Generic Adaptation of Jini Services

Matthias Lampe, Egbert Althammer, Wolfgang Pree
Software Research Lab
University of Konstanz
D-78457 Konstanz, Germany
lampe@lampe.net, althammer@acm.org, pree@acm.org
www.SoftwareResearch.net

***Abstract*--In the network-centric computing paradigm the net gains more importance. Software and hardware components, often abstracted as services, form an application that is distributed over the network and only loosely coupled. Clients that depend on those services communicate remotely and download the necessary code they need.**

Jini is a framework, based on Java that provides an infrastructure for network-centric computing. A Jini environment is a collection of interacting Java programs, called services, which are distributed over a network, and which need to register themselves with the Jini lookup service (LUS). The LUS works like the yellow pages allowing clients to lookup services. An exact type match between the service interface registered with the LUS and the one requested by the client is needed for a lookup. Therefore a client cannot use a service integrated into the network, which does not match a known interface.

The adapter pattern applied to Jini services solves this problem by creating adapters for services unknown to the clients. An adapter implements the known interface making it accessible by the client and operating as a link between the client and the unknown service. Since manually creating these adapters would result in a redundant coding effort, we propose using a tool, called the Generic Adapter, which combines automatic adaptation and adaptation requiring user interaction as strategies to facilitate the adapter creation. An intuitive graphical user interface reduces the effort for the user to a minimum allowing drag-and-drop operations to define mapping relations and to simply add custom code. The Generic Adapter tool can be easily integrated into a network-centric architecture like Jini. Service adaptation is performed on the basis of Java Reflection, XML or Javadoc service descriptions, and the mapping relations, which are either automatically determined or specified by the user, and which are stored in a central mapping table.

***Index Terms*--Module adaptation, adapter design pattern, Java, network-centric computing, Jini, component based software, reflection.**

I. INTRODUCTION

A. *Network-Centric Computing*

Network-centric computing is the next step following client/server computing and disk centric computing [3, 9]. In general, the net gains more importance, for instance, software can be downloaded from central servers instead of being locally installed from a CD. The crucial point of network-centric computing, however, is that software and hardware components (often called services) that form an application are distributed over the network and are loosely coupled. Clients that depend on those services communicate remotely and download the necessary code they need.

To show the power of this approach, we can take a look on how an application connects to a printer. In the disk centric approach the printer would be installed locally and connected to the computer where the application runs. Thus each computer would need its own printer. In the client/server approach the server manages the single components such as the client computer and the printer. To install the printer the client would have to install the printer driver (for instance, by downloading it from the server) and would connect to the printer through the server. This approach is has the advantage that more clients can share one printer. The printer is still managed by the server. In the network-centric approach, the printers represented by printing services register themselves when they are available. In order to print out a document, the application searches the net to find the available printing services, connects to the one that fits and downloads the necessary drivers from the printer service directly. This is possible without pre-installing any other components. In case the printing service is not available any more, the application disconnects and connects to another printer.

An example domain where network-centric computing can be demonstrated is the so-called building control system. In buildings we find a lot of services and gadgets such as lighting and heating components, electronic equipment (coffee machines, washing machines, stereos), alarms and smoke detectors. These services are distributed and self-contained. However, the installation and management of services can be very expensive, especially in larger buildings. To reduce the effort a central management unit could be installed to which the services connect (like in the printer example) and offer their functionality. Services are managed and grouped by the central unit. There are several standards in Europe, USA and Japan for building control systems [6]. Both examples show the dynamic nature of network-centric computing. Services should be attached to and detached from the running system without interrupting other processes.

The nature of networks, the limited bandwidth, inherent latency and the possibility of partial failure also bring some challenges to software architectures. This paper first introduces Jini [2] as a network architecture based on Java that addresses these aspects. It then points out a fundamental problem that can occur while integrating new services – when interfaces do not match. Finally, the paper demonstrates service adaptation as a possible solution and presents a case study, which uses adaptation on top of Jini.

B. *Overview of the Jini Environment*

Jini is a framework, based on Java that provides an infrastructure for network-centric computing. A Jini system (also called Jini federation or Djinn) is a collection of interacting Java programs, the services, which are distributed over a network. At the core of the Jini infrastructure is the so-called Jini lookup service (LUS) where services are registered. The LUS can be compared to the yellow pages. It contains only the interface of the services and the information to find them.

All other components of Jini are services. A Service can be any piece of software or hardware. Each service needs to keep the LUS up to date by registering its interface, which contains the public methods offered to the world, and attributes, which contain extra information about the service, when they join the Djinn. This is accomplished by sending the service proxy to the LUS as shown in Figure 1a. To keep service information valid in a network environment with partial failures, the service registration has to be renewed after certain duration. Otherwise the service entry is automatically cancelled from the LUS.

To use a service, a client has to contact the LUS first. It performs a service request by sending a service template, which contains one or more Java interfaces and/or attributes. The LUS returns the service proxies whose interfaces (attributes) are type compatible to the service template, which can be seen in Figure 1b. Figure 1c shows how the client invokes the service through its proxy without using the Jini infrastructure.

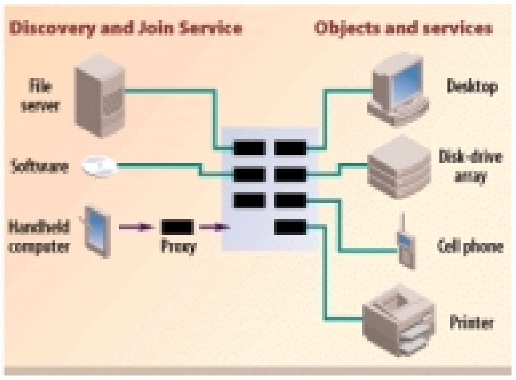


Figure 1a Djinn [10]

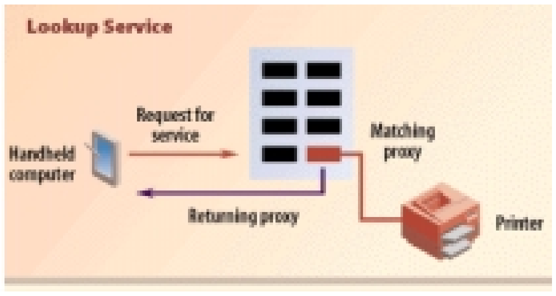


Figure 1b Service request [10]

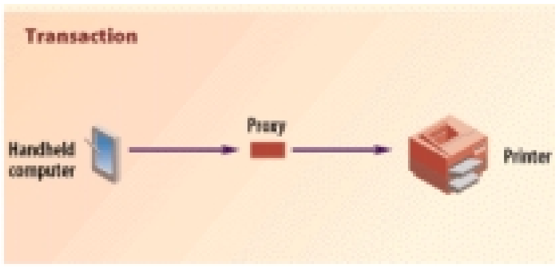


Figure 1c Service invocation [10]

Jini uses remote method invocation (RMI) as its remote communication protocol, thus Jini services are normally implemented in Java (at least Java 2 [1]) and have to be remote objects (throwing a RemoteException). If the service is implemented in a different language, it only has to offer a RMI proxy for the LUS. The communication protocol between proxy and service can be chosen arbitrarily.

C. *The Problem of Syntactically Unknown Or Incompatible Interfaces*

As mentioned, the Jini query model requires an exact type match between the service interface and the service template. This presumes that the client knows the syntax of the service interface. In other words, a client may not use a certain service because it does not know the syntax of its interface (i.e. its type). In the printer example this would be a printer with an incompatible interface. The following approaches attack this fundamental problem:

1. **Standardization of Interfaces:** To avoid this problem, clients and services need to be compliant to standards that specify how interfaces should look.
   Example: The Printing Service API [12].
   Advantage: Interfaces are known through standards and fit together.
   Disadvantage: It is difficult to establish one standard that covers all possible situations; normally there are deficiencies or multiple incompatible standards exist.
2. **Rewriting Services and Clients:** Services or clients are rewritten in order to cope with the counterpart.
   Advantage: Good performance through optimized interaction; is useful for special cases.

Disadvantage: Implementation effort, not possible when services and clients are black box implementations, which cannot be changed.

3. **Service Adaptation:** A connecting link is put between an incompatible client and service, which interprets the service interface and adapts it to the interface the client expects.
   Advantage: More elegant than the second approach because client and service remain unchanged and the adapter is flexible and adjustable to changes of interfaces. This is the only possible way for black-box services. Service adaptation is also used as a supplement to standardization (see 1).
   Disadvantage: Creating an adapter can be difficult depending on the interfaces and is not trivial. Performance could suffer by the indirection over the adapter.

Service adaptation relies on a semantic understanding of the service's interface, i.e. the functionality of the offered methods and values of attributes. This is done using Java reflection, the Javadoc [11] documentation or an XML-based [4, 13] service description.

Both approaches, the first one that relies on service standardization as well as the second one that requires a service description, have in common that they need the adaptation of components that do not exactly fit. Component-oriented systems solve this problem more or less elegantly through adapters as discussed in the Adapter design pattern [5]. As adaptation will become a frequent task in open distributed systems, an automation of this programming activity should be provided.

II. Concepts of Automating Service Adaptation

A. *Adaptation of Interfaces*

If classes cannot work together because of incompatible interfaces it is necessary to create an adapter, which converts the interface of a class into another interface expected by a client. This adapter pattern is described in [5] and shown in Figure 2.

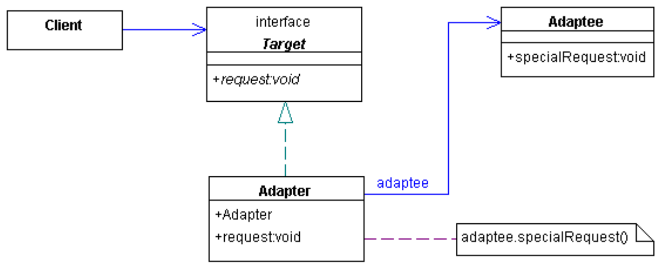


Figure 2 Structure of the Adapter Pattern [5]

The participants of the adapter pattern are the *Target*, which defines the domain-specific interface that the *Client* uses. The *Client* collaborates with objects conforming to the *Target* interface. The *Adapter* adapts the existing interface, *Adaptee*, to the *Target* interface. If the *Clients* call operations on an *Adapter* instance, they are handled by calls to *Adaptee* operations that carry out the request.

The basic structure and principle of the adapter is the same for various types of adapters, which can differ in the number of methods and the complexity of the adaptation of these methods. Manually implementing these adapters would result in an extra coding effort, which can be automated as facilitated as far as possible in systems where adaptation is often needed. The adaptation is defined through the mapping of the methods of the *Target* interface to the methods of the *Adaptee* interface. In this terminology the method *request*() of the *Target* would be mapped to the method *specialRequest*() of the *Adaptee*. All methods of the *Target* interface need to be mapped to methods of the *Adaptee* interface for the adapter to be complete.

The mapping can be classified in two main categories. The first is *automatic mapping*, i.e. the straightforward cases, which can be done automatically. However, most cases belong to the second category, *mapping requiring user interaction*, which need the user to define the mapping. The second category can again be divided into cases where the mapping can be defined using only a graphical user interface and cases where the user also needs to provide custom source code. The main assumption in the automatic mapping process is that methods with the same name provide the same functionality. This naming convention could be misleading in certain cases, which would require a user to verify the results with the help of the interface documentation. The two categories are:

1. Automatic Mapping
- o The name of an *Adaptee* and *Target* interface method and the parameter types and names are the same but the order of the parameters differs.
- o The name of an *Adaptee* and *Target* interface method and the parameter names are the same, but the parameter types differ. However, the parameter types of the *Target* interface method are subtypes of the corresponding parameters of the *Adaptee* interface method.
- o The name of an *Adaptee* and *Target* interface method is the same, but the return parameter type differs and the return parameter type of the *Adaptee* interface method is a subtype of the return parameter type of the *Target* interface method.

2. Mapping Requiring User Interaction

2.1. Definition of mapping relations through a user interface
- o The method name of the *Adaptee* and *Target* interface differs, but the parameter types, parameter names and the functionality are the same.
- o The parameter name of a non-primitive type, which corresponds in an *Adaptee* and *Target* interface method, differs but they are type compatible.

- A method of the *Target* interface cannot be mapped to a single method of the *Adaptee* interface, but to a series of methods of the *Adaptee* interface.

2.2. Providing custom source code to define mapping

- Two parameters in an *Adaptee* and *Target* interface method correspond to each other, but the parameter types are incompatible.
- The return types of two corresponding *Adaptee* and *Target* interface methods are incompatible.
- A method of the *Target* interface cannot be directly mapped to a method of the *Adaptee* interface.
- A parameter in a *Target* interface method cannot be mapped directly to a parameter in the corresponding *Adaptee* interface method.

B. Adaptation in the Context of Jini Services

In the Jini environment the general structure of the adapter pattern will be the same. The classes and interfaces in the pattern are only extended by the functionality necessary for classes to be clients and services in the Jini environment, including registering with the Jini LUS and performing lookups for services. Using the Adapter Pattern in the Jini environment also has the advantage that it allows the service, which implements the *Adaptee* interface, to change its implementation while keeping its interface without any changes to the *Adapter*, therefore supporting the loose coupling between the network components.

The following example from the domain of building control systems should demonstrate the advantage of adaptation in a Jini environment using a service that would be incompatible to an existing system without adaptation. The first interface, *Thermostat*, is the *Target* interface known to *Clients* in an existing building control system. However, there exists no service in the network implementing this interface. The only service providing similar functionality implements another interface called *TemperatureControl*, the *Adaptee* interface. Type incompatibility makes this service inaccessible to *Clients*, but a fairly simple adaptation allows integrating this service in the existing network. The methods of the two interfaces are shown in Figure 3.

Only the method *setSchedule()* of the interface *Thermostat* and the corresponding method *schedule()* of the interface *TemperatureControl* are explained in more detail to demonstrate the adaptation of these methods. The method *setSchedule()* schedules a rated temperature (*ratedTemp*) specified in Fahrenheit at the specified date (*date*). The rated temperature is the temperature the user wants to set. The service advises the connected gadgets to heat or cool in order to reach the temperature. The user can specify if this schedule should be done only once or repeated weekly (*weekly*).

The method *schedule()* offers nearly the same functionality, but has a different signature. Not only the method name differs, also the type of the date is *Date* and not *Calendar*, the second and third parameters are changed, the temperature is measured in Celsius and the repetition rate can be an arbitrary number of days specified as an integer value, where 0 means no repetition, i.e. a slightly extended functionality in respect to the first method.

To adapt the *TemperatureControl* interface to the *Thermostat* interface, the method *schedule()* has to be adapted to *setSchedule()*. In this case an adaptation strategy would be to adapt the three parameters in the implementation of *setSchedule()* to be able to invoke *schedule()*: the date parameter is converted from *Calendar* to *Date*, the second parameter is converted from Fahrenheit to Celsius and the third parameter is converted from *boolean* to *int* where *true* is converted to 7 (weekly) and *false* to 0 (once). The parameter conversions are realized as three separate methods, called custom methods, in the service adapter *TemperatureControlAdapter*.

The interaction between these components in the Jini environment is shown in Figure 4. The service implementing the *TemperatureControl* interface (*Adaptee*) first needs to register itself with the *Jini LUS*. The *TemperatureControlAdapter* (*Adapter*) performs a lookup for the *TemperatureControl* interface and receives the remote reference to the instance in the Jini environment, which allows the *TemperatureControlAdapter* to use remote method invocation of the adapted methods. In order to be accessible by clients in the network the

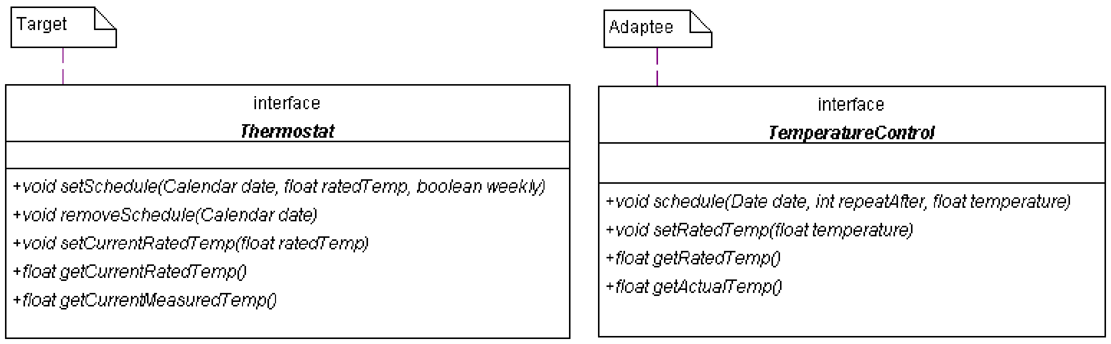

Target
interface
Thermostat
+void setSchedule(Calendar date, float ratedTemp, boolean weekly)
+void removeSchedule(Calendar date)
+void setCurrentRatedTemp(float ratedTemp)
+float getCurrentRatedTemp()
+float getCurrentMeasuredTemp()
Adaptee
interface
TemperatureControl
+void schedule(Date date, int repeatAfter, float temperature)
+void setRatedTemp(float temperature)
+float getRatedTemp()
+float getActualTemp()

Figure 3 Class diagram of the interfaces *Thermostat* and *TemperatureControl*

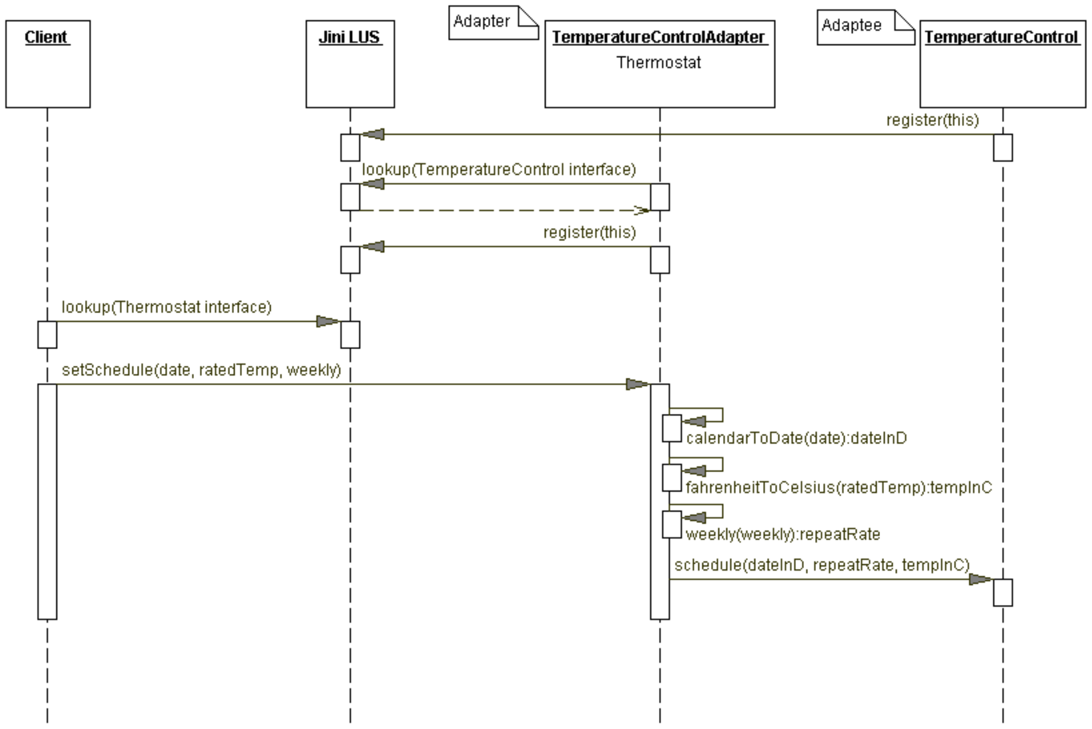

Adapter
Adaptee
Client
Jini LUS
TemperatureControlAdapter
Thermostat
TemperatureControl
register(this)
lookup(TemperatureControl interface)
register(this)
lookup(Thermostat interface)
setSchedule(date, ratedTemp, weekly)
calendarToDate(date):dateInD
fahrenheitToCelsius(ratedTemp):tempInC
weekly(weekly):repeatRate
schedule(dateInD, repeatRate, tempInC)

Figure 4 Functionality of the *TemperatureControlAdapter* in the Jini Environment

TemperatureControlAdapter registers itself with the *Jini LUS*. Since it implements the *Thermostat* interface (*Target), Clients* that perform a lookup for a service using the *Thermostat* interface will get a remote reference to the *TemperatureControlAdapter* instance from the *Jini LUS*. A remote method invocation of *setSchedule*() send to the *TemperatureControlAdapter* results in method calls to the custom methods *calendarToDate*(), *fahrenheitToCelsius*() and *weekly*(), and to a remote method invocation of the service method *schedule*() send to the *TemperatureControl* according to the Adapter Pattern.

III. Generic Service Adaptation Tool

The advantages of using adaptation in the Jini environment to allow unknown services to communicate with each other and the combination of *automatic adaptation* and *adaptation requiring user interaction* to facilitate creating adapters, resulted in the design and development of a tool called Generic Adapter, which integrates into the Jini environment allowing to create adapters for arbitrary service interfaces. Information necessary for the adaptation process is obtained from the interface definitions using Java Reflection. The relationships between the methods and parameters of the two interfaces are stored in a mapping table, which allows the generation of the adapter source code. The user interaction is supported through the Generic Adapter GUI that gives the user the possibility to use simple drag-and-drop operations and add custom code to specify the mapping relations.

A. Integration of the Generic Adapter Tool into the Jini Environment

The Generic Adapter is integrated into the Jini environment as a Jini service known to an extended LUS, which will launch the Generic Adapter if a service lookup fails, in order to allow adapting existing services. The sequence diagram in Figure 5 shows the interaction between *Client*, *Extended LUS*, and *GenericAdapter* and also the steps involved in creating the service adapter for the *TemperatureControl* service.

If the Client performs a lookup for the *Thermostat* interface, the *Extended LUS* will search for a matching service, but instead of returning a failure to the *Client*, since no registered service implements the given interface, it will invoke the *Generic Adapter*. The *Generic Adapter* will generate the service adapter with the help of the user and start it. The *Extended LUS* will return the remote reference of the service adapter adapting the *TemperatureControl* to the *Client*.

The steps in generating the service adapter are first to allow the user to choose a service to adapt from a list of registered services. Optional, the service with the highest probability for adaptation could be selected automatically. In this case the *TemperatureControl* service is chosen. Using the two interfaces *Thermostat* and *TemperatureControl* the service adapter is generated. Since full-automatic adaptation is not possible in this case, the Generic Adapter GUI is opened and the user can define the mapping between the two

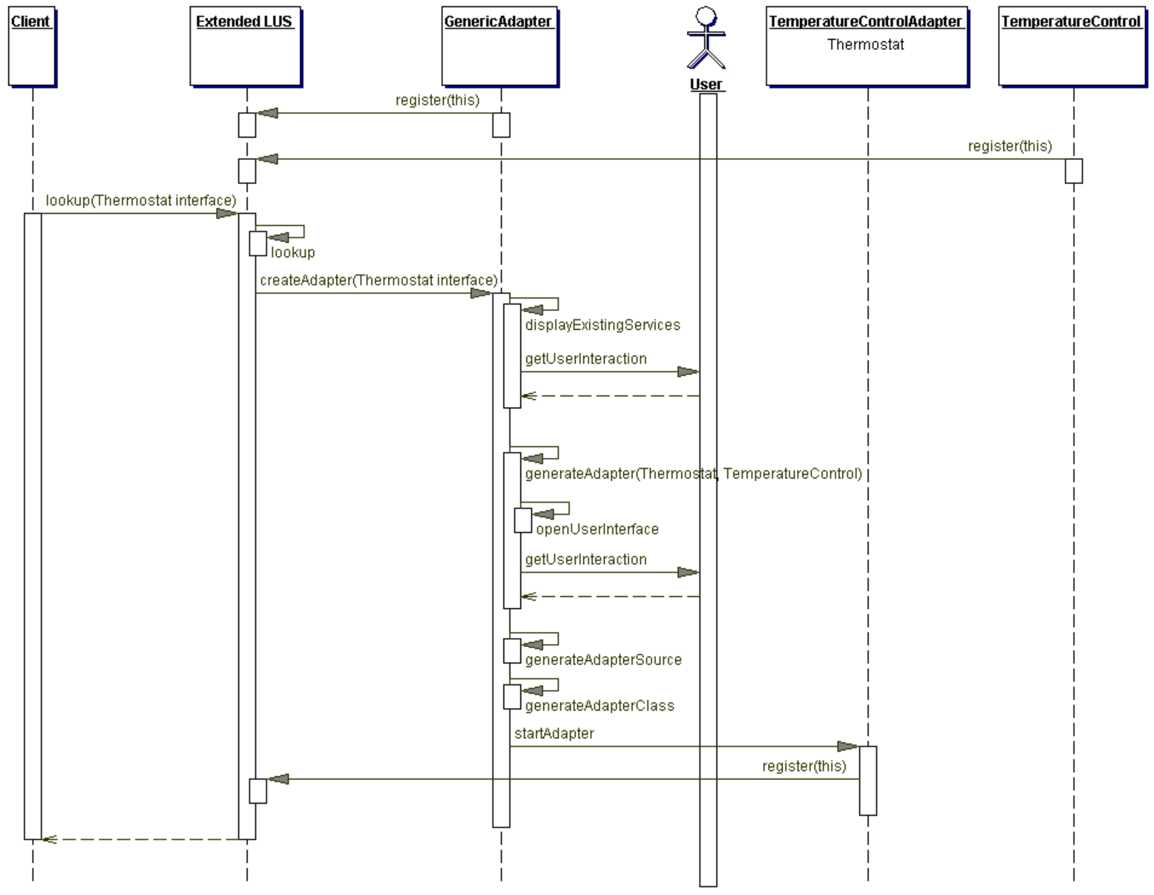


Figure 5 Generation of the *TemperatureControl* service adapter

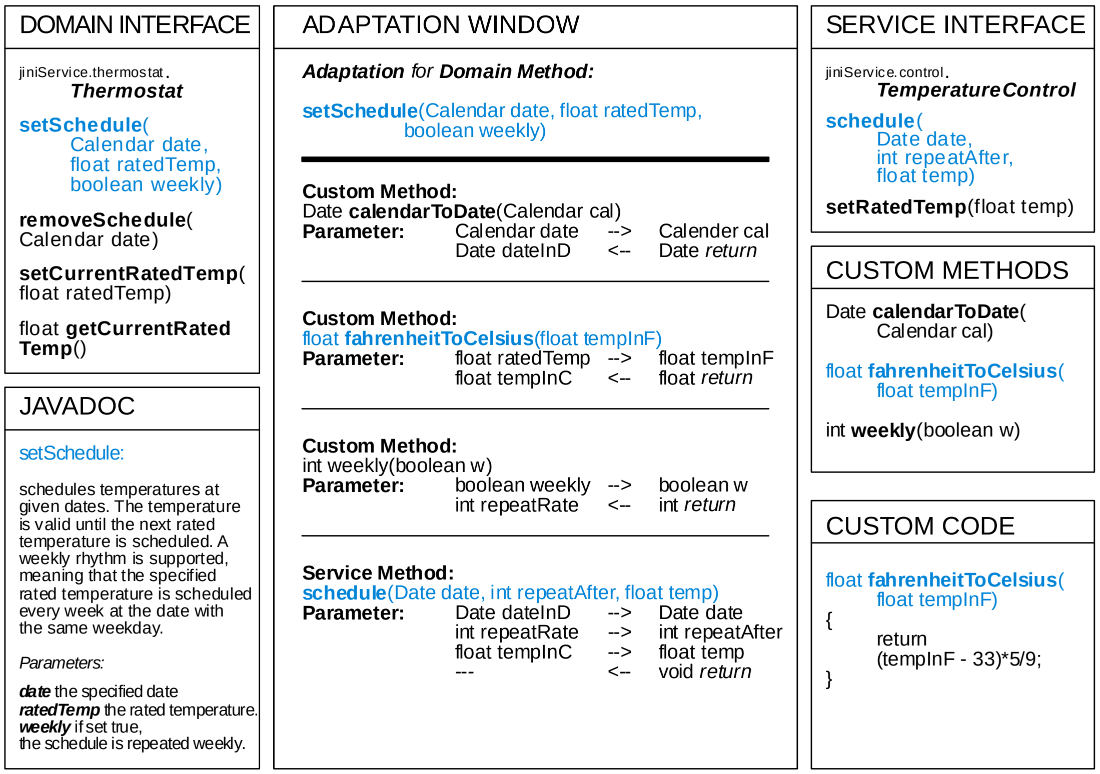
interfaces and add custom code to the adaptation. When the mapping definition is completed the source code for the adapter is generated, compiled into Java byte code, and the class file is dynamically loaded. The service adapter also includes special methods to handle the Jini specific tasks including registering and performing lookups. When started the service adapter registers itself with the *Extended LUS* in order to be integrated in the building control system.

B. Implementation Aspects of the Generic Adapter Tool

The information necessary to generate the service adapter is the interface definition of the Target and Adaptee interface, including the syntactic information. The interface definitions are loaded dynamically and examined using the Java Reflection API. Information about the interface methods including method names, parameter and return types can be obtained. In addition, the interface documentation, including the semantic information, is helpful for the user to determine valid mapping relations. The automatic mapping mentioned above, which is based on naming conventions also needs the parameter names of the interface methods, which cannot be obtained using Reflection. Using XML based interface documentation would provide both the interface documentation for the user and the parameter names needed for the automatic adaptation. As an alternative, the Javadoc interface documentation could be used by parsing it for the parameter names.

The core of the Generic Adapter is the mapping table [7], which stores the meta-information about the Target and Adaptee interfaces, and the mapping relations between the methods of the interfaces. These are not only one-to-one relationships, and often the sequence of the adapted methods is important. Additionally, custom code might be added. These facts make it important to keep a simple structure for the mapping definition. The Generic Adapter keeps for each Target interface method a sequence of Adaptee interface methods and custom methods, encapsulating the custom code. This sequence clearly defines the adaptation for the Target interface method. The mapping of the parameters of the Target interface method to parameters of the Adaptee and custom methods in the sequence is also defined in the method mapping. Return parameters of methods in the sequence are handled the same way.

The information stored in the mapping table is sufficient to generate the source code for the service adapter. It is important to test the completeness of the mapping between the two interfaces before generating the code. A source code template for the service adapter contains the Jini specific code, which is independent from the adapted



DOMAIN INTERFACE

jiniService.thermostat.
Thermostat

```
setSchedule(
        Calendar date,
        float ratedTemp,
        boolean weekly)

removeSchedule(
Calendar date)

setCurrentRatedTemp(
float ratedTemp)

float getCurrentRated
Temp()
```

JAVADOC

setSchedule:

schedules temperatures at given dates. The temperature is valid until the next rated temperature is scheduled. A weekly rhythm is supported, meaning that the specified rated temperature is scheduled every week at the date with the same weekday.

Parameters:

date the specified date
ratedTemp the rated temperature.
weekly if set true,
the schedule is repeated weekly.

ADAPTATION WINDOW

Adaptation** for **Domain Method:

```
setSchedule(Calendar date, float ratedTemp,
            boolean weekly)
```

Custom Method:
Date **calendarToDate**(Calendar cal)

Parameter:			
	Calendar date	-->	Calender cal
	Date dateInD	<--	Date *return*

Custom Method:
float **fahrenheitToCelsius**(float tempInF)

Parameter:			
	float ratedTemp	-->	float tempInF
	float tempInC	<--	float *return*

Custom Method:
int weekly(boolean w)

Parameter:			
	boolean weekly	-->	boolean w
	int repeatRate	<--	int *return*

Service Method:
schedule(Date date, int repeatAfter, float temp)

Parameter:			
	Date dateInD	-->	Date date
	int repeatRate	-->	int repeatAfter
	float tempInC	-->	float temp
	---	<--	void *return*

SERVICE INTERFACE

jiniService.control.
TemperatureControl

```
schedule(
        Date date,
        int repeatAfter,
        float temp)

setRatedTemp(float temp)
```

CUSTOM METHODS

```
Date calendarToDate(
        Calendar cal)

float fahrenheitToCelsius(
        float tempInF)

int weekly(boolean w)
```

CUSTOM CODE

```
float fahrenheitToCelsius(
        float tempInF)
{
        return
        (tempInF - 33)*5/9;
}
```

Figure 6 Design of the Interactive Adaptation GUI

interfaces. This template is filled with the information from the mapping table to complete the Java source file.

C. Interactive User Interface of the Generic Adapter Tool

In case the complete automatic mapping is not possible and *mapping with user interaction* is required, we specified, designed and implemented the user interface for the interactive adaptation of the Generic Adapter. Figure 6 shows the design including the *Thermostat* example explained above. The left part of the GUI shows the methods of the *Target* interface, here called *domain interface*. The right part shows the methods of the *Adaptee* interface, here called *service interface*, and the custom methods for the adaptation, which contain the custom code implemented by the user of the Generic Adapter. The middle part shows the mapping for one selected domain interface method in detail.

The mapping of the domain interface method is expressed as a sequence of method invocations, either service methods or custom methods. The service methods correspond to the methods of the service interface and the custom methods encapsulate source code which is not part of the service interface and which is either implemented manually or taken from a library. For example, custom methods are necessary to adapt parameters or to implement a different control flow. Each method invocation includes a parameter list, which shows how the parameters of the domain interface method are mapped to the parameters of the invoked service or custom method. Return parameters are mapped to special return variables, which can be used in further method calls in the method sequence. More detailed information about the selected methods is shown in the bottom left and right part. This information is extracted from the Javadoc or XML description of the interfaces.

In the example the domain method *setSchedule()* is selected. The window in the bottom left shows the Javadoc details about the method and the window in the middle the existing adaptation for *setSchedule()*. The adaptation sequence consists of four methods: custom method *calendarToDate()*, custom method *fahrenheitToCelsius()*, custom method *weekly()* and service method *schedule()*. The order of the methods in the adaptation window corresponds to the order of execution. In the GUI most actions are achieved with drag-and-drop operations, for example inserting a service or custom method into the adaptation sequence.

Results from one method serve as input for the ones underneath. This is realized with the so-called mapping parameters. For example, the first parameter *date* (of type *Calendar*) of the domain method *setSchedule()* serves as input parameter *cal* (of type *Calendar*) of the first method *calendarToDate()*. This so-called parameter mapping is realized by introducing an extra variable (we call it a mapping parameter) of type *Calendar* that saves the value. To map the parameters the user makes a drag and drop from the source parameter to the target parameter (in this case from *date* to *cal*). The generic adapter makes a plausibility check (the source must come before the target) and a type check (covariant and contra variant in the sense of [8]) and displays the type and the name of the mapping parameter in the left field besides the input parameter.

The generic adapter is designed in a way that it allows the user to automate the adaptation process. On the basis of the actual status of the adaptation process, the generic adapter will try to automatically complete the adaptation process by inserting the missing methods and parameters. If this is not possible it will try to generate the mapping with a highest probability (on the basis of naming conventions and type checks), which the user can change afterwards.

IV. Conclusion

Network-centric computing creates new situations of component interaction. One significant problem using typed languages like Java is the interface mismatch. Adaptation as the general idea solves these problems facilitating the component interaction in the network federation through adapters switched between incompatible interfaces. Since manually creating these adapters would result in a redundant coding effort, we propose using a tool, called the Generic Adapter, which combines automatic adaptation and adaptation requiring user interaction as strategies to facilitate the adapter creation. An intuitive graphical user interface reduces the effort for the user to a minimum allowing drag-and-drop operations to define mapping relations and to simply add custom code. The Generic Adapter tool can be easily integrated into a network-centric architecture like Jini. Service adaptation is performed on the basis of Java Reflection, XML or Javadoc service descriptions, and the mapping relations, which are either automatically determined or specified by the user, and which are stored in a central mapping table.

V. References

[1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Third Edition.* Addison-Wesley, 2000
[2] K. Arnold et al., *The Jini Specification.* Addison-Wesley, 2000
[3] B. Cole, *The Emergence of Net-Centric Computing: Network Computers, Internet Appliances, and Connected PCs*, Prentice Hall, 1998
[4] S. Czerwinski et al., *An Architecture for a Secure Service Discovery Service.* Mobicom ’99 Seattle Washington USA, 1999
[5] E. Gamma et al., *Design Patterns—Elements of Reusable Object-Oriented Software*, Reading, Addison-Wesley, Massachusetts, 1995
[6] M. Jedamzik, *Smart House*, http://ls7-www.informatik.uni-dortmund.de/research/gesture/argus/intelligent-home
[7] D. Rine, N. Nada, K. Jaber, *Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Design*, SSR ’99, Los Angeles, CA, 1999
[8] C. Szyperski, *Component Software*, Addison Wesley, 1997
[9] P. Varhol, *Netcentric computing*, Computer Technology Research, 1998
[10] Web reference: Infoworld. http://www.infoworld.com/pageone/gif/980928sb1-spotlight.gif
[11] Web reference: Javadoc. http://java.sun.com/products/jdk/javadoc/
[12] Web reference: Printing Service API, draft version. http://developer.jini.org
[13] Web reference: XML. http://www.w3.org/xml