T U M

I N S T I T U T F Ü R I N F O R M A T I K

AutoMate - From UML Models to Multi-Tier-Architectures

Klaus Bergner, Andreas Rausch, Marc Sihling

TUM-I0015
Oktober 00

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-10-I0015-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

AutoMate – From UML Models to Multi-Tier-Architectures

Klaus Bergner, Andreas Rausch, Marc Sihling

{bergner,rausch,sihling}@in.tum.de
Institut für Informatik
Technische Universität München
80290 Munich, Germany

Abstract

To create multi-tier architectures with a transparent access of distributed, persistant objects on the application layer, a technical solution is required which usually drastically complicates the overall implementation. In this paper, we present the tool AutoMate which generates on one hand large parts of the corresponding realization, releaving the developer from schematic code. On the other hand, AutoMate provides a convenient framework to manage the application´s business objects in an elegant, comprehensible way. This way, the overall time and cost of the development process is noticeably reduced.

Keywords

Multi-tier architecture, Client/Server, Codegeneration, OODBMS, CORBA

1 Introduction

Multi-tier architectures have grown to be standard solutions in various application domains in the last couple of years. For example, the three-tier architecture consisting of the presentation layer, the application layer and the database layer is already called the “classic” solution for all kind of information systems [5]. The success and acceptance of this kind of architecture has been especially stressed by the support of standardized interfaces as, for instance, to the respective database.

Application development in the object-oriented world is technically rather straightforward and additionally well supported by tools in the presentation and application layer. Various CASE tools help in elaborating analysis and design of the system model, graphical tools offer support for the design of the user interface, and most development environments keep a stock of standardized, prefabricated components which are easy to be adapted to the given application domain. However, development of the server side is almost always hand-made although it often brings along the same set of technical problems:

Most applications require some kind of communication middleware. For example, CORBA-based object request brokers allow for cooperation within a single layer or also between several layers of the architecture. This way, method calls are transparently forwarded to the called object eventually switching to other processes or computers. However, modern middleware technologies drastically complicate development as they require, for example, a plethora of additional helper objects like stubs and skeletons. Similarly, the actual transparent usage of persistent objects stored within the database is preceded by the rather complicated development of a custom-made solution for persistence.
By and large, the implementation of transparent access from "everywhere" to persistent objects of the application layer still lacks standard solutions and thus renders development of the server side unnecessarily complicated. This is especially due to the tricky interplay of application objects, middleware, and the underlying database.
In this paper, we present the tool AutoMate which currently offers transparent access to object-oriented Java databases via CORBA. Its special feature: the server code of a distributed three-tier architecture can be generated automatically from UML class diagrams, thus greatly reducing the required programming effort.
AutoMate builds on established commercial components. In its current version, the application layer relies on the CORBA object request broker Iona OrbixWeb [2]. The database layer uses the object-oriented database system Versant [7]. Clients in the presentation layer access Java objects on the application layer via standardized, easy-to-use interfaces, while the data is stored on the database layer transparently. The resulting three-tier architecture is very flexible with respect to the distribution of data and functionality, making it especially suitable for large enterprise applications like computer-aided engineering environments.
The software consists of two parts: a generator for the object model of the CORBA Java object database, and a set of manager components in the application layer of the server. The object model generator is integrated within the CASE tool Rational Rose [4]. It uses UML class diagrams to generate the Java code for the server objects, and starts up the application and database servers. Implementations for server-side methods on the application layer may then be added to the server at runtime. With respect to the manager components, the current version of AutoMate contains components for object creation, naming, queries, and transaction management. Additional functionality like schema evolution, version management, and object migration, as well as support for other CORBA object request brokers, object-oriented database systems, and CASE tools is considered in future versions.
In the following, we present our vision of the overall architecture of computer-aided engineering environments. We show how a tool like AutoMate can help software engineers in developing such systems, and discuss the basic concepts and design decisions of AutoMate. Finally, we demonstrate the usage of AutoMate and present a small application example.

2 Architecture Overview

The overall architecture of an AutoMate system is shown in Figure 1. It consists of the following three tiers:

- The database layer is responsible for storing and accessing persistent Java objects. The interface of the layer is currently based on the ODMG 2.0 standard [1]. Thus, different

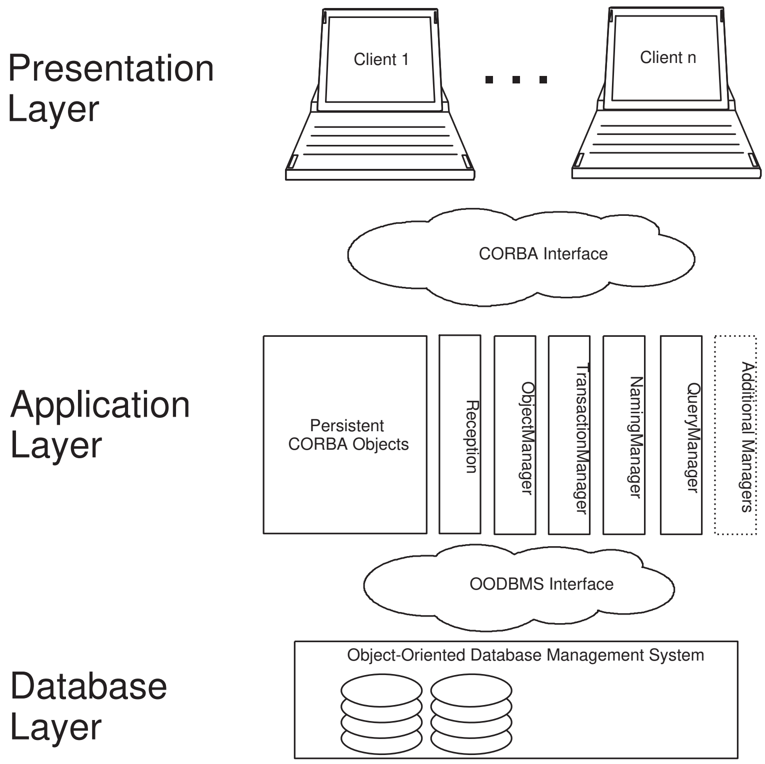

Figure 1: Overall Architecture of an AutoMate System

even distributed databases can be integrated behind this layer. However, the current version of AutoMate is based on the Versant OODBMS.

- The application layer uses Iona's CORBA ORB OrbixWeb. On the one hand, it provides CORBA interfaces to the system's base functionality, represented by a set of manager objects. On the other hand, it makes the persistent Java objects in the database layer accessible via CORBA. The whole application layer can be distributed as each manager and each persistent CORBA object may be a CORBA server itself.

- The clients in the presentation layer access the application layer via portable CORBA operation calls.

The IDL interfaces to the persistent CORBA database objects are described further in Section 3. Basically, they offer support for accessing the internal state of the object attributes as well as functionality for navigating between objects. The current version does also support the execution of additional, user-defined Java code in this layer.
The IDL interfaces to the manager objects are explained in Section 5. They offer support for object creation and destruction, naming and lookup of named objects, and transaction management. We also offer support for querying objects by means of standard OQL queries.
Note that the application and database layers may be fully generated from an UML model created with Rational Rose. Custom programming is only necessary to implement the clients in the presentation layer. In the future, we will provide support for the implementation of clients, for example, by providing a mechanism for event notification between the server and the clients.

Other planned extensions are concerned with scalability improvements for large applications, for example, by means of client-side caching, support for activation and passivation, and resource pooling.

3 Modeling Techniques and Generated IDL Interfaces

AutoMate fully automates the creation and startup of a CORBA object database server. Before we go into details about the generation itself and the technical infrastructure we first explain how UML class diagrams modeled in Rational Rose are used to generate IDL interfaces for accessing remote persistent CORBA objects. The next section will show how a running server may be generated and started.
Currently, AutoMate only relies on UML class diagrams for the generation of a CORBA object database server. In the following, the supported modeling elements are explained. For each modeling element, the IDL interfaces concerned with accessing modeled objects on the server are given. Note that the current version of AutoMate offers only simplistic, minimal CORBA access interfaces for objects. Although this is already sufficient for a wide range of applications, future versions will provide more comfortable interfaces with added functionality and support for additional modeling concepts.

3.1 Classes and Packages

To generate support for a certain class within the CORBA object database, the persistency flag for this class has to be set. This flag can be found in the "Detail" tab in the "Class Specification" dialog, as shown in 2.

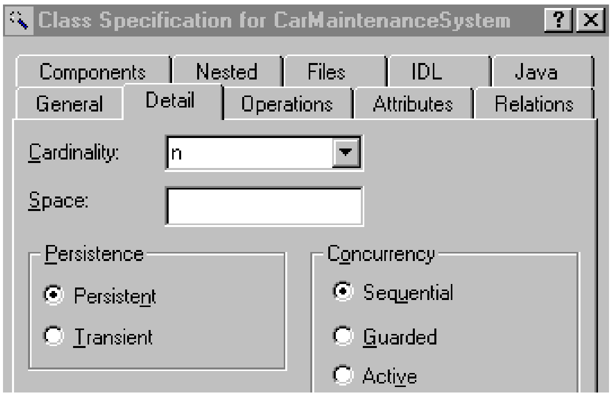

Figure 2: Marking a Class as Persistent

Classes that are not marked as persistent will be treated as transient. This means identical interfaces are generated for this classes, but they are never made persistent. Thus, programmers can use persistent and transient classes in an identical manner — they are also part of the transactional context.
Classes belong to Rose packages. Each Rose package maps to a corresponding IDL module on the client, and to a corresponding Java package on the server. Note that each class for a persistent CORBA object has to be contained in at least one package.

All generated interfaces extend the (empty) IDL interface **de::tum::automate::core::Element** which serves as a common base type for all persistent objects. The implementation of this basic class provides the hooks for the manager framework, like the transaction or query manager:

```
module server
{
  interface Car : de::tum::automate::core::Element
  {
    // generated attribute access

    // generated associations and  navigations

    // generated user-defined methods
  };
};
```

3.2 Enums

In contrast to the object models of C++ and CORBA, the object models of Rational Rose and UML do not include the concept of an enum, that is, a type whose instances are attributes with a restricted number of possible, specified values. Do not confuse user-defined enums with Java enumerations used to iterate over the objects in a container. (cf. Section 3.4).
In AutoMate, support for enums is provided with the help of the special stereotype `<<Enum>>`. To create an enum, you have to add the stereotype `<<Enum>>` to a Rose class. This can be done on the "General" tab in the "Class Specification" dialog, as shown in Figure 3.

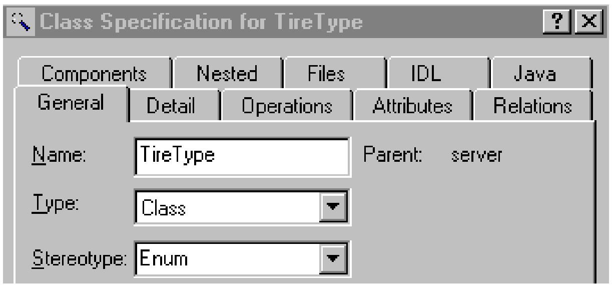

Figure 3: Creating an Enum

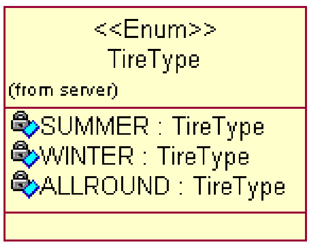

Figure 4: Specifying the Possible Values of an Enum

The possible values of an enum are specified as attributes of the same type as the enum itself. An example can be seen in Figure 4. The IDL interface generated from this example is

```
enum TireType { SUMMER, WINTER, ALLROUND };
```

3.3 Attributes

Attributes can be of the base types: `short`, `float`, `double`, `char`, `boolean`, `int`, `long`, and `java.lang.String` or they may be user-defined enums, or references to user-defined persistent CORBA objects. For an attribute `x` of type `A`, two access methods are generated in the IDL interface:

```
A getX();
void setX(in A);
```

3.4 Associations and Aggregations

The current version of AutoMate treats aggregations as simple associations and does not generate special code for aggregations. All AutoMate associations are bidirectional.

Multiplicities: For each association or aggregation between persistent CORBA objects, the multiplicity for both directions has to be specified. The current version of AutoMate only supports the following multiplicities:

```
0..1-to-0..1
0..1-to-*
```

Multiplicities of exactly `1` (like, for example, in `1-to-0..1`) may be specified, but are treated as `0..1`.
AutoMate does not include the concept of a `*-to-*` association. If such associations are needed, they must be broken up into two `0..1-to-*` associations.

Iteration via Enumerations: When an `0..1-to-*` association is used, a single object may be associated with arbitrarily many other objects. For accessing these objects, we provide typed iterators. They are designed following the standard Java `Enumeration` class. Note that enumerations are not related to CORBA enums (cf. Section 3.2).
For each class `A`, a corresponding IDL enumeration interface is generated as follows:

```
interface AEnumeration
{
   boolean hasMoreElements();
   A nextElement();
};
```

Navigating, Creating, and Removing Associations: The generated access operations in the IDL interfaces obey the following two rules, iff there are no name conflicts with other access operations:

- On the `x` side of a `0..1-to-x` association (with `x` being `0..1` or `*`) between two classes `A` and `B`, two access operations

  ```
  A getA();
  void setA(in A);
  ```

  are generated in `B`'s interface.

- On the `x` side of an `x-to-*` association (with `x` being `0..1` or `*`) between two classes `A` and `B`, two access operations

  ```
  BEnumeration getBs();
  void addB(in B);
  ```

  are generated in `A`'s interface.

If this transformation would result in name conflicts with other access operations, role names for the association endpoints have to be used in modeling. The access operations are then generated based on these role names instead of the class names. Name conflicts generally arise only when there are two or more associations between two classes or when there is a `0..1-to-0..1` association from a class to itself. If role names are used, the generated access operations in the IDL interfaces obey the following two rules:

- On the `x` side of a `0..1-to-x` association (with `x` being `0..1` or `*`) between two classes `A` and `B` with the role name `R` on the `A` side, two access operations

  ```
  R getR();
  void setR(in A);
  ```

  are generated in `B`'s interface.

- On the `x` side of an `x-to-*` association (with `x` being `0..1` or `*`) between two classes `A` and `B` with the role name `R` on the `B` side, two access operations

  ```
  BEnumeration getRs();
  void addR(in B);
  ```

  are generated in `A`'s interface.

Note that the current, minimal IDL interfaces contain no operations for removing association instances. To remove an `x-to-0..1` association link between objects of classes `X` and `Y`, the method `setX` with parameter `null` has to be called on the instance of `Y` which wants to be excluded.

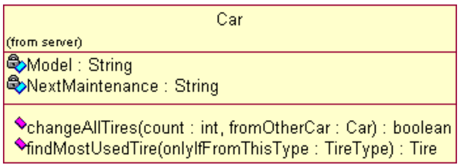

Figure 5: Specification of Server-Side Methods

3.5 Server-Side Methods

AutoMate allows the execution of Java code on the application server layer. Methods must be specified with the `public` visibility in Rational Rose, as shown in Figure 5.
The code of server-side methods is not specified in Rational Rose, but has to be written in special Java files, located in the `servermethods` directory below the working directory with the corresponding Rose model file.
The subdirectory structure within `servermethods` corresponds to the package structure of the project. The code for the example class `server.Car` of Figure 5 is, therefore, located in the file `servermethods\server\Carmethods.java`.
Server-side methods are specified in a special class with the suffix `Methods`. This class must extend the corresponding persistent server class with suffix `ImplPers`. For the example of Figure 5, the class definition looks like this:

```
package server;

import de.tum.automate.manager.*;
import de.tum.automate.manager.impl.*;
import de.tum.automate.core.*;
import de.tum.automate.core.impl.*;

public class CarMethods extends server.CarImplPers
{
   ...
}
```

Server-side methods should only rely on server-side classes and other server-side methods, as programmers should not see and use the types of the AutoMate framework. Instead of the CORBA method and class names specified in Rational Rose, their server-side counterparts should be used, therefore.
The code for the server-side method

```
  public changeAllTires(count : int, fromOtherCar : Car) : boolean
```

of Figure 5 may be as follows

```
public boolean changeAllTiresImpl(int count, CarImplPers fromOtherCar)
{
 if (getImplModel() == fromOtherCar.getImplModel())
 {
```

```
  TireEnumerationImpl myTires = getImplTires();
  TireEnumerationImpl otherTires = fromOtherCar.getImplTires();
   while(myTires.hasMoreElementsImpl() &&
         otherTires.hasMoreElementsImpl())
   {
    TireImplPers myTire = myTires.nextElementImpl();
    TireImplPers otherTire = otherTires.nextElementImpl();
    myTire.setImplAbrasion(otherTire.getImplAbrasion());
   }
  return true;
 } else {
  return false;
 }
}
```

Note that attribute calls have to be performed via calls to the corresponding `getImpl` and `setImpl` methods. Furthermore, an AutoMate server may be started without all server-side method Java files present. Calls to such server-side methods return default values, and are logged in the log file. This way, the tool can be used very easily for prototyping and incremental development.

4 Creating the Server

Generating a server requires that the OrbixWeb Java Daemon is running. All remaining activities are fully automated and can be selected and started from the AutoMate main dialog window, which may be opened in Rose via the menu bar. Once the generation is started the following steps are executed:

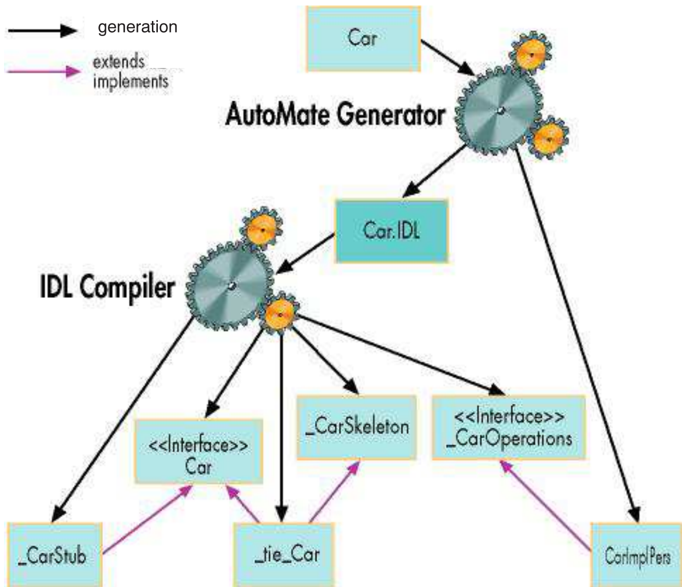

Figure 6: Overview of an exemplary generation process

1. Generate, Compile, and Enhance
   This step relates to the generation and preparation of the IDL files and the classes of the persistent CORBA objects on the server.
2. New Database
   In this step, a new Versant Java database is created and its schema is populated with the Java classes generated in the previous step.
3. CORBA Startup
   In this step, OrbixWeb is initialized with the generated IDL interfaces, connected with the Versant database, and made available as a new server in the network.

Figure 6 shows all files that have been generated for the example class `Car`. The interface `Car` represents the CORBA client interface. The stub and skeleton classes `CarStub`, `_tie_Car`, `_CarSkeleton`) are usually transparent to programmers. The persistent class `CarImplPers` has the same operations as the stub and skeleton classes, but implements a different interface, the interface `CarOperations`. Thus, CORBA objects and database objects are clearly separated.

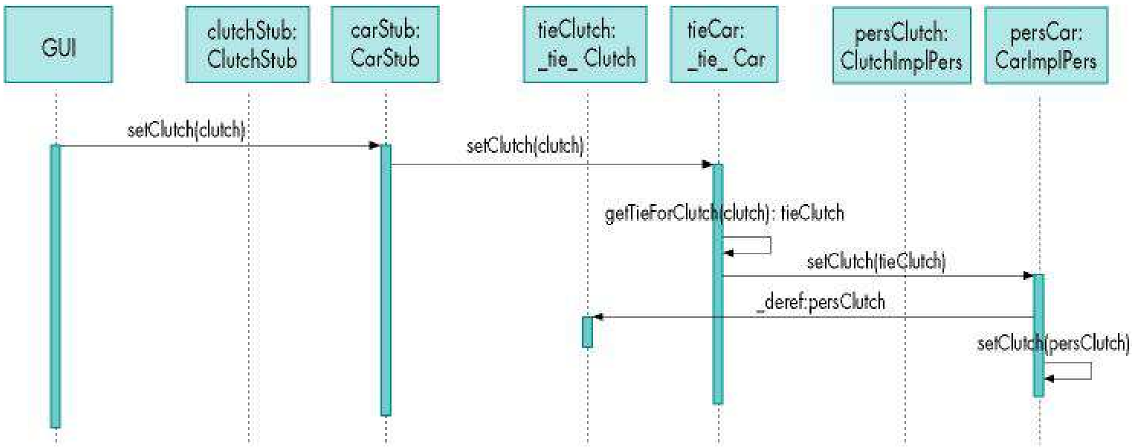

Figure 7: Examplary interplay of the generated classes

Finally, if all the files are generated and compiled and if the database and CORBA have been properly started, programmers can use the `Car` CORBA object. Figure 7 shows the appearing interactions if the method `setClutch` is called via CORBA on a `Car` object.

5 Services and Managers

To access the persistent CORBA objects living in an AutoMate object database, clients need to access so-called manager objects providing base functionality. In contrast to the persistent objects living on the AutoMate server, which may be created and deleted dynamically, the manager objects are part of the static architecture of AutoMate (cf. Section 2). All managers are so-called singleton objects—there exists only a single instance of each manager class.
The IDL interfaces of all managers are defined in the module `de::tum::automate::manager` provided in the file `Manager.idl`. This file includes the `Core.idl` file where the `Element` interface (cf. Section 3.1) and a basic exception interface named `BasicException` are defined. Almost all interfaces follow the corresponding standards of the OMG interfaces (c.f. [3]).

5.1 Transactions

The Transaction context interface offers the three standard transaction operations: The begin operation starts a new transaction, the commit operation tries to perform the changes made within the transaction, and the abort operation takes back all those changes. All three operations may result in exceptions.
It is possible to begin and end multiple transactions within one transaction context by calling the begin and commit operations repeatedly on a single Transaction object. However, each begin must be followed by a commit or abort. Calling begin twice in a row results in an exception.

```
interface Transaction {
  void begin() raises
    (de::tum::automate::
       core::BasicException);
  void commit() raises
    (de::tum::automate::
       core::BasicException);
  void abort() raises
    (de::tum::automate::
       core::BasicException);
};
```

Currently, optimistic as well as pessimistic transaction logic is supported. Database objects may be locked according to four different strategies: exclusive, read/write, read, and not repeatable read. This allows the programmer to optimize the strategy used in AutoMate.
The TransactionManager interface (cf. Figure 8) provides support for creating and deleting transaction contexts via the open and close operations. Note that these two operations are rather heavyweight compared to the transaction operations of Section 5.1.
In the current version of AutoMate, CORBA object references to objects of type Element are only valid within a single transaction (starting with a begin and ending with a commit or abort). If a durable object reference is needed, it can be acquired by calling the objectToHandle operation on an arbitrary persistent Element. This results in a portable string that may be stored persistently on the client, for example. The converse operation handleToObject converts this string to an object reference again.

5.2 Naming

The naming functionality (cf. Figure 9) is used to assign (or bind) string names to database objects. Names may be unassigned using the unbind operation. The lookup operation returns the database object assigned with a certain name.
Note that the bind operation does not need a Transaction parameter because its Element parameter is already connected with a transaction context.
Objects of type Element returned by the lookup method have to be cast to a more specific CORBA type before their specific methods can be accessed. This can be done with the help of the narrow method of the corresponding Helper interface, as demonstrated in the following client code example:

```
interface TransactionManager {

  Transaction open(in string dbName, in Locking locking,
    in TransactionLogic transactionLogic)
    raises (de::tum::automate::core::BasicException);

  void close(in Transaction transaction)
    raises (de::tum::automate::core::BasicException);

  string objectToHandle
    (in de::tum::automate::core::Element element)
    raises (de::tum::automate::core::BasicException);

  de::tum::automate::core::Element handleToObject
    (in Transaction transaction, in string handle)
    raises (de::tum::automate::core::BasicException);
};
```

Figure 8: Interface of the transaction manager

```
interface NamingManager {

  void bind
    (in de::tum::automate::core::Element element, in string name)
    raises (de::tum::automate::core::BasicException);

  void unbind
    (in Transaction transaction, in string name)
    raises (de::tum::automate::core::BasicException);

  de::tum::automate::core::Element lookup
    (in Transaction transaction, in string name)
    raises (de::tum::automate::core::BasicException);
};
```

Figure 9: Interface of the naming manager

```
interface ObjectManager {
  de::tum::automate::core::Element create
    (in Transaction transaction, in string objectType)
    raises (de::tum::automate::core::BasicException);
  void delete(in de::tum::automate::core::Element element)
    raises (de::tum::automate::core::BasicException);
};
```

Figure 10: Interface of the object manager

```
  Car carModel = CarHelper.narrow(namingManager.lookup(transaction,
                                "Donatas 2000 GLX"));
```

5.3 Queries

Apart from finding existing objects via the `lookup` operation of the naming manager, OQL queries may be used. The syntax of these queries follows the Versant variant of OQL as described in the Versant manuals.

```
interface QueryManager {

  de::tum::automate::core::ElementEnumeration query
    (in Transaction transaction, in string query)
     raises (de::tum::automate::core::BasicException);
};
```

The query manager returns an enumeration of generic `Element` CORBA objects. Analogous to objects returned by the `lookup` method of the `NamingManager`, the objects in the enumeration have to be cast to a more specific CORBA type before their specific methods can be used.
Furthermore, that queries rely on persistent server classes, and not on CORBA interfaces. Given the example class of Figure 5 on page 8, a possible query string could be

```
  select selfoid from server.CarImplPers where _Model == "Beetle"
```

5.4 Object Creation and Destruction

New persistent objects on the AutoMate server may be created by calling the generic factory method `create` of the AutoMate `ObjectManager`. On creation, the new object has to be connected with a transaction context. The `objectType` string parameter must contain a full, qualified Java class name with the package qualifiers according to the corresponding UML packages and IDL modules (an example is the string `"server.Car"`, which denotes class `Car` within module `server`). If one wants to remove a persistent object from the server AutoMate provides the `delete` operation.

```
interface Reception {
  TransactionManager getTransactionManager()
    raises (de::tum::automate::core::BasicException);
  ObjectManager getObjectManager()
    raises (de::tum::automate::core::BasicException);
  NamingManager getNamingManager()
    raises (de::tum::automate::core::BasicException);
  QueryManager getQueryManager()
    raises (de::tum::automate::core::BasicException);
};
```

Figure 11: Interface of the reception manager

5.5 Reception

The `Reception` object (cf. Figure 11) serves as central entry point to the AutoMate system, analogous to the reception of a hotel. Its purpose is to provide clients with references to the other managers on the server, e.g. the transaction, object, naming, and query managers. Thus the receptions can be used to authentify and authorify users. Future managers, like the planned version manager, will also be available via the reception.

6 The Client Side

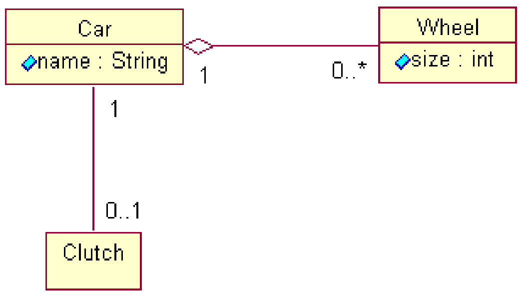

Figure 12: Example Class Diagram

In the previous sections, we demonstrated features as well as usage of the AutoMate framework using the simple example of a car. Now, we'd like to have a look from the viewpoint of the application programmer using the very same example.
We start out by the creation of the business class diagram as depicted in Figure 12. Basically, a car is modeled to have several wheels and an optional clutch. This specification is enriched within the tool Rational Rose with further information regarding the classes' persistency. Then, the server objects are created using the AutoMate-extension within Rational Rose which takes roughly five to ten minutes on a regular PC with Windows NT. After the generation process (see Figure 6), the AutoMate-framework together with the respective application objects is ready to use. Consider, a simple example of a client that instantiates a car and does some modifications (cf. Figure 13).

```
// initialize the ORB
ORB orb = ORB.init();

// get a CORBA reference to the reception
Reception reception = ReceptionHelper.bind(
  "AutoMateReception", "automate.tum.de");

// get CORBA references to the managers
TransactionManager transactionManager =
  reception.getTransactionManager();
ObjectManager objectManager = reception.getObjectManager();
NamingManager namingManager = reception.getNamingManager();
QueryManager queryManager = reception.getQueryManager();

// create and start a new transaction on the car database
Transaction transaction = transactionManager.open(
  "cars", Locking.READ_WRITE,
  TransactionLogic.PESSIMISTIC);
transaction.begin();

// get a reference to the object named "Donatas 2000 GLX"
Car carModel = CarHelper.narrow(
  namingManager.lookup(transaction, "Donatas 2000 GLX"));

// set the size of all wheels to 5
WheelEnumeration wheels = carModel.getWheels();
while (wheels.hasMoreElements())
  wheels.nextElement().setSize(5);

// create a new persistent Clutch object
Clutch clutch = ClutchHelper.narrow(
   objectManager.create(transaction,
   "de.tum.automate.cars.Clutch"));

// add the created clutch to the car model
carModel.setClutch(clutch);

// search for all cars named "Bug" and rename them to "Beetle"
ElementEnumeration bugCars = queryManager.query(transaction,
 "select selfoid from server.CarImplPers where _Name == \"Bug\"");
while (bugCars.hasMoreElements())
  CarHelper.narrow(bugCars.nextElement()).setName("Beetle");

// commit and destroy the transaction
transaction.commit();
transactionManager.close(transaction);
```

Figure 13: Implementation of the car client example

After initialization of the object request broker and the database, the code gets to know all managers as, for instance, the query manager. An appropriate transaction context is created and at this point, the technical preparation is done. Now, the business model is easily created and modified. Consider, in particular, the transparent usage of all application objects. It doesn't make a difference whether they are transient or persistent, remote or local.

7 Conclusion and Further Work

In this work, we presented and demonstrated AutoMate, a development tool essentially speeding up the creation of software systems based on three-tier architectures. Using the tool, application programmers can abstract away from technical details associated with underlying mechanisms and at the same time, exploit the properties of a software tool that incorporates a variety of existing standards, implementations , and specifications. For example, the ODMG 2.0 interface for object-oriented databases, the middleware CORBA, and existing transaction services.
But although the current version of AutoMate is already robust and fast enough to be used for some small to medium-sized three-tier applications, it is mainly intended as a first initial version. The basic architecture has the potential for many optimizations and extensions with respect to functionality as well as to scalability for large systems. We plan to offer support for

- the portable object adapter specification by the OMG. Besides being independant from the actual ORB in use, we result in an eased usage of object identifiers. In particular, we could get rid of the method `ObjectToHandle` of the TransactionManager interface.
- the new ODMG 3.0 interface specification. Here, a variety of new data types is introduced and ready to use also for navigational purposes. Moreover, several databases might be used at the same time.
- scheme evolution which is essentially model evolution resp. class diagram evolution. Basically, the designer specifies a set of transformation primitives which application results in the new model. For this purpose, we implemented another tool called *ShapeShifter* which does exaclty these steps using XMI/XML specifications. ShapeShifter is about to be integrated in AutoMate within the next couple of months.

Those interested in trying out AutoMate are invited to download a fully functional copy from our website [6].

References

[1] R. Cottell. *Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[2] Iona Technologies. *Iona Home Page,* `http://www.iona.com`, 1998.

[3] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, 2nd edition, 1998.

[4] Rational Rose. *Rational Rose,* `http://www.rational.com`, 1998.

[5] Klaus Renzel and Wolfgang Keller. Three Layer Architecture. *Software Architectures and Design Patterns in Business Applications*, TUM-I9746, 1997.

[6] The Automate Homepage, Technische Universität München. `http://automate.informatik.tu-muenchen.de`, 2000.

[7] Versant Object Technology. *Versant Home Page,* `http://www.versant.com`, 1999.