3. The protocol of the `ChatServer` implementation corresponds to a protocol that follows iteratively any of its method protocols. We could modify the AST to insert method invocations and adds identity checks.

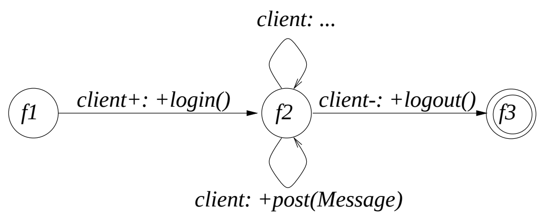


Figure 3: Extrated protocol of the Chat server component.

4 Explicit protocol in the EJB component model

Enterprise JavaBeans [2] is an industrial-strength component model, which supports the development, deployment, and management of transactional business systems, using distributed components implemented in Java. In the previous sections, we introduced the notions of component protocols and of coherence between protocols in a general component model. We now study the integration of these notions in the EJB component model.

First, we present the EJB component model. Second, we discuss the integration of protocols in the EJB component model at the specification level. In particular, we study how to include the notion of coherence in the EJB component model. Third, we discuss some implementation issues met when enhancing the EJB component model.

4.1 EJB component model

The definition of a bean is strongly bound to the Java environment. The structure of a bean, as shown in Figure 4, can be seen as being a set of classes, interfaces, and configuration files. A bean has two main interfaces. A remote interface describes the component functionality and a home interface specifies the methods controlling the life-cycle of the instances of the component. The structure of a bean also contains a deployment descriptor, an XML file containing non functional properties and definitions of the component environment.

[1]The framework is available at `http://www.emn.fr/farias`.

Non-functional properties allow configuration of three EJB services: transactions, persistence, and security. Modification of such descriptor file can change dramatically the behavior of a component without modifying the classes that define them.

The life-cycle of a bean has four main phases: development, where components are created; assembly, where components are composed to create more complex components; deployment, where components are configured to run in a particular environment; and execution, where components are running.

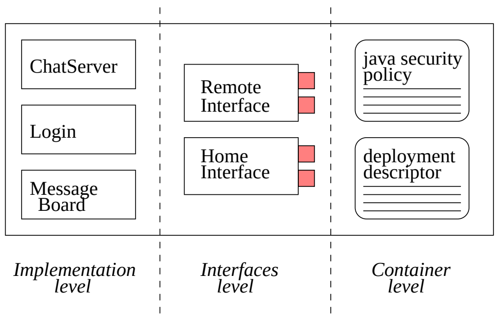


Figure 4: EJB component model.

An EJB application is a composition of beans join together via environment variables. The application is instantiated by a client that must create component instances to work with. Therefore, every other components instance are created indirectly when the instances are executed. The granularity of a component specification (via deployment descriptor) is only a bean.

4.2 On Component protocols and EJB component model specifications

The integration of protocol specifications can be performed during the different phases of a bean life-cycle and in the three different levels of the EJB component model: code source, interface, and descriptor files. This integration impact differently the whole life-cycle process, depending on when and where the specifications are integrated.

At the *development phase*, protocols can clearly specify constraints on components services, which are dependent with one another. At this phase of the life-cycle, protocols can be specified either at the source code level or at the interface level. As discussed in Section 3, protocols can be specified separately and then, by means of source code transformations, components are modified to follow the protocol specifications. Integrating protocols at the level of interfaces implies to change the traditional structure of the bean interface. Either way, the main advantage of specifying protocols at the development phase is that the specification remains separated from the implementation.

The distinction between remote and home interfaces can be used to separate the protocol in corresponding parts. A protocol specifying only services requests by collaborators (as the protocols of Nierstrasz [5]) can be integrated with the remote interface, while a protocol specifying the requests that the associated component makes to its collaborators can be specified in the home interface. For example, in the chat server application the remote interface can expose the `ChatServer` services and specifications about the `ChatServer` services availability, while the home interface can specify the methods modifying the life-cycle of the component instances as well as interactions (the services it requests) with its collaborators (`Login` and `MessageBoard` for example).

At the *assemby* phase, we think difficult to specify protocols for components being assembled. However, we can specify the way in which the respective protocols of the components will be composed. Protocol composition can thus be specified by means of protocol operators such as the union and concatenation. In the context of the chat server, we can imagine concatenating the `Login` component protocol with the `ChatServer` component protocol.

At the *deployment* phase, beans are now in binary form, so they cannot be modified. This means that protocols can only be specified at the container level where components are configured for deployment, by means of the deployment descriptor. The deployment descriptor plays a fundamental role in extending the container behavior, thus we can introduce protocol specifications in the container.

Finally, at the *execution* phase, protocol specifications related to running instances of components can take place in the execution environment (dynamic verification).

4.3 Implementation issues when integrating component protocols

Integrating protocol specifications at different levels of the EJB component model and in different life-cycle phases of the beans raises implementation issues, as well as commercial drawbacks.

Integration of protocol specifications at the phase of *development* by transforming the code source of components does not need to modify the semantics of Java or of the EJB server implementation, because the protocol specifications are separated from the components implementation. Protocol can still be specified at the *assembly* time by wrapping components and adding the corresponding glue code for integrating protocol specifications. Introducing protocol specifications at the *deployment* phase means modifying the container implementation or redefining the semantics of the home and remote interfaces. Either approach has for consequence that components are no longer compliant with the EJB specifications. At the *execution* phase, protocol specifications can only be integrated by modifying directly the EJB server and the deployment tools, which, once again, makes this approach not compliant with the EJB specifications.

The result of this discussion is that the later component protocols are integrated in the components life-cycle, the less compliant with the standard EJB specifications the integration is. On the one hand, integration at the source code-level is compliant with EJB specifications but does not let deployers to modify protocol specifications as when adding them at the deployment or execution phase. Integrating protocols at the deployment and execution phases let deployers add protocol specifications to COST components while the tradeoff is loosing compliance. Compatibility plays against the degree of integration.

5 Related Work

Components and protocols. Plazil *et al.* from the SOFA project at Charles University in Prague propose an enhanced architectural description language for component behavior with explicit protocols [8]. They have consider enhancing the EJB component model but no concrete work has been performed [4]. Yellin and Strom [13] also integrate protocol into components. Their work is genuine in that it considers automatic generation of adapter code among component protocols to satisfy compatibility property. However, they do not consider applying their research to real component models.

Separated component specifications. There are numerous approaches supporting specifications of component-related properties separated from the underlying component model. The UML [10], for example, has been applied to specify components. Similarly, Message Sequence Charts [9] is a trace language used to describe interactions among components based on finite-state systems. Architectural description languages, such as Wright [1], have also been used for similar purposes. However, none of these approaches has concretely try to enhance commercial component models and to analyze impact.

6 Conclusions and Future work

We presented a model of components with protocols. Then, we introduced a notion of coherence between specification-levels component protocols and implementation-level component protocols. We discussed the importance of structural entities in the EJB component model *w.r.t.* protocol integration. We also discussed the impact of integrating protocols specifications at the different levels of the beans.

We proposed a tool that helps in verifying and in integrating behavioral protocol specifications in the development phase of beans, while integrating protocols in the interface of beans still implies a strong loss of compatibility with the EJB component model specifications.

Future work. Next step is to experiment our theory by means of concrete modifications in open source EJB servers, such us JOnAS [7]. Also, work remains to enhance the EJB component model in such a way that the different roles distinguished by the EJB component model (developers and deployers) can extend specifications using our model of protocols.

References

[1] Robert J. Allen, *A formal approach to software architecture*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[2] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan, *Enterprise javabeans^{TM} specification*, SUN Microsystems, August 2001, Version 2.0, Final Release.

[3] Andrés Farias and Mario Südholt, *On components with explicit protocols satisfying a notion of correctness by construction*, DOA 2002, Distributed Objects and Applications, Lecture Notes in Computer Science, Springer-Verlag, October 2002, To appear.

[4] Vladímir Menel, Jiří Adámek, Adam Buble, Petr Hnetynka, and Stanislav Visnovsky, *Enhancing ejb component model*, Tech. Report 2001/7, Dep. of SW Engineering, Charles University, Prague, December 2001.

[5] Oscar Nierstrasz, *Regular types for active objects*, Object-Oriented Software Composition (Oscar Nierstrasz and Dennis Tsichritzis, eds.), Prentice-Hall, 1995, pp. 99–121.

[6] Object Technology International, Inc., *Eclipse platform – A universal tool platform*, July 2001.

[7] ObjectWeb, Open source middleware, *JOnAS: Java (tm) open application server*, 2002.

[8] Frantisek Plasil and Stanislav Visnovsky, *Behavior protocols for software components*, IEEE Transactions on Software Engineering, IEEE, January 2002.

[9] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski, *Tutorial on Message Sequence Charts*, Computer Networks and ISDN Systems **28** (1996), no. 12, 1629–1641.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The unified modeling language reference manual*, 1 ed., Addison-Wesley, Reading, Massachusetts, USA, 1999.

[11] Jan van den Bos and Chris Laffra, *PROCOL: A parallel object language with protocols*, OOPSLA'89, Conference Proceedings. Object-Oriented Programming, Systems, Languages and Applications (New Orleans, Louisiana, USA) (N. Meyrowitz, ed.), ACM SIGPLAN Notices, vol. 24(10), October 1989, pp. 95–102.

[12] Bart Wydaeghe, *PACOSUITE, component composition based on composition patterns and usage scenarios*, Ph.D. thesis, Vrije Universiteit Brussels, 2001.

[13] Daniel M. Yellin and Robert E. Strom, *Protocol specifications and component adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), no. 2, 292–333.

A ChatServer Source Code

```
package fr.emn.chat.server;

import java.beans.*;
import java.util.*;

public class ChatServer extends Thread {
    private Security aSecurityLibrary;
    private Vector clients;
    private Vector moderators;
    private Message currentNewMessage;

    public Server() {
        this.aSecurityLibrary = new Security();
        this.clients = new Vector();
        this.moderators = new Vector();
        this.currentNewMessage = null;
    }
    private void setCurrentNewMessage(
        Message aMessage)
        throws PropertyVetoException {

        PropertyChangeEvent newMessage =
            new PropertyChangeEvent(
                this,
                "currentNewMessage",
                this.currentNewMessage,
                aMessage);
        this.currentNewMessage = aMessage;
        this.firePropertyChange(newMessage);
    }
    private Message getCurrentNewMessage() {
        return this.currentNewMessage;
    }
    public void addVetoableChangeListener(
        VetoableChangeListener aModerator) {
        this.moderators.add(aModerator);
    }
    public void addPropertyChangeListener(
        PropertyChangeListener aClient) {
        this.clients.add(aClient);
    }
    public void removeVetoableChangeListener(
        VetoableChangeListener aClient) {
        this.clients.remove(aClient);
    }
    public void run() {
        System.out.println("Server running...");
        while (true) {
        }
    }
    public void post(Message aMessage)
        throws PropertyVetoException {

        System.out.println(
            "New Message received from: "
                + aMessage.getFrom()
                + " at "
                + aMessage.getDate().toString());
        System.out.println(aMessage.getValue());

        Iterator mIterator = moderators.iterator();

        while (mIterator.hasNext()) {
            Moderator m = (Moderator) mIterator.next();
            m.vetoableChange(
                new PropertyChangeEvent(
                    this,
                    "currentNewMessage",
                    this.currentNewMessage,
                    aMessage));
        }

        this.setCurrentNewMessage(aMessage);
    }
    public void login(User aUser, String password) {
        this.aSecurityLibrary.login(aUser, password);
    }
    public void logout(User aUser) {
        this.aSecurityLibrary.logout(aUser);
    }
    private void firePropertyChange(
        PropertyChangeEvent newMessage) {
        Iterator clientIterator = this.clients.iterator();

        while (clientIterator.hasNext()) {
            Client c = (Client) clientIterator.next();
            c.propertyChange(newMessage);
        }
    }
}
```