The TENA Middleware

Model-Based Distributed Application Development for High-Reliability DoD Range Systems

J. Russell Noseworthy, Ph.D.
TENA Software Development Lead
j.russell.noseworthy@objectsciences.com

Abstract

The Test and Training Enabling Architecture (TENA) Middleware is the result of a joint interoperability initiative of the Director, Operational Test and Evaluation (DOT&E) of the Office of the Secretary of Defense (OSD). The goals of the initiative are to enable interoperability among ranges, facilities, and simulations in a quick and cost-efficient manner, and to foster reuse of range assets and future range system developments.

The TENA Middleware uses Unified Modeling Language (UML)-based model-driven code generation to automatically create a complex Common Object Request Broker Architecture (CORBA) application. This model-driven automatic code-generation greatly reduces the amount of software that must be hand-written and tested.

Furthermore, the TENA Middleware combines distributed shared memory, anonymous publish-subscribe, and model-driven distributed object-oriented programming paradigms into a single distributed middleware system. This unique combination yields a powerful middleware system that enables its users to rapidly develop sophisticated yet understandable distributed applications.

The TENA Middleware offers powerful programming abstractions that are not present in CORBA alone and provides a strongly-typed Application Programmer Interface (API) that is much less error-prone than the existing CORBA API. These higher-level, easy-to-understand programming abstractions combined with an API designed to reduce programming errors enable users to quickly and correctly express the concepts of their applications.

Re-usable standardized object interfaces and implementations further simplify the development of applications. The net result of this combination of features is a significant reduction of application programming errors and hence increase overall reliability.

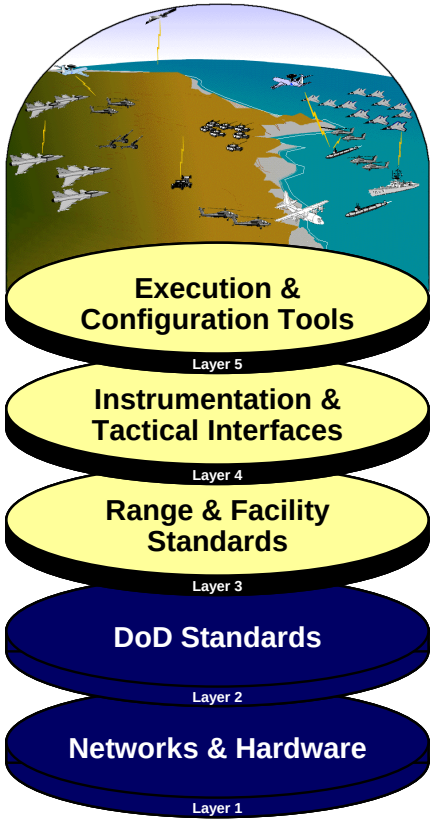
Distributed applications developed using the TENA Middleware exchange data using the publish-subscribe paradigm. Although many publish-subscribe systems exist, the TENA Middleware represents a significant advance in the field due to many the high-level, model-driven programming abstractions it presents to the programmer.

The TENA Middleware API relies heavily on compile-time type-safety to help ensure reliable behavior at runtime. Careful API design allows a great number of potential errors to be detected at compile-time that might otherwise go unnoticed until run-time—where the cost of an error could be extremely high!

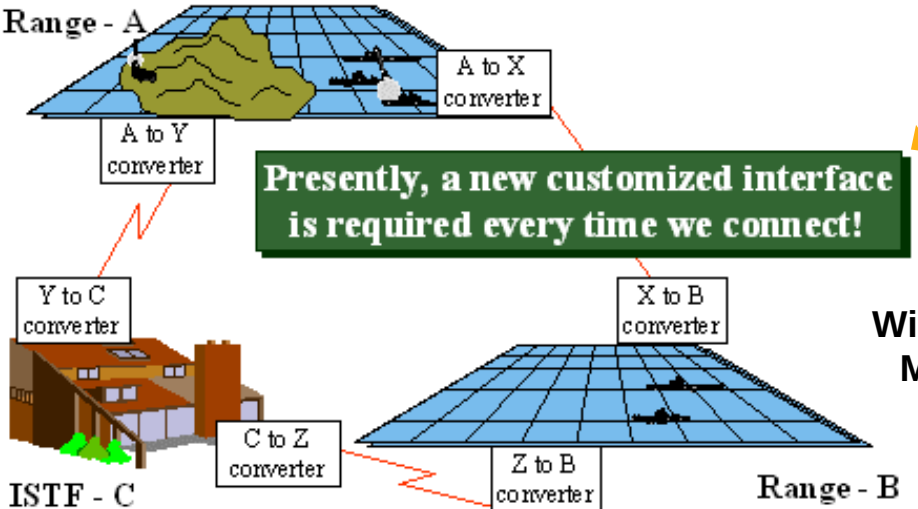
The implementation of the TENA Middleware uses C++, as well as a real-time CORBA ORB. The TENA Middleware is currently in use at dozens of Department of Defense (DoD) testing and training ranges across the country. It is available, subject to U.S. export control laws, at http://www.fi2010.org/.

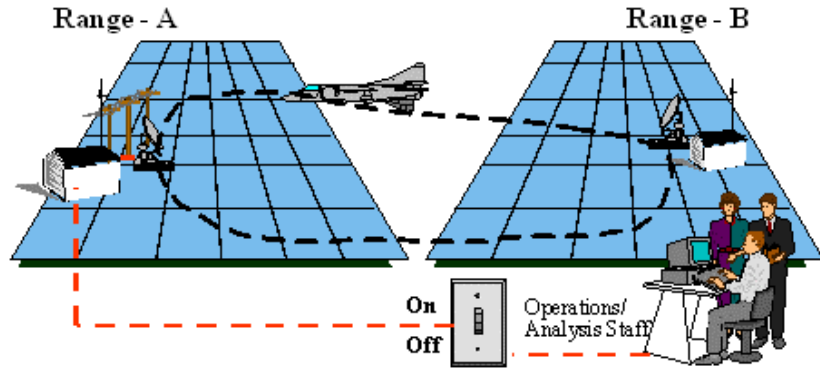
Foundation Initiative 2010

Foundation Initiative (FI) 2010 is a joint interoperability initiative of the Director, Operational Test and Evaluation (DOT&E) of the Office of the Secretary of Defense. The vision of FI 2010 is to enable interoperability among ranges, facilities and simulations in a quick and cost-efficient manner, and to foster reuse of range assets and future range system developments.

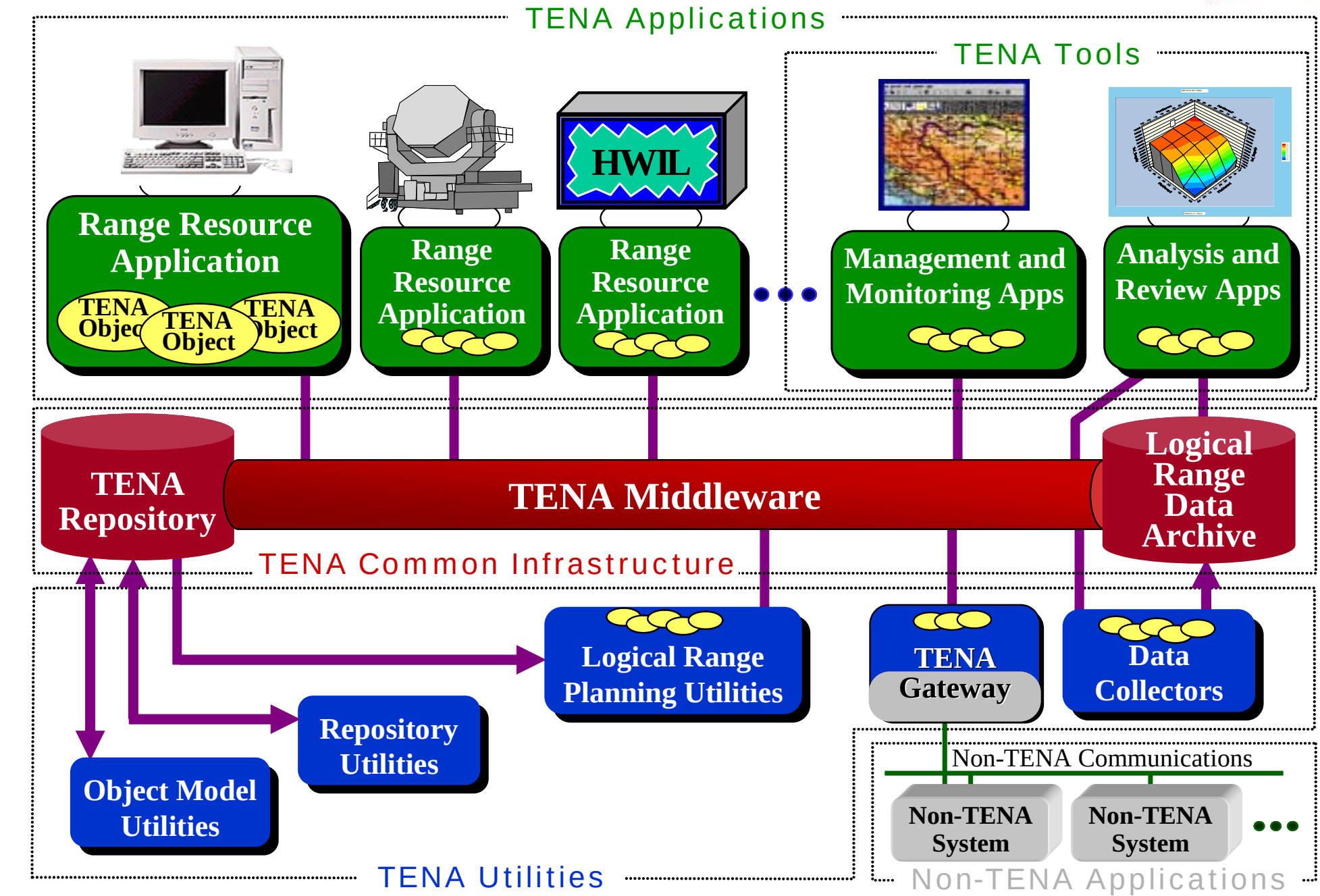


The central product of FI 2010 is TENA, the Test and Training Enabling Architecture (http://www.fi2010.org/documents/tena2002.pdf). The TENA Middleware is the central product of the architecture (http://www.fi2010.org/).





TENA Architecture Overview



Distributed, Real-time and Embedded Systems and the TENA Middleware

DoD ranges use systems of sensors to take measurements for the purpose of testing and/or training.

Many of the these sensor systems are embedded systems.

The testing and training events occur in the **real** world. Thus, **real** missiles are launched, **real** tanks are driven, **real** planes are flown, etc.; and so **real** measurements must be taken in **real-**time.

The sensor systems are themselves inherently distributed, typically over a large geographic area.

The sensor systems can include half a dozen to several hundred individual component sensors.

So, DoD ranges are large-scale, distributed, real-time and embedded (DRE) systems.

The TENA Middleware is intended to support DoD ranges.

Thus, the TENA Middleware must support large-scale DRE systems.

How Does the TENA Middleware Approach the Challenges of Large-scale DRE Systems?

Successfully developing applications for large-scale, distributed, real-time and embedded systems is very difficult for most (all?) programmers.

The TENA Middleware must make it easier for programmers to develop such applications.

Reliability, maintainability, and understandability are more critical features for these applications than maximum possible performance, as long as these applications run “fast enough”.

The TENA Middleware provides:

- Model-based, high-level programming abstractions.
- Bug prevention through compile-time type checking and an API that’s hard to use wrong.
- Model-driven code generation of custom-tailored core middleware software.
- Complete, working, model-based applications, ready for customization by programmers.

The TENA Middleware uses model-driven automated code generation to reduce the amount of software that must be written (and tested) by humans. Furthermore, the TENA Middleware provides the application developer with a powerful programming abstractions. These programming abstractions are easy for the application developer to understand, resulting in applications with fewer mistakes.

Ultimately, reliability is determined through the course time, after considerable testing. However, the time required to achieve the desired reliability can be greatly reduced with the help of middleware and tools to reduce the possibility of human error.

Distributed Programming Communication Paradigms

Range applications communicate various types of information over computer networks, *e.g.*, test measurements, telemetry streams, command and control signals, simulation data, *etc.*

Several different types of distributed application development paradigms are commonly employed to solve these types of distributed computing problems:

Message Passing

A programming paradigm where application programmers address packets of data (*i.e.*, messages) to one or more recipients.

Anonymous Publish-Subscribe

A common form of message passing that does not require (or even allow) the message sender to declare the intended message recipients. Instead, each message sender (*i.e.*, publisher) indicates the type of message being sent. Message recipients (*i.e.*, subscribers) indicate the type of message they would like to receive.

Distributed Shared Memory (DSM)

A programming paradigm that projects the illusion of working on a shared memory multi-processor machine onto a distributed computing system.

Remote Method Invocation

Distributed object-oriented computing (DOC) is a programming paradigm designed to mimic method invocation on objects with *remote* method invocation (RMI) on distributed (*i.e.*, remote) objects. Where DSM systems strive to provide the illusion that reads and writes of certain data structures are local to a single application even though they are in fact shared amongst multiple applications, DOC systems strive to provide the illusion that method invocations on certain objects are performed locally even though they are in fact, performed on an object implemented in a remote application.

Model-Driven Programming Paradigm

Model-driven programming is a paradigm focused on the use of abstract models to describe a given programming problem. The model is used as the basis for the automatic generation of programs used to solve the problem. Since the model describes the problem instead of a particular solution to the problem, model-driven programming offers to promise to readily change the techniques (*e.g.*, the type of middleware) used by the generated programs to solve the problem. The Object Management Group (OMG) has been championing a model-driven programming paradigm called Model Driven Architecture (MDA®).

> MDA is an approach to system development, which increases the power of models in that work. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.
>
> > *MDA Guide*, http://www.omg.org/docs/omg/03-06-01.pdf, Object Management Group, Jun. 2003, version 1.0.1

Combining Paradigms to Create the TENA Middleware

Each of the above programming paradigms have their strengths and weaknesses. By carefully combining the above communication paradigms with model-driven programming, the TENA Middleware has made a significant advance in the field of distributed programming systems. **The TENA Middleware combines distributed shared memory, anonymous publish-subscribe, and model-driven distributed object-oriented programming paradigms into a single distributed middleware system.** This unique combination of high-level programming abstractions yields a powerful middleware system that enables its users to rapidly develop complex yet reliable distributed applications.

The TENA Middleware provides these high-level abstractions by using models to drive the automatic code-generation of complex CORBA applications. In this way, the TENA Middleware offers programming abstractions not present in CORBA and provides a strongly-typed Application Programmer Interface (API) that is much less error-prone than the existing CORBA API. The added high-level programming abstractions combined with an API designed to reduce programming errors enable users to quickly and correctly express the concepts of their applications. Re-usable standardized object interfaces and implementations further simplify the application development process.

TENA Middleware MetaModel

A *metamodel* is "a model that defines an abstract language for expressing other models" *Common Warehouse Metamodel—Glossary*, http://www.omg.org/ docs/formal/01-10-25.pdf, OMG, Nov. 2001

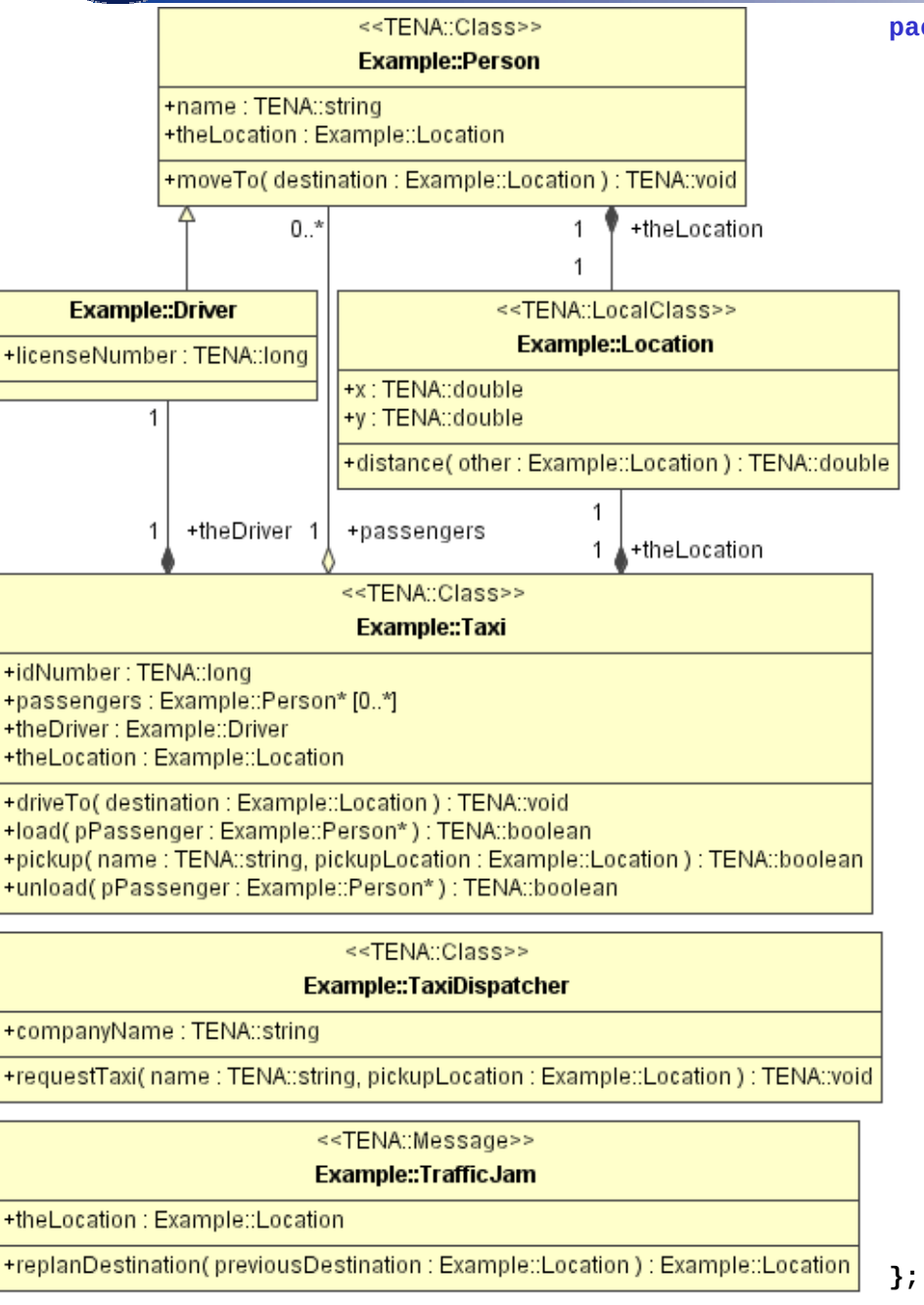
Every modern programming language has a metamodel. For example, the metamodel for the C++ programming language says that a class may have attributes and operations, that one C++ class may inherit from another, and that one C++ class may contain another.

The TENA metamodel defines several fundamental abstract concepts, constructs, and the relationships between them. The TENA Middleware supports these abstract concepts and constructs. A TENA Object Model (OM) is constructed from specific TENA metamodel concepts and relationships between those specific concepts.

A TENA Object Model can be represented in two ways. The first is using a Unified Modeling Language (UML) class diagram. The second is using an text-based representation. The following slide demonstrates each of the two types of representations using an identical example Object Model.

This particular example Object Model might be useful for creating a distributed control application for a taxi company, or to create a taxi company simulation or video game. This example Object Model uses many of the features provided by the TENA metamodel.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

The Stateful Distributed Object (SDO)

A Stateful Distributed Object SDO is an abstract concept formed by the combination of a CORBA distributed object with data or *state*. The state is data attributes of the SDO that are disseminated via publish-subscribe and cached locally at each subscriber.

Traditional CORBA middleware provides the illusion that a distributed object's methods exist on a local object in the native programming language of the application. Unbeknownst to the programmer, the distributed object's methods may in fact involve a remote method invocation to another application on the network.

The SDO extends this concept to include, not only methods, but data attributes as well. The TENA Middleware provides the illusion that both an SDO's methods and state can be accessed *locally*, as if the SDO was a object in the native programming language of the application.

An SDOs state is disseminated via anonymous publish-subscribe to applications based on subscription characteristics, such as the type of the SDO. Subscribers can read the state of an SDO as if it were a local object. The state making up an SDO can consist of the typical fundamental data types, *e.g.*, long, double, string, enums, *etc.* Furthermore, any of the other complex constructs from the TENA metamodel described below are allowed to be SDO attributes.

In addition to state (*i.e.*, attributes), every SDO may have remotely-invocable methods. As with attributes, any of the fundamental data types or complex constructs from the TENA metamodel may be used as method parameters or return types, with the notable exception of an SDO itself. This is because an SDO cannot be passed by value.

The SDO (Continued)

Combining the concepts of a CORBA distributed object with the distributed shared memory and anonymous publish-subscribe programming paradigms to create an SDO, results in a programming abstraction possessing the benefits of all these systems:

An SDO supports the remote method invocation concept that is very natural to distributed object-oriented system programmers.

An SDO provides direct support for disseminating data from its source to multiple destinations.

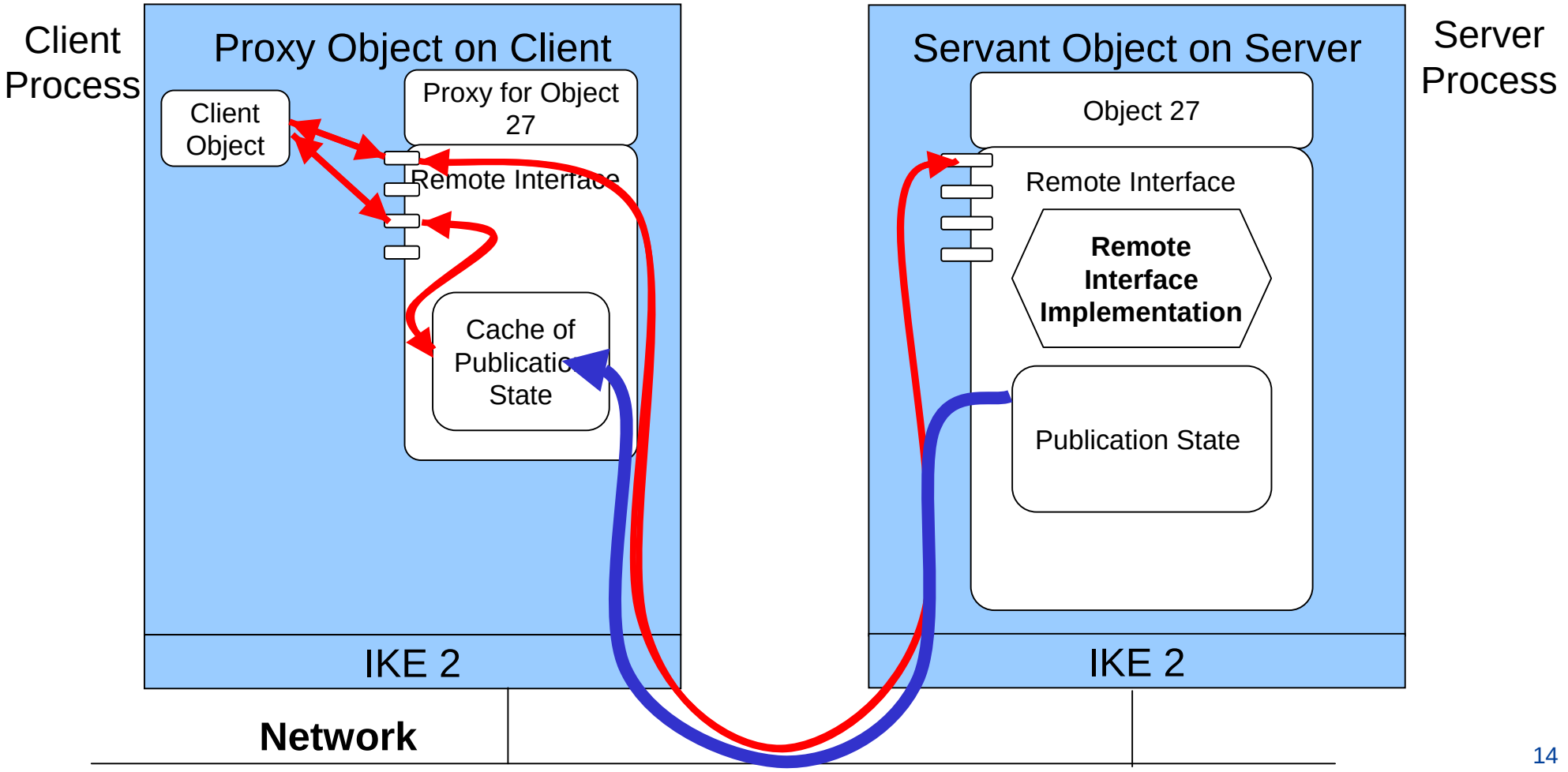
An SDO supports reads and writes of data as if it were any other local data—a concept familiar to virtually every modern programmer.

An SDO's model-driven automatically generated code eliminates the tedious and error-prone programming chores common to distributed programming.

An SDO's API is straightforward and easy to understand. An SDO is shown in a TENA Object Model with either the TENA::Class stereotype, if the Object Model is represented in UML, or with the keyword class, if the Object Model is represented in TDL. In the Object Model shown on the preceding slide, Person, Driver, Taxi, and TaxiDispatcher are all SDOs.

Clients & Proxies; Servers & Servants

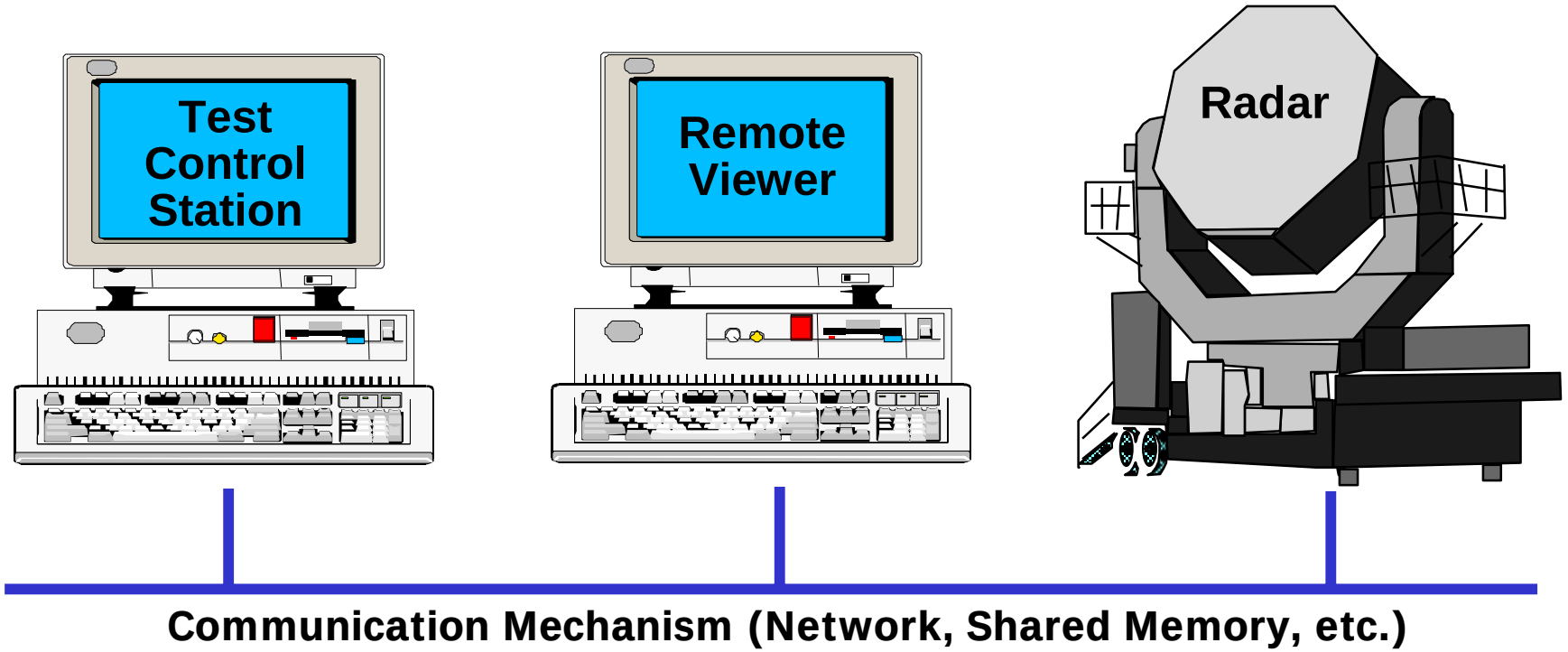
The abstract concept of a single SDO instance is implemented using a servant object in a server process and proxy objects in client processes. Method invocations on a proxy are delegated to the servant via RMI. Data attribute accesses on a proxy are performed via a local cache read.



Simple Example of a Logical Range

TENA specifies a peer-to-peer architecture. Applications can be both **clients** and **servers** simultaneously. Server applications contain one or more SDO **servants**. Client applications obtain **proxies** to servants by subscribing. Only the SDO's servant can modify its publication state.

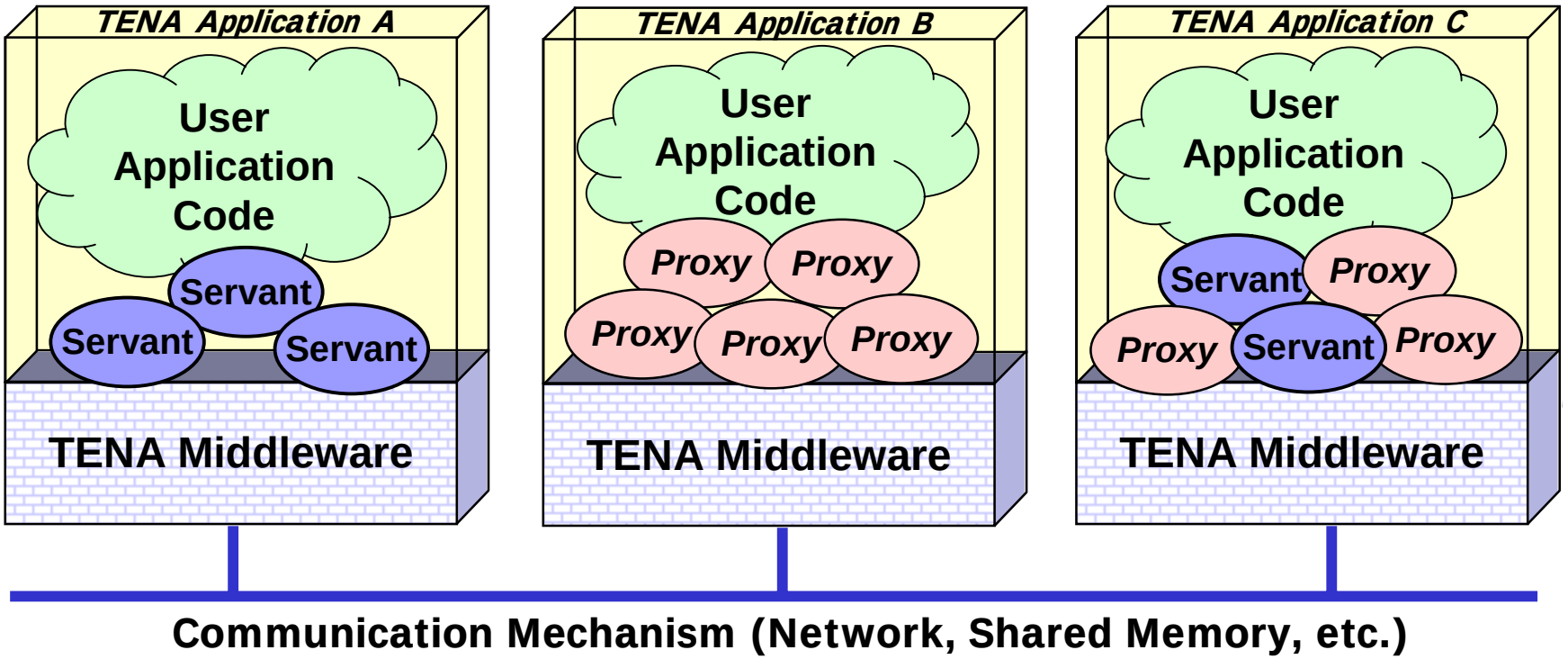
The TENA Middleware, the TENA objects, and the user's application code are compiled and linked together.



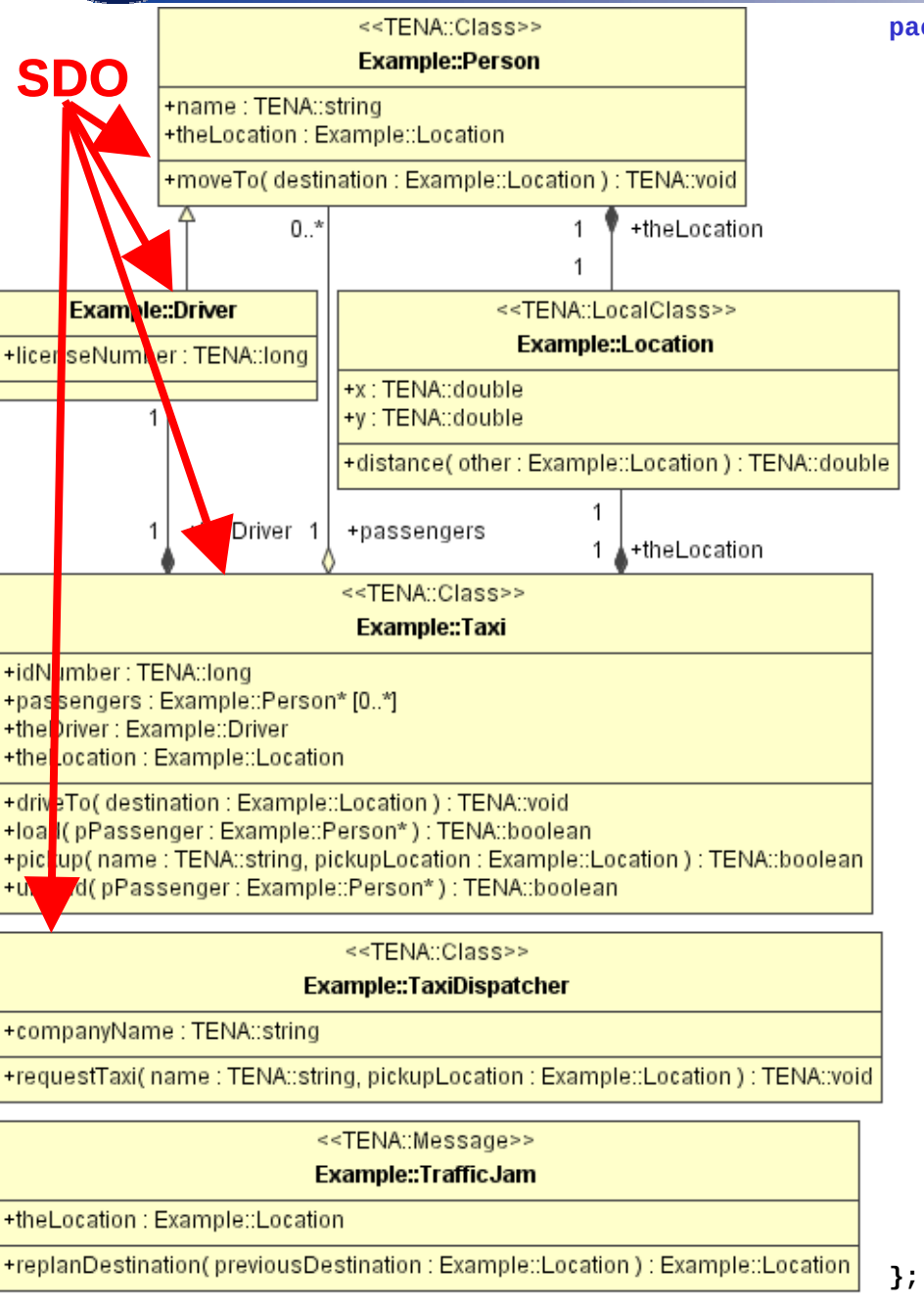
Simple Example of a Logical Range

TENA specifies a peer-to-peer architecture. Applications can be both **clients** and **servers** simultaneously. Server applications contain one or more SDO **servants**. Client applications obtain **proxies** to servants by subscribing. Only the SDO's servant can modify its publication state.

The TENA Middleware, the TENA objects, and the user's application code are compiled and linked together.



Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

Interface

An interface in the TENA metamodel is very similar to a conventional distributed object. That is, it may describe the signatures to a collection of remotely invocable methods. However, unlike an SDO, an interface may not have attributes. Furthermore, since an interface in the TENA metamodel in inherently abstract, every interface must be implemented by an SDO in order to be used in a distributed application.

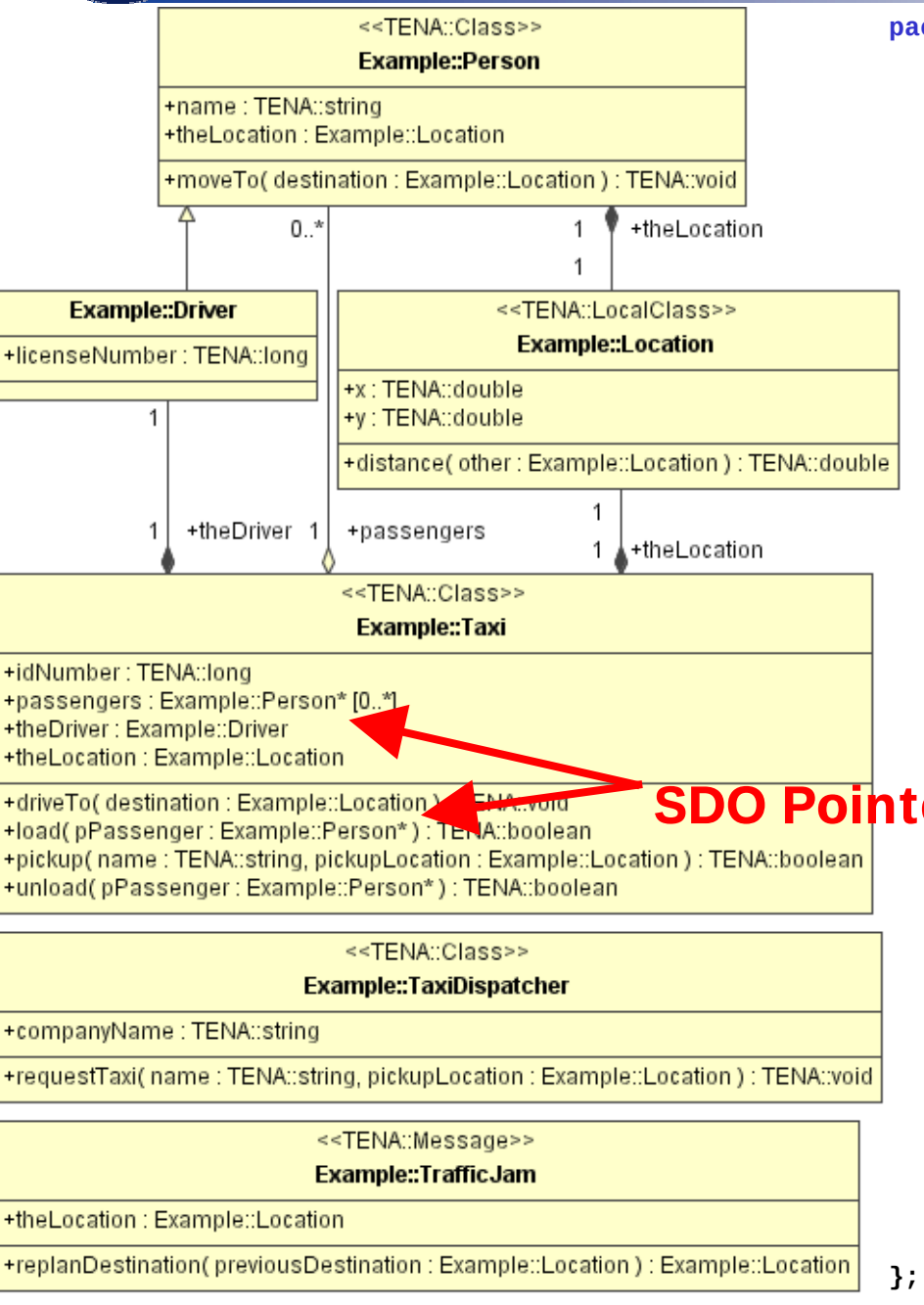
An interface is shown in a TENA Object Model with either the TENA::Interface stereotype, if the Object Model is represented in UML, or with the keyword interface, if the Object Model is represented in TDL. Note that the example Object Model shown in Figures 2 and 3 on the previous page does not happen to contain any interfaces.

SDO Pointer

In the TENA metamodel, a pointer to an SDO behaves in the same was as pointers to objects in the metamodels of other programming systems. An SDO pointer can be used to refer to or *point to* a particular SDO instance. The particular SDO instance pointed to can be changed. In fact, an SDO pointer could be set point no SDO instance, *i.e.*, to point to "nothing". Finally, an SDO pointer can be *de-referenced* to yield the SDO to which the pointer referred.

An SDO pointer is shown in a TENA Object Model by naming the SDO type followed by the attribute or argument name followed by an asterisk (*). This is true for both the UML and TDL representations. In the Object Model shown in Figures 2 and 3 on the preceding page, the Taxi SDO's load() and unload() methods each have SDO pointers to a Person SDO as parameters. Furthermore, the Taxi SDO has an attribute named passengers that is a vector of SDO Person pointers. Vectors are discussed in Section II-H. The concept of an SDO pointer enables the ability to model ideas such as those shown in the example Object Model. The Taxi SDO can load() and unload() instances of Person SDOs using an SDO pointer, thereby modeling the behavior of a real taxi.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

Local Class

A local class in the TENA metamodel is similar to a class (i.e., an SDO) in that it too is composed of both methods and attributes (i.e., state). However the methods and attributes of a local class are always local with respect to the application holding an instance of the local class.

Invoking a method on an SDO in general involves an RMI over the network to a remote machine where the method's computation is actually performed. Invoking a method on a local class always results in the method's computation being performed locally in the invoking application.

Similarly, the attributes (i.e., state) of an SDO are accessed as if they were local to the application. In fact their values may have been set in a remote application and cached in the local application after having been disseminated. Each time they are re-set by the remote application, they are disseminated and the cache in the local application is updated. In contrast, the attributes of a local class are always completely local to the application holding the instance of the local class.

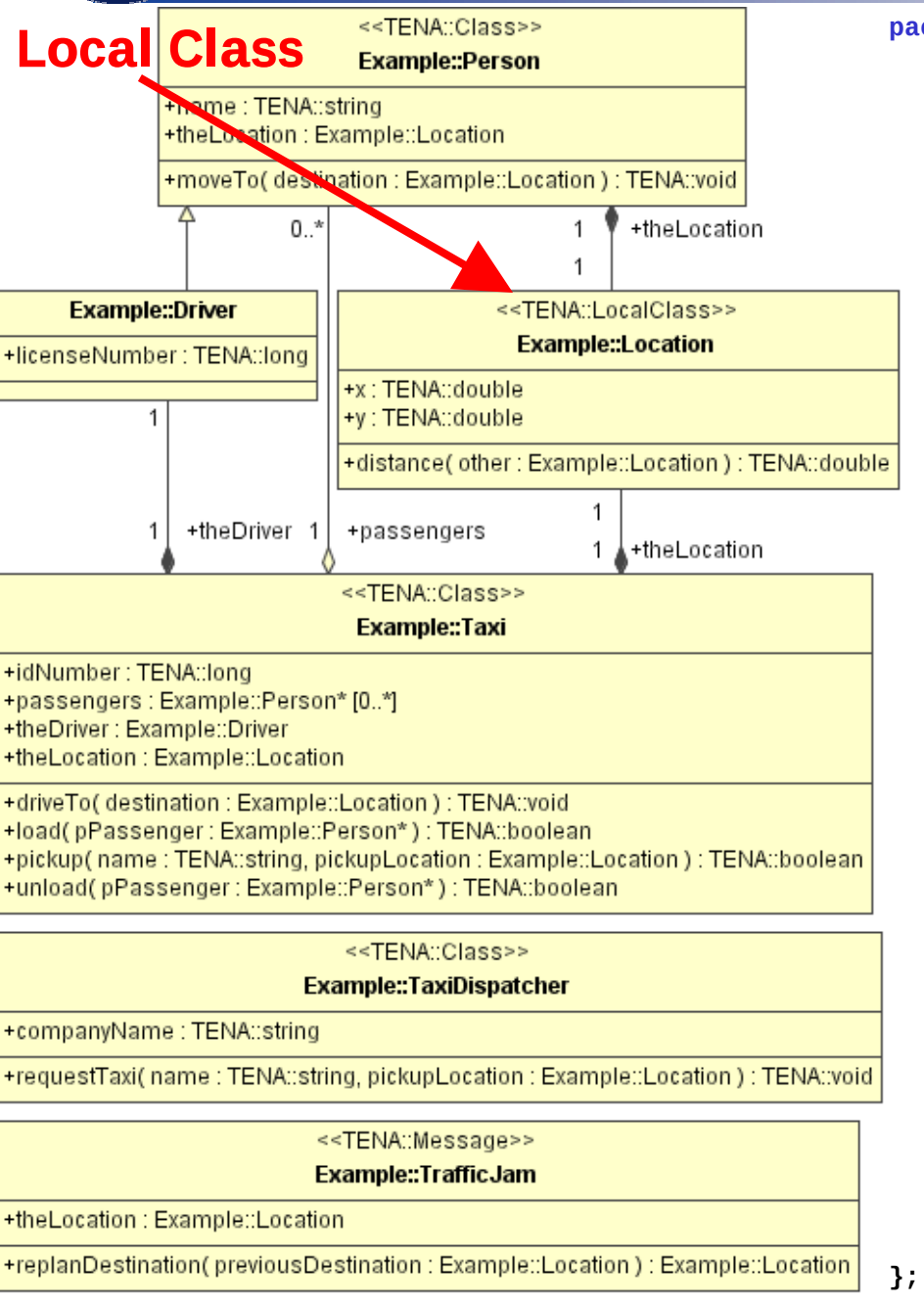
Local classes are analogous to CORBA valuetypes, a name given to connote the idea of an object (or type) that is passed by value to a method—even a remote method. The parameters or return types of methods on a local class allow everything in the TENA metamodel is except an SDO. Since local classes are always passed by value, they themselves are legal parameters and return types.

Similarly, any of the fundamental data types or complex constructs from the TENA metamodel may be used as an attribute of a local class, with the exception of an SDO. Since a local class is inherently local, composing it from constructs that are generally remote (i.e., SDOs) is impossible. A local class is shown in a TENA Object Model with either the <<TENA::Local Class>> stereotype, if the Object Model is represented in UML, or with the keywords local class, if the Object Model is represented in TDL. In the Object Model shown in Figures 2 and 3, Location is a local class.

The local class concept greatly improves the power and expressiveness of the TENA metamodel. While the SDO concept is very adept at modeling many real-world problems, often situations arise where it is impractical to use RMI or remotely modifiable attributes.

The local class named Location in the example Object Model demonstrates just such a case. The distance() method on Location calculates the distance between two Locations. Performing such a simple computation as an RMI, while possible, would be grossly inefficient in most real distributed systems. This same Location concept could be extended to provide methods to convert from one coordinate system to another, thereby greatly enhancing interoperability, without sacrificing efficiency.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

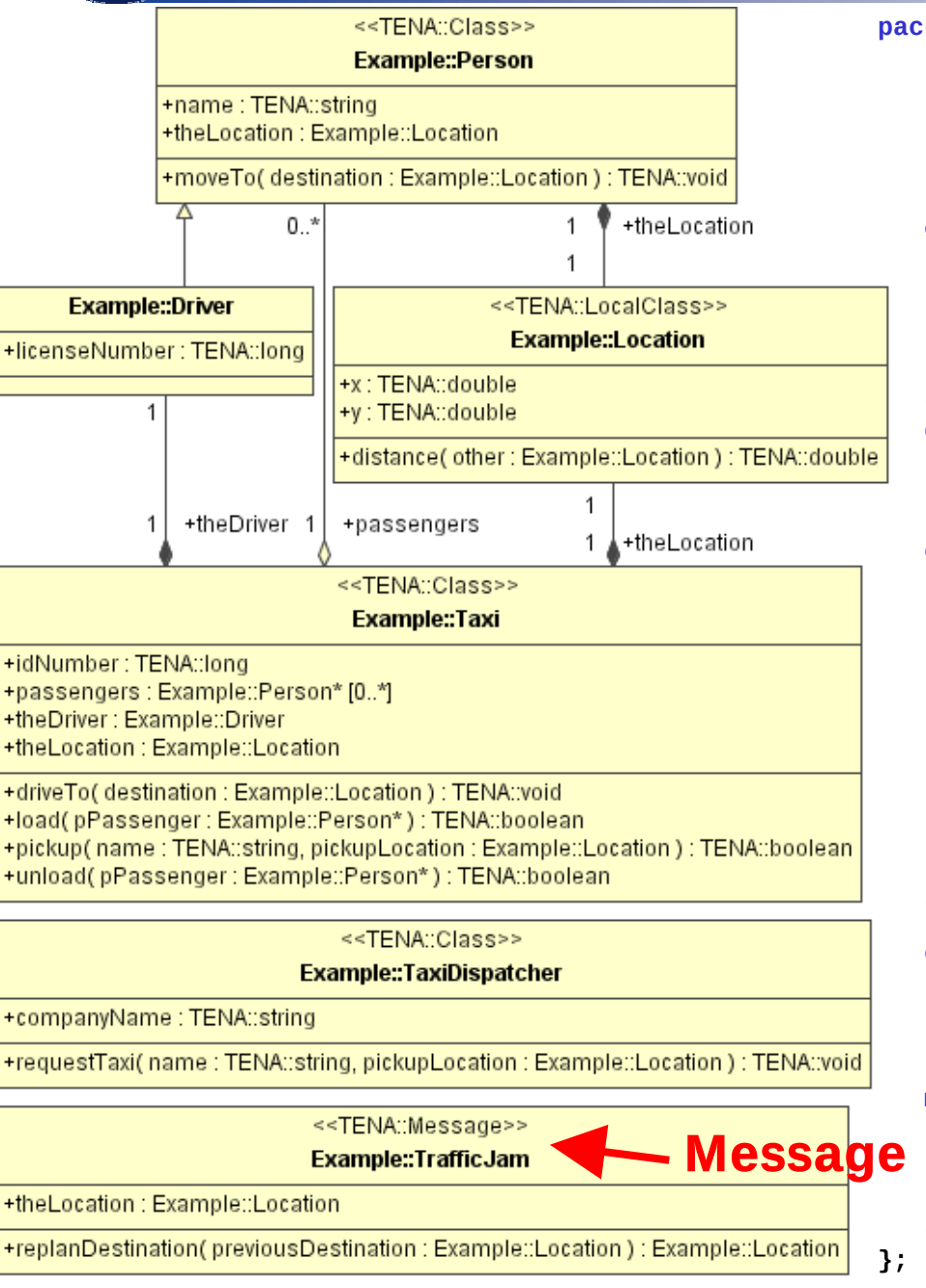
Message

A message in the TENA metamodel is identical to a local class with one important distinction. A local class can only be indirectly disseminated to a remote application via an RMI on an SDO or as an attribute of the state of an SDO. A message can be directly disseminated to remote applications, i.e., it can be sent and received.

A message is shown in a TENA Object Model with either the <<TENA::Message>> stereotype, if the Object Model is represented in UML, or with the keyword message, if the Object Model is represented in TDL. The example Object Model shown in Figures 2 and 3 has a message named TrafficJam.

Messages are provided in the TENA metamodel to allow application developers to model events that do not persist. An SDO, is intended to model a concept in the distributed application that persists and whose state may get modified over time, such as the Taxi SDO in the example Object Model. A message is indented to model a one-time event that typically needs to be delivered to multiple applications, such as the TrafficJam message in the example Object Model.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

Inheritance

Inheritance is a core concept of the metamodel of every object-oriented programming system. Inheritance, often referred to as the is-a relationship, provides the notions of extensibility and substitutability. For example, in Figure 2 there is an arrow from the Driver SDO to the Person SDO. That arrow is the UML symbol for inheritance. This means that a Driver is-a Person.

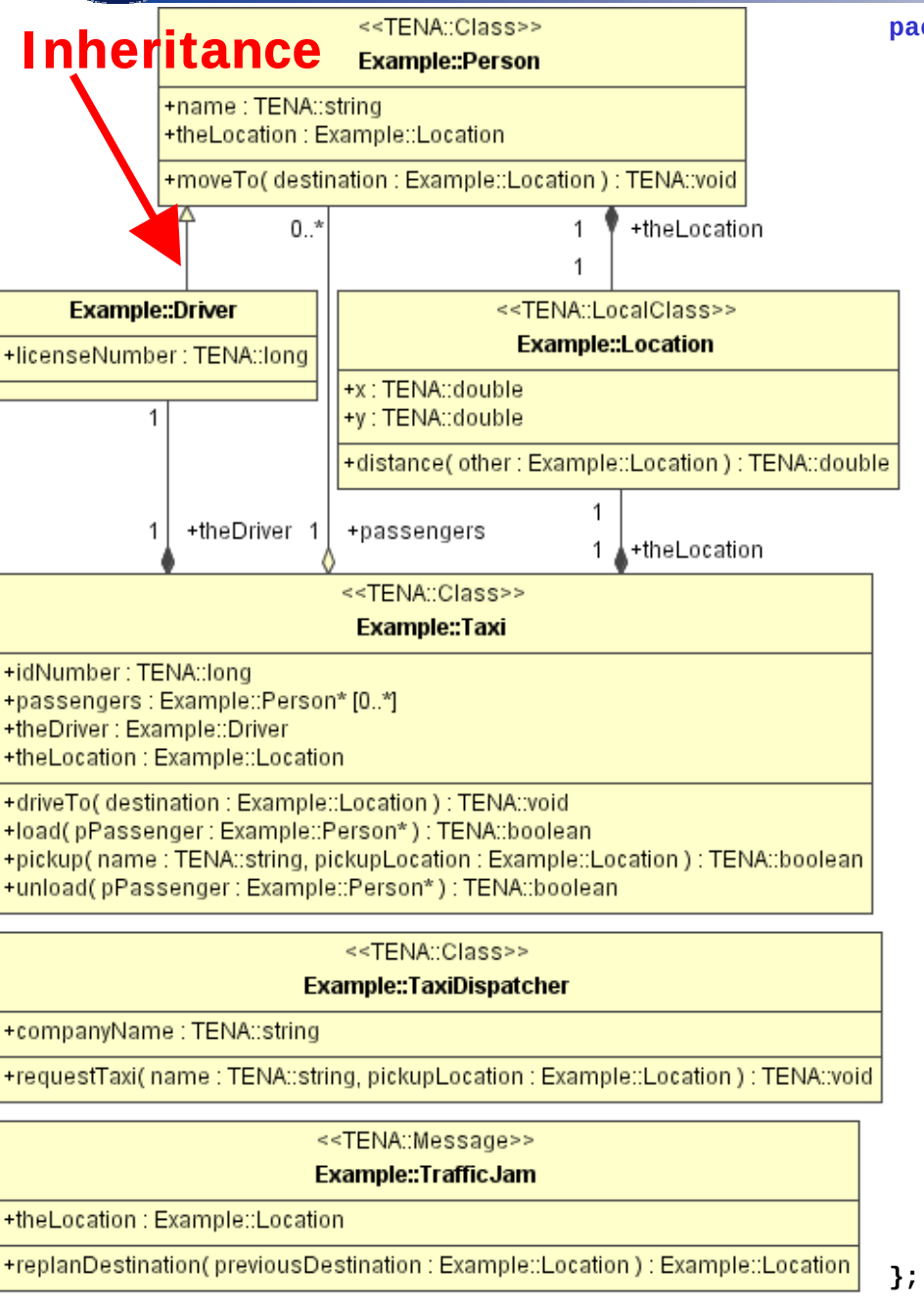
Thus, anywhere a Person is required, e.g., in the Taxi::load() method, a Driver may be substituted. Furthermore, the Driver has all the properties of a Person and more (i.e., it has a licenseNumber attribute). Thus the Driver is an extension of the Person. In Figure 3, the TDL definition of Driver uses the extends keyword to indicate that it inherits from Person.

The TENA metamodel allows an SDO to extend, or singly inherit from, another SDO. Likewise, a local class can singly inherit from another local class, and a message can singly inherit from another message. Interfaces can singly or multiply inherit from other interfaces.

The TENA metamodel also allows an SDO to multiply inherit from an interface. Although not shown in the example Object Model, this is represented in TDL using the implements keyword.

Any TENA Middleware application that subscribes to SDOs of type Person, will have access to all the Person SDO instances. In addition to this, they will have access to all the Driver SDO instances but only those parts of the Driver that the Driver inherited from Person. That is, the Driver SDOs will appear to the Person subscriber as just a Person, and the licenseNumber attribute of the Driver will be unavailable.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

Containment

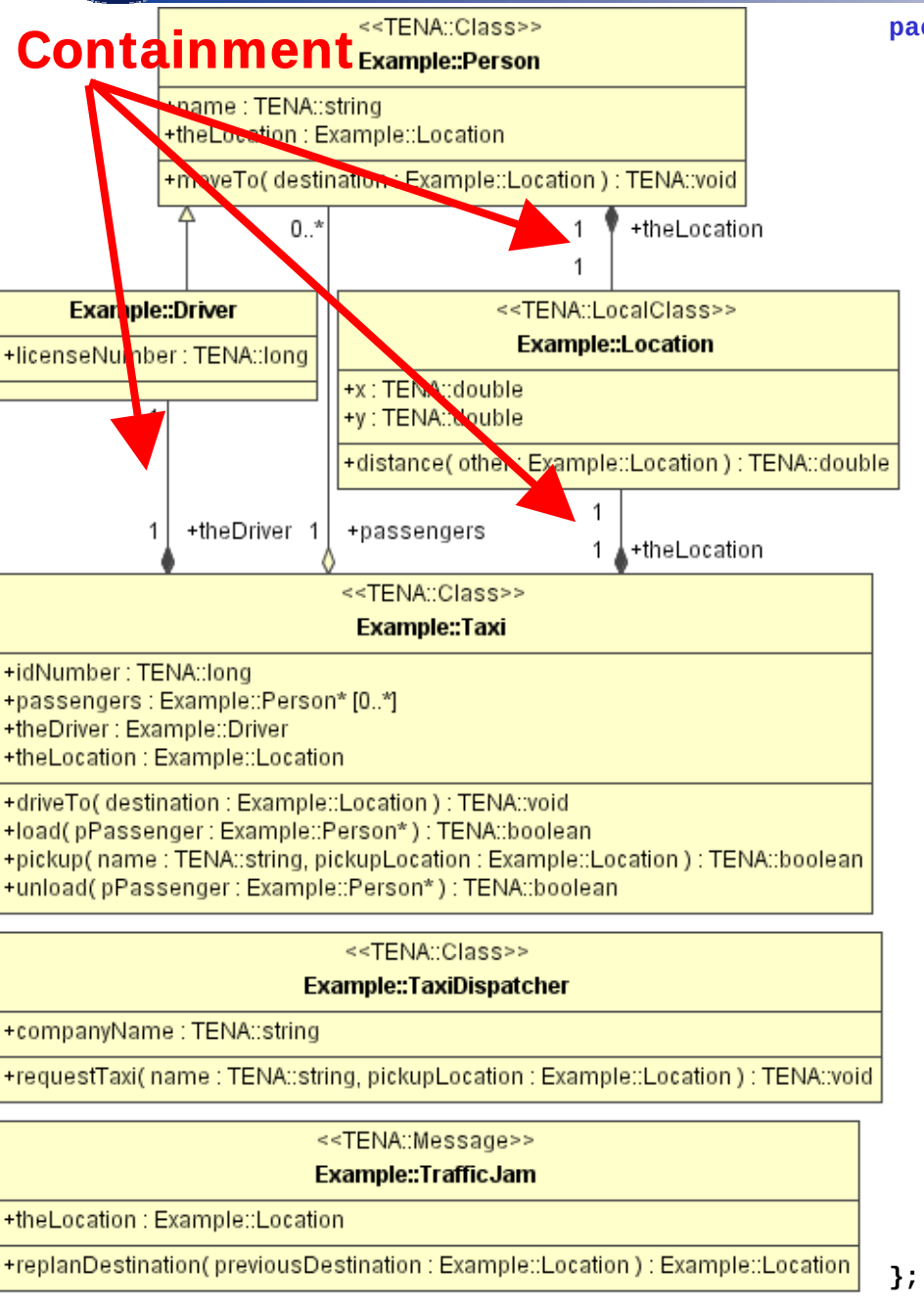
Containment is another core concept of every object-oriented programming system's metamodel. Containment is often referred to as the *has-a* relationship. The containment relationship guarantees that the lifetime of the contained object is never longer than the container, *i.e.*, the part never "outlives" the part.

The TENA metamodel allows an SDO to contain one or more other SDOs, with no theoretical limit to the nesting. Likewise, a local class may contain one or more other local classes. In addition to this, an SDO may contain an arbitrary number of local classes and messages. Similarly, a local class may contain an arbitrary number of messages. Finally, a message may contain any number of local classes and other messages.

By definition, an attribute is contained in it's class. Thus every attribute in the example Object Model shown in Figure 2 and 3 is an example of containment. Even the fundamental types (*e.g.*, the licenseNumber of the Driver) are trivial examples of containment.

But consider the Taxi SDO, which contains a Driver SDO and a local class named Location. Any application that subscribes to Taxi SDOs will have access to, not only every Taxi SDO instance, but also every Driver SDO instance contained in each Taxi. Furthermore, any application that subscribes to Driver SDOs will have access to every Driver SDO instance—even those Driver SDOs contained within a Taxi SDO.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

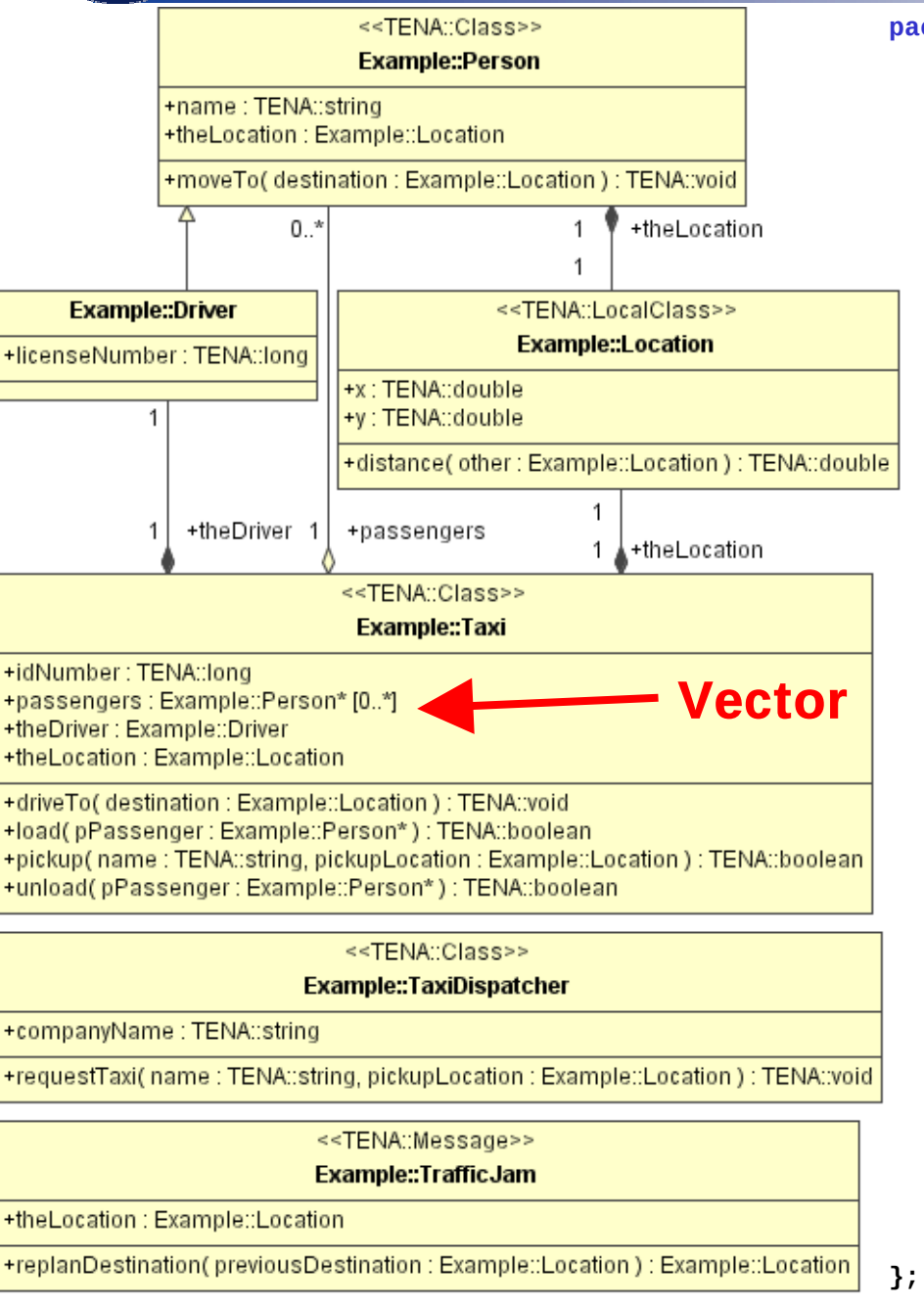
Vectors

Often, real-world systems can be best modeled using objects with a variable number of one type of attribute. For example, a real-world taxi does not always have the same number of passengers. For these types of situation, the TENA metamodel offers a construct called “vector”. A vector in the TENA metamodel is provided to model the concept of varying cardinality of attributes.

The TENA metamodel supports vectors of fundamental data types (*e.g.*, long, string, enum, *etc.*) as well as vectors of local classes, messages, and SDO pointers. A vector is depicted in a TENA Object Model using UML attribute multiplicity notation, if the Object Model is represented in UML, or using the keyword vector if the Object Model is represented in TDL. Examples of each can be seen in Figures 2 and 3, respectively.

The example Object Model in those figures demonstrates the combination of the vector concept with the SDO pointer concept. The Taxi SDO contains an attribute named “passengers” that is a vector of SDO pointers to Person SDOS. This is a natural way to model the behavior of a real-world Taxi, which holds not only a variable *number* of passengers, but also changes the *particular* passenger(s) it holds.

Example TENA Object Model

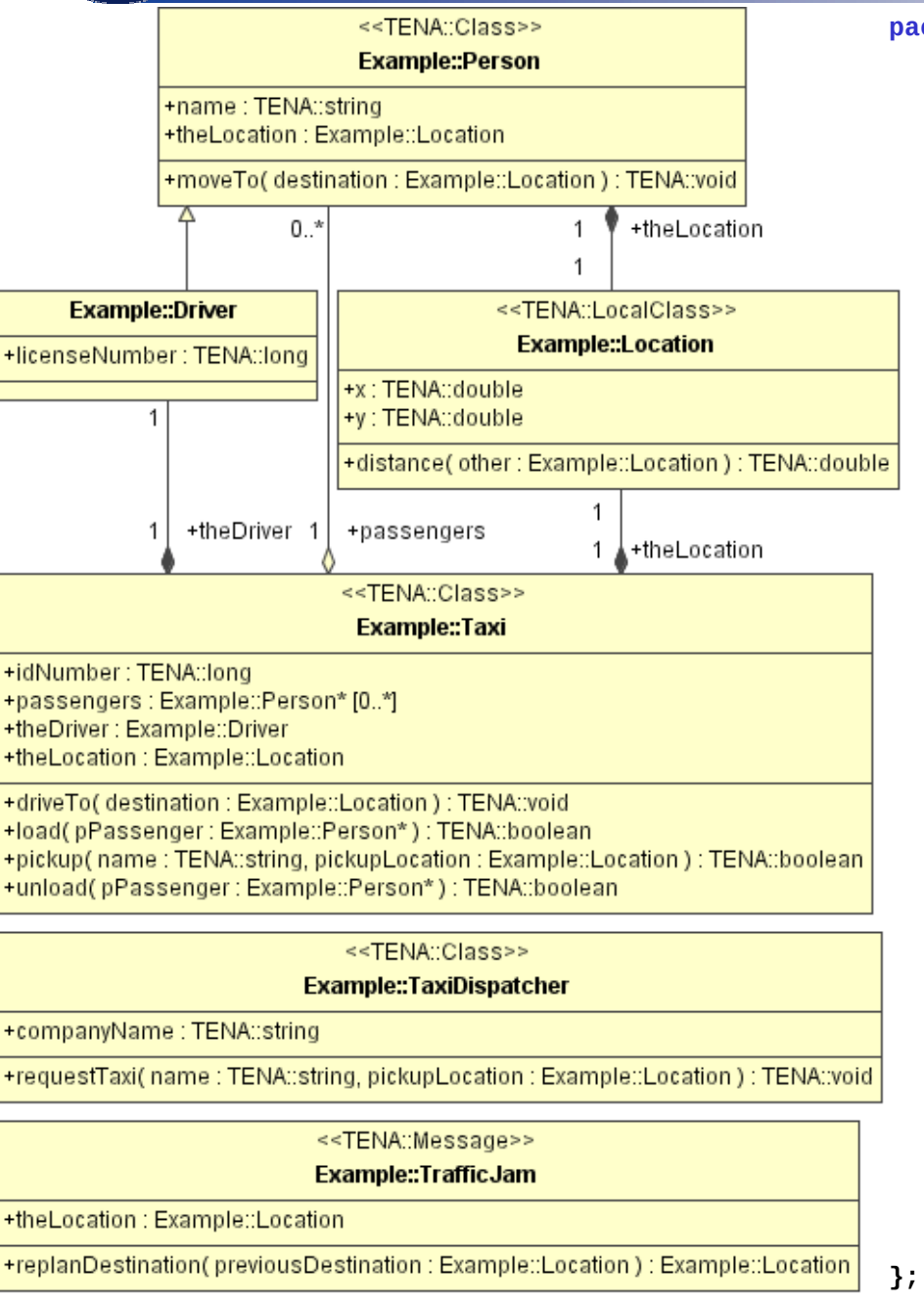



```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

Package

Finally, the TENA metamodel allows grouping of related constructs using a *package*. Packages reduce name conflicts between multiple Object Models, or even just a single, complex Object Model. In the example Object Model, all the constructs are contained in a package named “Example”.

Example TENA Object Model




```
package Example {
   local class Location {
      double x, y;
      double distance( in Location other );
   };
   class Person {
      string name;
      Location theLocation;
      void moveTo( in Location destination );
   };
   class Driver : extends Person {
      long licenseNumber;
   };
   class Taxi {
      long idNumber;
      vector< Person * > passengers;
      Driver theDriver;
      Location theLocation;
      void driveTo( in Location destination );
      boolean load( in Person * pPassenger );
      boolean pickup( in string name, in Location pickupLocation );
      boolean unload( in Person * pPassenger );
   };
   class TaxiDispatcher {
      string companyName;
      void requestTaxi(in string name, in Location pickupLocation);
    };
   message TrafficJam {
      Location theLocation;
      Location replanDestination( in Location previousDestination);
   };
};
```

The TENA Middleware API

The TENA Middleware API relies heavily on compile-time type-safety to help ensure reliable behavior at run-time. Careful API design allows a great number of potential errors to be detected at compile-time that might otherwise go unnoticed until run-time—where the cost of an error could be extremely high.

Strong typing prevents the accidental misuse of an object, increasing the likelihood that an application that compiles will run successfully. Modern C++ makes it possible to eliminate the simple memory errors that have been the bane of many a developer's application.

The TENA Middleware API provides clear, consistent, and unambiguous memory semantics. Application developers **never** use **delete** on TENA Middleware objects. All dynamically allocated objects manage their own memory. In addition to preventing memory leaks, this provides exception safety.

The API includes Elements of the OM that are Compiled into Applications

Why use compiled-in object definitions?

Strong type-checking

Don't wait until runtime to find errors that a compiler could detect

Performance

Interpretation of methods/attributes has significant impact

Ability to easily handle complex object relationships

Conforms to current best software engineering practices

How do you support compiled-in object definitions?

Use a language like CORBA IDL to define object interface and object state structure

Use a compiler to implement the required functionality

Thus the concept of the TENA Definition Language (TDL) was created

Very similar to IDL and C++

All TENA object model constructs are defined in TDL.

API for Instantiating an SDO Servant

```
// Create an object that describes the implementation details
// of the Taxi SDO servant to be instantiated.
Example::Taxi::PublicationInfoPtr
 pTaxiPubInfo(
   new Example::Taxi::BasicImpl::PublicationInfoImpl );

// Inform the Middleware that you will publish Taxi SDOs
// Provide it some publication info and get back an
// Example::Taxi::ServantFactory.
Example::Taxi::ServantFactoryPtr pTaxiServantFactory(
 pSession->
   createServantFactory< Example::Taxi::ServantTraits >(
     pTaxiPubInfo ) );

// Instantiate a Taxi SDO servant using a TaxiServantFactory.
// In this example the servant's state will be disseminated
// using BestEffort (UDP multicast).
Example::Taxi::ServantPtr pTaxiServant(
 pTaxiServantFactory->createServantUsingDefaultFactory(
  TENA::Middleware::BestEffort ) );
```

Updating the Servant's Publication State

To change a servant's publication state, the TENA Middleware uses the concept of "updaters".

Updaters allow sets of publication state attributes to be modified "atomically" (all at once, all or none).

```
// Get an updater
std::auto_ptr<
Example::Taxi::PublicationStateUpdater >
  pTaxiUpdater(
    pTaxiServant->createPublicationStateUpdater());

// Update the publication state
pTaxiUpdater->set_idNumber( 17 );

// Commit the changes atomically
pTaxiServant->modifyPublicationState(pTaxiUpdater);
```

API for Subscribing to an SDO

```
// Instantiate a CallbackInfo object which is used to
// exchange information between the main program and
// the callbacks
Example::Taxi::BasicImpl::CallbackInfoPtr
  pTaxiCallbackInfo(
    new Example::Taxi::BasicImpl::CallbackInfo );

// Instantiate an object to describe the type of SDO
// to subsribe to
Example::Taxi::SubscriptionInfoPtr
  pTaxiSubscriptionInfo(
    new Example::Taxi::BasicImpl::SubscriptionInfoImpl(
      pTaxiCallbackInfo ) );

// Subscribe to Taxis
pSession->subscribeToSDO< Example::Taxi::ProxyTraits >(
      pTaxiSubscriptionInfo );
```

Reading the PublicationState

- **Assume discovery occurred and the proxy was assigned to `pTaxiProxy`**
- **Using the "get" method to read the publication state**

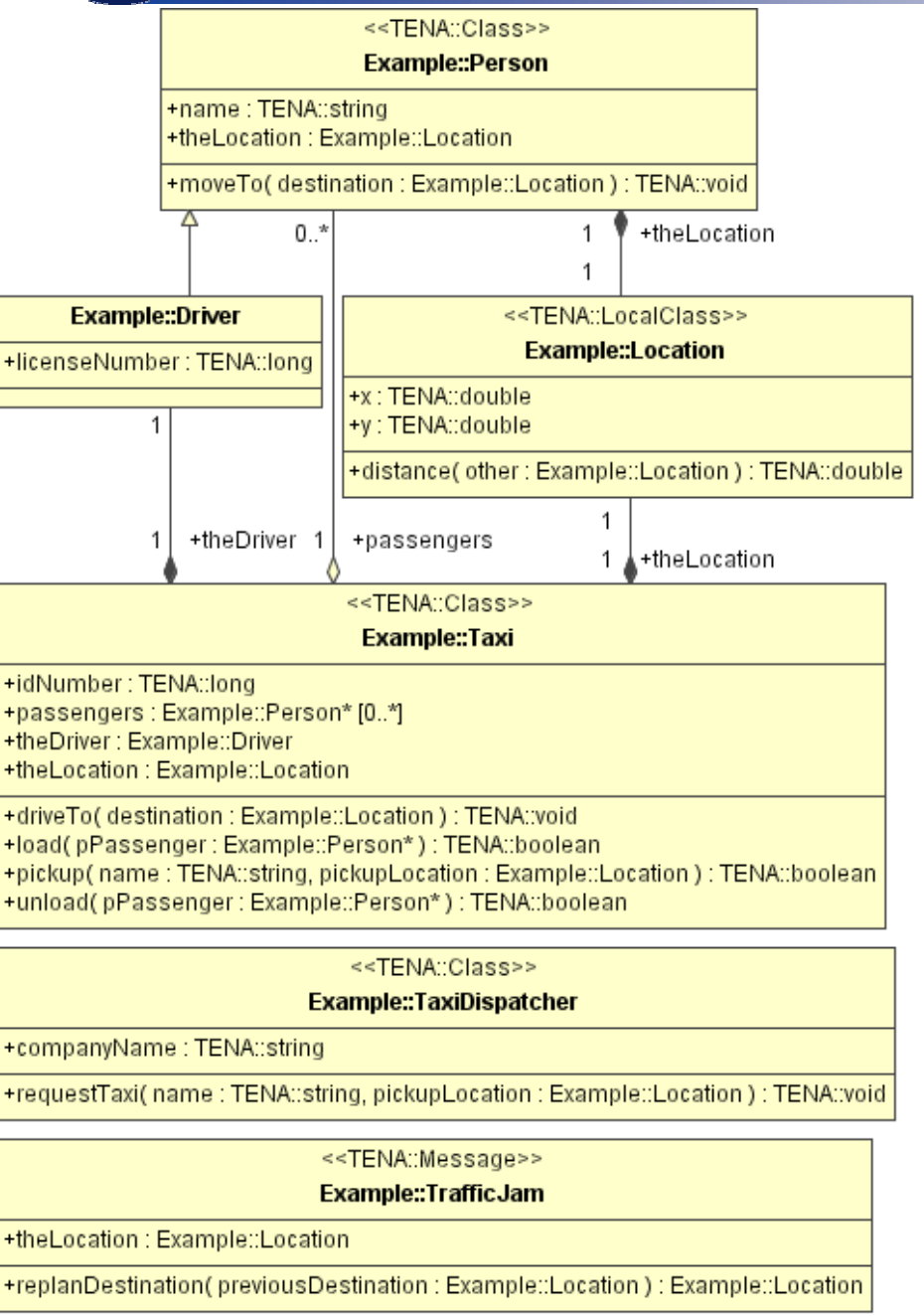
```
// Get a read-only copy of the publication state
Example::Taxi::ImmutablePublicationStatePtr
  pTaxiPublicationState (
    pTaxiProxy->getPublicationState() );

// read an attribute
 long taxiIDnumber(
   pTaxiPublicationState->get_idNumber() );
```

- **Or do it all at once:**

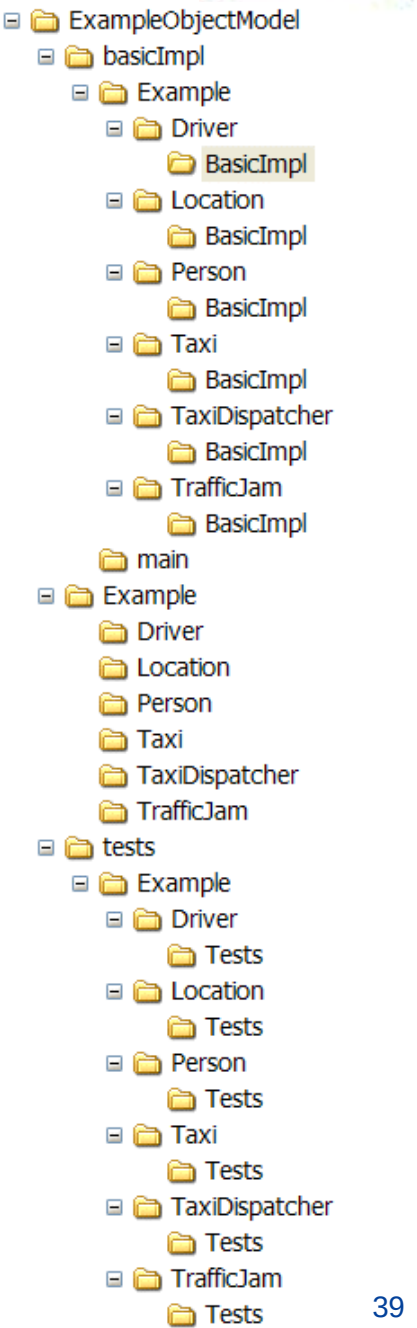
```
std::cout << "Taxi idNumber is "
  << pTaxiProxy->getPublicationState()->get_idNumber()
  << std::endl;
```

Auto Code Generation



The TENA Middleware object model compiler turns this example object model into 1064 files in 37 directories.

Note that only a fraction of these files are “user API” files.



Auto Code Generation

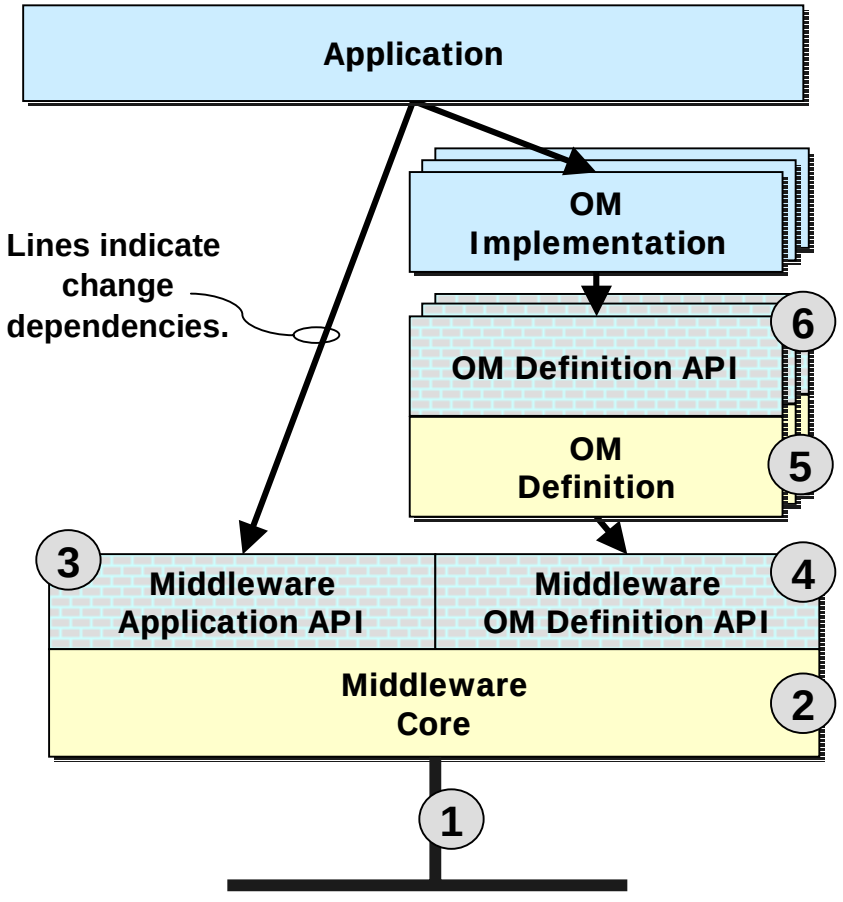
- **CORBA has been using auto-code generation for over a decade**
  - But it's still requires a reasonably high level of expertise to write a correct CORBA application.
  - And a correct real-time CORBA application requires even more expertise!
- **The TENA Middleware uses auto-code generation to generate IDL and the accompanying CORBA code to provide the high-level programming abstractions of the TENA meta-model**
- **The TENA Middleware also auto-code generates example applications**
  - Ease of use means more programmers can use the TENA Middleware
    - Auto-code generation of entire applications greatly lowers the barrier to entry for the average programmer that needs to create distributed, real-time applications.
  - Rapid application development
    - Auto-code generation of entire applications greatly reduces the time required to actually start getting work done
  - Improved application reliability
    - Auto-code generation of entire applications greatly reduces the likelihood of programming errors that can't be detected at compile time.

Remaining Challenges for TENA Middleware-based Large-scale DRE Systems

- **Monitoring & Diagnostic Tools**
- **Distributed Debugging Tools**
- **Deployment & Configuration Tools**
  - How many computers connected via how many LANs should be used to support an envisioned large-scale DRE system?
  - How should a given set of applications be deployed on a given computer network to best (or even adequately) support an envisioned large-scale DRE system?
- **Model-Driven Custom Application Generation**
  - Currently, example applications are generated that must then be hand-customized by the programmer.
- **Automated Application Maintenance and Porting**
  - Currently, only primitive tools and techniques are used to update applications when the object model is modified, when a new TENA Middleware version is released, etc.

Why is Automated Application Maintenance and Porting Necessary

There are a lot of moving parts in a TENA application!



1 – Middleware interoperability change
- Apps – need to upgrade
- **OM defs – no change**
- **OM impls – no change**

2 – Internal middleware change
- **Apps – no change**
- **OM defs – no change**
- **OM impls – no change**

3 – Middleware user API change
- Apps – change required
- **OM defs – no change**
- **OM impls – no change**

4 – Middleware OM API change
- **Apps – no change**
- OM defs – change required
- **OM impls – no change**

5 – OM definition change
- **Apps – no change**
- **OM defs – change implied**
- **OM impls – no change**

6 – OM definition API change
- **Apps – no change**
- **OM defs – no change**
- OM impls – change required

Acknowledgments

The TENA Middleware was developed for the Foundation Initiative 2010 Project Management Office [http://www.fi2010.org/] under contract 1435-04-01-CT-31085 in which the regulations from DFARS 252.227-7013 and 252.227-7014 are enforced. This presentation describes the work of a team of software professionals, including not just the presenter, Russ Noseworthy, but also the members of TENA Middleware development team: Steve Bachinsky, Jon Franklin, Marvin Greenberg, Braden McDaniel, Sean Paus, Ed Powell, and Ed Skees, as well as our new team member, Keith Snively, and alumni's Jon Franklin and Mike Mazurek. The entire TENA Middleware development team would like to thank the FI 2010 project team, in particular George Rumford, Jason Lucas, Gene Hudgins, Jerry Santos, Kevin Alix, Kurt Lessmann, Dee Beddard, Stephanie Clewer, and Keith Poch.