EXHIBIT 1009

U.S. Patent Publication No. 2003/0058277 to Bowman-Amuah ("BowmanAmuah") (Part 3 of 3)

pulling data into the system, while the core business functionality is performed by passive filters.

[3207] Because active and passive filters demonstrate different levels of pro-activity, it is useful to further break down the type of consumers and suppliers into four general types: push suppliers, pull suppliers, push consumers and pull suppliers. These four simple abstract interfaces help segregate the fundamental, yet disparate, behaviors. Active filters inherit both from PullConsumer and PushSupplier. Active filters' sources inherit from PullSupplier, and their destinations inherit from PushConsumer. Passive filters inherit from PushConsumer and PushSupplier. Passive filters' sources inherit from PushSupplier, and their destinations inherit from PushConsumer.

[3208] Pipes

[3209] While filters define the basic processing steps, pipes define how to flexibly configure the system. Pipes can be used to connect filters in a wide range of configurations.

[3210] Acting as a choke point for data to be pulled from an active filter

[3211] Connecting serial filters defined as independent processes

[3212] Multiplexing to/demultiplexing from several filters of the same type running in parallel

[3213] Pipes may use buffering, multiplexing and demultiplexing techniques in order to transfer data between filters. Some examples of useful pipe implementations include:

[3214] Channeled Pipes. Perhaps the most generally useful form of a pipe is based on the CORBA Event Channel object, which can connect any number of Push/Pull Suppliers to any number of Push/Pull Consumers.

[3215] Multithreaded Pipes. These pipes route data to one of several filter threads. The data can then be joined back to the primary thread on the other end of the filter with a demultiplexing pipe.

[3216] Database Queue Pipes. These pipes wrap around a database queue to enable seamless data transfer between processes.

[3217] The various command shells enable filter programs to be tied together into a Processing Pipeline.

[3218] Collaborations

[3219] Abstraction Factory. Often filters will need to produce new data objects from input but are only aware of the data's abstract interfaces. As a result of this generality, the filters will need to utilize an abstraction factory to produce concrete objects without knowing their concrete class types.

[3220] Business Logic Service Patterns (1024)

[3221] As is stated in the Component Technology Architecture Framework, "Business components are the core of any application, they represent concepts within the business domain. They encapsulate the information and behavior associated with those concepts. Examples of business components include, Customer, Product, Order, Inventory, Pricing, Credit Check, Billing, and Fraud Analysis." These are the components that in many cases have been the most elusive for reuse but hold the highest promise for attacking the cost of development. In this area there are at least three targeted categories of business components, Common Business Components, Common Business Services and Common Business Facilities.

[3222] Common Business Components are those components from the preceding list that encapsulate key business concepts. At one level these components represent cross application components that are common to a plethora of applications. These include concepts like Customer, Company, Account, Shipment, etc. These common components normalize how basic behavior surrounding common business concepts can be normalized. Common Business Components are very concerned about the validity of the relationships they have with other components and ensuring that the information relationships are maintained correctly.

[3223] Common Business Services deal with the higher level services that abstract out the "Business Unit of Work" or more transactional aspects of business processing. Having components that capture key processing concepts normalizes the processes for handling business events. These are services like credit checks, ordering, servicing problems, shipping, product selection, etc. They tend to capture business practices and when reused enable a company to increasingly leverage the value of these practices.

[3224] Common Business Facilities are those services that deal with areas of more engineering component type reuse. These include base common facilities like reason codes, currency management, telephone and address manipulation and validation of these common business types.

[3225] How do the patterns in this section help?

[3226] The patterns described in this section represent some initial attempts to capture basic concepts that are useful in the area of Common Business Facilities. They are by no means exhaustive but represent building blocks in a complete solution. Both provide tremendous value in solving two key challenges which appear on every engagement.

[3227] The Constant Class pattern describes a facility for ensuring correct data at the attribute level.

[3228] The Attribute Dictionary describes a facility for encapsulating architectural mechanisms within business objects.

[3229] Attribute Dictionary

[3230] FIG. 58 illustrates a flowchart for a method 5800 for controlling access to data of a business object via an attribute dictionary. A plurality of attribute values for a business object are stored in an attribute dictionary in operation 5802. A plurality of attribute names are provided in the attribute dictionary for the stored attribute values in operation 5804. Next, in operation 5806, it is verified that a current user is authorized to either set or get one of the attribute values upon a request which includes the attribute name that corresponds to the attribute value. The attribute value in the attribute dictionary is obtained or updated if the verification is successful and an indicator is broadcast upon the attribute value being updated in operations 5808 and 5810.

[3231] In one embodiment, a list of the attribute names may be outputted in response to a request. Additionally, the

list may also include only the attribute names of a portion of the attribute values of the business object that are present.

[3232] In one aspect, the attribute values may be obtained for auditing or rollback purposes. In another aspect of the present invention, a dirty flag may be set upon the attribute value being updated.

[3233] Typically, business objects include "getter" and "setter" methods to access their data. How can I support value-added processing, such as logging events for changes, without impacting application code?

[3234] Typically, business objects store attributes in instance variables. The application code for a typical setter for an attribute is depicted as:

```
public void setBalance(Float newBalance) {
    myBalance = newBalance;
    return;
}
```

[3235] Initially, this is straightforward. However, after all of the attribute setters and getters have been coded, the need may arise for an event to be broadcast each time an attribute is updated. The code for a simple setter would need to change to become:

```
public void setBalance(Float newBalance) {
    myBalance = newBalance;
    this.notifyChanged("Balance");
    return;
}
```

[3236] Now each attribute setter must contain the call to the 'notifychanged' architecture method. This implementation forces architecture mechanisms to be intrusive to application code. Moreover, addition or extension of architecture processing should not impact business logic. One new line of code alone may not seem like a large burden on application developers. However, many other architecture requirements might later affect each setter or getter.

[3237] As another example, before updating an attribute, a check may be required to determine if the current user has security rights to update attributes. Also, after successful update, a dirty flag may be set, or an audit log may be performed. The code for each setter now looks as follows:

```
public void setBalance( Float newBalance ) {
    // keep track of any original balance,
    // for post-change processing, then do
    // some pre-processing to check
    // that the user has access rights
    Float oldBalance = myBalance;
    this.assertUserAttributes( "Balance" );
    // finally update the balance, then
    // broadcast, set the Dirty Flag,
    // and log
    myBalance = newBalance;
    this.notifyChanged( "Balance" );
    this.setDirty( );
    this.logChanged( "Balance", oldBalance );
}
```

[3238] Thus, each added architecture framework for gets and sets must be manually added to all getters and setters. Such changes impact application developers during coding and maintenance. Moreover, they also complicate business logic with technical details.

[3239] Therefore, the application architecture should control access to a business object's data. This will separate out reusable, technical, architecture details. Business objects should use an Attribute Dictionary to provide an architectural hook for attribute getters and setters. Moreover, this framework should handle all architectural processing related to the update and access of data, transparently to application logic.

[3240] Rather than using instance variables, the Attribute Dictionary holds all attribute values for the object. This dictionary is a collection, keyed by attribute names. Then the architecture can provide generic architecture methods to get and set attributes in the dictionary.

[3241] Business objects could each delegate directly to the Attribute Dictionary within the attribute getter and setter. However, rather than having each business object talk directly to the Attribute Dictionary, simple helper methods can be created in a superclass for business objects. This simplifies the interface for application developers, who do not need to know about the Attribute Dictionary. This also allows for business object specific logic to also be added prior to and after the dispatch to the Attribute Dictionary.

[3242] The code for a simple setter now would look like:

```
class Account extends BusinessObject {
public void setBalance( Float newBalance ) {
    // set my balance with the new value
    // passed in. The architecture will handle
    // any technical details related to
    // setting the data.
    this.setAttribute( "Balance", newBalance);
}
}
```

[3243] The architecture superclass will then perform the following:

[3244] get the original value, perhaps for auditing or rollback purposes

[3245] check if the user has security access to set the attribute

[3246] update the attribute on the Attribute Dictionary

[3247] if successful, broadcast and log the change

[3248] The Attribute Dictionary would then contain the code to:

[3249] update the value for the given attribute name

[3250] set the dirty flag

[3251] This illustrates that both the superclass facade and the Attribute Dictionary can have different processing. In

general, one generic location for getting and setting attributes supports (but is not limited to).

[3252] logging

[3253] broadcasting

[3254] dirty flag

[3255] security checking

[3256] NULL field value handling

[3257] This logic will be either in the facade methods (for any code that is business object specific), or the generic methods on the dictionary, thereby shielding developers from this added complexity.

[3258] Benefits

[3259] Maintainability. Architecture code can be added and changed in one place for all objects, without change to the application code.

[3260] Flexibility. The implementation of the storage mechanism can be changed as needed to improve performance.

[3261] Readability. The methods used in application code to retrieve and update fields on the object are generic. These methods do not have excess architecture code to detract from the purpose of the method.

[3262] Object Model

[3263] **FIG. 60** illustrates the manner in which the AttributeDictionaryClient **6000** is the facade which delegates to the AttributeDictionary **6002**. For example, business objects would inherit this behavior. AttributeDictionaryClient **6000** probably wouldn't be the immediate superclass, but it would be somewhere in the hierarchy. In this manner, stateful business objects, like Account or Customer, can easily take advantage of the Attribute Dictionary.

[3264] The attributeValues attribute on the Attribute Dictionary is shown as an instance of the HashMap class **6004**, which stores key value pairs. The HashMap Collection is used to provide access to attribute values based on the attribute name. This is required for a direct lookup of values associated with attribute names. Such lookup can use string representation of the attribute names.

[3265] Object Interaction Diagrams

[3266] There are four interactions for this framework: Simple Attribute Getter, Simple Attribute Setter, Failed Attribute Getter, and Retrieval of Attribute Names. **FIG. 61** illustrates the internal implementation of the dictionary.

[3267] **FIG. 61** depicts the use of the containsKey( ) method **6100** on the HashMap to ensure that the value will exist before the get( ) method is used. This proactive search for the value ensures that the nullPointerException is not thrown from the AttributeDictionary. The performance of such methods will be checked during testing. If such processing is not performant, the code can be altered and the call to containsKey( ) removed. In that case, the HashMap will need to wrap a try-catch block around the get( ) method. **FIG. 62** illustrates a method **6200** that dictates that any nullPointerException that is thrown would be caught and rethrown as the more user-friendly exception in the attribute dictionary pattern environment. **FIG. 63** illustrates the Get the Attribute Names method **6300** in the attribute dictionary pattern environment.

[3268] Public Interface

[3269] The following are methods on the AttributeDictionary. The AttributeDictionaryClient exposes similar public methods.

[3270] public Object getAttribute(String attributeName) raises AttributeNotFoundException;

[3271] The return value of getAttribute( ) is typically a wrapped primitive, or Java type, for most attributes. This includes, for example, an account balance (Float) or account number (String). The return value of these wrapped primitives must be cast, as illustrated in the following example:

```
class Account extends BusinessObject {
    public Float getBalance( ) {
    // get my balance using the superclass facade
    // cast the return value before returning it
    return (Float)this.getAttribute( "Balance" );
    }
}
```

[3272] Other methods on the AttributeDictionary include:

[3273] public void setAttribute(String attributeName, Float attributeValue);

[3274] public void setAttribute(String attributeName, String attributeValue);

[3275] public void setAttribute(String attributeName, BusinessObject attributeValue);

[3276] These overloaded methods create a generic interface to the AttributeDictionary for attribute setters. They ensure type checking, such that no attributes will be set to a value other than those for which an overloaded method exists.

[3277] public String[ ] attributeNames( );

[3278] The method attributeNames( ) returns a collection of the names of only those attributes that have been populated (or set) on the dictionary. This might be useful for other frameworks, which may want to iterate over all attributes. At any particular time, a business object may not contain all of its attributes (e.g., because of partial retrieval from the database). So this may be a subset of the full attribute list for the object.

[3279] Constant Class

[3280] **FIG. 64** illustrates a flowchart for a method **6400** for managing constants in a computer program. In operation **6402**, a plurality of constant names are provided with each constant name having a corresponding constant value. The constant names are grouped into constant classes based on an entity which the constant values represent in operation **6404**. Access is allowed to the constant values in operation **6406** by receiving a call including the corresponding constant name and corresponding constant class.

[3281] In one aspect, the constant values may be changed upon being accessed. In another aspect, the constant value may also include an enumeration. Also, in one embodiment, accessor logic modules may be assigned to a plurality of the named constants with the accessor logic modules being executed upon the accessing of the corresponding constant value via the accessor logic module. Also, the accessor logic modules may be edited per the desires of a user. Additionally, the constant values may be accessed without the accessor logic modules.

[3282] Literals are hard-coded constants referenced in multiple places. How can source code refer to literals in a maintainable fashion?

[3283] The concept and value of named constants have been realized for quite some time. The idea can date back to Assembler language naming memory locations where data was stored. The purpose is to give the ability to refer to fixed values by the name of what they represent rather than by the quantity they are set to.

[3284] Named constants allow a programmer to "parameterize" a system. This allows a programmer to change a constant's value in a single place rather than every place the constant is used. In addition to the maintenance gain, readability is also increased.

[3285] Many languages offer mechanisms to implement named constants. These include PoolDictionaries (Smalltalk), enums (C and C++) and public static final declarations (Java). Difficulties arise during implementation of these mechanisms with respect to type constraints, visibility, and type checking.

[3286] Using these traditional approaches, results in global namespace for these literals. This can result in name collisions. For example, the name HIGH to define a large magnitude could translate into different values for different uses. A HIGH temperature could be 95 while a HIGH altitude could be 39000.

[3287] In addition, constants often belong to logical groupings. For instance, STOCK, BOND, and OPTION are all types of financial instruments. These belong in a some sort of collection.

[3288] A consistent, quality method to represent constants in an object-based system is required.

[3289] Therefore, represent named constants in a separate class, grouping categories of constant values together within one name space. Constants tend to naturally fall into logical groupings. Each grouping should be represented by its own class. For instance, all of the constants used by a PhoneNumber object to capture the various types of PhoneNumber (i.e. home, business, fax, cell, pager, etc) can be accessed through a PhoneTypeConstants class.

[3290] If constants are obtained by other means than explicit language constructs like "public final int HOME_ADDRESS" than public accessors are used to insulate a client from changes in how the constant is obtained. In this case the values of each of the constants should be defined privately inside the Constant Class. Public accessors are then provided for clients to obtain the constant values. This allows for "changing constants". Business-related values that may seem constant at design and construction time very often are not. Some of these "constants" may eventually require some logic to determine their value. If clients obtain constants through accessor methods, no changes (except within the accessor) will have to be made if the logic is added. This is a particularly safe practice when programming rules dictate all constants to be stored and retrieved from database tables.

[3291] In the case where constants are defined within the class itself most OO languages, excepting Smalltalk, allow for some type of const definition. In this case by using a const construct (i.e. static final int PhoneNumberType.FAX= new PhoneNumberType( )) it is not necessary to have public accessors and private definitions. Declare the class type, create static final instances of the type and do not provide a public constructor. This ensures the type safety and provides easy to access members in the Eiffel style.

[3292] Moreover, public accessors in either strategy provide for type-safe enumerations. Enumeration is a special type of constant that deserves attention. A TypeConstant class can provide enumeration by implementing some key methods that provide for supporting iteration over the elements of the enums. In Java, for example, this entails implementing the Enumeration interface.

[3293] Benefits

[3294] Maintainability. Groups all valid values together and ensures they can not be created or passed as parameters by any other method.

[3295] Type Safety. Enumeration values can be type-checked by a compiler in method parameters and return values.

[3296] A common application pattern where this use of constants was applied was in the modeling of instances vs instance types where the types added no additional behavior. In two different customer care applications this came through as the objects like PhoneNumber, PhoneNumberType, RatePlan & RatePlanType, etc. This example has not yet been updated to JavaBeans.

```
package Entity;
import java.util.*;
public class PhoneNumberType {
    static final Vector types = new Vector( );
    static final PhoneNumberType FAX = new PhoneNumberType(0,
"Fax");
    static final PhoneNumberType CELL = new PhoneNumberType(1,
"Cell Phone");
    static final PhoneNumberType HOME = new PhoneNumberType(2,
"Home");
    static final PhoneNumberType WORK = new PhoneNumberType(3,
"Work");
    static final PhoneNumberType PAGER = new PhoneNumberType(4,
"Pager");
    private final int phoneNumberTypeOrd;
    private final String typeId;
    private PhoneNumberType(int i, String id) {
        phoneNumberTypeOrd = i;
        typeId = id;
        types.addElement(this);
    }
    public final static Enumeration elements( ) { // allows for
enumeration
        return types.elements( );
    }
    public static void main(String args[]) {
        Enumeration elements = PhoneNumberType.elements( );
```

-continued

```
        PhoneNumberType pt;
        while (elements.hasMoreElements( )) {
            pt = (PhoneNumberType)elements.nextElement( );
            System.out.println(pt.toString( ));
        }
    }
    public String toString( ) {
        return typeId;
    }
}
```

[3297] This type partition is used by *PhoneNumber*. See main( ) for uncommenting a line that demonstrates the type safety protection through the use of static final and private constructors.

```
package Batty;
import java.io.PrintStream;
import java.io.StringWriter;
public class PhoneNumber
{
    private PhoneNumberType phoneNumberType;
    private String areaCode;
    private String prefix;
    private String suffix;
    public PhoneNumber( ) {
    areaCode = null;
    prefix = null;
    suffix = null;
}
public PhoneNumber(String aPhoneNumber)
{
    parsePhoneNumber(aPhoneNumber);
    setPhoneNumberType(PhoneNumberType.HOME);
}
public PhoneNumber(String anAreaCode, String aPrefix, String aSuffix)
{
    areaCode = anAreaCode;
    prefix = aPrefix;
    suffix = aSuffix;
}
public String areaCode( )
{
    return areaCode;
}
public void areaCode(String anAreaCode)
{
    areaCode = anAreaCode;
}
public boolean equals(String aPhoneNumber)
{
    PhoneNumber tempPhoneNumber = new
PhoneNumber(aPhoneNumber);
    return equals(tempPhoneNumber);
}
public boolean equals(PhoneNumber aPhoneNumber)
{
    if (areaCode( ) == null && aPhoneNumber.areaCode( ) != null ||
aPhoneNumber.areaCode( ) == null && areaCode( ) != null)
        return false;
    if (areaCode( ) != null)
    {
        if (areaCode( ).equals(aPhoneNumber.areaCode( )) &&
prefix( ).equals(aPhoneNumber.prefix( )) &&
            suffix( ).equals(aPhoneNumber.suffix( )))
            return true;
        else
            return false;
    }
```

-continued

```
    if (prefix( ).equals(aPhoneNumber.prefix( )) &&
suffix( ).equals(aPhoneNumber.suffix( )))
        return true;
    else
        return false;
}
public static void main(String args[])
{
    PhoneNumber aPhoneNumber;
    System.out.println("Testing construction & comparisons");
    if (args.length == 0)
    {
        System.out.println("Test with no area code - no masks");
        aPhoneNumber = new PhoneNumber("5578255");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.FAX);
        System.out.println(aPhoneNumber.toString( ));
        System.out.println("Test with area code - no masks");
        aPhoneNumber = new PhoneNumber("2055572580");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.WORK);
        System.out.println(aPhoneNumber.toString( ));
        System.out.println("Test with normal masks");
        aPhoneNumber = new PhoneNumber("(216) 557-9580");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.PAGER);
        System.out.println(aPhoneNumber.toString( ));
        System.out.println("Test equality 5578255 557-8255");
        PhoneNumber temp1 = new PhoneNumber("5578255");
        temp1.setPhoneNumberType(PhoneNumberType.CELL);
        PhoneNumber temp2 = new PhoneNumber("557-8255");
        temp2.setPhoneNumberType(PhoneNumberType.CELL);
        // temp2.setPhoneNumberType(new PhoneNumberType(8,
"TOY")); // assert type safety w/private ctor
        System.out.println(temp1);
        System.out.println(temp2);
        System.out.println("temp1 == temp2: " +
temp1.equals(temp2));
        return;
    }else {
        aPhoneNumber = new PhoneNumber(args[0]);
        System.out.println(aPhoneNumber.toString( ));
    }
}
private void parsePhoneNumber(String aPhoneNumber)
{
    StringBuffer sBfr = new StringBuffer(aPhoneNumber.length( ));
    int i = 0;
    do
        if (Character.isDigit(aPhoneNumber.charAt(i)))
            sBfr.append(aPhoneNumber.charAt(i));
    while (++i < aPhoneNumber.length( ) - 1);
    String tempString = new String(sBfr);
    if (tbfr.length( ) == 7)
    {
        prefix(tempString.substring(0, 3));
        suffix(tempString.substring(3, 7));
        return;
    }
    areaCode(tempString.substring(0, 3));
    prefix(tempString.substring(3, 6));
    suffix(tempString.substring(6, 10));
}
public String prefix( )
{
    return prefix;
}
public void prefix(String aPrefix)
{
    prefix = aPrefix;
}
/**
* This method was created by a SmartGuide.
* @param sw StringWriter
*/
public void printOn (StringWriter sw) {
    sw.write((areaCode != null || areaCode == "") ? ("(" +
areaCode( ) +
") ") :"");
```

-continued

```
        sw.write(this.prefix( ));
        sw.write("-");
        sw.write(this.suffix( ));
        return;
}
public void setPhoneNumberType(PhoneNumberType pnt) {
    phoneNumberType = pnt;
}
public String suffix( )
{
    return suffix;
}
public void suffix(String xSuffix)
{
    suffix = xSuffix;
}
public String toString( )
{
    return new String(phoneNumberType.toString( ) + ": " + ((areaCode
!= null || areaCode == "") ? "(" +
areaCode( ) + ")" : "") + prefix( ) + "-" +suffix( ));
}
}
```

[3298] Alternatives

[3299] Smalltalk allows for grouping logical constants in PoolDictionaries as in TextConstants. This is simply a global dictionary with key value pairs that simplifies and improves readability by using well understood names like "Space" and "Tab". However, they are global variables and they are not automatically recreated when you file in code that depends on them.

[3300] When constants are implemented in a class within Smalltalk accessors must be used. There is no real language notion of final or const in Smalltalk that would allow for accessing member variables.

[3301] Communications Services Patterns (1008)

[3302] An original tenet of component-based design has been simplified distribution of functionality. According to the original argument, up-front definition of component boundaries and their interfaces would simplify the configuration of functionality on the network. Even though component-based design has simplified distribution, it has not guaranteed success. Networks introduce performance issues and failure conditions that didn't exist in non-distributed solutions. If a solution is to be successful, these issues can't be ignored.

[3303] Each pattern in this section addresses a difficulty associated with distributed computing. Every pattern reflects a problem and a solution to issues encountered by other development teams.

[3304] Legacy systems running on mainframes, Unix boxes, etc. are an important part of today's client projects. The majority of today's clients have existing computer systems that cannot be easily rewritten or ignored. Integration of these older systems with the newly developed applications is often imperative to the success of the project. Any newly developed components must leverage the existing functionality on these Legacy systems. The Legacy Wrapper pattern addresses this problem. It describes a common pattern for tackling the integration issues associated with reusing functionality from existing systems.

[3305] Server-side components implement services for use by the Clients in an application. These components should clearly specify the interfaces and services they provide, but how should they make them available? A well-known central service, e.g., a Name Service or Trader Service can be used to make the interfaces available to all Clients, but is that always warranted or prudent? Should every Client have access to every service? The Locally Addressable Interface (LAI) and Globally Addressable Interface (GAI) patterns describe two approaches to this problem.

[3306] The performance characteristics of remote components are very different from "in process" components. The cost of requesting and transmitting data between remote components is much higher and should be considered in a distributed solution. As a result, distributed solutions often call for communication patterns that improve upon the performance aspects most important to the system. The Structure Based Communication pattern addresses the "chattiness" associated with distributed applications. It helps reduce network load and increases system response time. The Paging Communication pattern addresses the common need to retrieve and display large lists of data. It shows how incremental fetching can be used to provide much better perceived responsiveness in GUI based applications.

[3307] The cost of locating a remote service and establishing a connection to that service can also be a costly endeavor. The Refreshable Proxy Pool pattern describes a robust and efficient way to minimize this "lookup" activity.

[3308] Most recent component-based systems use middleware such as CORBA, DCOM or Java RMI to specify the interfaces provided by components and the associated data types. However, such middleware is not always available, or directly applicable. In such situations the Stream Based Communication pattern, or one of its descendants, the Fixed Format Stream and SelfDescribing Stream patterns might be applicable. These patterns describe different techniques for efficiently streaming data between processes. While they all share a common solution to a common problem, the solutions present different tradeoffs between implementation simplicity, performance and flexibility.

[3309] A Null value represents the "empty set" and is an important value in distributed component solutions.

[3310] Some languages, such as Java, support Null as a specific value, whereas other languages do not (e.g., C++ which uses zero and context to determine Null). This language mismatch can cause problems in distributed systems that use more than one language. The Null Structure pattern describes this problem and proposes a solution.

[3311] Fixed Format Stream

[3312] FIG. 65 illustrates a flowchart for a method 6500 for providing a fixed format stream-based communication system. In operation 6502, a sending fixed format contract or interface code is defined for a sending system. A receiving fixed format contract or interface code is also defined for a receiving system. A message to be sent from the sending system to the receiving system is translated in operation 6504 based on the sending fixed format contract. The message is then sent from the sending system and subsequently received by the receiving system in operations 6506

and **6808**. The message received by the receiving system is then translated based on the receiving fixed format contract in operation **6510**.

[3313] In one embodiment of the present invention, information in the translated message received by the receiving system may also be stored in a relational database. In one aspect, the fixed format contracts may be included in meta-data of the message. Also, in another aspect, the message may include an indication of a version thereof.

[3314] In one situation, one of the systems is an object-based system and one of the systems may be a non-object-based system. In another situation, both of the systems are may be object-based systems. In a third situation, both of the systems may be non-object-based systems.

[3315] Stream-based communication is a very effective pattern for relaying data, data structures, and meta-data. Meta-data is information about the data like data structure, data types, etc. using a shared, generic format. How can the message format be shared between systems so as to create the most straightforward and best performing stream-based mechanism?

[3316] Often, it is determined that a stream-based communication mechanism should be used to transport information between systems. Stream-based communication is a pattern where information is transported from one system to another using a simple stream and a shared format to relay the data structure and meta-data information.

[3317] **FIG. 66** illustrates two systems **6600** communicating via a stream-based communication **6602** and using a common generic format to relay the meta-data information.

[3318] However, when implementing Stream-based Communication, a number of factors influence the method for enabling each system with a "shared format." The "shared format" provides the meta-data information needed to interpret the raw data in a stream. This shared format is like a secret decoder ring for systems sending and receiving messages. It allows the systems to convert structured data (objects, strings, etc.) into raw data and raw data back into structured data. This is needed to transmit the structured data across the network.

[3319] On many projects, the following factors influence the details of communicating using a stream.

[3320] High performance—System performance is always a factor, but sometimes it is one of the most important factors in a system.

[3321] Short development time—The system must be operational in the shortest possible timeframe.

[3322] Stable information characteristics—In some solutions, the data and the structure of the data are stable and unlikely to change.

[3323] In cases like this, how can one optimize the benefits of stream-based communication and implement only the most basic capabilities that one requires?

[3324] Therefore, use the Fixed Format Stream pattern to create a stream-based message that uses fixed format contracts to share the formatting information and meta-data between systems.

[3325] Fixed format contracts are maps that contain the meta-data information such as the data structure, data separators, data types, attribute names, etc. They describe how to translate Fixed Format messages onto a stream and off of a stream.

[3326] **FIG. 67** illustrates an example of a Fixed Format message **6700** associated with the fixed format stream patterns. The location and size of each attribute in the message is fixed and known at design time. In the example below, it is known that the command will be in bytes **1-9**, the first name will be in bytes **10-29**, the last name in bytes **29-49**, etc. This information (meta-data) is used by the Fixed Format contracts to convert Fixed Format messages from data structures to raw data and back again.

[3327] **FIG. 68** depicts the complete Fixed Format Stream pattern associated with the fixed format stream patterns. A data structure on System A **6800** is translated to a Fixed Format message (raw data) using a Fixed Format contract. The message is put in the stream **6802** and sent to System B **6804**. System B **6804** receives the Fixed Format Message (raw data) and uses its Fixed Format contract to recreate the data structure. The same process works in reverse when System B **6804** responds to the message request.

[3328] Benefits

[3329] Performance. Because there is no time spent on look-ups or dynamic translation of the message, performance is better than with other variations of Stream-Based Communication.

[3330] Small Message Size. Each Fixed Format message contains only data to be sent to the other system. These messages contain no meta-data and are smaller than those in Self-Describing Streams.

[3331] Simplicity. Translating and parsing information onto and off of the stream is straightforward and easier than with the other variations of Stream-Based Communication. The behaviors for the Fixed Format Streaming are contained in the fixed format contracts on the interfaces of the sending and receiving systems and thus easy to find.

[3332] Object Friendly. This pattern is very straightforward to implement in object based systems. The objects contain the fixed format contracts and manage the translation and parsing onto the stream. These objects can access their own private behaviors which makes the interface much simpler.

[3333] Implementing this pattern is very straightforward. Define corresponding fixed format contracts on the interface code of both the sending and receiving systems. **FIG. 69** illustrates fixed format contracts **6900** containing meta-data information for translating structured data onto and off of a stream.

[3334] In non-object systems, define a fixed format contract on the parsing interface module of the sending system. The interfacing module on the sending system can use the contract as a map for how to translate and write the data onto the stream. Define a corresponding fixed format contract on the interface modules of the receiving system. The interface module on the receiving system can use the contract to read and translate the data off of the stream.

[3335] In object-based systems, make each object responsible for its own fixed format contract. Using this contract, each object is able to retrieve and parse its attribute values onto a stream as strings (streamOn) and each object class should be able to parse attributes off of a stream and put them into a new instance of an object (streamOff). Also, it is good practice to include the version of the format within the stream so that concurrent format versions can be accommodated in the design.

[3336] Below is a pseudo-code example of an object-based system communicating with a non-object system using stream-based communication and a fixed format contract.

[3337] **FIG. 70** illustrates a Customer object **7000** in an object-based system **7002** streaming itself into a stream **7004**, the stream being sent to a non-object system **7006**, this stream being read and the data inserted into a relational database **7008**.

[3338] 1. The CustomerObject with attributes name, sex, and age has a method "streamOn: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOn:" method goes through each of the object's attributes and parses each values as a string onto the stream.

[3339] The fixed format contract here is embodied in the order that this method parses the attributes onto the stream. A pseudo-code example in Java is the following: Note: Assume that "asString( )" converts the receiver to a string and that "padWithSpaces( )"pads the string with spaces and makes the string the length specified.

```
/** Stream my attribute values on aStream **/
public void streamOn (OutputStream aStream)
{
    aStream.write(this.getName( ).asString( ).padWithSpaces(10));
    aStream.write(this.getSex( ).asString( ).padWithSpaces(7));
    aStream.write(this.getAge( ).asString( ).padWithSpaces(5));
}
```

[3340] 2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.

[3341] 3. Once at the non-object system, interface code reads through the stream, parses the values off of the stream, converts them to the appropriate types if required, and puts them in a copybook with the appropriate structure. In this example, the fixed format contract is embodied in the structure and type of the WS-SHARED-FORMAT-CUSTOMER working-storage copybook. Refer to the pseudo-COBOL example below.

```
. . .
DATA DIVISION.
FD FILE-STREAM-IN
    RECORD CONTAINS 20 CHARACTERS
. . .
WORKING-STORAGE SECTION.
*** THIS COPYBOOK CONTAINS THE COMMON FORMAT OF
THE
*** CUSTOMER IN THE DATA STRUCTURE AND DATA TYPES
```

-continued

```
01 WS-COMMON-FORMAT-CUSTOMER.
   05 WS-COMMON-FORMAT-NAME        PIC X(10).
   05 WS-COMMON-FORMAT-SEX         PIC X(7).
   05 WS-COMMON-FORMAT-AGE         PIC 999.
*** THIS COPYBOOK IS THIS SYSTEMS VIEW OF A CUSTOMER
01 WS-CUSTOMER.
   05 WS-NAME                      PIC X(10).
   05 WS-AGE.                      PIC 999.
   05 WS-SEX                       PIC X(07).
. . .
PROCEDURE DIVISION.
. . .
*** OPEN THE FILE STREAM AND PUT THE CONTENTS IN THE
*** WS-COMMON-FORMAT-CUSTOMER COPYBOOK.
OPEN FILE-STREAM-IN
READ FILE-STREAM-IN INTO WS-COMMON-FORMAT-
CUSTOMER
    AT END CLOSE FILE-STREAM-IN
END-READ.
*** MOVE THE VALUES INTO FROM THE COMMON FORMAT
INTO
*** THE WS-CUSTOMER VARIABLES.
MOVE WS-COMMON-FORMAT-SEX      TO WS-SEX.
MOVE WS-COMMON-FORMAT-AGE      TO WS-AGE.
MOVE WS-COMMON-FORMAT-NAME     TO WS-NAME.
. . .
*** CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
*** RELATIONAL DATABASE
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER.
. . .
STOP RUN.
```

[3342] Conversely, a stream could be created by a non-object system (or another object-based system for that matter) and sent to one's object-based system. In this case, CustomerObject could use a "streamOff: aStream" method and instantiate a new instance of an aCustomerObject and populate it with the appropriate attribute values.

[3343] Eagle Architecture Framework: Uses Stream Based Communication in a number of ways. First of all, it uses it to embed tracing information in CORBA distributed requests. Second of all, it is used to replicate state between fault-tolerant services.

[3344] MCI: Invoice Development Workbench. This workbench helps MCI create error-free invoice definitions for the various Local Bells. Stream-based communication was used as part of an efficient, lightweight persistence mechanism.

[3345] Java Serialization: This is a Java defined fixed format for streaming objects.

[3346] Object Request Brokers (ORBs) that use CORBA, DCOM, or Java Remote Method Invocation (RMI)—ORBs that use one of these standards implement this pattern. They define an Interface Definition Language (IDL) that is the format or contract of the stream and use stream-based communication as the communication medium.

[3347] Collaborations

[3348] Stream-based Communication. This is the parent pattern to Fixed Format Stream. In this pattern, infor-

mation is transmitted using a simple stream and a shared, generic format. The Fixed Format Stream is a more specific implementation of Stream-Based Communication.

[3349] Structure Based Communication—This pattern uses a Fixed Format Stream to transmit data structure between systems. It is often used to obtain data from a Server for display in a Client UI.

[3350] Bridge (from the Gamma book Design Patterns) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collaborations between a business object and a format object while decoupling the business object from its specific stream format.

[3351] Abstract Factory (from the Gamma book Design Patterns) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

[3352] Alternatives

[3353] Self-Describing Stream. This pattern is a specific implementation of Stream-Based communication where the messaging format is parameterised and stored on the stream. A message language is used to read and write the format of the message from the stream.

[3354] Downloadable Format Stream—This pattern is a specific implementation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

[3355] Globally Addressable Interface

[3356] **FIG. 71** illustrates a flowchart for a method **7100** for delivering service via a globally addressable interface. A plurality of interfaces are provided in operation **7102** and access is allowed to a plurality of different sets of services from each of the interfaces in operation **7104**. Each interface has a unique set of services associated therewith. Each of the interfaces is named in operation **7106** with a name indicative of the unique set of services associated therewith. The names of the interfaces are then broadcast to a plurality of systems requiring service in operation **7108**.

[3357] The access may be allowed via structured-based communication. As another option, the names may be broadcasted using a naming service. Also, the naming service may provide the systems requiring service with a location of the interface on a network. In addition, the systems requiring service may be capable of looking-up the interfaces using the naming service.

[3358] In a client-server environment, a client makes requests of services on a Server. In such an environment, how might a Server expose its services for use by one or more clients?

[3359] In a typical two or three-tiered client-server application, the services are maintained away from the users (Client) on separate Server machines. Whenever a user needs to use a service, the user must send a request across the network to the Server machine

[3360] Before a client can utilize a service, it must find the service. If the client is unable to find the service, it can't ever use it. **FIG. 72** depicts a client **7200** that is unable to find the services provided by a server **7202** via a network **7204**.

[3361] The client could look for services in a naming service. However, if the services don't exist in the lookup or naming service, the client still can't find and use the service.

[3362] Therefore, use a Globally Addressable Interface to expose services to all available clients.

[3363] A Globally Addressable Interface builds upon the Interface pattern and the Naming pattern. When implementing a Globally Addressable Interface, a Server's operations are bundled into logical groups using the Interface pattern.

[3364] **FIG. 73** illustrates the grouping of services **7302** using interfaces **7304**.

[3365] For example, all the operations for accessing and viewing customer information (Get Customer, Get Customer Address, etc.) could be bundled into one interface. All the operations for changing customer information (Change Customer, Address, Change phone number, etc.) might be bundled into another interface. Keep in mind, this is an example "bundling" of operations and not the definitive method for bundling operations.

[3366] Once all the operations have been grouped into an interface, the interface is given a name appropriate to the operations it bundles. Then the interfaces are announced using the Naming pattern. The Naming pattern enables registration of interfaces in a globally available naming service. **FIG. 74** illustrates a customer server **7400** is publicly announcing its interfaces **7402**.

[3367] Until that time, a client can't find the operations and can't use them. Thus, the Server must use the Lookup or Naming pattern to register its interfaces (not methods). Once the interfaces have been registered with such a service, any client can go to the Naming Service, locate an interface, and access an operation in that interface.

[3368] Benefits

[3369] Public Addressability—Every Globally Addressable Interface is publicly available for use by any client. As a result, any Client can find these interfaces and access these operations.

[3370] Stateless Load Balancing. Globally Addressable Interfaces are generally implemented by stateless Servers. When Stateless Servers are used, it is a lot easier to balance the incoming load. Since state or context is always passed into the Server, any call can be directed to any Server that supports a particular operation. If one is busy, the Client can be forwarded on to the next one.

[3371] The following is a message trace diagram depicting the interactions associated with a Globally Addressable Interface.

[3372] The Message Trace diagrams depict a common Client-Server scenario. A client requests customer data from a Server. The Server finds the data in a database and forwards it back to the client. The Client can then display the data in a User Interface for a user.

[3373] The scenario was broken into two message trace diagrams. The first message trace sets the stage for the second. In the first message trace, the Server registers two Globally Addressable Interfaces with a Naming Service. The Client then "looks-up" an interface and establishes a connection to that interface.

[3374] Assumptions

[3375] CORBA ORB connects Client and Server

[3376] CORBA Naming Service used to lookup GAIs

[3377] **FIG. 75** illustrates a method **7500** including the registering and then locating of a globally addressable interface.

[3378] Collaborations

[3379] 1a. "Bind" the interface name (Update Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the Interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

[3380] 1b. "Bind" the second interface in the same manner as the first.

[3381] 2. The client instantiates a Proxy (Browsing Interface Proxy) to the Browsing Interface on the Customer Server.

[3382] 3. The Proxy "looks up" the network location of the Browsing Interface. It makes a request of the Naming Service. It requests the network location of the Browsing Interface.

[3383] 4. The Naming Service returns the Remote Object Reference (network location) for the Browsing Interface. The Proxy now has all the information it needs to access an operation on the Browsing Interface.

[3384] The second message trace builds upon the first. In this message trace diagram, the Client calls the Server through a Globally Addressable Interface. The server finds the appropriate customer data and returns it to the Client. The Client can then display it in the UI.

[3385] **FIG. 76** illustrates the present invention using a method **7600** wherein a globally addressable interface is used to obtain data from a server. The steps associated with the method **7600** of **FIG. 76** will now be set forth.

[3386] Collaborations

[3387] 5. The Client asks the Browsing Interface Proxy for the data associated with customer **1234**.

[3388] 6. The Browsing Interface Proxy forwards the request across the network to the Browsing Interface.

[3389] 7. Same as 6.

[3390] 8. The request is forwarded to the Customer Server. The Customer Server requests the customer data from the Database.

[3391] 9. The Database returns the customer data for Customer **1234**.

[3392] 10. The Customer Server creates a structure and populates it with the customer data.

[3393] 11. The Customer Structure is forwarded through the Browsing Interface, across the network and back to the Browsing Interface Proxy.

[3394] 12. The Browsing Interface Proxy forwards the Customer Structure to the Client. The Client can now display the data in a UI for a user.

[3395] IDL Interfaces and Structures

[3396] The following IDL defines the two Interfaces and Structures used in the message trace diagram s above.

```
module CustomerServer
{
// CORBA IDL for the Update Interface
    interface CustomerUpdateInterface
    {
        void changeCustomer( long onId,
commonDefs::CustomerStructure aCustomer);
        void changeAddress(long onId,
commonDefs::AddressStructure aNewAddress);
        string changePhoneNumber(long onId, string
aNewPhoneNumber);
    };
    // CORBA IDL for the Browsing Interface
    interface CustomerBrowsingInterface
    {
        commonDefs::CustomerStructure getCustomer(long onId);
        commonDefs::AddressStructure getAddress(long onId);
        string getPhoneNumber(long onId);
    };
};
// This module defines the structures passed through the two // customer
interfaces.
module commonDefs
{
    struct AddressStructure
    {
        string street;
        string city;
        string state;
        string zip;
        string phoneNumber;
    };
    struct CustomerStructure
    {
        string id;
        string status;
        string firstName;
        string lastName;
    };
};
```

[3397] Sample Code

[3398] The following is some Sample Java code for the Skeleton portion.

```
    //Pass all requests on to the Component for processing
    public CustomerStructure getCustomer(
    String aCustomerId)
    {
        CustomerComponent aCustomerComp = this.getComponent( );
        return(aCustomerComp.getCustomer(aCustomerId));
    }
```

-continued

```
//Pass all requests on to the Component for processing
public String getPhone(String aCustomerId)
{
    .
    .
    .
}
```

[3399] The next snippet of code is for the Customer Component (Server). The interface delegates the processing to the component.

```
// Get the Customer's data and return it
public CustomerStructure getCustomer(String aCustomerId)
{
    // Go to the database and get the
    // customer with the appropriate ID
    Customer aCustomer = ...
    // Create a structure and populate it with the
    // customer data retrieved from the DB.
    CustomerStructure aCustomerStructure = new
CustomerStructure( );
    aCustomerStructure.id = aCustomer.getId( );
    aCustomerStructure.status = aCustomer.getStatus( );
    aCustomerStructure.firstName =
aCustomer.getFirstName( );
    aCustomerStructure.lastName =
aCustomer.getLastName( );
    return (aCustomerStructure);
}
public String getPhone(String aCustomerId)
{
        .
        .
}
public AddressStructure getAddress(String aCustomerId)
{
        .
        .
}
```

[3400] Additional Considerations

[3401] The GAI class is actually represented by two different classes (and the Component itself). Each GAI is made up of a Proxy and a Skeleton. The Proxy represents the interface on a Client while the Skeleton represents the interface on a Server.

[3402] Collaborations

[3403] Proxy—The proxy pattern is generally used to communicate from a Client to a Globally Addressable Interface on a Server.

[3404] Structure Based Communication—Often times, a client needs to display data in a UI for a user (e.g. Customer Information, Order Information, etc.). When communicating through a Globally Addressable Interface, this data is transmitted from the Server to the Client using Structure Based Communication.

[3405] Load Balancing—When a number of servers implement the same Globally Addressable Interface, the Load Balancing pattern is used to balance Client requests between these Servers.

[3406] Proxy Pool—The Proxy Pool pattern helps balance the cost of instantiating Remote Proxies and retaining Proxy "freshness." The Proxy Pool pattern can be used to create a pool of Proxies to Globally Addressable Interfaces.

[3407] Locally Addressable Interface—Locally Addressable Interfaces are private interfaces that aren't easily located. Generally, a well-known interface (like a Globally Addressable Interface) is used to find a LAI. A Client easily find and access a service on a Globally Addressable Interfaces and request a reference to a Locally Addressable Interface in return.

[3408] Interface—The Interface pattern defines methods or functions or services rather than implementation. The Interface pattern is expanded upon by the Globally Addressable Interface pattern.

[3409] Naming—The Naming pattern describes a pattern for registering and finding services or objects etc. where they can be easily found in an application. The Naming pattern is often used to register Globally Addressable Interfaces so they are publicly available.

[3410] Alternatives

[3411] Locally Addressable Interface—The Locally Addressable Interface pattern is both a collaborating and alternative pattern. It can be used to retrieve information from Servers instead of Globally Addressable Interface.

[3412] Legacy Wrapper

[3413] FIG. 77 illustrates a flowchart for a method 7700 for affording access to a legacy system. A plurality of components coupled to a client via a component integration architecture are provided for servicing the client in operation 7702. A legacy system is interconnected to the client via the integration architecture using a legacy wrapper in operation 7704. In operation 7706, the legacy system and the client are interfaced via the legacy wrapper by communicating with the client by way of a first protocol and by communicating with the legacy system by way of a second protocol.

[3414] As an option, the legacy wrapper may include a legacy wrapper component coupled to a component adapter which, in turn, may be coupled to a legacy adapter via a legacy integration architecture. In this aspect, the legacy adapter may also coupled to the legacy system. As another option, the component adapter may also reformat call parameters of the message into an acceptable format for the legacy system.

[3415] As an additional option for this aspect, the legacy wrapper component may also include a pure legacy wrapper component. As even a further option to this aspect, the legacy wrapper component may include a hybrid legacy wrapper component. Also, the interfacing may further include: sending a message from the client to the legacy wrapper component via the component integration architecture; sending the message via the component adapter to the legacy integration architecture; forwarding the message to the legacy adapter; formatting the message to match an application program interface (API) of the legacy system; executing calls on the legacy system based on the formatted message, executing functions of the calls and returning results to the legacy adapter, legacy integration architecture, component adapter, and legacy wrapper component which

reformats the results, and forwarding the reformatted results to the client via the component integration architecture.

[3416] Legacy systems pose a unique situation for developers of component-based solutions. Commonly hosted on mainframes, Legacy Systems often communicate through proprietary protocols, have no standard data or process APIs and don't integrate easily with component based systems. How does a developer access a Legacy System in a component-based solution?

[3417] A legacy system is an existing system that does not conform to the technology, architecture and standards of the current project. A large IBM 3090 Mainframe running programs to calculate automobile insurance rates is an example of a Legacy System. It is large, very important to an insurance company, and runs on older, proprietary hardware.

[3418] Legacy Systems generally utilize different communication protocols than those used by newly developed component systems. As a result, communicating between a newly developed component and a Legacy System is very difficult.

[3419] **FIG. 78** depicts the communication difficulties associated with Legacy Systems **7800** attempting to communicate with a client **7802** via a component integration architecture **7804**. The newly developed application (client and components) communicates through a different protocol than the existing Legacy System. **FIG. 78** illustrates heterogeneous Interfaces from Components.

[3420] Legacy Systems are critical to an organization and usually represent a significant investment. They are tightly controlled to reduce the incidence of system failures and clients may be unwilling or unable to replace these older systems.

[3421] New applications could be developed on the mainframe system, however, this generally is not considered strategic and takes a lot of time and effort. Organizations want to add new functionality (new processes) without investing in the old legacy system.

[3422] As a result, the current Legacy Systems represent significant investments, are often crucial to a business and aren't easily replaced. Investing in new Legacy applications isn't practical or strategic and Legacy Systems can't communicate with newer componentized systems.

[3423] Therefore, the component-based solution should use a Legacy Wrapper to communicate with the existing Legacy Systems. The Legacy Wrapper is a component built to adapt the front end of a legacy system to the rest of the component-based solution.

[3424] This solution encapsulates the concerns of a Legacy System away from the new application. It allows other components in the solution to communicate with the legacy component in the exact same manner as the rest of the component-based solution. Further, this solution can also be used to partition the existing Legacy System functionality. **FIG. 79** illustrates homogenous interfaces from components **7900** which rectify the problems with Legacy Systems **7901** attempting to communicate with a client **7902** via a component integration architecture **7904**.

[3425] Benefits

[3426] Reuse. The Legacy Wrapper pattern allows reuse of an existing Legacy System. New component applications can be developed that leverage the rich store of business processes and data that already exist on the Legacy System.

[3427] Migration. Allows for slower migration of functionality from the Mainframe to components. By continuing to use the functionality of the existing legacy system, the immediate need to build the same functionality in a pure component-based solution is lessened.

[3428] Encapsulation. Provides a separation of concerns between the new system and the Legacy System. By encapsulating the Legacy System, the impact of host changes is largely limited to the Legacy Wrapper.

[3429] The implementation of Legacy Wrapper is usually very specific to the type of Legacy System it is integrating. The implementation in this section attempts to give a high level overview of the components of typical legacy systems.

[3430] **FIG. 80** shows how a Legacy Component is integrated into a component-based model **8000**.

[3431] The upper part of **FIG. 80** depicts the main units **8002** of a component-based solution. The lower part of the picture depicts the Legacy Component **8004** in greater details.

[3432] The following is a description of the participants in the upper portion of **FIG. 80**.

[3433] The Client (**8006**) is the application running on the user's machine. It is responsible for UI presentation, local business objects, and communication using client resident proxies.

[3434] The Component Integration Architecture (**8008**) is the component that allows clients to communicate and remotely invoke functions on the server components. Typically this is based on some middleware standard (e.g., CORBA or MTS).

[3435] The Components (**8010**) in this **FIG. 80** represents the server components. These are the business entity components and the business process components. They are invoked from the Client via client proxies.

[3436] From the outside, the Legacy Component **8004** looks identical to any other component. However, internally is performs a very specialized function.

[3437] The lower part of **FIG. 80** expands the Legacy Component **8004**. The expansion shows the individual elements, which comprise the Legacy Component **8004**. These elements are:

[3438] The Legacy Wrapper Component (**8012**) is responsible for presenting the same functionality provided by the legacy system to the rest of the component-based solution. Other components of the new component-based solution will interact and communicate with this component. Although this component wraps the existing legacy system, it should behave as any other component in the newer solution.

[3439] The Component Adapter (8014) is a custom component responsible for the translation from the Legacy Wrapper Component to the particular implementation of the Legacy Integration Architecture.

[3440] The Legacy Integration Architecture (8016) is responsible for sending and receiving messages between the server and host machines. This architecture is usually based on some specific communication implementation. Examples of this include message queues and common databases accessible by both legacy systems and component-based solutions.

[3441] The Legacy Adapter (8018) is a custom component responsible for translation from the particular implementation of the Legacy Integration Architecture to the Legacy System.

[3442] The Legacy System (8020) is the existing system that will be accessed by the newer component-based solution. Changes to the Legacy System should be minimized when accommodating the new component-based solution.

[3443] The application on the host is responsible for translating messages between the Legacy Integration Architecture and the Legacy System. For example, the application must know how to format calls to CICS appropriately, as well as interpret results and reformat them in a way appropriate for the Legacy Wrapper server component.

[3444] The degree to which the wrapper components are specialized to partition the functionality of the existing legacy system can vary.

[3445] Pure Legacy Wrapper Component

[3446] One type is the Pure Legacy Wrapper Component. This component simply adapts the legacy system to the new component-based solution. No new business processes are added. The interface methods on the Legacy Wrapper Component "pass through" to the legacy system, as shown in FIG. 81. FIG. 81 illustrates Legacy Wrapper Components of a Pure Legacy Wrapper Component including a Legacy Wrapper Component 8112, a Component Adapter 8114, a Legacy Integration Architecture 8116, a Legacy Adapter 8118, and a Legacy System 8120.

[3447] Hybrid Legacy Wrapper Component

[3448] Another type of Legacy Wrapper Component is the Hybrid component. FIG. 82 illustrates a Hybrid Component type of Legacy Wrapper Component. As shown, the hybrid includes a Legacy Wrapper Component 8212, a Component Adapter 8214, a Legacy Integration Architecture 8216, a Legacy Adapter 8218, and a Legacy System 8220.

[3449] It is a mix of legacy system adapter and some new business processes built in a single component. Some of the interfaces 8222 of the wrapper component 8212"pass through" to the legacy system, while other interfaces communicate with objects, which may in turn call the legacy system.

[3450] There are potentially more variations, including use of an Event Service to allow the mainframe to initiate work from the wrapper components.

[3451] Example

[3452] FIG. 83 shows an abstract example of the control flow in a Legacy Component. Although, the example is at a very high level, it should provide some insight as to how the Legacy Component functions and how it invokes work on the legacy system.

[3453] From the example of FIG. 83, the following steps are shown:

[3454] 1. The Client component wants to invoke some functionality, which is located on the legacy system. The Client sends a message via the Component Integration Architecture (e.g. ORB) on the way to the Legacy Wrapper Component.

[3455] 2. The Component Integration Architecture (e.g. ORB) forwards the call to the appropriate Legacy Wrapper Component.

[3456] 3. The Legacy Wrapper Component sends the call via the Component Adapter to the Legacy Integration Architecture. When necessary, the Component Adapter reformats the call parameters into an acceptable format for the Legacy System.

[3457] 4. The Legacy Integration Architecture receives a call for the host-based Legacy application and forwards it to the Legacy Adapter.

[3458] 5. The Legacy Adapter receives the message from the Legacy Integration Architecture and formats it to match the API of the Legacy System. It makes the appropriate calls on the Legacy System. The Legacy System executes the function and returns the results to the Legacy Adapter.

[3459] 6. The Legacy Adapter receives the results and returns them to the Legacy Integration Architecture.

[3460] 7. The Legacy Integration Architecture receives the result and forwards it to the Legacy Wrapper Server Component through the Component Adapter.

[3461] 8. The Legacy Wrapper Component receives the result, reformats the parameters for the component system and forwards it to the Component Integration Architecture.

[3462] 9. Finally, Component Interaction Architecture receives the result and forwards it to the Client.

[3463] Collaborations

[3464] Message Queued Legacy Integration is a specific implementation of this pattern. It uses message queues as the legacy integration architecture.

[3465] Adapter (from the Gamma book Design Patterns) describes at a more abstract level how to convert the interface of a class into another interface that clients expect.

[3466] Proxy—This pattern is documented in Design Patterns by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. The Proxy would be used to communicate across the Component Integration Architecture to a Legacy Wrapper.

[3467] Alternatives

[3468] Screen Scraping is a more specialized version of legacy wrapping. It describes how to convert a user interface to that of the server (i.e., the legacy system in this case). In this solution, the host-based application generates 3270 type screens and then passes them to CICS. The advantage of this solution is that it is non-invasive to CICS and reacts as if it were just another terminal interacting with CICS. This may be necessary with legacy systems which must be leveraged, but can not be modified and provide no common API set.

[3469] Locally Addressable Interface

[3470] **FIG. 84** illustrates a flowchart for a method **8400** for delivering service via a locally addressable interface. In operation **8402**, a plurality of globally addressable interfaces and a plurality of locally addressable interfaces are provided. Access is allowed to a plurality of different sets of services from each of the globally addressable interfaces and the locally addressable interface in operation **8404**. Each interface has a unique set of services associated therewith. In operation **8406**, the globally addressable interfaces are registered in a naming service for facilitating access thereto. Use of the locally addressable interfaces is permitted only via the globally addressable interfaces or another locally addressable interface in operation **8408**.

[3471] In an option, the use of the locally addressable interfaces may be facilitated by structured-based communication. As another option, the access may be allowed via a customer interface proxy, a customer server and a database of the globally addressable interface.

[3472] In one embodiment, a request may be received by the customer interface proxy for a reference to one of the locally addressable interfaces. The request may then be forwarded across a network to the database of a server of the globally addressable interface. Also, data from the database may be returned in response to the request. Additionally, an object may be instantiated and populated it with the data by the server of the globally addressable interface. The object may also be associated with one of the locally addressable interfaces. Also, the locally addressable interface may be forwarded to the globally addressable interface. As even a further option, a reference may be forwarded to the locally addressable interface across the network and to the customer interface proxy. In addition, the use of the customer interface proxy may be also used to access the locally addressable interface across the network.

[3473] In a client-server environment, a client makes requests of Services on a Server. In such an environment, how might a Server expose its services for use to a client in a tightly controlled manner?

[3474] Quite often a component wants tight control over the visibility of its interfaces or does not have a need to make its interfaces widely available. Examples of such situations include:

[3475] Security—A component may provide multiple interfaces, some of which have sensitive operations that should not be exposed to all clients. For example, an insurance company's customer service desktop application gets full access to all interfaces and services on a Customer component, but an Independent Agent application has restricted access to services.

[3476] Interface Design—From a design standpoint it may make sense to limit access to some interfaces. For example, a system operations interface might allow clients to query Server components for the number of requests being serviced, or disable future requests on a particular Server. In this type of situation, it's best to limit access to the appropriate user group. In this case, the operations tools specifically designed for administering a system.

[3477] Large number of interfaces—If the component design calls for a large number of interface instances (objects), then it would be detrimental to use the GAI pattern. The sheer number of interfaces could overcrowd and overburden the Naming or Trader service. The Naming or Trader service would slow down as it searched its large list of entries. Additionally, the system would slow down as every client attempted to access the Naming or Trader service for every interface.

[3478] Thus, it's sometimes best to keep interfaces with limited appeal out of a Naming or Trader Service.

[3479] No need—If a particular interface or service only has one client, why bother registering it globally? It doesn't make sense and causes additional administration.

[3480] **FIG. 85** illustrates Problems with Globally Addressable Interfaces in a system **8500** including clients **8502** and servers **8504** with a plurality of interfaces **8506**.

[3481] The last couple of points are quite common for stateful components. The above samples clearly do not call for the GAI pattern—an alternative manner of making interfaces available to clients is required.

[3482] Therefore, the Locally Addressable Interface pattern should be used to control access to interfaces in an efficient manner.

[3483] **FIG. 86** illustrates the manner in which the present invention uses a Locally Addressable Interface **8600** to hide functionality and lessen the load on the Naming or Trading Service **8602**.

[3484] All components maintain a Globally Addressable Interface **8604** This interface is registered with a Naming or Trader service **8602** and can have any of its services accessed by any client on the network. The services on a GAI **8604** are generally stateless and potentially shared by many clients.

[3485] Locally Addressable Interfaces **8600** are not registered with a Naming or Trading service **8602** and can only be obtained through a Globally Addressable Interface **8604** or another Locally Addressable Interface **8600**.

[3486] **FIG. 87** illustrates the manner in which the present invention obtains a Locally Addressable Interface **8700**.

[3487] Globally Addressable Interface **8702** services typically are used to obtain Locally Addressable Interfaces **8700** by providing some key information to the service, trigger global changes to all of the component's member objects, or to obtain component-maintained data that is not represented by a Locally Addressable Interface **8700**.

[3488] It is important to note that member business objects are never directly exposed to the client but, rather, communicated with through a component interface (global or local). This allows for changes to be made to the internal structure of the component without disturbing the way a client interfaces with the component. Encapsulation is preserved.

[3489] Benefits

[3490] Tight control. Servers providing LAIs have full control to determine which clients will receive them. This control could, for example, be based on client type, access rights or server load.

[3491] No central bottleneck. The pattern does not rely on a centralized service to hand out interfaces. This leads to a scalable architecture that can handle many interface instances.

[3492] Useful for stateful components. Stateful components often contain many objects, each accessed through a separate interface instance. The LAI pattern is very useful in such circumstances.

[3493] Complex server side relationships. The LAI pattern is better for managing complex object relationships than most alternatives. If an object is associated with a lot of other objects (an order holds a customer and an address and a line item etc.), it isn't practical to copy all of the objects to the client.

[3494] The following is a message trace diagram depicting the interactions associated with a Locally Addressable Interface.

[3495] The Message Trace diagrams depict a common Client-Server scenario. The Client would like to interact with a specific Customer on the Server. The client requests a Locally Addressable Interface to a Customer Object on the Server and communicates with that object.

[3496] The scenario was broken into two message trace diagrams. The first message trace sets the stage for the second. In the first message trace, the Server registers a Globally Addressable Interface with the Naming Service. The Globally Addressable Interface will be used to get the Locally Addressable Interface.

[3497] Assumptions

[3498] CORBA ORB connects Client and Server

[3499] CORBA Naming Service used to lookup GAIs

[3500] **FIG. 88** illustrates the method in which the present invention registers and then locates a Globally Addressable Interface **8800**. The various steps shown in **FIG. 88** are set forth hereinbelow.

[3501] Collaborations

[3502] 1a. "Bind" the interface name (Customer Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

[3503] 2. The client instantiates a Proxy (Customer Interface Proxy) to the Customer Interface on the Customer Server.

[3504] 3. The Proxy "looks up" the network location of the Customer Interface. It makes a request of the Naming Service. It requests the network location of the Customer Interface.

[3505] 4. The Naming Service returns the Remote Object Reference (network location) for the Customer Interface. The Proxy now has all the information it needs to access an operation on the Customer Interface.

[3506] The second message trace builds upon the first. In this message trace diagram, the Client calls the Server through a Globally Addressable Interface. The server finds the appropriate customer data and instantiates an object with the data. A Locally Addressable Interface to the specific Customer object is then returned to the Client. The Client can then directly access the specific Client through the Locally Addressable Interface.

[3507] **FIG. 89** illustrates the manner in which the present invention uses a Globally Addressable Interface **8900** to obtain a Locally Addressable Interface **8902** to a specific Customer Object **8904**. Note the steps set forth below.

[3508] Collaborations

[3509] 5. The Client asks the Customer Interface Proxy for a reference to a Locally Addressable Interface for Customer **1234**.

[3510] 6. The Customer Interface Proxy forwards the request across the network to the Customer Interface.

[3511] 7. The request is forwarded to the Customer Server. The Customer Server requests the customer data from the Database.

[3512] 8. The Database returns the customer data for Customer **1234**.

[3513] 9. The Customer Server creates instantiates an object and populates it with the customer data. The Customer object is associated with a Locally Addressable Interface (Update Interface).

[3514] 10. The Locally Addressable Interface is forwarded to the Customer Interface.

[3515] 11. The Customer Interface forwards a reference to the Locally Addressable Interface, across the network and back to the Customer Interface Proxy.

[3516] 12. The Customer Interface Proxy instantiates an Update Interface Proxy with the reference to the Update Interface.

[3517] 13. The Customer Interface Proxy forwards the Update Interface Proxy to the Client.

[3518] 14. The Client sends a new address for the customer to the Update Interface Proxy.

[3519] 15. The Update Interface Proxy forwards the information across the network to the Update Interface.

[3520] 16. The Update Interface forwards the new address to the Customer Object. The Customer Object updates its address based upon the new information.

[3521] Collaborations

[3522] Proxy—The proxy pattern is generally used to communicate from a Client to a Locally Addressable Interface on a Server.

[3523] Interface—The Interface pattern defines methods or functions or services rather than implementation. The Interface pattern is expanded upon by the Locally Addressable Interface pattern.

[3524] Globally Addressable Interface—Locally Addressable Interfaces are private interfaces that aren't easily located. Generally, a well-known interface (like a Globally Addressable Interface) is used to find a LAI. A Client can easily find and access a service on a Globally Addressable Interface and request a reference to a Locally Addressable Interface in return

[3525] Structured Based Communications—Often times, a client needs to display data in a UI for a user (e.g. Customer Information, Order Information, etc.). When communicating through a Locally Addressable Interface, this data is transmitted from the Server to the Client using Structure Based Communication.

[3526] Alternatives

[3527] Globally Addressable Interface. The Globally Addressable Interface pattern is both a collaborating and alternative pattern. It can be used to retrieve information from Servers instead of Locally Addressable Interface—the right choice will depend on the context.

[3528] Null Structure

[3529] **FIG. 90** illustrates a flowchart for a method **9000** for communicating a null value. A query is first communicated in operation **9002** from a first system to a second system to determine whether a data structure is a null value. Next, in operation **9004**, a response to the query is received from the second system indicating whether the data structure is a null value. A request for the data structure is sent from the first system to the second system in operation **9006** only if the response indicates that the data structure is not a null value. Subsequently, the data structure is received from the second system in operation **9008**.

[3530] As one option, the response may be a Boolean indication. As another option, the response may be determined based on an attribute of the data structure. As a further option, the data structure may represent a set of a plurality of values. Also, the first system may, optionally, be a client and the second system is a server.

[3531] When transmitting data across a network between a client and server application, the middleware's "type system" does not always support null values. How can a remote service send or receive null values over a communications medium that does not support them?

[3532] It is expected that distributed Business Components will collaborate with other Business Components via some sort of communications medium. Communications between components is not usually handled by the components themselves but rather by some communications middleware (like an object request broker, or ORB).

[3533] A "null" value is a frequently used value in object-based systems. A "null" represents the empty set. It is often returned from a service that is unable to find the requested elements or is used as an optional parameter in a distributed service. For example, a Client might request all the customers with a last name of "Smith." If no Customers exist with a last name of "Smith," a "null" value would be returned.

[3534] Some legacy systems return -999 or 0 when no data exists. This is not an ideal solution as the system is using data to represent non-data. What if -999 or 0 are valid responses to a request? Instead, a "null" could be used to better represent this case. A "null" value provides extra flexibility since a specific data value need not be reserved to represent the empty set.

[3535] However, middleware cannot represent every data type that exists in every language. Since middleware is "language neutral", it can only represent the least common denominator of every language accessible via the middleware. Due to this constraint, "null's" often can not be represented in middleware. **FIG. 91** illustrates the problem associated with sending a NULL across many types of middleware **9100**.

[3536] A system should be able to take advantage of this important value and use middleware that may not support it.

[3537] Therefore, use the Null Structure pattern to pass a structure with an is Null attribute across the middleware. Unlike a "null", a structure can be passed across the middleware.

[3538] **FIG. 92** illustrates the manner in which the present invention passes a "null" structure across the middleware **9200**.

[3539] The extra attribute on the structure then determines whether or not the structure represents a "null" value. The structure can be queried to determine whether or not it represents a "null" value. **FIG. 93** depicts conversations **9300** with a "null" data structure **9302**. **FIG. 94** depicts conversations **9400** with a non- "null" data structure **9402**.

[3540] The is Null attribute could be added as shown in the IDL example below.

```
structure Generic
{
    boolean isNull;
    long typeIdentifier;
    long otherIdentifier;
    double one;
};
Benefits
    Flexibility. This pattern allows for "null" values to be utilized by
    distributed components.
```

[3541] The following example assumes a CORBA implementation. In order to pass Null Structures across an ORB, a structure must be defined in the ORB's interface definition language (IDL). The following IDL defines a structure that will represent an Integer or a "null."

```
struct CosmosInteger
{
    long value;
    boolean isNull;
};
```

[3542] In the code that prepares the data to be sent over the ORB, a check of the data is made and the structure is populated appropriately. If it is null, the is Null flag is set, otherwise it is cleared. Refer to the following code example:

```
public CosmosInteger convertIntegerForORB(Integer anInteger)
{
    CosmosInteger integerStructure = new CosmosInteger(); //
    if (anInteger == null)
    {
        integerStructure.isNull = true;
    }
    else
    {
        integerStructure.isNull = false;
        integerStructure.value = anInteger.intValue(); //
    }
    return integerStructure;
}
```

[3543] The receiving code that obtains the data from the ORB does the same conversion in reverse as shown in the method below:

```
public Integer convertIntegerFromORB(CosmosInteger
anIntegerStructure)
{
    Integer anInteger = null;
    if (anIntegerStructure.isNull == false) // structure not null
    {
        anInteger = new Integer(anIntegerStructure.value);
    }
    return anInteger;
}
```

[3544] Collaborations

[3545] Proxy. A Proxy is a placeholder that can accept requests meant for another object. This is typically used in distributed systems when one component wants to send a request to another. Thus, a proxy is often used to make request of servers that may return null structures.

[3546] Client-Server. Client-Server is a type of architecture that separates of Client portion of an application from the business logic or database portion of an application. When implementing a Client-Server application, the Client and Server often communicate across a middleware (like CORBA) that doesn't support "nulls."

[3547] Alternatives

[3548] Invalid Value. Determine an "invalid value" for each data type in the particular application. Return the "invalid value" when ever a null should be returned.

[3549] Paging Communication

[3550] **FIG. 95** illustrates a flowchart for a method **9500** for transmitting data from a server to a client via pages. In operation **9502**, pages of data sets are built from data in a database of a server. Upon receipt of a first request from a client for the data in the database of the server in operation **9504**, a first one of the pages of the data sets is sent to the client over a network in response to the first request in operation **9506**. When a second request from the client for the data in the database of the server is received in operation **9508**, a second one of the pages of the data sets is then transmitted to the client over the network in response to the second request in operation **9510**.

[3551] The second request may be sent to the server with an identifier of a last entry of the first page. Also, a size of the data sets of the pages may be defined dynamically. As an option, the pages may be displayed by the client upon receipt from the server. Also, a size of the data sets of each of the pages may be determined based on a user interface of the client. As another option, a size of the data sets of each of the pages may be determined based on an amount of data capable of being displayed at once by the client.

[3552] In a client-server environment, a client often needs to display or process a long list of data. Finding and transmitting this list of data can take a long time and negatively impact the user's response time. How can a client and server interact to improve the user's response time when retrieving a large list of data?

[3553] The speed with which a UI can respond to a "user initiated" request is important. This is generally called the UI response time and is an important attribute of every application. **FIG. 96** depicts the response time **9600** for a User Interface **9602** to display a list of customers in a list box **9604**.

[3554] Users expect an "acceptable" level of UI response in their applications. Applications that don't meet this criteria, will not be successful.

[3555] Many UIs allow users to query databases for lists of data. In **FIG. 96**, for example, the user clicks the "Get Customers" button to initiate a database query. The query will retrieve every customer from the database and the UI will display the customers in a list box. The user can then scroll through the data and select a particular entry for further investigation.

[3556] **FIG. 97** shows a request that returns a large amount of data. As shown, in a three-tiered client-server environment, each query must travel from the client UI **9700**, across a network **9702**, to a Server **9704**, and eventually to a Database **9706**. Then, the result of the query must travel all the way back to the client.

[3557] When the query results in a large amount of data, the time to search the database and return the data across a network can become prohibitive. As a result, the UI response time will quickly degrade to an unacceptable level.

[3558] To make things worse, the average user only looks at half of the data returned from the database. The user is just as likely to find their data in the first half of the list as the second half of the list. As a result, the user may wait a long time for data that is not used.

[3559] FIG. 98 shows a graphical depiction of a paging communication pattern 9800.

[3560] Therefore, provide Paging Communication between the client and server tiers of an application. Paging Communication describes a pattern for transmitting a large amount of data while maintaining an acceptable UI response level.

[3561] Rather than send all of the data at one time, a subset or "page"9802 of data is transmitted. When the client needs more data, another "page"9802 of data is transmitted. This continues until the client has seen enough data or all the data has been transmitted.

[3562] Benefits

[3563] More Responsive UI. This pattern improves upon the user's response time. The server only retrieves and transmits a "page" of data at a time. This is a lot faster than retrieving and transmitting all of the data at one time. The pattern breaks-up the total search and transmission time into smaller page-sized chunks. This greatly improves upon the user's perceived performance.

[3564] Additionally, the Server searches the database for a "page" of data at a time until the user finds what they are looking for. As a result, unless the needed data is in the last page of data, the search is limited to a portion of the total search.

[3565] Configurable Page Size. The page size can be "tuned" to best fit the application. As a result, the page size can be altered to best fit a particular network, application design, etc.

[3566] Stateless Servers. The paging mechanism can be managed from the client-side requester. Thus, this pattern can be used with stateless servers just as easily as with stateful servers.

[3567] UI Tunable. The page size can be changed to match a particular User Interface.

[3568] List Box Friendly. A list box can only display a limited amount of data at one time. As a result, it isn't as important to have all of the data immediately available for the list box. The List box can display a page of data, and then request additional pages of data as the user scrolls through the list.

[3569] Scenario: A user is searching for a particular customer. The user doesn't remember the exact name of the customer, but the user believes they will recognize the name when they see it. Thus, the user requests a list of all customers.

[3570] Technical Parameters

[3571] Static Page Size=4

[3572] List Box can only display 2 lines of data at a time.

[3573] FIG. 99 illustrates a message trace diagram showing the interactions between a Client 9900 and a Server 9902 using Paging Communication to satisfy the previously mentioned scenario.

[3574] Definitions

[3575] Starting Key

[3576] The Starting Key is the initial starting point for the search. The database will begin searching for data (customers in the message trace above) at the Starting Key. An example starting key could be "A*".

[3577] Last Found Key

[3578] The Last Found Key is used to request subsequent pages of data from the Server and the database. The "last found key" defines the starting point for the next data request. The Server will begin searching for data at the "last found key" and continue until it has retrieved a full "page" of information.

[3579] When all of the data has been retrieved from the Server and Database, the Last Found Key is left blank. This notifies the Client that all the data has been sent.

[3580] Intermediate Page

[3581] An intermediate "page" is returned for every request but the last. When a client receives an intermediate page and a "last found key", the client knows more "pages" of data exist on the server.

[3582] In order to obtain an intermediate "page," a "last found key" must be passed from the client to the server. When the Server has retrieved a full "page" of data, the new "last found key" is saved. It is then passed back with the intermediate "page." The new "last found key" defines the starting point for the next data request.

[3583] Last Page

[3584] When the Server has retrieved all of the data meeting the search criteria, the Server builds the last "page." When the last page is returned to the client, the "last found key" is left blank. This notifies the client the search is complete and no more data matching the search exists on the Server. Note that the last page is usually smaller than the other pages.

[3585] Empty Page

[3586] When no data are selected from the search criteria, the server builds an empty page signaling to the client no more data exist on the server.

[3587] Static or Dynamic Page Size

[3588] The page size can be defined statically or dynamically. The message trace diagram in FIG. 99 depicts a static page size.

[3589] If you'd like a dynamic page size, the client must pass an additional parameter with each request to the Server. The additional parameter would be the page size.

[3590] The steps associated with FIG. 99 will now be set forth.

[3591] Collaborations

[3592] 1. The user "clicks" the "Get Customers" button on the User Interface. The Client UI makes a getAllCustomers request of the Server and passes a Starting Key as a parameter. Since the user wants to view all of the customers, a Starting Key of spaces is used. Message sent=getAllCustomers("");

[3593] 2. The Server receives the request from the Client. The Server realizes the Starting Key is blank and knows this is a new request. Thus, the Server requests first four customers (the page size) from the database.

[3594] 3. The database returns the first four customers (Albert Abraham, Ned Abraham, Sally Abraham and Alice Allen) and a "Last Found Key" ("Alice Allen") to the Server. The "Last Found Key" denotes the last entry found during the search. It will be used for subsequent searches.

[3595] 4. The Server builds a page with the four customers retrieved from the database. The Server returns the page and the Last Found Key to the Client.

[3596] Page Type=Intermediate

[3597] Page="Albert Abraham", "Ned Abraham", "Sally Abraham" & "Alice Allen"

[3598] lastFoundKey="Alice Allen"

[3599] The Client receives the "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the first two customers (Albert Abraham, Ned Abraham).

[3600] The User clicks the "scroll down" arrow twice and can now see two additional customer (Sally Abraham, Alice Allen).

[3601] 5. The User clicks the "scroll down" arrow again. No more data exists on the Client so the Client must request another page from the server. The Client UI makes a getAllCustomers request of the Server and passes the Last Found Key of Alice Allen. Message sent=getAllCustomers("Alice Allen");

[3602] 6. The Server receives the request from the Client. The Server requests the next four customers (page size) after Alice Allen. Message sent=getPageOfcustomer("Alice Allen")

[3603] 7. The database returns the next four customers (Jason Allen, Fred Allen, Sam Allen & Zack Allen) and a "Last Found Key" ("Zack Allen") to the Server.

[3604] 8. The Server builds a page with the four customers retrieved form the database. The Server returns the page and the Last Found Key to the Client.

[3605] Page Type=Intermediate

[3606] Page="Jason Allen", "Fred Allen", "Same Allen" & "Zack Allen"

[3607] lastFoundKey="Zack Allen"

[3608] The Client receives the "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the first two customers and one new customer (Alice Allen, Jason Allen).

[3609] The User can now scroll through the next three customers. When scrolling past customer Zack Allen, the Client will request another page of data from the Server. It will follow the same basic pattern as described in steps 5-9.

[3610] Eventually, the end of the list of Customer will be reached

[3611] n-3. Once again, the client clicks the "scroll down" arrow and no more customers exist on the client. The Client must request another page from the server. The Client UI makes a getAllCustomers request of the Server and passes the Last Found Key of Jim Ziegler. Message sent=getAllCustomers("Jim Ziegler");

[3612] n-2. The Server receives the request from the Client. The Server requests the next four customers (page size) after Jim Ziegler. Message sent=getPageOfcustomer("Jim Ziegler")

[3613] n-1. The database can only find two more customers. The database returns the final two customers (Sam Ziegler and Ziggy Ziegler) and no Last Found Key.

[3614] n. The Server builds a page with the two remaining customers retrieved from the database. The Server returns the page and the blank Last Found Key to the Client.

[3615] Page Type=Last Page

[3616] Page="Sam Ziegler", "Ziggy Ziegler"

[3617] lastFoundKey=""

[3618] The Client receives the final "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the following two customers (Jim Ziegler, Sam Ziegler).

[3619] The User clicks the "scroll down" arrow once and can now see the final two customers (Sam Ziegler, Ziggy Ziegler) in the List Box.

[3620] Subsequent "clicks" on the scroll down arrow no longer request data from the Server. The Client knows (due to the blank last found key) that it has already received all of the available data.

[3621] Additional Details

[3622] Context isn't generally stored on the Server when implementing Paging Communication. As a result, it is important to request a minimum collection of data from the server. Most of the relational database are using a count mechanism that defines the maximum number of data to search. That will minimize CPU and memory usage.

[3623] As explained in the example, page size may be adapted to the client requirements, however that does not mean the page size must exactly fit the widget size. Ideally the client application will anticipate future user actions and request more than one page.

[3624] Collaborations

[3625] Proxy—The Proxy pattern is often used to communicate between Clients and Servers in a distributed environment. A Proxy is often used to make requests for a "page" of data from a Server.

[3626] Interface Control Model—The ICM pattern addresses the separation of the Interface (Viewing portion) from the Control from the Model (the data portion) in an application. Paging Communication is

often used when implementing this separation of functionality. A user through the Interface uses the pattern to retrieve large lists of data from the Model for viewing.

[3627] Globally Addressable Interface—Globally Addressable Interfaces are often used to obtain a Page of data from a Server for display in a Client UI.

[3628] Alternatives

[3629] Paging with Server Caching—This pattern builds upon the Paging Communication pattern. Rather than querying the database for a "page" of information, the Server would retrieve all of the data at one time. Then the Server would pass the data to the Client one page at a time.

[3630] Refreshable Proxy Pool

[3631] **FIG. 100** illustrates a flowchart for a method **10000** for interfacing a naming service and a client with the naming service allowing access to a plurality of different sets of services from a plurality of globally addressable interfaces. In operation **10002**, the naming service calls for receiving locations of the global addressable interfaces. As a result of the calls, proxies are generated based on the received locations of the global addressable interfaces in operation **10004**. The proxies are received in an allocation queue where the proxies are then allocated in a proxy pool (see operations **10006** and **10008**). Access to the proxies in the proxy pool is allowed for identifying the location of one of the global addressable interfaces in response to a request received from the client in operation **100010**.

[3632] The proxy pool may employ load balancing. As another option, the proxies in the proxy pool may be renewed based on an age thereof. As a third option, a handle may interface the proxy pool and the client. This handle may additionally interface a plurality of the proxy pools and the client.

[3633] In distributed systems with many clients, it is important to establish connections with remote servers in an efficient manner. In a manner where clients evenly utilize the available servers. How can this be performed in a consistent manner for all clients?

[3634] In production systems it is quite common for long-lived clients to "stay up" for days and interact with a collection of different servers. Oftentimes, a client process will establish connections to the same type of server a bunch of times during its lifetime. The lifetimes of the client and its servers are often different as a result of server maintenance and failures. However, such failures should have minimal impact on the client. Clients in such systems usually retrieve a Globally Addressable Interface (GAI) from a naming or Trader Service (see GAI Pattern).

[3635] A GAI retrieved by a client will usually go through three phases: 1) Initial retrieval of a GAI from the Trader Service that is subsequently wrapped up in a proxy, 2) Invocations of businesses functions supported by the GAI and 3) Release of the GAI proxy. This often means a long-lived client will repeatedly ask the Trader Service for the same type of interface during its lifetime.

[3636] **FIG. 101** illustrates repeated requests to the Trader Service **10100** for the same interfaces are neither efficient nor necessary.

[3637] Repeatedly requesting the same interface from a Trader Service is neither efficient or necessary. However, the Trader Service does maintain load balancing information and allocates the least busy interface at any given time. Thus, some value exists in repeated reallocation

[3638] Therefore, use a Refreshable Proxy Pool mechanism that standardizes the usage, allocation and replenishment of proxies in a client's pool. Initially, the Proxy Pool will allocate a bunch of Proxies to remote services using some sort of a Lookup Service (e.g. Trader Service, Naming Service). The Proxy Pool will hold onto these Proxies and allocate them to Clients as they need them. When the client asks the Proxy Pool for a proxy, the pool will hand out a new Proxy.

[3639] **FIG. 102** illustrates how a pool **10200** can be created that reuses GAI proxies.

[3640] In order to balance the system load evenly, the Proxy Pool should implement a Load Balancing approach (e.g. Round Robin) for handing out proxies within the pool.

[3641] Each proxy in the pool has a "retirement age." The "retirement age" determines the time to refresh a given Proxy. When a Proxy reaches its retirement age, it is taken out of the pool and replaced with a freshly allocated Proxy. This ensures the Proxy Pool is refreshed regularly with new GAIs retrieved from the Trader Service. The retirement mechanism helps dynamically balance the systems load.

[3642] Benefits

[3643] Performance. Establishing connections to servers will take less time because clients will go to the trading service less often.

[3644] Balanced Load. This ensures all clients are implementing the same strategy for utilizing available servers.

[3645] Standard. One single approach to pooling increases maintainability and predictability across the enterprise and decreases confusion.

[3646] Ease of use. Client developers do not have to design and implement their own version of pooling.

[3647] Maintenance. This mechanism allows for centralized development. When a bug is found it can be fixed and distributed to all build centers.

[3648] Dynamic. Client threads will not have to worry about allocating retired or bad proxies.

[3649] Robustness. By pooling GAI location information, clients are less susceptible to Trader Service failure. This is because the Proxy Pool can operate using the GAIs it has information on for as long as those references main valid.

[3650] The Proxy Pool should be packaged and distributed to client developers so that it is non-intrusive and easy for them to use.

[3651] Parameters such as pool size and retirement age should be configurable.

[3652] The client thread using the proxy should not pay a penalty for pooling or allocation.

[3653] The pool should recover gracefully from server failure.

[3654] The pool should recover gracefully from Trader Service failure.

[3655] **FIG. 103** illustrates the implementation of a Refreshable Proxy Pool **10300**. The Refreshable Proxy Pool is based on a pool-queue approach. In this design, the pool holds allocated proxies **10302** while the queue allocates and replenishes the pool with proxies. To handle the allocation and replenishment, a worker or allocation thread **10304** runs on the queue and makes calls to the Trader Services as needed.

[3656] There can be numerous proxy pools, but this implementation supports typed pools Using C++ templates, i.e., each pool will only contain proxies of one type. This allows the client to create a class that is passed to the proxy pool and supports client specific properties in the pool such as pool size, retirement age, etc. Also, due to synchronization issues with the rest of the architecture, there can be only one allocation queue.

[3657] Clients who wish to use a pooled proxy will create a handle as a wrapper. This handle wrapper takes care of the problems associated with sharing resources across threads such as lazy initialization, reference counting, allocation, and de-allocation.

[3658] Handles are classes that abstract the users away from the implementation. Handles are generally stack based and exist for the lifetime of a method invocation or an object. The handle destructor insures that the underlying proxy is dereferenced.

[3659] Suggested Classes

[3660] **FIG. 104** illustrates the class relationships between the patterns primary classes.

Class	Description
PooledProxy (10402)	This is the base class for the pooled proxy. It actually acts as a wrapper for a Proxy and maintains all usage and reference counting information.
ProxyPool (10404)	This is the proxy pool, where clients go to retrieve a proxy. It should be thread-safe in that multiple threads are automatically synchronized. This pool should only contain valid proxies that have been allocated by the AllocationPool. When a proxy is requested, the usage count is incremented. After the "usage" passes retirement age, the proxy is removed from the pool and placed back into the allocation pool.
AllocationPool (10406)	This is the pool that actually does the proxy allocation. This pool is populated with no-allocated proxies and a "reader" thread will allocate them. Since there will only be one, this class should be implemented as a Singleton. This pool however can allocate proxies of any type.
ProxyHandle<T> (10408)	This is the Handle that clients should use to manage pooled proxies. The handles must use a static_cast<T> (C++ template) to validate the correct proxy. T is defined by a client template instantiation, and assumes the client knows exactly what type of proxy the pool is actually holding. Clients must take care to assure that pools only contain proxies of one type.

[3661] Collaborations

[3662] Globally Addressable Interface—This is a pattern for making interfaces publicly available. Distributed connections to Globally Addressable Interfaces can be pooled using the Refreshable Proxy Pooling pattern.

[3663] Proxy—This pattern is documented in the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. Proxies are pooled using the Refreshable Proxy Pool pattern.

[3664] Trader—The Trader service defines how distributed architectures locate components based on the types of services they provide. The allocation queue interacts with a Trader Service to allocate the correct type of proxy.

[3665] Naming—The Naming Service provides a mapping between names and object references. A Naming Service could be used to store the GAI references that the Refreshable Proxy Pool pattern requires.

[3666] Alternatives

[3667] Single Use—As opposed to pooling connections to a remote server a client can request a new connection each time a GAI is needed. This would work best when a client infrequently needs GAIs.

[3668] Proxy Pool—This pattern addresses the pooling of proxies without periodic refreshing. It is a simpler version of the Refreshable Proxy Pool that may be of use when server load is fairly constant.

[3669] Self-Describing Stream

[3670] **FIG. 105** illustrates a flowchart for a method **10500** for providing a self-describing stream-based communication system. Messages are sent including data between a sending system and a receiving system in operation **10502**. Meta-data is attached to the messages being sent between the sending system and the receiving system in operation **10504**. The data of the messages sent from the sending system to the receiving system is translated based on the meta-data in operation **10506**. The meta-data includes a first section that identifies a type of object associated with the data and a number of attribute descriptors in the data. Also included is a second section that includes a series of the attribute descriptors defining elements of the data.

[3671] As an option, the sending system and receiving system may each be equipped with logic for interpreting the meta-data of the messages. As another option, the elements may be defined in terms of size, type, and name. Versions of the present invention include a version where one of the systems may be an object-based system and one of the

systems may be a non-object-based system, a version where both of the systems may be object-based systems, and even a version where both of the systems may be non-object-based systems.

[3672] Stream-based communication is a very effective pattern for relaying data, data structures, and meta-data. Meta-data is information about the data, such as data structure, data types, etc. using a shared, generic format. How can the message format be shared between systems so as to create the most flexible stream-based communication mechanism?

[3673] Often, it is determined that a stream-based communication mechanism should be used to transport information between systems. Stream-based communication is a pattern where information is transported from one system to another system using a simple stream and a shared format that relays both the data and meta-data information.

[3674] **FIG. 106** illustrates two systems **10600** communicating via Stream-Based Communication **10602** and using a shared generic format to relay the meta-data information.

[3675] However, when implementing Stream-based Communication **10602**, a number of factors influence the method for enabling each system with a "shared format." The "shared format" provides the meta-data information needed to interpret the raw data in a stream. This shared format is like a secret decoder ring for systems sending and receiving messages. It allows the systems to convert structured data (objects, strings, etc.) into raw data and raw data back into structured data. This is needed to transmit the structured data across the network.

[3676] Many additional factors influence the detailed design of this communication mechanism. Some systems support volatile and constantly changing object models, data models and data structures. In these systems, flexible, de-coupled communication is extremely important.

[3677] In a constantly changing system, a statically defined "shared format" doesn't work very well. Every change to the object model, data model of data structure causes a reimplementation of the "shared format." Each reimplementation results in a redesign, recompile, and retest of the changed code.

[3678] **FIG. 107** illustrates an object-based system **10700** with a frequently changing object model **10702** communicating via Stream-Based Communication **10704**.

[3679] **FIG. 107** depicts a constantly changing system. Initially, the object-based system **10700** is designed to send Poodle objects through a stream to a non-object system **10702**. As time passes, the system requirements change. Now, the object-based **10700** system must send German Shepherd objects through a stream to the non-object system **10702**. If the "shared format" for converting dog objects to raw data is inflexible, this will break the system.

[3680] In cases like this, it would be better to implement a communication mechanism or "shared format" that can better handle changes to the systems.

[3681] Therefore, use a Self-Describing Stream and create a stream that contains message data AND descriptive meta-data. Then use a message language to read the formatting information and meta-data off of the stream.

[3682] **FIG. 108** illustrates a stream-based message that contains both message data **10800** and descriptive meta-data **10802**.

[3683] **FIG. 108** depicts a message sent using a Self-Describing Stream. The first 30 bytes contain descriptive meta-data **10802**. This meta-data **10802** describes the formatting of the "real" data **10800** in the remainder of the message. It describes the data type, attribute names, location in the message, etc. of the "real" data **10800** in the message. The remaining 70 bytes are the "real" data **10800** transmitted between the two systems.

[3684] Additionally, each system must implement a message language. The message language defines the rules for writing and interpreting the descriptive meta-data. It describes how the meta-data is parameterized and embedded in the message.

[3685] These Self-Describing messages usually contain three distinct sections: a generic header, an attribute descriptors section, and a data section. The header portion contains generic information about the message. It contains such information as the type of object, the number of attributes descriptors, the target environments, etc. The attribute descriptors section contains a series of attribute descriptors that define the various data elements of the information. The number of these attribute descriptors is usually defined in the header section. The last section contains only data.

[3686] **FIG. 109** illustrates the manner in which a message language defines how to parameterize the meta-data **10900** and put it on the stream.

[3687] Benefits

[3688] Greater Flexibility. Because the information about the structure of the data has been parameterized and stored as additional data within the message, changes to the data structure would have no effect on this interface mechanism. This means the interface mechanisms will not need to be re-designed/re-built/re-tested/re-deployed, etc for each change in meta-data.

[3689] Interfacing systems are better de-coupled. Because the message format is embedded in the actual stream, this format does not need to be stored or kept in synch across different systems. It can be "discovered" at run-time when the interface is invoked

[3690] For object-based systems, the implementation is quite straightforward. Simply make each object responsible for implementing streaming behaviors based on the format and message language. Each object should know how to get and parse its attribute values onto a stream as string values (streamOn) and each object class should know how to parse attributes off of a stream and put these values into a new instance of the object (streamOff).

[3691] Below is an example of a Self-Describing stream. It is used to stream an object's information from an object-based system to a non-object system.

[3692] **FIG. 110** illustrates a Customer object **11000** in an object-based system **11002** streaming itself into a stream **11004**, the stream **11004** being sent to a non-object system **11006**, this stream **11004** being read and the data inserted

into a relational database **11008**. The steps illustrated in **FIG. 110** will now be set forth.

[3693] 1. The CustomerObject with attributes name, sex, and age has a method "streamOut: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOut:" method goes through each of the object's attributes and parses each value as a string onto the stream.

[3694] In the Java pseudo-code below, the message language defines the format of the header, the format of the attribute descriptors, and the delimiter used in the parsing.

```
Note: Assume that "asString()" converts the receiver to a string and that
"padWithSpaces()" pads the string with spaces and makes the string the
length specified.
/** Stream my attribute values on aStream **/
public void streamOn: (OutputStream aStream)
{
    // CREATE THE HEADER
    aStream.write("CUSTOMER "); // This is a customer object
    aStream.write("003"); // with three attributes
    aStream.write("002"); // this is the format version
    // DESCRIBE EACH ATTRIBUTE
    aStream.write(Stream.Delimiter);
    aStream.write("NAME ");
    aStream.write("STG ");
    aStream.write("010");
    aStream.write(Stream.Delimiter);
    aStream.write("SEX ");
    aStream.write("STG ");
    aStream.write("001");
    aStream.write(Stream.Delimiter);
    aStream.write("AGE ");
    aStream.write("NUM ");
    aStream.write("003");
    // WRITE OUT THE ATTRIBUTE VALUES AS DATA
    aStream.write(Stream.Delimiter);
    aStream.write(this.getName( ).asString( ).padWithSpaces(10));
    aStream.write(this.getSex( ).asString( ).padWithSpaces(1));
    aStream.write(this.getAge( ).asString( ).padWithSpaces(3));
}
```

[3695] 2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.

[3696] 3. Once at the non-object system, interface code reads the stream, parses the values off, converts and moves the values into a copybook with the appropriate structure, and saves the information in relational database. A pseudo-COBOL example is listed below. In reality, this interface code would be more dynamic than depicted in this example.

```
. . .
DATA DIVISION.
FD FILE-STREAM-IN
    RECORD CONTAINS 100 CHARACTERS
. . .
WORKING-STORAGE SECTION.
01 WS-FILE-STREAM-IN                PIC X(100).
01 WS-SHARED-FORMAT-HEADER
05 WS-HEADER-OBJECT-TYPE            PIC X(10).
05 WS-HEADER-NUM-OF-                PIC X(3).
ATTRIBUTES
05 WS-HEADER-VERSION-OF-            PIC 999.
FORMAT
01 WS-SHARED-FORMAT-
ATTRIBUTE
02 WS-ATTRIBUTE-NAME               PIC X(10).
03 WS-ATTRIBUTE-TYPE               PIC X(03).
03 WS-ATTRIBUTE-SIZE               PIC 999.
01 TEMP-VARIABLES
03 WS-INDEX                        PIC 9999.
. . .
01 WS-CUSTOMER
03 WS-NAME                         PIC X(10).
03 WS-SEX                          PIC X(01).
03 WS-AGE                          PIC 999.
. . .
88 LH-HEADER-SIZE                  PIC 99 VALUE 27.
88 LH-ATTRIBUTE-DESCRIPTOR-SIZE    PIC X(1) VALUE 14.
88 LH-DELIMINATOR                  PIC X(1) VALUE "|".
88 LH-STRING                       PIC X(1) VALUE "STG".
88 LH-NUMBER                       PIC X(1) VALUE "NUM".
. . .
PROCEDURE DIVISION.
. . .
*** OPEN THE FILE STREAM AND READ IT INTO THE
TEMPORARY ***
*** VARIABLE WS-FILE-STREAM-IN ***
OPEN FILE-STREAM-IN.
READ FILE-STREAM-IN INTO WS-FILE-STREAM-IN
AT END CLOSE FILE-STREAM-IN
END-READ.
*** MOVE THE HEADER INFORMATION INTO THE HEADER
COPYBOOK ****
MOVE (WS-FILE-STREAM-IN FROM ZERO TO LH-HEADER-SIZE)
TO WS-SHARED-FORMAT-HEADER.
*** FIND WHAT BYTE THE DATA STARTS AT AND SET THE
INDEX ****
MOVE (LH-ATTRIBUTE-DESCRIPTOR-SIZE * WS-HEADER-NUM-
OF-ATTRIBUTES)
TO WS-INDEX.
*** PARSE THE APPROPRIATE OBJECT STRUCTURE OFF OF ***
*** THE STREAM ***
IF WS-HEADER-OBJECT-TYPE EQUALS "CUSTOMER" THEN
PERFORM 1000-PARSE-CUSTOMER-STREAM THRU
1000-PARSE-CUSTOMER-STREAM-END.
ELSE IF WS-HEADER-OBJECT-TYPE EQUALS "EMPLOYEE"
THEN
. . .
ELSE IF
. . .
ELSE
*** END THE PROGRAM
RUN-STOP.
END-IF.
1000-PARSE-CUSTOMER-STREAM.
*** READ WHICH VARIABLE IT IS AND POPULATE THE
CORRECT ***
*** VARIABLES ***
IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX +6)) =
"NAME"
THEN
MOVE WS-INDEX TO START-INDEX.
*** FIND THE DELIMINATOR AFTER THE NAME STRING AND
***
*** MOVE THE NAME VALUE INTO THE SEX VARIABLE ****
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT INDEX) = LH-
    DELIMINATOR
END-PERFORM.
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX) TO WS-SEX.
PERFORM 1000-PARSE-CUSTOMER-STREAM
THRU 1000-PARSE-CUSTOMER-STREAM-END.
ELSE IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX +
```

-continued

```
S) = "SEX"
THEN
*** FIND THE DELIMINATOR AFTER THE SEX STRING AND
MOVE ***
*** THE SEX VALUE INTO THE SEX VARIABLE ***
MOVE WS-INDEX TO START-INDEX.
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT WS-INDEX) = DE-
    DELIMINATOR
END-PERFORM
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX) TO WS-SEX
PERFORM 3050-PARSE-CUSTOMER-STREAM
THRU 3000-PARSE-CUSTOMER-STREAM-END.
ELSE IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX +
5)) = "AGE"
THEN
*** FIND THE DELIMINATOR AFTER THE AGE STRING AND ***
*** MOVE THE AGE VALUE INTO THE AGE VARIABLE ***
MOVE INDEX TO START-INDEX.
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT WS-INDEX) = DE-
    DELIMINATOR
END-PERFORM
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX) TO WS-AGE
PERFORM 3050-PARSE-CUSTOMER-STREAM
THRU 3000-PARSE-CUSTOMER-STREAM-END.
ELSE
PERFORM 2050-SAVE-CUSTOMER THRU
2060-SAVE-CUSTOMER-END.
END-IF.
3050-PARSE-CUSTOMER-STREAM-EXIT.
2060-SAVE-CUSTOMER.
*** CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
*** RELATIONAL DATABASE
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER
. . .
2050-SAVE-CUSTOMER-END.
```

[3697] Conversely, a stream could be created by a non-object system or another object system, populated with customer information, and sent to one's object-based system. Once in the object-based system, the CustomerObject could use a "streamOff: aStream" method, instanciate a CustomerObject, and populate it with the appropriate attribute values.

[3698] Collaborations

[3699] Stream-based Communication. This is the parent pattern to the Self-Describing Stream pattern. In this pattern, information is transmitted using a simple stream and a shared, generic format. The Self-Describing Stream is a more specific implementation of Stream-Based Communication.

[3700] Bridge (from the Gamma book Design Patterns) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collaborations between a business object and a format object while decoupling the business object from its specific stream format.

[3701] Abstract Factory (from the Gamma book Design Patterns) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

[3702] Alternatives

[3703] Fixed Format Stream—This pattern is a specific variation of Stream-Based communication where the messaging format is defined and stored on both the sending and receiving systems.

[3704] Downloadable Format Stream—This pattern is a specific implementation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

[3705] Stream-based Communication

[3706] **FIG. 111** illustrates a flowchart for a method **11100** for providing a stream-based communication system. A shared format is defined on interface code in operation **11102** for a sending system and a receiving system. A message to be sent from the sending system to the receiving system is translated based on the shared format in operation **11104**. The message is sent from the sending system and received by the receiving system in operations **11106** and **11108**. The message received by the receiving system is translated based on the shared format in operation **11110**.

[3707] As an option, information in the translated message received by the receiving system may be stored in a relational database. As another option, the shared format may be based on an order of attributes in the message.

[3708] In one version, one of the systems may be an object-based system and one of the systems may be a non-object-based system. In another version, both of the systems may be object-based systems. In a third version, both of the systems may be non-object-based systems.

[3709] In order to successfully transmit a formatted message, both the sending and receiving systems must understand the format and structure of the transmitted information. Some communications mediums, however, do not inherently transmit the formatting information with the data. How can information be easily communicated between systems when the communication mechanism does not inherently convey data structure or other meta-data information?

[3710] For two systems to successfully communicate, they must understand the structure of the data they are passing. The sending system needs to convert standard programming constructs (objects, structure, strings etc.) into bytes of data that can be transmitted along a network. The receiving system needs to receive the bytes of data on a wire and reconstitute it back into objects, structure, strings etc.

[3711] Many communication mechanisms inherently provide this functionality to a software developer. CORBA and DCOM are two examples of this type of middleware. Using CORBA, one system can send a structure of information to another system. CORBA will convert the structure into a network appropriate format, transmit it across the network and reformat it on the receiving end.

[3712] Other types of middleware, however, do not provide this full range of functionality. Lower level communication protocols like TCP and UDP as well as higher-level protocols like HTTP and telnet do not provide support for sending data structures.

[3713] Additionally, popular message queuing software (IBM'S MQSeries and BEA MessageQ) and many custom communications mechanisms are lacking support for transmitting structured data across the network.

[3714] These communication protocols do not inherently convey meta-data information. 'Meta'-data is information about the data. Meta-data could describe the data structure (senders address in bytes **1-16**, receivers address in bytes **11-20**, data in **21-100**, etc.), data types, etc.

[3715] When the highest-level common communication protocol between two systems cannot convey this meta-data information, an alternative communication mechanism is needed.

[3716] **FIG. 112** illustrates how systems **11200**, **11202** of the present invention communicate over a communication mechanism **11204** that cannot inherently convey meta-data information.

[3717] How can they exchange structured data?

[3718] In object-based systems, issues with conveying meta-data are even more prevalent. Object-based systems often need to transfer object information across non-object communication mechanisms (e.g. DCE, . . .) or to non-object systems. Because neither non-object communication mechanisms nor the non-object systems understand the notion of objects, how can the structure of the objects be meaningfully conveyed?

[3719] **FIG. 113** is an illustration of an object-based system **11300** communicating with a non-object system **11302** using a communication mechanism **11304** that cannot convey meta-data information.

[3720] How can they exchange structured data?

[3721] Therefore, use Stream-based Communication to transmit information between systems. Stream the data between the two systems and use a generic format to relay the information and its associated meta-data between the systems.

[3722] A stream is simply a buffer that data is "written to" and "read from" in discrete quantities. The size of the buffer is predetermined and can be very small (i.e. one byte in length) or very large (i.e. a page). The buffer can't hold objects or structures, but just raw data. Buffers are quite dumb and don't understand anything about their raw data. Thus, it does not have meta data for the information in the buffer.

[3723] The "shared format" provides the meta-data information needed to interpret the raw data in the buffer. This shared format is like a secret decoder ring for systems sending and receiving messages. The sending system uses the decoder ring to convert objects, structures, etc. into raw data on a stream. The receiving system uses another decoder ring to reconstitute the raw data back into objects or structures. If objects aren't supported, the raw data is converted into a comparable format for use by the receiving system.

[3724] **FIG. 114** depicts an example of Stream Based Communication with two disparate systems **11400**, **11402** communicating via stream-based communication **11404**.

[3725] In **FIG. 114**, the object-based system **11400**uses a shared format (decoder ring) to convert an object into raw data. The raw data is then copied onto the stream. The stream then delivers the data to the Non-Object system **11402**. The Non-Object system **11402** reads the raw data and reconstitutes the data using its shared format.

[3726] In this example, the sending system is sending objects while the receiving system doesn't understand objects. Thus, the receiving system can only convert the raw data into a data equivalent of the object sent.

[3727] Benefits

[3728] Maintainability. When using this pattern, a shared, generic format is used to interpret the data. As a result, the two systems are de-coupled and less dependent upon each other. As long as the format remains unchanged, changes to the internal implementation of either system will not affect the other system. Maintenance of decoupled systems is easier.

[3729] Batch Compatible. Strings of data can be concatenated and transmitted as a group. This enables batch messaging (e.g. Request Butcher) and processing.

[3730] Enables Lightweight Persistence. Stream-based communication can be used to interface with a lightweight persistence mechanism. Objects, structures, etc. can be can be converted to raw data and streamed to a flat file for saving. At a future time, the file can be opened, the raw data can be streamed out of the file and reconstituted into full blown objects or structures.

[3731] The implementation of Stream Based Communication is very straightforward. Simply define interface code on the sending system that creates a stream and parses the data onto this stream using a format shared by the both the sending and receiving systems. On the receiving system, define interface code that reads the stream and, using the same shared format, parses the data off of the stream and into a data structure compatible with the receiving system.

[3732] The specific implementation of the formats can be, and most likely will be, different from system to system but the actual format must be shared and should be generic between systems. Shared so that the information is accurately relayed and generic to keep the systems as de-coupled as possible by not exposing any implementation details of either system in the format. Further, this shared format can be implemented in a variety of places depending upon the specific requirements of the interface.

[3733] For object-based systems, make each object responsible for implementing streaming behaviors that use this shared format. Each object should use the format as a map to parse the attribute values onto a stream (streamOn). Conversely, each object class should use the format as a map to parse its attribute values off of a stream and put them into a newly instantiated instance of the object (streamOff).

[3734] In the example below, an object within an object-based system uses stream-based communication to stream

its attribute values onto a stream. Then a communication mechanism transports the stream to a non-object system, and a non-object system reads the information off of the stream and inserts it into its relational database.

[3735] **FIG. 115** is an illustration of a Customer object **11500** in an object-based system **11502** streaming itself into a stream **11504**, the stream **11504** being sent to a non-object system **11506**, this stream **11504** being read and the information is inserted into a relational database **11508**.

[3736] 1. The CustomerObject with attributes name, sex, and age has a method "streamOn: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOn:" method goes through each of the object's attributes and parses each values as a string onto the stream.

[3737] The fixed format contract here is embodied in the order that this method parses the attributes onto the stream. A pseudo-code example in Java is the following. Note-Assume that "asString( )" converts the receiver to a string and that "padWithSpaces( )" pads the string with spaces and makes the string the length specified.

```
/** Stream my attribute values on aStream **/
public void streamOn (FixedFormatStream aStream)
{
aStream.write(this.getName( ).asString( ).padWithSpaces(30));
aStream.write(this.getSex( ).asString( ).padWithSpaces(2));
aStream.write(this.getAge( ).asString( ).padWithSpaces(3));
}
```

[3738] 2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.

[3739] 3. Once at the non-object system, interface code reads through the stream, parses the values off of the stream, converts them to the appropriate types if required, and puts them in a copybook with the appropriate structure. In this example, the fixed format contract is embodied in the structure and type of the WS-SHARED-FORMAT-CUSTOMER working-storage copybook. Refer to the pseudo-COBOL example below.

[3740] . . .

[3741] DATA DIVISION.

```
SD FILE-STREAM-IN
    RECORD CONTAINS 80 CHARACTERS
. . .
WORKING-STORAGE SECTION.
*** THIS COPYBOOK CONTAINS THE SHARED FORMAT OF
THE
*** CUSTOMER IS THE DATA STRUCTURE AND DATA TYPES
01 WS-SHARED-FORMAT-CUSTOMER
   03 WS-SHARED-FORMAT-NAME        PIC X(30).
   03 WS-SHARED-FORMAT-SEX         PIC X(2).
   03 WS-SHARED-FORMAT-AGE         PIC 999.
*** THIS COPYBOOK IS THIS SYSTEMS VIEW OF A CUSTOMER
01 WS-CUSTOMER
   03 WS-NAME                      PIC X(30).
   03 WS-SEX                       PIC 99.
   03 WS-AGE                       PIC X(03).
. . .
PROCEDURE DIVISION
. . .
*** OPEN THE FILE STREAM AND PUT THE CONTENTS IN THE
*** WS-SHARED-FORMAT-CUSTOMER COPYBOOK.
OPEN FILE-STREAM-IN
READ FILE-STREAM-IN INTO WS-SHARED-FORMAT-
CUSTOMER
    AT END CLOSE FILE-STREAM-IN
END-READ.
*** MOVE THE VALUES INTO FROM THE SHARED FORMAT
INTO
*** THE WS-CUSTOMER VARIABLES.
MOVE WS-SHARED-FORMAT-SEX      TO WS-SEX.
MOVE WS-SHARED-FORMAT-AGE      TO WS-AGE.
MOVE WS-SHARED-FORMAT-NAME     TO WS-NAME.
. . .
*** CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
*** RELATIONAL DATABASE.
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER.
. . .
STOP-RUN.
```

[3742] Conversely, a stream could be created by a non-object system (or another object-based system for that matter) and sent to one's object-based system. In this case, CustomerObject could use a "streamOff: aStream" method and instanciate a new instance of a sCustomerObject and populate it with the appropriate attribute values.

[3743] Again, there are several variations of this pattern depending upon what the specific requirements are. Some of these variations are further explained in the children patterns. Refer to Fixed Format Stream, Downloadable Format Stream, and Self-describing Format Stream.

[3744] Collaborations

[3745] *Fixed Format Stream*—This child pattern is a specific variation of Stream-Based communication where the messaging format is defined and stored on both the sending and receiving systems.

[3746] Downloadable Format Stream—This child pattern is a specific variation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

[3747] Self-Describing Stream. This child pattern is a specific variation of Stream-Based communication where the messaging format is parameterized and stored on the stream. A message language is used to read and write the format of the message from the stream.

[3748] Structure Based Communication—This pattern uses a Fixed Format Stream to transmit data structure between systems. It is often used to obtain data from a Server for display in a Client UI.

[3749] *Bridge* (from the Gamma book *Design Patterns*) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collabora-

tions between a business object and a format object while decoupling the business object from its specific stream format.

[3750] Abstract Factory (from the Gamma book *Design Patterns*) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

[3751] Structure-Based Communication

[3752] **FIG. 116** illustrates a flowchart for a method **11600** for efficiently retrieving data. A total amount of data required for an application executed by a client is determined in operation **11602**. In a single call, the total amount of data from a server is requested over a network in operation **11604**. All of the data is bundled in operation **11606** into a data structure by the server in response to the single call. In operations **11608** and **11610**, the bundled data structure is sent to the client over the network and the data of the data structure is cached on the client. The cached data of the data structure is used as needed during execution of the application on the client in operation **11612**.

[3753] The data structure may be bundled on the server by a business object. In addition, the business object may be instantiated by an action of the client. Also, the network may be at least one of a local area network and a wide area network. As a further option, the request may be administered by a proxy component. Further, the data structure may contain no logic.

[3754] In a client server application, the client communicates with a server over a network. Depending upon the speed of the network and the number calls across the network, an application can experience performance problems. How can a client update a server while minimizing the network traffic and maintaining an acceptable level of system performance?

[3755] Acceptable system performance is an important attribute of every application. When creating a client-server application, the performance of the network must be considered during the design and development of an application. The speed at which data can be transmitted across the Local Area Network or Wide Area Network, can make or break a client-server application.

[3756] In a typical three-tiered client-server application, the business objects are maintained away from the users (Client) on separate Server machines. Whenever a user needs the expertise of a business object, the user must send a request across the network to the Server machine.

[3757] **FIG. 117** illustrates the manner in which a client **11700** requests information from server objects **11702** via a network **11704**.

[3758] Depending upon the size of the message and the speed of the network, this could take a long time. This is a reality of three-tiered client-server applications.

[3759] When a client-server application is introduced to the world of distributed objects, the network can become an even larger bottleneck. In a pure distributed object approach, the client is passed an object reference to a business object on a server machine. The client then accesses the specific business object over the network as if it resided on its local machine. Using this "pure" distributed object approach, the application's calling pattern begins to look like the schematic of **FIG. 118**.

[3760] **FIG. 118** illustrates the method of the present invention in which a client **11800** requests attributes from a server object **11802** via a network **11804**.

[3761] This is an excellent programming model that frees the developer to access local and remote objects in the same fashion. Unfortunately, it makes it easier for the application developer to forget about the physical realities of the network. A network call is always slower than a call within a single machine. Ignoring this reality may result in an unacceptably slow application.

[3762] On a very fast LAN where the number of network calls is small, this calling pattern may be acceptable. On a slower LAN, any WAN or when the number of network calls is large, this pattern will yield unacceptable network performance. Something must be done to maintain an acceptable level of system response for the users.

[3763] **FIG. 119** illustrates the transmitting of all data in a Data Structure **11900** from a client **11902** to a server **11904** and visa-versa. As shown, to maximize the performance on the client, it is best to bundle all the necessary data into a single data structure that can be transmitted as a structure across the network.

[3764] The Client would first determine the sum total of everything it will need from the business object on the Server machine. The Client makes a request for all of this data from the business object. The business object bundles all the data into a data structure and returns it to the client. The Client will cache this data (using the Caching Proxy pattern) on its local client machine and use it as needed.

[3765] Benefits

[3766] Better System Performance—This pattern will improve the performance of an application by reducing the network traffic between the client and the server. The client makes one network request to retrieve all of the data from the server. Regardless of the number of displayable attributes the application will only make one network send.

[3767] Without this pattern, the client could make a network send for every attribute retrieval. If the user wants to retrieve a customer's name, address and phone number, that could result in three network requests (one for each attribute). Without this pattern, the network traffic can become prohibitive and the performance would suffer.

[3768] Structure Based Communication assumes a client needs information from an object that exists on a server. Thus, this pattern assumes the existence of an "interesting" object on the server machine.

[3769] Even though the "finding" and "instantiating" of a server object isn't part of this pattern, it does establish context and sets the stage for the pattern. As a result, a message trace diagram for finding and instantiating a particular object instance is shown below. This will set the stage for the implementation of the Structure Based Communication pattern.

[3770] **FIG. 120** illustrates the method in which a client **12000** finds and instantiates a Customer Object from a customer component **12002**. The various steps shown in **FIG. 120** will now be set forth.

[3771] Collaborations

[3772] 1. The client instantiates a Proxy (Customer Component Proxy) to the Customer Component. The Client then asks the Proxy for Customer Jimbo Jones.

[3773] 2. The Customer Component Proxy forwards the request across the network to the Customer Component.

[3774] 3. The Customer Component requests the information for Jimbo Jones from the database.

[3775] 4. The Database returns the data associated with customer Jimbo Jones.

[3776] 5. The Customer Server Component instantiates a customer object using the Jimbo Jones data from the database.

[3777] 6. The Customer Server Component returns a remote object reference to the "Jimbo Jones" object running on the Server.

[3778] 7. The Client creates a proxy to the "Jimbo Jones" object using the remote object reference.

[3779] Now that a customer object (Jimbo Jones) exists on the server component, Structure Based Communication can be used to pass the needed data from the server to the client.

[3780] **FIG. 121** illustrates a Structure Based Communication that builds upon the method of **FIG. 120** and depicts the flow of control during Structure Based Communication. The various steps shown in **FIG. 121** will now be set forth.

[3781] Collaborations

[3782] 8. The Client asks the Customer Proxy for the data associated with the Jimbo Jones object.

[3783] 9. The Customer Proxy forwards the request across the network to the Customer Component.

[3784] 10. The Jimbo Jones object creates a data structure and populates it with its data.

[3785] 11. The Data Structure is passed across the network to the Customer Proxy on the Client.

[3786] 12. The Customer Proxy forwards the data structure containing Jimbo Jones' data to the Client component.

[3787] Participants

[3788] Client—The "client" for the transaction. This could be a User Interface that displays customer data for viewing by a Customer Service Representative.

[3789] Network—A LAN or WAN network that connects the Client with the Customer Component.

[3790] Customer Component—A server component that encapsulates the data for all of the customers in a system.

[3791] Customer Component Proxy—A proxy to the Customer Component. Any request it receives, it forwards across the network to the Customer Component.

[3792] Customer Proxy—A proxy to the Jimbo Jones Customer Object Any request it receives, it forwards across the network to the Jimbo Jones Customer Object.

[3793] ACustomerStructure—A data structure. It contains the data (but no methods) from the Jimbo Jones object.

[3794] Database—Any relational database.

[3795] Jimbo Jones Object—An object that represents the Jimbo Jones customer. This object contains Jimbo Jones' data and methods associated customer methods.

[3796] Sample Java Code

[3797] The following java example accompanies the previous message trace diagrams. The java code follows the same scenario as the message trace diagrams, but it has been simplified in some areas.

[3798] The following snippet of code defines the data structure used to pass customer information.

```
public CustomerStructure{
        String firstName,
        String lastName,
        short streetNumber,
        String street,
        String city,
        String state,
        String zipCode;
```

[3799] The following block of code is the "main" routine for the Client. Even though all distributed calls are made through a proxy, the proxy code is not shown in this example. The two proxy classes encapsulate the details associated with distributed network calls.

```
// Client side code here.
Main( )
{
// Create a Proxy to the Customer Component.
CustomerComponentProxy aCustomerComponentProxy =
    new CustomerComponentProxy( );
// Get a Proxy to the Jimbo Jones Customer
// Object.
CustomerProxy aCustomerProxy =
    aCustomerComponentProxy.getCustomer("Jimbo Jones");
// Get Customer data from the Customer Server
// Component(Call across the network)
CustomerStructure aCustomerStructure =
    aCustomerProxy.getCustomerAsStructure( );
// Use the Customer data received in the
// structure. For Example, display the data
// structure data (aCustomerStructure) in a UI.
                .
                .
                .
                }
```

[3800] The following code is a sample Customer Server Component. The Customer Server Component is used to retrieve the data associated with customer Jimbo Jones from the database. It also instantiates a customer object using the data retrieved from the database.

```
// Customer Component Code here
public class CustomerComponent
{
    // Put the data associated with a Customer Object
    // into a data Structure. This data structure
    // will be sent across the network to a client.
    public Customer getCustomer(String aCustomerName)
    {
        // Find the Customer in the database
        .
        .
        .
        // Instantiate the Customer Object
        Customer aCustomer = new Customer(...
        .
        .
        // Return a "remote object reference" to the
        // Jimbo Jones Customer object.
        return (aCustomer);
    }
}
Finally, this is the code for the Jimbo Jones Customer object.
    // Customer object code here
    public class Customer
{
    // Put the data associated with a Customer Object
    // into a data Structure. This data structure
    // will be sent across the network to a client.
    public CustomerStructure getCustomerAsStructure( )
    {
        CustomerStructure aCustomerStructure = new
CustomerStructure( );
        aCustomerStructure.firstName = this.getFirstName( );
        aCustomerStructure.lastName = this.getLastName( );
        aCustomerStructure.streetNumber = this.getStreetNumber( );
        aCustomerStructure.street = this.getStreet( );
        aCustomerStructure.city = this.getCity( );
        aCustomerStructure.state = this.getState( );
        aCustomerStructure.zipCode = this.getZipCode( );
        return aCustomerStructure;
    }
    // Getters and Setters for all attributes
    // (code not shown here)
        .
        .
        .
}
```

Note: The "Main" code example above contains a Proxy to customer "Jimbo Jones" and then a Structure of data through the proxy. The code was written for ease of understanding, but causes two calls across the network. In reality, it is preferred to perform both functions using one network call. Limiting the number of network calls will improve system performance. The recommended code might look something like this:

[3801]

```
// Create a Proxy to the Customer Component
    // (same as above)
    CustomerComponentProxy aCustomerComponentProxy = new
    CustomerComponentProxy( );
// Get a Proxy to the Jimbo Jones Customer
// object
//              AND
// customer data at the same time.
CustomerProxy aCustomerProxy;
```

-continued

```
CustomerStructure aCustomerStructure =
    aCustomerComponentProxy.getCustomerData("Jimbo Jones",
aCustomerProxy);
```

[3802] Collaborations

[3803] Proxy—This pattern is documented in the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. The Proxy pattern is often used to retrieve data structures from a server component.

[3804] Cached Proxy—This pattern is documented in "The Proxy Design Pattern Revisited" section of the Pattern Languages of Programming Design 2 book. A Cached Proxy caches data locally on the client. Structure Based Communication uses and builds upon this pattern.

[3805] Globally Addressable Interface—This pattern often works in conjunction with Structure Based Communication. Oftentimes, a Globally Addressable Interface is used to obtain Structures of data for display on a Client.

[3806] Locally Addressable Interface—This pattern can also be used in conjunction with Structure Based Communication. After establishing a relationship with an LAI, a client may obtain data from the Server object using Structure Based Communication.

[3807] Alternatives

[3808] Distributed Objects—The "pure" distributed object approach is an alternative to Structure Based Communication. Using this pattern, individual objects are queried for each piece of information needed by a client.

[3809] Presentation Services (1000)

[3810] Presentation Services enable an application to manage the human-computer interface. This includes capturing user actions and generating resulting events, presenting data to the user, and assisting in the management of the dialog flow of processing. The Presentation Services forward on the user requests to business logic on some server. Typically, Presentation Services are only required by client workstations.

[3811] In addition to Presentation Services on the client, some business logic will usually reside on the client as well to aid the Presentation Services. Even on thin clients some sort of validation logic is usually included with the Presentation Services. A quick review of the Gartner Group's five styles of client/server computing help to illustrate this.

[3812] FIG. 122 shows the Gartner Group's Five Styles of Client/Server Computing 12200.

[3813] The way that Presentation Services interact with client-side business logic is very important to the overall scalability and maintainability of the application. An application's business logic is expected to be highly reusable, even on the client. If business logic is coupled with the Presentation Services too tightly, it will be very difficult to

separate and reuse the business logic if the Presentation Services ever need to be altered (not an uncommon occurrence).

[3814] The patterns in this section help to guide application architects on strong, proven techniques to safely integrate client-side business logic with an application's Presentation Services. The Activity pattern lays the groundwork for separating the Presentation Services and business logic on the client by assigning non-presentation logic to a type of object called an Activity. The View Configurer pattern helps to assign new views with their appropriate Activity. Finally, the User Interface Validator pattern describes how to implement customizable, extendable validation logic on a user interface.

[3815] Activity

[3816] **FIG. 123** illustrates a flowchart for a method **12300** for providing an activity module. A server and a presentation interface of a client are interfaced to permit the receipt of requests for service from the presentation interface of the client in operations **12302** and **12304**. A portion of the requests are handled on the client in operation **12306**. In operations **12308** and **12310**, another portion of the requests are forwarded to the server for further handling purposes and changes are effected in the presentation interface.

[3817] A plurality of presentation interfaces may be interfaced. Optionally, a model may be interfaced for management purposes. With such an option, the model may further include a proxy. As another option, errors and exceptions may also be handled. As a third option, events intended to be received may be triggered by the presentation interface.

[3818] Many client/server applications maintain some amount of business logic on the client. How can an application represent and reuse "client-side" business logic across multiple, volatile user interfaces?

[3819] Imagine a typical client/server system design. In almost all cases, a typical system executes data access logic on the server and presentation logic on the client, business logic is split across both the client and server. The majority of this business logic is maintained on the server. This logic is represented by various components and business objects that can communicate with each other to complete a variety of system use cases.

[3820] The client, on the other hand, is mostly responsible for supporting user interactions with the system. To be successful, the client must also execute some degree of business logic. While this can vary from implementation to implementation, some categories of logic are invariably located on the client. This includes simple data validations, representing data structures and relationships, error and exception handling, and communications with the server.

[3821] To complete a single use case, the client may need to interact with a number of server components. From the user's perspective, one unit of work is being performed but it may involve multiple, discrete interfaces and multiple server invocations. Some business logic is required to manage the complex flow to complete this unit of work. For example, suppose a use case for a network inventory management system is "Add Network Card". This may require the user to input information in three or four screens and client communication with more than one server component. Managing this flow is not the responsibility of the presentation logic but still needs to be executed on the client.

[3822] The system may also require a number of interfaces to complete the same use case. Depending on the user category executing the use case, the interface may be a PC, a handheld device, or a telephone. Even on the same type of device, the interface may differ depending on the user category. Some users may want to access the application via a standard Windows interface while others may want to access it via a "web-centric" interface (internet browser). In all of these cases, the unit of work to be completed by the user is not changed and should be reused.

[3823] **FIG. 124** illustrates multiple interfaces to an application **12400** including a handheld device **12402**, a desktop PC **12404**, and a telecommunications device **12406**.

[3824] Often these user interfaces will be changed over time to fit user's changing needs. While the tasks completed by the user may not change, the interface to complete those tasks will need to. Windows users will want to move to the Web. Web users will want to move to handheld devices. The presentation code should be able to be changed without causing a rewrite of the business logic on the client.

[3825] Therefore, bundle business logic executed on the client separate from the presentation logic. This new type of class is an Activity.

[3826] An Activity is responsible for:

- [3827] managing client logical units of work
- [3828] maintaining client representation of a business model
- [3829] validation across multiple interfaces (complex business logic)
- [3830] error and exception handling
- [3831] communication with server and other services
- [3832] creating other Activities
- [3833] triggering events intended to be "caught" and acted on by the presentation logic

[3834] An Activity resides between the actual user interface and the business model and server components as shown in the Entity Relationship diagram below:

[3835] **FIG. 125** illustrates an activity entity relationship diagram.

[3836] While any user interface maintains a reference to the Activity **12500** it provides an interface **12502** for, the Activity is unaware of what (if any) interfaces exist on it. This decoupling allows for a large amount of flexibility with the interfaces to an application. Multiple types of interfaces can exist on a single type of Activity. Code is reused and none is lost if presentation logic is replaced with something different.

[3837] While a user interface can communicate directly with its associated activity, an activity should never directly communicate with any of its interfaces. This would set up a dependent relationship that would reduce the flexibility of the activity. Instead, an activity can communicate to its interfaces through an event mechanism. Interfaces are set up

as dependents of the activity and the activity sends events to all of the interfaces on it. Each interface can decide how to handle the event.

[3838] FIG. 126 illustrates a roles and responsibilities diagram.

[3839] Benefits

[3840] Maintainability. By separating the presentation and non-presentation logic, the client is easier to understand and maintain.

[3841] Reuse. The presentation layer may be replaced or reused without affecting the non-presentation logic.

[3842] FIG. 127 illustrates a typical implementation between a user interface and its activity.

[3843] The diagram shows the various "layers" of interaction where the lightest shaded boxes are the presentation, the next darkest is the activity, and the darkest is the component (proxies 12700).

[3844] A user request is captured and processed by the presentation object (NetworkInventoryUserInterface 12702). In this case, the processing involves simple validation (format and "is null" checking).

[3845] The presentation object then copies its data into a structure representing some business entity (aNetworkItem 12704) and passes it to the activity 12706. The presentation object then triggers the activity to start its processing of the new network item.

[3846] The activity then performs possibly more complex validation and communicates with the server components to complete the use case.

[3847] Collaborations

[3848] Façade—The Activity acts as a Façade to all of the server components by coordinating a user interfaces interaction with them.

[3849] Separation of Concern—Dividing defined responsibilities into separate classes (presentation logic into UI classes and client-side business logic into activity classes).

[3850] Observer—An activity's interfaces are observers of that activity.

[3851] User Interface Validator

[3852] FIG. 128 illustrates a flowchart for a method 12800 for structuring validation rules to be applied to a user interface for maximum maintainability and extensibility. In operations 12802 and 12804, a plurality of user interface widgets are provided along with a plurality of validation rules which govern use of the user interface widgets. A user is allowed in operation 12806 to select the validation rules to associate with the user interface widgets of a first user interface. The validation rules of the user interface widgets of the first user interface are automatically associated across a plurality of separate different user interfaces in operation 12808.

[3853] The validation rules may be created at the time the first user interface is created. As another option, the validation rules may be implemented by a different class for each type of validation. As a further option, an indicator may be displayed upon one of the validation rules being violated. Additionally, each validation rule class may extend an abstract validation rule class that defines what types of widgets are supported. Also, a request for the validation rules may optionally be received from one of the user interfaces.

[3854] How can you structure validation rules to be applied to a user interface for maximum maintainability and extensibility.

[3855] Imagine a typical Windows or web-based client/server application. In most cases where a "windows" type of user interface is provided, an application supports some business rules by validating data entered by the user. A common example of this is checking the format of data in an entry field or ensuring that a required field is not left empty.

[3856] The business rules supported by user interface validation is usually somewhat limited. The scope of these rules is generally constrained to checking if a field is empty, checking the format of a field (date, time, numeric, currency, etc.), and checking if a field has alpha-characters, numeric-characters, or both. In addition, due to fact that many widgets provide constrains through their own form (list boxes, radio buttons), the types of widgets that require this type of validation checking is also somewhat limited (text fields, editable combo boxes, etc.).

[3857] FIG. 129 illustrates widgets with their validation requirements 12900.

[3858] Because this type of validation will most likely be required across all of an application's user interfaces and the fact that the types of validation rules and widgets needed to validate are limited, this behavior is a strong candidate for a framework.

[3859] The framework would provide a common approach to validating user data across all of an application's user interfaces. Rules would be applied consistently throughout the application. While some common validation rules would be provided, the framework needs to allow their behavior to be modified (overridden) and make it easy for new rules to be added.

[3860] Finally, both immediate and batch validation should be provided by the framework.

[3861] Therefore, for each user interface in an application, encapsulate their validation logic in a User Interface Validator. FIG. 130 illustrates a user interface validator association diagram. A User Interface Validator 13000 associates various validation rules with the user interface widget they are to be applied to.

[3862] The associations are created at the time the user interface is created. The validations are triggered when deemed necessary by the user interface. Any validations that fail are displayed to the user including the type of validation that failed and the widget that it failed on.

[3863] The rules are implemented by a different class for each type of validation. Each of these validation rule classes must know how to check its rule for every type of widget that can be checked. As mentioned in the Context section of this pattern, this will most likely be limited to text entry type widgets. In addition, each validation rule class extends an

abstract validation rule class that defines what types of widgets are supported. This is an implementation of the Visitor pattern.

[3864] **FIG. 131** illustrates a validation rule class diagram.

[3865] Note that the check operations accept a Validate **13200** type of class. Each widget that can be validated with this framework must implement a validateRule method. This simple method accepts some ValidationRule **13202** as a parameter and simply turns around and calls the check method on the rule passing itself in as a parameter. This interaction is shown in **FIG. 132**, which illustrates a rule validation interaction diagram.

[3866] The concrete implementation of the check method will be invoked. This method knows how to extract the data from the particular widget provided and verify the rule.

[3867] The User Interface Validator's job is to associate these rule instances with all of the widgets they pertain to. When the validate method is invoked on the Validator, all of the rules are sent to each of the appropriate widgets via the validateRule method.

[3868] New rules can be added by creating new classes that extend off of the abstract Validation Rule class. No changes need to be made to the widgets.

[3869] Benefits

[3870] Consistency. All user interface validation rule checking is done in the same way using the same rule logic.

[3871] Extensibility. New rules can be added without affecting any other part of the application.

[3872] Automation. Application of validation rules can be automated with a GUI based tool rather easily.

[3873] The associations between a widget and the rule to apply to it should be set up when the user interface is created. A user interface can implement a method that accepts a rule and widget and passes it on to the User Interface Validator as shown in the code example below:

[3874] ValidateTextField userNameField=new TextField("user name");

[3875] ValidateTextField passwordField=new TextField("password");

[3876] ValidateTextArea commentsArea=new TextArea("comments");

[3877] this.addValidation(userNameField, new NotEmptyValidationRule( ));

[3878] this.addValidation(passwordField, new NotEmptyValidationRule( ));

[3879] this.addValidation(passwordField, new NotNumericValidationRule( ));

[3880] this.addValidation(commentsArea, new MaxLengthValidationRule(255));

[3881] The addValidation method on the user interface is shown below.

[3882] public void addValidation(ValidateWidget aWidget, ValidationRule aRule)

```
{
    UIValidator myValidator = this.getValidator( );
    MyValidator.addValidation(aWidget, aRule);
}
```

[3883] The addValidation method on the User Interface Validator is shown below.

```
public void addValidation(ValidateWidget aWidget, ValidationRule
aRule)
{
    Hashtable rulesAndWidgets = this.getRulesAndWidgets( );
    if (rulesAndWidgets.containsKey(aRule))
    {
      aRule = aRule.clone( );
    }
    rulesAndWidgets.put(aRule, aWidget);
}
```

[3884] In the above code, three widgets are created and then associated with various validation rules. The user name and password fields are required and cannot be left blank, the password may not contain any numbers, and the comments text area may not be longer than 255 characters.

[3885] Note that each of the widgets is created with a string that describes a name for the widget that the user would recognize. This name is used in the error list to help a user identify which widget failed validation.

[3886] At some appropriate time, the user interface sends the validate message to the User Interface Validator. This method steps through each of the rules provided to it when the user interface initialized and passes them to their associated widget by the validateRule method. The code is shown below:

```
public Vector validate( )
{
    Vector errors = new Vector( );
    Hashtable rulesAndWidgets = this.getRulesAndWidgets( );
    Enumeration rules = rulesAndWidgets.keys( );
    while (rules.hasMoreElements( ))
    {
        ValidationRule aRule = (ValidationRule)rules.nextElement( );
        ValidateWidget aWidget = (ValidateWidget)
            RulesAndWidgets.get(aRule);
        String anError = aWidget.validateRule(aRule);
        if (anError != null)
        {
            errors.addElement(anError);
        }
    }
    return errors;
}
```

[3887] Note that in the code example above, the validateRule message returns a String rather than a boolean. This string can be passed back to the user interface that invoked the validate and used to describe the errors that occurred to the user.

[3888] Collaborations

[3889] Visitor Each of the rules are implemented as a Visitor according to the GoF pattern of the same name.

[3890] View Configurer

[3891] FIG. 133 illustrates a flowchart for a method 13300 for assigning a view to an activity. Notification is received that a startup event of an activity has occurred in operation 13302. A reference to a first instance of an object created by the startup event of the activity is also received in operation 13304. In operation 13306, a view to launch is determined in response to the receipt of the notification and the reference. The view is based on predetermined criteria. The view is associated with the activity and displayed in operations 13308 and 13310.

[3892] The predetermined criteria may include user preferences, an experience level of a user, security profiles, and/or workflow settings. Also, the activity may be allowed to run without a corresponding view. The activity may also operate on a machine separate from a machine of an end user.

[3893] As an option, a request may be sent to be notified when a new instance of an object is created. As another option, a configuration file may be read for obtaining configuration information.

[3894] How do I associate a new view with the appropriate business activity underneath, in a configurable manner?

[3895] Consider a user interface that displays and collects data for an activity object underneath.

[3896] The ICM/MVC patterns provide for a layered architecture. Each layer talks to a layer below it, and no lower layer talks to an upper layer. For example, the view messages downward to the activity and the business objects, and the activity messages to the business objects. Layers talk down. No layer messages back upward.

[3897] Traditionally, activities launch their views directly. In the example illustrated below, the Search View tells the Search Activity to launch the Customer Maintenance Activity, which then opens up its own view. But this violates the ICM approach, because then the model is talking directly up to the view.

[3898] FIG. 134 illustrates a manner in which the maintain customer activity operation 13400 of the present invention launches its view 13402.

[3899] It might be more appropriate to let the view layer, rather than the activity layer, make decisions about launching other views. The view layer already knows about usability preferences, positioning on the screen, etc. However, one wants the activities to control conversational flow for preconditions, postconditions, workflow, and any other additional business logic.

[3900] A view should not be able to launch a new, separate activity, because that involves business logic. Instead, one wants activities to launch other activities. When the activity layer controls conversational flow, one needs a mechanism to launch views on top of these activities, without violating ICM.

[3901] Therefore, a View Configurer 13500 will be created to manage the relationship between activities 13502, 13504 and views. This would likely be a singleton.

[3902] FIG. 135 illustrates the view configurer 13500 launching the maintain customer view operation.

[3903] The View Configurer is a generic mechanism which allows launching of different views, based on certain criteria. It uses an observable relationship with activity factories to solve this problem. With the View Configurer, developers do not hard code the particular policies for the selection of a view. Moreover, this mechanism allows activities to run without a corresponding view.

[3904] Communication from the activity to the View Configurer will be conducted through broadcasting (as described in the Observer pattern). In this manner, the activity doesn't know about the existence of the View Configurer, it listens to activity broadcasts such as when the controller starts up. This configurer can use the Observable Factory to get a handle to the activity instance.

[3905] There are four main steps involved with the View Configurer observer/observable interface:

[3906] Prior to the flow depicted above, the View Configurer has registered with the Activity Factory saying "Tell me when a new instance is created". This is an example of an "Observable Factory," which can be thought of as a Factory which implements the subject/observable role of the Observer pattern. The Factory needs to be a singleton, so the View Configurer has visibility to it.

[3907] The Search View tells the Search Activity to launch the Maintain Customer Activity.

[3908] The factory for the Maintain Customer Activity creates a new instance of the Maintain Customer Activity.

[3909] Because the View Configurer has pre-registered an interest in the startup of activities, it will receive a broadcast message. In this step, the View Configurer should receive a minimum of two parameters:

[3910] Notification of the startup event that has just occurred.

[3911] A reference to the new instance of the object that was just created.

[3912] The View Configurer then determines which view to launch. This can be based on a variety of criteria, such as user preferences, experience level, security profiles, or workflow settings. The configurer determines the correct view and attaches it to the activity underneath.

[3913] Benefits

[3914] Development. Depending on the distribution model in place, business processing can be executed and tested before the appropriate views have been implemented.

[3915] Automated testing. The View Configurer is particularly useful when you want to use scripts and avoid bringing up windows with automated testing. This is especially true for performance testing, where you might want to run 100 transactions, which might involve instantiating 100 instances of the same activity.

[3916] Running processes in batch mode. The View Configurer allows processes to run without a View, and makes it very simple to connect, disconnect, or reconnect related views.

[3917] Distribution Transparency. In a distributed environment, the process might live on a different machine from the end user's machine. In that case, it cannot launch the view directly, within its own executable. (Unless using a remote windowing system like X-windows, etc.) So the View Configurer allows application architects to transparently move process logic around, depending on the distribution model.

[3918] Collaborations

[3919] The Observer Pattern (Gang of Four Pattern) describes how to provide visibility to other entities via a one to many relationship. A singleton activity factory will create new activity instances, and broadcast the startup of the new activities to the View Configurer.

[3920] An interface for the creation of activities is used in conjunction with the Observer Pattern. In this way, the startup of new activity instances can be broadcast to the View Configurer. This is described in the Factory Pattern (Gang of Four Pattern)

[3921] Generally, there will only need to be one View Configurer per executable. The View Configurer would likely be a singleton, as described in the Singleton Pattern (Gang of Four Pattern).

[3922] Environment Services (1016)

[3923] Environment Services provide miscellaneous application and system level services that do not deal directly with managing the user-interface, communicating with other programs, or accessing data. These services are divided into:

[3924] Operating System Services

[3925] Runtime Services

[3926] Version Management

[3927] Licensing Services

[3928] Error Handling/Logging Services

[3929] Properties

[3930] Task and Memory Management

[3931] Security

[3932] "Miscellaneous services" should not be interpreted as "less important services." In fact, they are vitally important. Developers are more productive when they are not required to be concerned over logging and auditing, error handling and context issues. Obtaining the freedom to largely ignore these issues requires close attention to providing facilities which are well thought out and meld into the application structure.

[3933] Despite the pervasive demands of environmental considerations, many forms of documentation largely gloss over these issues. For example, many times when reading API documentation we find authors disclaim the fact that no error handling or "programming by contract" is shown within the code examples to improve readability. Yet, getting error handling right is key to stability in the execution environment. Programming by contract with the use of preconditions and post-conditions is perhaps the most aggressive style of programming known to date to assure correct programs. Assertion, Exception Hierarchies, Exception Response Table and Polymorphic Exception Handler tackle these problems vigorously by helping to define clearly how to solve some of these key kernel application architecture considerations. The Exception patterns provide a blueprint illustrating how architectural issues can be abstracted into a service level component so that the impact to the application code is minimal.

[3934] A demanding issue in distributed systems is gathering and using trusted information about the clients interacting with the systems. In earlier generations of systems the number of users was a fairly precise calculation—just count the number of workstations which could potentially connect to an application. Information about the users was also a fairly simple matter since they connected directly to the resources from which they were requesting services. Today, with clients offering web services within n-tiered architectures this is no longer easily predictable. In addition, requirements to support these less predictable numbers of users and to have a personal "one-to-one" relationship with them is key to many web strategies. The LoadBalancer and UserContext pattern offer some help in this area. The former addresses strategies for ensuring maximal leverage of the system resources and services and the latter helps in addressing the issue of maintaining state and context about the user. These facilities are mandatory when security, auditing and logging are considered essential properties of the environment.

[3935] Assertion

[3936] FIG. 136 illustrates a flowchart for a method 13600 for testing successfulness of an operation having pre-conditions and post-conditions that must be satisfied for the operation to be successful. A first assertion is raised in operation 13602 asserting a pre-condition that must evaluate to true if the operation is successful. The operation is then executed in operation 13604. A second assertion is raised in operation 13606 asserting a post-condition that must evaluate to true if the operation is successful. An error message is outputted upon failure of at least one of the assertions in operation 13608.

[3937] Optionally, an error handler may be provided for detecting a failure of the assertion of one of the conditions and shutting down a system running the operation upon the detection of the failure. As another option, an assertion may be raised at the beginning and end of every operation of a plurality of operations. Also, a check may be performed prior to raising the assertions for determining the propriety of raising the assertions.

[3938] Each assertion may be raised with descriptions for helping to identify where the assertion failed. Also, each assertion may be raised with parameters for helping to identify why the assertion failed. In one embodiment, two types of assertion classes may be provided. In such an embodiment, one of the assertion classes may implement assertion-checking logic and the other assertion class may implement only null operations, with one of the assertion classes being selected to be raised.

[3939] Every operation has a set of preconditions and postconditions that must be met for the operation to be

considered successful. If these expectations are not met, the system state is in error. How can operations check for these errors so that the handling of these critical errors are consistent across the application?

[3940] Methods typically obtain and return a value, set an attribute based on a passed in parameter, or modify the state of the application based on some complex business rule or ruleset. While there is always some expected result of the invocation of an operation, there are also other, less expected possibilities. The provided parameters may not be within the expected range, thereby causing an error. A communications failure could cause the operation to fail to complete or, worse yet, return incorrect data or leave the system in an inconsistent state.

[3941] Any complete design determines that some formal specification is required to ensure operations complete correctly. This specification is most often in the form of pre- and post-conditions. These conditions define a set of logical expressions that must hold true for the operation to begin and end as expected. These conditions are usually defined during the Design Phase of development. An example is shown in the Operation Diagram below.

[3942] FIG. 137 illustrates an operation diagram depicting an example of pre-conditions 13700 and post-conditions 13702.

[3943] The conditions, in short, define the contract for the method. All of these pre-conditions must hold true before an operation's execution and all of the post-conditions must hold true after an operation's execution. Only then is the operation said to be successful. If any of these conditions fail, a critical error has occurred. The system must assume it is in an inconsistent state and cannot continue processing.

[3944] It is expected that the system programmers will check for pre- and post-conditions systematically in the operations they are coding. This seemingly simple requirement becomes non-trivial when some issues are considered:

[3945] How can multiple developers implement these checks in a consistent manner?

[3946] Some condition checks may be expensive to complete (database and remote component queries). How can these be turned on and off to meet performance expectations? Problem with deferred evaluation; see below.

[3947] How can the exceptions raised when a condition check fails be handled in a consistent manner throughout the system?

[3948] Therefore, a type of object should be developed to represent a check against an operation's conditions. This generic class of objects is known as an Assertion. Application developers should then raise Assertions throughout the system to check the correctness of the code and system state.

[3949] An Assertion accepts conditions that must always evaluate to true. If any of these conditions ever fail, a critical error has occurred and the system should shut down. Pre-conditions and post-conditions are good examples of the type of conditions that should be asserted during an operation's execution.

[3950] The Assertion class is passed a condition that, if evaluated to be false, raises the appropriate errors and shuts the system down. The purpose of this pattern is to formally recognize the pre- and post-conditions of a method in the actual code rather than through developer comments. By implementing an assert( ) method on a common superclass, the interaction with the Assertion class can be hidden from the functional developer. An example of the use of assertions is shown below:

```
public Customer createCustomer(int newCustomerNumber)
{
    Customer newCustomer = null; // declare the new customer
    this.assert(newIdentifier > 0); // pre-condition, a customer's
                                    // identifier must be greater than
                                    // zero
    // code to create the customer
    this.assert(newCustomer != null); // post-condition, the customer
was
                                    // created
    return newCustomer;
}
```

[3951] Assertions can be raised with descriptions and parameters. A description can help to identify where the Assertion failed and a parameter list can help to identify why the Assertion failed.

[3952] Assertions should be raised at the beginning and end of every operation. Prior to raising the Assertion, a check should be made to see if it appropriate to raise one (if assertions are enabled, if performance sensitive assertions are enabled). This can be accomplished by querying the Assertion class for its state before checking the assertion:

```
if(!Assertion.isPerformanceSensitive( ))
{
    // assert
}
```

[3953] All operations will have both pre- and post-conditions. Even in cases where an operation defines an input parameter as something as broad as an integer, it is doubtful that all integers are acceptable to the operation. In this case, an Assertion should be raised to check if the parameter is in the appropriate range.

[3954] A "top-level" error handler should be defined to catch all AssertionExceptions and handle them in a clean and consistent manner. This should include reporting the assertion failure and shutting down the system following an orderly procedure.

[3955] It is important to note the difference between assertions and standard error-handling. Assertions are condition checks that can be turned on and off during runtime whereas standard error handling is always enabled. This is because assertions must always be true. The burden is on the testing process to catch all failed assertions. Thus, a failed assertion should simply never happen in deployed code. However, exceptions can happen, and therefore cannot simply be turned off.

[3956] Benefits

[3957] Ease of Error Identification. Many error are caused by invoking an operation with improper data

(parameters). By formalizing these conditions, it is very obvious is an error was caused by bad data or bad code.

[3958] Correctness. Properly placed assertions assure that the system is in a correct state and responses can be trusted. Assertion checking complements, but does not replace, a comprehensive testing program. The responsibility remains with the designer to identify the correct conditions to assert.

[3959] Consistency. All checks will be made and handled in a similar fashion.

[3960] Control. The enabling and disabling features of the Assertion allows an operations controller to determine when and what checks should be made at runtime rather then development time.

[3961] Flexibility. All handling and clean-up of incorrect assertions is located in one place making changes to this logic much easier to implement.

[3962] Readability. Polices concerning how assertions are actually thrown and handled is not in the functional code.

[3963] Documentation. The code actually documents the design assumptions. This can also be used by documentation generators which read through the code.

[3964] The Assertion class can be defined as shown in the following specification.

[3965] Class Assertion

[3966] void raise(boolean condition) throws AssertionException

[3967] void raise(boolean condition, String description) throws AssertionException

[3968] void raise(boolean condition, Vector parameters) throws AssertionException

[3969] void raise(boolean condition, Vector parameters, String description) throws AssertionException

[3970] boolean isEnabled( )

[3971] boolean isPerformanceSensitiveEnabled( )

[3972] Class AssertionException extends Exception

[3973] One possibility on how to handle the enabling and disabling of assertion checking would be to have two possible types of Assertion class. One which implements the actual assertion-checking logic and another which only implements no-ops. The Assertion instance is then obtains through an AssertionFactory which can be set as to which of the two types to distribute. These settings are determined at runtime.

[3974] It should also be noted that in Java, the exception that is thrown should be a generic run-time exception that doesn't need to be caught by the method or mentioned in the method's throw clause.

[3975] Collaborations

[3976] Factory

[3977] Distributed Garbage Collection

[3978] FIG. **138** illustrates a flowchart for a method **13800** for detecting an orphaned server context. A collection of outstanding server objects is maintained and a list of contexts is created for each of the outstanding server objects in operations **13802** and **13804**. A compilation of clients who are interested in each of the outstanding server objects are added to the list in operation **13806**. Recorded on the list in operation **13808** is a duration of time since the clients invoked a method accessing each of the contexts of the outstanding server objects. The list is examined at predetermined intervals for determining whether a predetermined amount of time has passed since each of the objects has been accessed in operation **13810**. Contexts that have not been accessed in the predetermined amount of time are selected in operation **13812** and information is sent to the clients identifying the contexts that have not been accessed in the predetermined amount of time in operation **13814**.

[3979] After waiting a preselected amount of time for receiving a response from one of the clients, the context may optionally be deleted if a response from one of the clients is not received within the predetermined amount of time. Also, a response may optionally be received from one of the clients requesting that one of the contexts be maintained. In such a situation, upon receipt of the response, a time the context was last updated may be updated to a current time.

[3980] As a further option, a queuing delay may be accommodated for a response from the clients. Also, each of the clients may maintain a collection of all objects the client is interested in. The clients then may send requests to keep alive any objects the clients are currently interested in.

[3981] A client requests a server process but due to abnormal circumstances fails to clean up. How is the orphaned process detected and removed?

[3982] In the design of a stateful server, the LUW Context pattern facilitates the server process constructing domain objects at the request of the clients and maintaining these objects within a given context. Domain objects are entered into a registry with their appropriate context which the server maintains and updates when a request is received to create or delete an object. Each time a context is accessed then a notification is broadcast to the registry, regardless of a state change. With a simple context management, each time a context is referenced by a client a reference counter is incremented and similarly decrements when the reference is destroyed. Once the reference count returns to 0 then the context can be removed from the registry.

[3983] If the context is not explicitly deleted by the client then it will remain in the registry as the server has no way of detecting that the context is orphaned.

[3984] Even if the client application is rigorously designed to ensure all redundant contexts are deleted, an abnormal client event may result in its termination leaving an orphaned server context.

[3985] FIG. **139** illustrates a Client **113900** that has instantiated A **13902** and C **13904**, deletes C but fails to delete A.

[3986] The server still has a reference counter greater than 1 even though the client is no longer interested.

[3987] Therefore, Distributed Garbage Collection should be implemented to ensure that orphaned server contexts are deleted on the server. In the registry for the Garbage Collection the server maintains a collection of outstanding server objects and for each object a list of its contexts, the clients currently interested and the duration since a method was invoked upon a given context by a client. Periodically this list is examined to establish if any of the objects have not been accessed for some configurable time and are candidates for reaping. So, for example, a value of 5 minutes could serve as a default poll event or keep alive interval. If a candidate for a orphaned server process is identified then the clients are sent a message, requesting if they are still interested in the context. This might be performed by publishing an "is anyone interested" message to the registered clients to establish if anyone is interested in the object in its assigned context or by asking the clients explicitly depending on the nature of the architecture.

[3988] The client side also maintains a collection of all of the objects that it is interested in. When it is queried, it instructs the server to keep alive any objects it has an interest in for which a query has been received.

[3989] **FIG. 140** illustrates a GarbageCollector **14000** requesting for interest in context A **14002**. No responses are received from any clients so the server assumes it is orphaned and deletes it.

[3990] If the period configured for a client to respond expires then the context is deleted. This accounts not only for an abnormal termination of the client but for failure of the client application to clean up. However, if a request is received from a client to maintain a context then the time the context was last accessed is updated to the current time and it remains in the Garbage Collection registry.

[3991] **FIG. 141** illustrates a GarbageCollector **14100** requesting for interest in context B **14102**. Client 2 registers interest so the reaper updates the access time stamp and maintains B.

[3992] Benefits

[3993] Cleanup on the Server. Reduces the amount of redundant resources on the server to a minimum. This is especially important if a stateful component is held in a transaction by a client and the architecture prevents additional clients from accessing it, e.g. with BEA's M3.

[3994] Performance. Ensures that only the required contexts are maintained on the server, minimizing the work that the server is required to do, especially during the cleanup process at the end of a UOW.

[3995] Centralization. The collector has a central view over all of the contexts that are currently accessed by all of the clients within a given context. This simplifies the persistence of a context at the end of processing.

[3996] In order to prevent potential race conditions the client must be given sufficient time to respond to the keep alive message from the server before the context is deleted. Typically the client has a separate listener for upward messages originating at the server, so queuing is not an issue at the client end. However, a server is more likely to queue on the receiving end, especially in a system with high message rates.

[3997] Unless there is a dedicated listener on the server it must be configured to accommodate for any queuing delay on receipt of the client response.

[3998] Collaborates With

[3999] Context Pattern Language describes the architecture that is required before the Distributed Garbage Collection is required.

[4000] Variation Of

[4001] Java Shared Namespaces with distributed garbage collection.

[4002] Objectstore PSE WeakArrays.

[4003] Exception Hierarchies

[4004] **FIG. 142** illustrates a flowchart for a method **14200** for creating a common interface for exception handling. Naming conventions of exceptions are determined in operation **14202**. A prefix and/or a suffix is added to each exception interface name in operation **14204** for indicating that the exception interface is an exception. In operations **14206** and **14208**, where an exception error occurred is indicated and a determination is made as to what caused the exception error. Context is provided as to what was happening when the exception error occurred in operation **14210**. Streaming of the exception is allowed to a common interface in operation **14212**. An error message is outputted indicating that an exception error has occurred in operation **14214**.

[4005] As an option, a layer and/or domain may be added from which each exception originates to each of the names of the exception interfaces. As another option, the exceptions may be partitioned into classes based on the way exceptions are handled, exceptions associated with different layers of a system, domains, and/or the source of the exceptions. As a further option, a class may be created which represents a source of the exception and holds an original copy of the exception for avoiding creation of duplicate exceptions. Also, arbitrary exceptions may each optionally support a clone method which creates a copy of the arbitrary exception.

[4006] Developing exception handling logic without classifying and organizing exceptions makes the handling logic cumbersome and fragile to change. How should exceptions be structured?

[4007] The traditional way of conveying errors is by passing error codes from callee to caller. This approach is adequate in some cases, but in general, it is less powerful and more error prone than an exception based approach. In the traditional approach, only minimal information can be passed, such as a failure to locate a configuration file (information on which file has to be provided by some other means). It is also very easy, and common, to ignore the return code. Projects which faithfully test every return code end up mixing a high percentage of error logic with the primary logic. This increases the complexity, and the development, review, and maintenance effort.

[4008] Some computer languages (Java, C++) support an error reporting mechanism based on exceptions. In these languages an exception can be a class type and hold arbitrary information, such as the name of the configuration file that was missing. Also, exceptions cannot be as easily ignored as return codes. If the callee raises an exception and the caller doesn't handle it, the caller's caller is checked to see if it handles the exception. This continues until the exception is handled or the program terminates. Designed properly, the error handling logic will be somewhat separated from the primary logic and will be less dense than the traditional approach.

[4009] The exception class designer is free to create any interface for the class, and each exception class can have its own unique interface. The exception handling logic **14300** will know which exception **14302** was raised (via runtime support) and can make use of the interface particular to the given exception. You can think of the exception handling logic being a set of "chunks" of logic where each chunk handles a specific type of exception. With this in mind, you can see how having many different exception types will cause the exception handling logic to grow. As a new exception type is added to the system, a new "chunk" might have to be added to the handling logic. This is not good. The code is not flexible to change and is in several places. Note **FIG. 143**.

[4010] Suppose you have all these chunks of handling logic and discover that the logic is pretty much the same. For example, assume your architecture is layered and you want to treat all exceptions from the persistence layer the same, such as logging the error and notifying the user. Also assume that the persistence layer can raise any one of fifty exceptions, and more are expected to be added in the future. This is fifty chunks of code that must be present in the exception handling logic, and again, this logic may be in several places. Wouldn't it be nice to write one chunk of handling logic and be done with it?

[4011] Let's take another scenario. Suppose you want to prevent any raised exception from bringing down your system, at least not without a fight. In some cases the error will be unrecoverable and there is not much you can do but release resources (locks, communication channels, . . .) and terminate. What caused the problem is going to be on the tops of the minds of the production support people, and yours when you get their call (always in the middle of the night). You could write the exception handling logic chunks for each exception type—remembering that each exception has its own interface and will require separate logic to handle each interface—for each exception, but now you have to handle all the exceptions in the system. Wouldn't it be nice to write one chunk of handling logic and be done with it?

[4012] Therefore, to simplify the error handling logic and be able to treat groups of exceptions the same, a few techniques should be used to organize and define the exception interfaces.

[4013] The first step is to create an exception interface that all other interfaces will use or extend. It is not possible to provide one here as it greatly depends on the requirements at hand. But here are some guidelines:

[4014] Determine the exception naming conventions. Use either a prefix or suffix to indicate that the interface is an exception. Also consider naming exceptions with the layer or domain they originate from. For example you may have an exception, CsAddressExcp, which is owned by the Customer Acquisition domain.

[4015] Provide a means to determine where the error occurred (file, line, client or server, layer, . . .) so that it can be investigated.

[4016] Provide a means to determine what happened (could not open file: XYZ).

[4017] Provide context as to what was happening (Saving account information).

[4018] Provide a way to stream the exception or stringify it.

[4019] Consider separate production messages versus debug messages.

[4020] Don't try to indicate severity. This is determined by the context of the caller, not the callee.

[4021] The intent is to be able to handle any arbitrary exception the same by having a common interface. Take time and get this right, to avoid updating several other exceptions later.

[4022] Now that this base exception interface is available, any handling logic can treat all exceptions alike; only one chunk of logic needs to be written. Specific exceptions can still be handled on a case by case basis as required. You can extend this concept to further partition the exceptions by creating a tree of exception types. By handling any exceptions at particular point in the tree, you effectively handle all exception types below that point. The trick is in creating a useful tree. Here are some guidelines:

[4023] Determine where handlers will be put and how they will respond to each exception. If you find that many are handled in the same way there may be a natural grouping that can be leveraged.

[4024] Consider the stability of your grouping. Is the group cohesive or is regrouping likely?

[4025] If parts of your system are layered, consider a branch that consolidates each layer. This enables a handler to deal with all exceptions emanating from a given layer.

[4026] Consider grouping by domains (Legal, Finance).

[4027] Consider grouping by subsystem.

[4028] Consider common problems such as parameter validation, pre- and post-conditions.

[4029] Consider the source (client or server).

[4030] **FIG. 144** illustrates that groupings are not always exclusive. It is possible to group some exceptions **14400**, **14402**, **14404** by layer and then domains within that layer.

[4031] Benefits

[4032] Simplicity. Simplifies handling logic by being able to write a handler that deals with the base exception type.

[4033] Consistency. Consistent approach to error handling.

[4034] Maintainability. Minimizes coding changes by reducing the multiple number error handling chunks

[4035] Manageability. Provides Conceptual Framework

[4036] The solution section covered many of the considerations in creating the exception tree so this section only provides some additional details to consider.

[4037] Wrapping and delegation can be used to simplify in certain situations. Consider a distributed application and the need or desire to handle server and client exceptions differently, or to know the source of the error. One way to avoid creating duplicate exceptions (one per source) is to create a class which represents the source and holds the original exception. For example AaServerExcp can hold a pointer to the base class AaExcp. The handling logic can catch AaServerExcp exceptions and then access the held exception. An alternative is to put a code in the base class with indicates source but then all logic needs to know to set this value and all handling logic needs to test for it.

[4038] To hold onto an arbitrary exception you need a way of creating a copy of it, but you may not know the actual type of the exception. In C++ the exception will be destroyed when you leave the handling logic, so you need the ability to create a copy to hold onto. A common technique is it have all exceptions support a "clone" method which creates a copy of themselves.

[4039] Consider how to stream an exception so it can be sent from server to client.

[4040] Exception Response Table

[4041] FIG. 145 illustrates a flowchart for a method 14500 for recording exception handling requirements for maintaining a consistent error handling approach. An exception response table is provided in which an exception is recorded in operations 14502 and 14504. The context of the exception is entered in the exception response table in operation 14506 and a response for the exception is listed in the exception response table in operation 14508. The response is subsequently outputted upon the exception occurring in the context in operation 14510.

[4042] A typical response and a last resort response may be listed in the exception response table. The typical response may also be outputted upon the exception occurring in the context. The last resort response may be outputted upon the exception occurring out of the context. Additionally, abbreviations may be used to reduce an output size of the exception response table. Further, the exception response table may also include an exception category field for permitting organizing multiple exceptions by source. Optionally, an optimization may be determined that can be made based on similar entries in the exception response table. Further, the optimization made may also include classifying the exceptions for organizational purposes.

[4043] The response to an exception may vary per exception type and the context in which it is thrown, such as being thrown on the client or server, and the context in which it is handled. How do you record the exception handling requirements?

[4044] During exception handling design there are several aspects to capture to achieve a consistent approach:

[4045] The set of exceptions to be handled

[4046] The set of responses to these exceptions

[4047] The context in which the exception is handled; e.g. client or server, batch or GUI

[4048] The set of exceptions to handle and their organization structure varies by project. Typically exceptions are organized into hierarchies to facilitate handling. The response to an exception may vary by exception type, the context in which it was thrown, and the context in which is handled. Here are some examples of error handling decisions of a hypothetical project.

[4049] "All exceptions thrown on the server, and not handled by the server logic, will be propagated to the client."

[4050] "The current transaction is aborted if a server exception is not recoverable"

[4051] "All Server exceptions derived from Excp will be logged if not handled by the server code. The last resort handler will ensure this."

[4052] "GUI clients will display the error information in a splitter window"

[4053] "Batch clients will send error information to Operations"

[4054] These few examples demonstrate how context (Batch, GUI, Client, Server, last resort) can affect the handling of exceptions, and that even in a given context, the exception type may play a role in the handling. In a real system there may be several other context and exception-type specific requirements.

[4055] There are two common exception handling contexts that should be present in most systems. One is referred to as the Typical Response and the other is referred to as the Last Resort Response. The Typical Response is the error handling code intentionally added to handle exceptions. For example, car.start( ) is likely to fail due to being out of gas. The Typical Response may be to fill the tank and retry. The Last Resort Response is what to do when an exception is not handled (the Typical Response could not handle the error, such as a hole in the gas tank). Last Resort Response is a way of capturing what should be done when application code fails to handle an error. Recovery is usually not possible at this point but the handler may be coded to log the error and notify Operations of the problem. Without this response, systems may crash unnecessarily, or without indicating what happened.

[4056] All these permutations of exception types, contexts, and responses need to be managed in order to maintain a consistent error handling approach.

[4057] Therefore, use an Exception Response Table to capture the exceptions in the system, and the appropriate responses by context. What is important to capture is the exception, context, response, information, documenting the error handling requirements.

[4058] The following table lists exceptions by category and type, with the typical and last resort response. Other

contexts and responses are listed within these columns. The exception category field is optional but can help to organize exceptions by their source (application, architecture, . . .) or hierarchy. This table can become quite packed with response information so a nomenclature may need to be developed to condense the information. The implementation section provides an example of this; Other ways of formatting this information are possible.

Exception	Typical Response	Last Resort Response
Exception Category		
Exception Name Description . . .		
Exception Category		
Exception Name Description		

[4059] Benefits

[4060] Requirements Traceability. Exceptions requirements are captured and managed through implementation.

[4061] Hierarchy Design. Analysis may show optimizations that can be made such as handling a subtree of exceptions with the same code, as the response is the same to any exception in the subtree.

[4062] Interface Design. Discovery of interface requirements on the exception classes to support a particular response is another benefit.

[4063] Handler design. Assists in exception handling design by identifying common responses that can be leveraged by the handlers.

[4064] The table below shows an example of an Exception Response Table for a fictitious client/server system. This is followed by the nomenclature section which is customized per project.

Name	Typical Response	Last Resort Response
Architecture Framework Exceptions		
AnAssertionExcep	C: N/A	C: L, Popup(severe), Shutdown
Assertion failure	S: N/A	S: L, N, P(AnServerAnExcep), Shutdown
AnExcep Base class for exceptions	C: N/A S: N/A	C: N/A S: N/A
Application Exceptions		
CoBalanceExcep Account out of balance	C: Popup(severe) S: P(AnServerAnExcep)	C: L, Popup(warn) S: L, N, P(AnServerAnExcep)
Nomenclature:		

Note: Abbreviations were used so that the table could be printed. The nomenclature section is only meant to serve as an example.

[4065] Context:

[4066] C=Client

[4067] S=Server

[4068] Response:

[4069] N/A=not applicable; don't handle

[4070] L=log error

[4071] L(diagnostic)=log errors for diagnostic purposes only

[4072] N=notify operations

[4073] Optional=application, context dependent. Not required to be caught

[4074] P=pass exception to client

[4075] P(<exception>)=pass given exception type to client, will be different from type caught

[4076] Popup(warn)=display warning message

[4077] Popup(severe)=display severe warning message

[4078] Popup(retry)=display retry message

[4079] Shutdown=release resources and shutdown gracefully.

[4080] Exception Hierarchy discusses how to organize exceptions.

[4081] Last Resort Exception Handling describes where handlers should be placed to prevent a program from terminating without warning.

[4082] Polymorphic Exception Handler describes how to design and code exception handlers that reduce the impact of changes and the overall size of the error handling logic.

[4083] Polymorphic Exception Handler

[4084] **FIG. 146** illustrates a flowchart for a method **14600** for minimizing the amount of changes that need to be made to exception handling logic when new exceptions are added. Exceptions are organized into hierarchies in a polymorphic exception handler in operation **14602**. A root of one of the hierarchies in which an exception occurs is caught in operation **14604**. The exception is instructed to rethrow itself in operation **14606**. The rethrown exception is caught and identified in operations **14608** and **14610**. A type of the rethrown exception is determined in operation **14612** and a message is outputted indicating the type of the rethrown exception in operation **14614**.

[4085] Single exception interfaces may be used as the roots of the hierarchies. Also, the polymorphic exception handler may handle each unique root. Further, an added exception may be organized into a hierarchy and handled by the polymorphic exception handler. As an option, handling behavior may be encapsulated in the polymorphic exception handler. As additional option, catch blocks may also be created to catch the rethrown exception.

[4086] Large systems can be quite complex and require error management integrating disparate components and/or libraries (i.e. DBMS APIs, data structures library, middleware, etc) How can exception handling logic be written so that little or no changes are required when new exceptions are added to the system?

[4087] A software system using exceptions as the error handling approach may have to respond to a variety of exceptions. Handling each exception type on a case by case basis is cumbersome and expensive, both in terms of initial development and subsequent maintenance. In languages such as Java and C++, the mechanism to handle exceptions is to use try-catch blocks which look like this:

```
try
{
    // perform some work here
}
catch (ExceptionTypeA& excp)
{
    // Exception A thrown.Handling logic here
}
catch (ExceptionTypeB& excp)
{
    // Exception B thrown. Handling logic here
}
catch (...)
{
    // Don't know what was thrown, but still need to handle it.
}
```

[4088] This example shows only two explicit exception types being handled but a system typically has several potential exceptions. If the development of the exception types is poorly designed the try-catch blocks can become quite large as they attempt to handle each exception. Imagine trying to handle, say, fifty more exception types, in several places, in the code. The error handling code expansion is exponential! **FIG. 147** depicts a program **14700** (i.e., the exception handler of the present invention) with a few try-catch blocks **14702**. As more exceptions are added these blocks expand to handle each new exception.

[4089] Another problem with exception handling logic is that it can be quite involved, such as logging the information to a persistent store, notifying Operations support, rolling back a transaction, etc. the example only showed one commented line to represent the code. Again, imagine each catch block requiring several lines of code. This logic may be repeated in each catch block.

[4090] Taken together, varying exception types and potentially repeating and complex logic in the catch blocks, the development and maintenance efforts regarding error handling are going to be much more expensive than they need to be.

[4091] Therefore, structure the exceptions into hierarchies, create an exception handler object that performs the catch block logic, and minimize the number of catch blocks required to support a given try-block.

[4092] Exception Hierarchies organizes exceptions into hierarchies and facilitates the design of exception handlers. Handlers can then be designed to handle the roots of hierarchies. This is much simpler than handling each exception type on a case by case basis. In custom development where the project has control of all code, a single exception interface can be used as the root. The more likely situation is some custom development and using third party libraries which may also use exceptions. In these cases, the exception handler will handle each unique root.

[4093] Using an exception handler, versus custom logic per catch block, reduces the maintenance and development effort as the code is easier to read, there is less of it, and any changes that need to be made can be made in one place.

[4094] The following code snippet shows the form of the try-catch blocks using the polymorphic exception handler. It may seem equivalent to the prior catch-block example but it is not. The first distinction is the type of exceptions handled. In this case, the roots of the exception hierarchies are caught, not the individual exception types. For this example there are only two exception hierarchies in the system, so only these roots are handled. What this means is that as new exceptions are added to the hierarchies, this code does not change, and remember, this code is in several places in the system.

[4095] The second difference with this code is the encapsulation of the handling behavior in the exception handler. The handle method can perform arbitrarily complex logic behind the scenes, and if this needs to change, is changed in one place. For example, if the current handling logic logs a message to disk and now needs to be extended to notify Operations personnel, this can be centralized in on place. The code as written does not need to change.

```
try
{
    // perform some work here
}
catch (ExceptionRoot& excp)
{
    ExceptHdlr hdlr;
    hdlr.handle(excp);
}
catch (OtherPartyRoot& excp)
{
  ExceptHdlr hdlr;
    hdlr.handle(excp);
}
catch (...)
{
  ExceptHdlr hdlr;
    hdlr.handle( );
}
```

[4096] **FIG. 148** depicts the same program **14800** (the polymorphic exception handler) with smaller catch blocks **14802**. A handler **14802** has been added which is consolidates the common code and the number of catch blocks has been reduced overall by making the handler responsible for handling each exception. The downside is that now the handler is subject to frequent change as exceptions are added to the system. The maintenance effort outweighs this disadvantage.

[4097] The examples have shown a single exception handler being used. In practice it is more likely that multiple will be used. For example, the exception handler on a server may have different requirements or constraints than a client, or one client may be GUI based and display pop-up error messages, where another client is a batch program that needs to send notification messages to Operations. This can be handled by creating multiple handlers or using the Strategy pattern to customize the behavior.

[4098] Benefits

[4099] Simplicity. Reduces development and maintenance effort required for exception handling.

[4100] Maintainability. Reduces impact of changes

[4101] Robustness. Centralizes/Encapsulates handling logic

[4102] Flexibility. Multiple handlers can be used

[4103] The exception base class declares a method, rethrow, which is used by the handler to determine the real type of the exception. Another approach is to use double dispatch which may be show in a future version. Below is an example of this interface only showing the essential detail.

```
//.....................................
// Base Class of Exceptions
//.....................................
class Exep
{
public:
    // Rethrow the exception. Throw *this;
    virtual void rethrow() const = 0;
};
//.....................................
// Example Derived Class of Exceptions
//.....................................
class Derived : public Exep
{
public:
    virtual void rethrow () const { throw *this; }
};
//.....................................
// Example Derived Class of Exceptions
//.....................................
class SubDerived : public Derived
{
public:
    virtual void rethrow() const { throw *this; }
};
```

[4104] When the exception handler is passed the exception from the catch-block all it knows is that it has a base exception type. For some projects this may be sufficient if the exception interface is rich enough and all exceptions are treated the same. In other cases, exceptions may require specialized treatment. With the rethrow mechanism in place, the handler can create a try-catch block and have the exception rethrow itself. The catch blocks are then used to catch the specific exception type.

```
//.....................................
// Exception Handler
//.....................................
class ExceptionHandler
{
public:
    ExceptionHandler( );
    // Handle the exact exception
    void handle(const Exep& );
    // Handle a third party exep
    void handle(const ThirdPartyExep& );
};
//.....................................
// Handle the exception
//.....................................
void ExceptionHandler::handle(const Exep& e)
{
    // Rethrow the exception to get the specific type
    // Note that catches are in the order of most specific to
    // most general.
```

-continued

```
try
{
    e.rethrow( );
}
catch(SubDerived& exep)
{
    // Handle SubDerived
}
catch(Derived& exep)
{
    // Handle Derived
}
catch(...)
{
    // Handle a pointer here since nothing matched it.
    }
}
ExceptionHandler::handle(const ThirdPartyExep& e)
{
    // Handle based on ThirdPartyExep interface
    // Can't rethrow because ThirdPartyExep doesn't support this.
    // Could use RTTI if needed.
}
```

[4105] Load Balancer

[4106] FIG. 149 illustrates a flowchart for a method 14900 for distributing incoming requests amongst server components for optimizing usage of resources. Incoming requests are received and stored in operations 14902 and 14904. An availability of server components is determined and a listing of available server components is compiled in operations 14906 and 14908. A determination is made as to which server component on the listing of available server components is most appropriate to receive a particular request in operation 14910. Each particular request is sent to the selected server component determined to be most appropriate to receive the particular request in operation 14912.

[4107] Optionally, the determination of which server component is the most appropriate may be performed by allocating the requests on a round-robin basis whereby requests are assigned to consecutive server components by traversing along the listing of available server components. As another option, the determination of which server component is the most appropriate may also include calculating an amount of utilization that each available server component is currently experiencing.

[4108] The amount of utilization of each available server components may be calculated based on current CPU utilization, kernel scheduling run-queue length, current network traffic at a node to the server component, and/or a number of requests currently being serviced. Also, a request may be remoted to a different available server component upon a crash of the selected server component. Additionally, the server components may be saved in a persistent store, wherein a check is made to determine whether a connection to a server component needs to be reestablished.

[4109] In order to support scalability in a high volume distributed component environment, resources tend to be replicated. How can incoming requests be distributed amongst the available server components in order to optimize the usage of system resources?

[4110] In a distributed system, server components provide functions and data that can be accessed by client compo-

ments. Many identical copies of a server component can be running on different platforms in the system in order to support large volumes of client requests.

[4111] In order to make use of the system's scarce resources, some way of routing an incoming request to the best server component available is required. In general, all requests take a similar length of time to service.

[4112] FIG. 150 illustrates server components 15000 receiving service requests 15002.

[4113] Therefore, use Load Balancer to select the best server component out of an available pool for the client to use.

[4114] FIG. 151 illustrates a load balancer 15100 mediating the requests of FIG. 150.

[4115] Incoming client requests are routed by the Load Balancer to the best available server component.

[4116] A number of possible strategies exist for deciding which server component is the most appropriate at a given point in time.

[4117] Round Robin—Allocate the received requests on a round-robin basis, whereby a list of the available server components is created and, as requests are received, they are allocated by traversing down the list. When the end of the list is reached, the next request is allocated to the server component at the beginning of the list.

[4118] Utilization Based—Allocate the received requests based on the utilization that each server component is currently experiencing. The definition of utilization can be tailored to meet specific requirements or deployment strategies. It may be based on a combination of current CPU utilization, kernel scheduling run-queue length, current network traffic at that node, number of requests currently being serviced, or any other factors particular to the environment.

[4119] Benefits

[4120] Performance. Based on the selection strategy employed, the client is connected to the server component that is best able to serve it.

[4121] Scalability. As the number of users and requests increase, processing can be distributed across the available resources.

[4122] Robustness. In the event of the server crashing, the client can then ask the Load Balancer to provide another server component for it to use. This can be extended still further by federating Load Balancers and their associated server component pools.

[4123] The following is the IDL that was used to define the Load Balancer:

```
interface LoadBalancer
{
    Object getService ( )
        raises ( ArchitectureException );
    void register ( in Object aServerComponent )
        raises ( ArchitectureException );
};
```

[4124] Collaborations

[4125] Round Robin Load Balancing

[4126] Utilization Based Load Balancing

[4127] User Context

[4128] FIG. 152 illustrates a flowchart for a method 15200 for maintaining a security profile throughout nested service invocations on distributed components. In operation 15202, interconnections are provided between distributed components each having nested service invocations. A user is identified in operation 15204. The user is associated with roles in operation 15206. In operation 15208, a user context instance is created upon successful identification of the user. The user context instance also includes information about the user including the roles. A request is received from the user to invoke a service on a component in operation 15210. The component invokes an additional service of another component. The user context is queried for the information about the user in operation 15212. The user information is compared with an access control list for verifying that the user has access to the component in operation 15214. The user information is also compared with an access control list for verifying that the user has access to the additional service of the other component in operation 15216.

[4129] Optionally, all user interactions may be logged as well. As another option, a user interface may be modified to provide access to actions that can be performed by the user based on an identity of the user and the roles associated with the user. The user context instance may also be passed along as a parameter of service invocations. Additionally, the service invoked may associate any objects created, updated, or deleted with the user context instance. As a further option, the user context instance may also encapsulate security certificates of the user.

[4130] For security and auditing purposes, user information must be maintained throughout a service's implementation across multiple, distributed platforms. How can this original security profile be maintained throughout nested service invocations on distributed components?

[4131] All mission-critical systems require some form of security and auditing capabilities. These capabilities restrict who can use the system and what they can and cannot do and, in the case of a security breach or dispute, resolve who did what and when.

[4132] To meet these capabilities, users must be identified, associated with roles and granted authorization before any operation proceeds. In addition, all user interactions and results of those interactions may be logged. On a user interface, access to certain panels and controls are granted according to a user's role.

[4133] In a distributed, component-based system, these complex requirements become even more difficult to implement. Typically, a client (or user) invokes some service on a component. That component may invoke any number of

additional services on any number of additional components to complete its designated task. These successive service invocations are a result of the initial client request so the security profile that allowed the initial request must also allow all successive requests.

[4134] **FIG. 153** illustrates a component interaction diagram showing an interaction between a number of components in a financial system. A user initiates an addStock( ) service on the Portfolio component **15300**. To perform the addStock( ) service, the Portfolio must use the getStockPrice( ) and the deductFromAccount( ) services on the Market and Finance components **15302**,**15304**, respectively. This implies that a user who can access the addStock( ) service must also have permissions to access the getStockPrice( ) and the deductFromAccount( ) services. This may need to be checked by each of the distributed components within the context of one logical service. In addition, auditing what has been done, or perhaps requested to be done, adds another common requirement that must be accounted for. A component servicing multiple clients must associate client requests with corresponding services invoked on business objects. This information must be persisted as each change is committed.

[4135] Therefore, represent information about a user in a shared User Context object. This object maintains a user's unique identification that can be subsequently checked against a resource's access control list (ACL). A User Context instance is created upon a user's successful, validated identification to the system (usually through some "login" mechanism). After that, the system user interface can modify itself to provide only the actions that can be performed by that particular user acting in a particular role. Controls may query the User Context and modify their own visual state as needed (enable/disable, hide/show).

[4136] The User Context can also be passed along as a parameter of service invocations. All public, stateless services on a component should provide for a User Context to be passed along as a parameter. The service being invoked can then associate any Business Objects created, updated, or deleted as a result of the service invocation with the User Context.

[4137] One example of this would be a User Manager **15400** associating a User Context instance **15402** with the Business Objects **15404** they are affecting. **FIG. 154** illustrates a user manager/user context relationship diagram.

[4138] These associations can be used for auditing purposes. When a change to a Business Object is committed, a log entry can be created tying the change with the user that triggered it.

[4139] Benefits

[4140] Common User Representation. One single representation of a user and their access rights can be shared across all areas of the system.

[4141] Extensible Security. Because there is one source for the User Context various policies or strategies could be used to identity and authenticate the User within a context. For example, it could encapsulate the User's certificates that allow more advanced security strategies to determine authorization.

[4142] Class UserContext

[4143] UserContext(Identifier identifier)

[4144] Identifier getIdentifier( )

[4145] String getName( )

[4146] void setName(String newName)

[4147] void addRight(String accessares, AccessLevel level)

[4148] void removeRight(String accessArea, AccessLevel level)

[4149] Vector getRights(String accessArea)

[4150] boolean canCreateIn(String accessarea)

[4151] boolean canReadIn(String accessArea)

[4152] boolean canUpdateIn(String accessArea)

[4153] boolean canDeleteIn(String accessArea)

[4154] Class AccessLevel

[4155] static AccessLevel create( )

[4156] static AccessLevel read( )

[4157] static AccessLevel update( )

[4158] static AccessLevel delete( )

[4159] boolean=(AccessLevel anAccessLevel)

[4160] It is expected that the User Context will be passed from component to component. In this case the User Context will have to be defined using some sort of interface language definition (IDL).

[4161] Collaborations

[4162] Permission

[4163] Policy

[4164] SecurityManager

[4165] Logging

[4166] Alternatives

[4167] MTS & EJB offer an environment that does not require the passing of the context with every operation. A container as a set context type> that provides a handle within the component for the methods to access the cached context.

[4168] Information Services Patterns

[4169] Reliable information access mechanisms in a multi-user environment are a crucial, technical issue for almost all systems that a user builds.

[4170] Most business information systems manage data which must be saved in non-volatile storage (e.g., disk). The data must live, or "persist," between invocations of any particular application or program. Persistence is the capability to permanently store this data in its original or a modified state, until the information system purposely deletes it. Relational databases, object databases, or even flat files are all examples of persistent data stores.

[4171] This section discusses issues and approaches for developing an object-oriented persistence architecture.

[4172] A key issue frequently encountered in the development of object-oriented systems is the mapping of objects in memory to data structures in persistent storage. When the persistent storage is an object-oriented database, this mapping is quite straightforward, being largely taken care of by the database management system.

[4173] In the more common situation where the persistent storage is a relational database, there is a fundamental translation problem or a so-called "impedance mismatch". The physical, logical, and even philosophical differences between a relational and object data storage approach are significant. Mapping between the two is hard. The architecture must, in this case, include mechanisms to deal with this impedance mismatch.

[4174] The impedance mismatch is due to the following contrasting features of objects/classes and tables:

[4175] Identity: Objects have unique identity, regardless of their attributes. Tables rely on the notion of primary key to distinguish rows. While a relational DBMS guarantees uniqueness of rows with respect to primary keys for data stored in the database, the same is not true for data in memory.

[4176] Inheritance: This is a meaningful and important notion for classes, it is not meaningful for tables in traditional RDBMSs.

[4177] Navigation: The natural way to access and perform functions on objects is navigational, i.e., it entails following references from objects to other related objects. By contrast, relational databases naturally support associative access, i.e., queries on row attributes and the use of table joins.

[4178] The patterns in this section focus on problems and solutions associated with using a relational DBMS with an object-oriented persistence architecture.

[4179] A key objective of a comprehensive object-to-relational persistence architecture is shielding the application business logic and developers from the relational structure. The benefits are a simplified environment for business developers, reduced distraction with technical issues, and increased focus on the business object model and functional logic. However, in order to reap these benefits, a significant investment in architecture development is typically required.

[4180] The scope of a persistence architecture can range across the following levels of transparency and automation:

[4181] Heavyweight, fully-automated, including the mapping of the object model to the database schema and generation of all the database access code. Variants of this architecture type may allow the customization of database access code (e.g., for optimization purposes).

[4182] Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se.

[4183] Minimal persistence approach in which each business object is directly responsible for database access.

[4184] Of course, there is a tradeoff between transparency, automation, and flexibility on the one hand, and architecture complexity and development cost on the other.

[4185] The patterns in this section solve several of the fundamental problems encountered in the development of an object-to-relational persistence architecture, including the mapping of classes to tables (Data Handler, Individual Persistence), identity management (Object Identifiers as Object), caching(Object Identity Cache), allocation of responsibilities (Data Handler, Piecemeal Retrieval, Persistent State Separate from Persistent Object), and data access optimization (Multi-Object Fetch) and the mapping of basic SQL types to object attributes (Attribute Converter).

[4186] In addition to providing persistence capabilities, reliable information access mechanisms in a multi-user environment must support transaction semantics. As the real-life implementation of all of the patterns in this section requires integration with transaction management frameworks, the Persistence patterns should be considered and used in conjunction with the patterns in the Transactions section.

[4187] Attribute Converter

[4188] FIG. **155** illustrates a flowchart for a method **15500** for translating an object attribute to and from a database value. A database is provided and a conversion process is determined for converting an object attribute to and from a database value in operations **15502** and **15504**. The conversion process is encapsulated in an attribute converter. A first object attribute is directed to the attribute converter for conversion to a first database value in operation **15506**. A second database value is directed to the attribute converter for conversion to a second object attribute in operation **15508**.

[4189] A different attribute converter may be created for each type of conversion of object attributes to and from database values. In addition, the attribute converters may also implement a common interface. Further, all attributes of the same type may be directed to a particular attribute converter. Optionally, a second attribute converter may be substituted for the attribute converter for altering the conversion of the attribute. As an another option, the attribute converter may be altered for relieving a performance bottleneck.

[4190] Object attributes must go through some translation before they are written to and after they are read from some persistent stores. How can you isolate the translation algorithm from the persistent object and the persistence mechanism?

[4191] When interacting with a relational data store, the attribute value doesn't always map directly to a database type. Other times, an attribute value maps to more than one database type.

[4192] For example, in an Object based system, an attribute with a Boolean value is often converted from a Boolean object to a "T" or "F" string before it is saved in the database. In another example, a phone number attribute might be composed of an area code (847), an exchange (714) and an extension (2731). These three field might be saved in three separate database columns or combined into one before they are saved in the database.

[4193] FIG. 156 illustrates that an attribute 15600 can't be saved directly into the persistent store 15602.

[4194] An impedance mismatch exists between the attribute and the data store and a conversion must take place.

[4195] The logic to perform this conversion can vary from one attribute to another. Based upon the attribute type, a different conversion must take place. In addition, special situations can arise where the same type of attribute will be stored differently in different situations. It is desirable to reuse this logic; however, the solution must be flexible enough that the developer is not locked into one single translation for an attribute type.

[4196] Therefore, use an Attribute Converter to translate database values to object attributes and vice versa.

[4197] FIG. 157 illustrates the use of an Attribute Converter 15700 to save an attribute 15702 into a database 15704.

[4198] The knowledge of how to translate an attribute value to and from a persistent store is encapsulated in a separate Attribute Converter object.

[4199] The attribute's value should not be obtained directly from the attribute prior to saving it in the database. Nor should an attribute be instantiated directly from the raw value obtained from the persistent store. Values should be obtained or attributes created exclusively via an Attribute Converter.

[4200] It is recommended that a different Converter be created for each type of conversion required. This keeps the Converter's knowledge very specialized. As a result, the combination of Converters required to persist an entire object to a persistent store is very flexible due to the modularity of the Attribute Converter objects.

[4201] Benefits

[4202] Reuse. For some types of attributes, the conversion process can be rather involved. If this knowledge is encapsulated in an Attribute Converter, it can be reused for converting other attributes of the same type.

[4203] Flexibility. If the conversion for a specific attribute needs to be altered, simply substituting a different Attribute Converter will alter the behavior and not disrupt the rest of the application.

[4204] Maintainability. Altering a single Attribute Converter can affect several attributes in the system. For instance, if the conversion of one specific type of attribute is identified as a performance bottleneck, altering the corresponding Converter can benefit a large part of the system.

[4205] Ideally, all Attribute Converters should implement a common abstract class or interface. This allows the architecture to treat all Converters equally. The architecture need not know the specific translator class it is using.

[4206] The interface may look something like this.

```
public interface AttributeConverter
{
    public String translateValueForDataStore(Object anAttribute);
    public Object translateValueFromDataStore(String aColumn,
java.sql.ResultSet aResultSet);
}
```

[4207] The first behavior, translateValueForDataStore, takes an attribute and translates it into a String that can be used in an SQL statement. The second behavior, translateValueFromDataStore, takes a column name and JDBC result set as arguments. It then answers the attribute translated from the given result set. Each implementation of AttributeConverter must then implement both behaviors in their own specific way.

```
public class BooleanConverter implements AttributeConverter
{
    public String translateValueForDataStore(Object anAttribute)
    {
        String value = null;
        if(anAttribute != null)
        {
            if((Boolean)anAttribute).booleanValue ())
            {
                value = "T";
            }
            else
            {
                value = "F";
            }
        }
        else
        {
            value = "NULL";
        }
        return value;
    }
    public Object translateValueFromDataStore(String aColumn,
java.sql.ResultSet aResultSet)
    {
    Boolean result = null;
    String value = null;
    value = aResultSet.getString(aColumn);
    if(value.equalsIgnoreCase("T"))
    {
        result = new Boolean(true);
    }
    else if(value.equalsIgnoreCase("F"))
    {
        result = new Boolean(false);
    }
    return result;
}
}
```

[4208] The Boolean Converter above knows how to translate a Boolean object to and from a character representation in the relational database.

[4209] Collaborations

[4210] Normalized Mapping—The Mapper contains all information required to store an object in a relational data store. Attribute Converter can be utilized by Normalized Mapping to store the knowledge needed to properly translate attribute state values to and from the persistent store.

[4211] Denormalized Mapping—Denormalized Mapper is another pattern for mapping objects to a relational database. The Attribute Converter pattern could be used by Denormalized Mapper to provide conversion between attribute values and database values.

[4212] Alternatives

[4213] Case Statements—Case statements aren't really a pattern, but they are an alternative to Attribute Converter. A case statement could be implemented in the super class to handle the translation of the data.

[4214] Data Handler

[4215] **FIG. 158** illustrates a flowchart for a method **15800** for controlling data. A data retrieval mechanism is provided in operation **15802** for retrieving data from a database. The data retrieval mechanism also writes data to the database. In operation **15804**, the data retrieval mechanism is encapsulated in a data handler. A request from a domain object is received for a retrieval of a portion of the data in the database in operation **15806**. The data retrieval mechanism is utilized in operation **15808** to retrieve the portion of the data from the database. The portion of the data is passed through the data handler to the domain object in operation **15810**.

[4216] The data retrieval mechanism may be capable of being used by a plurality of domain objects. Also, the data retrieval mechanism may capable of being used by only one of a plurality of domain objects. Dependencies on the data retrieval mechanism within the data handler may also be managed via code generation.

[4217] The data handler may physically partitioned into a component separate from the domain object. Optionally, the domain object may write attributes to a data stream. In such a case, the data handler may define an order in which the attributes are written to the data stream. Also, a row class may define the attributes in the same order as the attributes appear on the database. Further, the data handler may iterate over the attributes and may save them to the database.

[4218] Business Objects in memory generally store and retrieve their data members from some type of persistent store. When using Individual Persistence, how can we ensure that the retrieval mechanism used by the domain object is independent of the business logic?

[4219] Individual Persistence assigns responsibility for data access at the level of individual domain objects. Each domain object or class can retrieve, update, insert, and delete its data from a persistent store independently of other objects or classes. This promotes encapsulation and reuse across business transactions.

[4220] **FIG. 159** illustrates the data retrieval mechanism calls being placed directly within the domain object **15900** (in this example SQL is inserted into the Account business object).

[4221] When persistence is at the class level, it is typical to code the actual SQL, serialization, or CICS call directly in the class itself. In the example shown above, an "Account" object can contain the SQL needed to retrieve and save its state to the database.

[4222] This approach can reduce the flexibility of the domain object, in that, changes to the access logic or the backend database must result in changes to the business object or class. How can we ensure that the business logic is *independent* of the data retrieval mechanism for such a class?

[4223] **FIG. 160** shows the interrelationship between a database **16000**, a persist **16002**, and an account **16004**.

[4224] Business objects delegate their data retrieval mechanism to an appropriate handler. This Data Handler can be either be generic or specific to each type of domain object used. To minimize the impact of changes, dependencies on the database schema or data retrieval mechanism within the handler could be managed via code generation. In this manner, the physical data access is separated from pure business logic.

[4225] **FIG. 161** illustrates that the database retrieval mechanism is separated from the business object **16100** by encapsulating the logic within a data handler **16102**.

[4226] Benefits

[4227] Loose Coupling of Data Access. The business object is independent of the database access logic and the backend database. As a result, the method by which the domain object accesses the persistent store can be changed without impacting existing source code.

[4228] Distribution. Data handlers can be physically partitioned into a separate component from the business logic. For example, the data handler could be on a data server component near the DB, while the business logic is in an application component.

[4229] Multiple Data Handlers. Different strategies can be implemented based upon specific requirements. For example, on the client we can use serialization to communicate with the server; whereas the server can use standard DB access to communicate with DB.

[4230] Support for Testing. Similarly, during testing, hard-coded data handlers can be created to return dummy data. These can then be replaced at run-time or later in testing without impacting the code.

[4231] The following information focuses on the implementation of the Data Handler pattern (TiMapper) and the separation of the business domain objects from the data retrieval mechanism used on the project.

[4232] **FIG. 162** illustrates the TiPersistenceStream **16200** and TiMapper **16202**.

[4233] TiPersistenceStream and TiMapper

[4234] Within the Rapid Batch Persist Service, objects save and load themselves by writing to or reading from a Persistence Stream. This is undertaken via the base class (TiPersist) with specialized streaming code created via the Creation Code Generator. As a result domain objects are only "aware" of how to stream themselves, and not how the data storage mechanism works.

[4235] The first attribute an object writes to the stream is its CLASS_ID. The stream then expects the other attributes to be put to the stream in the order defined by the mapper class (the *data handler*). This relationship between data handler and domain object is controlled via a row class,

which defines the attributes in the same order as they appear on the DB (this class is also created via the Code Generator).

[4236] When the end of the stream is reached, the mapper class iterates over the list of attributes within the row and saves them via embedded Pro*C. When loading the reverse happens, in that, the mapper reads the information from Oracle and then populates a row based upon the CLASS_ID pulled from the database.

[4237] TiMapper

[4238] A Mapper for a class contains the column(s) and table that the class will be written to, the type of the data and the order that they will be read from/written to the stream. It also reads and writes rows of data from the database. It generates a where clause from the primary key information (PID). A database runtime context is obtained from the Transaction Service (using the current implicit transaction context). It also contains the code to query for sequence ids, for classes that use optimistic locking.

[4239] The mapper contains the data retrieval mechanism that interacts with the Oracle database instance. As a result, if a different technology is used to interact with Oracle (e.g. stored procedures, embedded Pro*C, Method/3 Pro*C) only the mapper class needs to change.

[4240] TiRow

[4241] A row contains the data to be written to a database row from a stream or to be written to a stream from a database row. It knows the column names on the table and knows the order in which they are read from streams.

[4242] TiMapperManager

[4243] The mapper manager is responsible for creating mappers for a given CLASS_ID. Each type of mapper registers with the mapper factory when the shared object is loaded into memory (using the dynamic registration factory pattern). The mapper factory is a singleton read only object.

[4244] The Factory pattern can be used to create the appropriate Data Handler for a specific business object. This pattern enables the data access method to be changed at runtime (e.g. batch mode, online mode or Request Batcher).

[4245] The Stream-Based Communication pattern can be used to stream the business object's data to the handler. The stream can then be either forwarded to a Request Batcher or can parsed and sent to database.

[4246] Individual Persistence

[4247] **FIG. 163** illustrates a flowchart for a method **16300** for organizing data access among a plurality of business entities. Data about a user is retrieved and packaged into a cross-functional data object in operation **16302** and **16304**. A request for data is retrieved from one of a plurality of business objects in operation **16306**. In operation **16308**, the business object are directed to the data object such that the business object retrieves the entire data object. The business object also selects the data from the data object.

[4248] Both locking and integrity may use a uniform mechanism. The business object may retrieve account, customer, and bill-related data from the data object. Also, the business objects may be able to update themselves independently of each other.

[4249] Optionally, new business objects may take advantage of existing data access routines. Also, each business object may use a uniform access architecture.

[4250] Create a data access architecture that supports reusable, independent business objects in the context of atomic, functionally-specific transactions.

[4251] A business unit of work, or business transaction, typically acts on multiple business entities. But for each individual entity, the transaction might only display and update a few data attributes.

[4252] For example, accepting bill payment over the phone might use the account number, customer name, amount due, date due, and credit card number. The transaction could therefore avoid accessing many attributes of the account, customer, or monthly bill entities. This might naturally lead to a data architecture which only fetches required attributes, based on the transaction's needs.

[4253] Indeed, a traditional client/server program retrieves data in a piecemeal fashion. In this case, the example program would typically allocate a single record to fetch and store the required data items. Then, an "accept bill payment" data access module would fill this structure. This couples data access to processing function.

[4254] **FIG. 164** illustrates retrieving data piecemeal.

[4255] This traditional style of data access may seem flexible. After all, developers can grab whatever data they need for a particular business transaction.

[4256] But such access is very unstructured. Different pieces of a cohesive account entity, for example, scatter across multiple windows. Some windows will use the account's effective date; others will not.

[4257] This also introduces redundancy. Retrieving the date requires determining the correct database, table, column, and type declaration. Yet another developer who needs this date, for a different data set, duplicates the effort. This does not encapsulate changes, thereby raising costs for testing, maintenance, and extension.

[4258] Moreover, each transaction must hand-craft its own retrieval procedure. Creating the thousandth new business transaction would require creating a thousandth new access module. Yet all data items would already have been retrieved by other modules. This style of data access is not reusable.

[4259] Finally, business entities are typically less likely to change than the transactions, or processes, which act on these entities. For example, an enterprise might re-engineer its billing function. Regardless of the resulting process, the account, customer, and monthly bill objects would likely remain. This suggests that transaction-based data access is brittle.

[4260] Therefore, data access should be organized around business entities, rather than transactions. Individual Persistence packages data into cross-functional objects, rather than transaction-specific data structures. Each individual business object, instead of the window or application, manages the retrieval of its data items.

[4261] A business object provides public methods for accessing, computing, displaying, and setting that data. Developers can therefore no longer plunder the persistent

store, selecting data items at will. Instead, they must access their data through encapsulated, self-requesting business objects.

[4262] With this architecture, the example billing function retrieves account, customer, monthly bill, and bill payment objects.

[4263] FIG. 165 illustrates the manner in which the present invention retrieves whole objects 16500.

[4264] For reuse, business objects should be able to request and update themselves independently of each other. Separating the data access for customer and account objects supports reusing them in isolation. Objects should therefore avoid explicitly requesting other linked objects, unless a formal containment relationship exists.

[4265] Finally, separation of concern allows business objects to remain blissfully unaware of the transactions which use them. A business object will not know which data items or services it may need to provide to its clients. Thus, the object must bring back all its data.

[4266] Benefits

[4267] Reuse. New transactions can take advantage of existing data access routines. For example, introducing a new business transaction, like perform credit check, would use existing customer and account objects. Yet, these domain model objects would already know how to update themselves. Therefore, the new application would build no new data access code.

[4268] Maintainability and extensibility. This approach supports "fix in one place." Any changes to particular data elements only need to be changed, tested, and recompiled in one access module, that of the owning business object.

[4269] Uniformity. Both optimistic locking and referential integrity (RI) are typically defined at the business object level. For example, separate account and customer objects typically have their own locking attributes. In addition, an RI rule usually relates one entity to another, such as "all accounts must have a customer." Organizing data access around business entities simplifies locking and integrity. Both can use a uniform mechanism, implying that the architecture can hide technical details. This avoids the hard-coding typical of the transaction-based approach.

[4270] Flexibility. Whole object retrieval is extensible. It allows a transaction to ask an object for any data. This supports maintenance and extension. A developer can easily change the particular data items a transaction uses. But whole retrieval still guarantees that those items will already have been retrieved. For example, a future release of the accept bill payment window could also display the social security number. Yet the data access routines would need no modification.

[4271] Each typical business class will support the standard CRUD flag capabilities, of:

[4272] Create

[4273] Retrieve

[4274] Update

[4275] Delete

[4276] This access architecture is uniform across business entities. The architecture can therefore standardize much of the processing. For instance, the architecture can handle, across business objects: dirty checks, CRUD flag management, optimistic locking, referential integrity, security checking, commit scope, request formatting, object streaming/unstreaming, message compression, and error handling. Moreover, business entities should support these capabilities through a consistent, polymorphic interface.

[4277] For example, all business objects could respond to the saveData message, to persist any changes. saveData could first check, privately, if the object was even modified. Then, using private CRUD flags, it could determine whether a save translates into an insert, update, or delete. Finally, saveData could stream out the business object's attributes, based on a defined layout. Then, a transaction persists its business objects by simply iterating over the collection, sending each member saveData.

[4278] The architecture should also encapsulate the data access protocols or products. For example, whether the business objects use a relational or object DBMS should be transparent to calling programs. This minimizes the impact of changes to the storage technology.

[4279] Individual Persistence naturally leads to multiple, small-grained request messages per transaction. Request Batcher solves performance problems with multiple network messages.

[4280] Data Handler encapsulates data access code from business objects. This protects business logic from changes in data access protocols and products.

[4281] Request Sorter handles referential integrity and deadlock avoidance in a uniform manner.

[4282] Multi-Object Fetch

[4283] FIG. 166 illustrates a flowchart for a method 16600 for retrieving multiple business objects across a network in one access operation. In operation 16602, a business object and a plurality of remaining objects are provided on a persistent store. Upon receiving a request for the business object in operation 16604, it is established which of the remaining objects are related to the business object in operation 16606. The related objects and the business object are retrieved from the persistent store in one operation and it is determined how the retrieved related objects relate to the business object and each other (see operations 16608 and 16610. A graph of relationships of the business and related objects is instantiated in memory in operation 16612.

[4284] An object navigation pattern of accessing the business object and then accessing relationships with the related objects may be used to retrieve the related objects. The relationships between the business and related objects for instantiating the graph of relationships may also be determined from a source object, a set of target objects, and the name of the relationship. Additionally, the establishment of which of the remaining objects are related to the business object and the determination of how the retrieved related objects relate to the business object and each other may be

pre-processed before retrieving the selected related objects and the business object from the persistent store.

[4285] As an option, a portion of the objects may also be retrieved from a cache. Also, as another option, a batch request may be sent to the persistent store for retrieving the remainder of the objects.

[4286] It is not unusual to retrieve multiple business objects within a unit of work. How can the persistent objects involved in a unit of work be efficiently retrieved?

[4287] A given business object maintains associations to several other business objects. A given unit of work needs to access a subset of the defined associations. In order to perform the unit of work, the business object and the required subset of associated objects must be retrieved from persistent store.

[4288] In the course of performing a unit of work, a set of related objects needs to be accessed. Typically, one starts from a "root" object and "navigates" relationships to access related objects. This process can be repeated on the related objects.

[4289] An entire graph of related objects can be accessed in this manner. A natural way to retrieve these objects is through lazy instantiation, i.e., objects are retrieved from persistent store as each relationship is traversed. This retrieval pattern typically requires multiple database/network accesses and can have serious performance implications, especially over a WAN.

[4290] Therefore, a mechanism is needed to perform the retrieval from persistent store in one access operation. This mechanism includes:

[4291] Support for a declarative multi-object fetch statement which defines what is going to be fetched. This multi-object fetch statement does two things. It declares what is going to be fetched. It also declares how the objects that are being fetched relate to each other.

[4292] Retrieval of the persistent data corresponding to the multi-object fetch statement.

[4293] Instantiation of the graph of related objects in memory.

[4294] Benefits

[4295] Performance. Performance is increased by making a single trip across the network and a single database access to fetch several instances of objects that participate in a transaction. The savings can be especially noticeable over a high-latency WAN.

[4296] Continuity. Within the application code, the object navigation pattern of accessing an object and then accessing relationships from that object can still be used to access objects.

[4297] Complexity. Requires more elaborate architecture than lazy instantiation.

[4298] **FIG. 167** illustrates an example of an implementation of the Multi-Fetch Object **16700**.

[4299] **FIG. 168** illustrates the Fetching of a Household object **16800** along with the other related objects using the multi object fetch results.

[4300] **FIG. 169** is an interaction diagram showing when the multi object fetch is not used.

[4301] Note that if there is a large household, and each hobbyist in the household has lots of hobbies and interests, several trips to the server will be made to fulfill this query. There needs to be a multi-object fetch specification that keeps enough detail to know what needs to be fetched and how the fetched object will relate to each other. Here is a structure that will capture that information.

```
struct MultiObjectFetchSpec
{
  ArrayOf(int) classId;
  char **associationNames;
};
```

[4302] This is a declaration of the multi object fetch using the example above with the Household, Hobbyist, Hobby and Interst.

```
const HOUSEHOLD_CLASSID = 1;
const HOBBYIST_CLASSID = 2;
const HOBBY_CLASSID = 3;
const INTEREST_CLASSID = 4;
const NUMBER_OF_HOUSEHOLD_RELATIONSHIPS = 1;
const NUMBER_OF_INDIVIDUAL_RELATIONSHIPS = 2;
static char *EmptyRelationship[1] = { 0 };
static char *
HouseholdRelationships[NUMBER_OF_HOUSEHOLD_RELA-
TIONSHIPS + 1] = {"Hobbyist", 0};
static char *
HobbyistRelationships[NUMBER_OF_INDIVIDUAL_RELA-
TIONSHIPS + 1] = {"Hobbies", "Interests", 0};
static MultiObjectFetchSpec HobbyInterestMcFSpec[5] =
{
        { HOUSEHOLD_CLASSID, HouseholdRelationships },
        { HOBBYIST_CLASSID, HobbyistRelationships },
        { HOBBY_CLASSID, EmptyRelationship },
        { INTEREST_CLASSID, EmptyRelationship },
        { 0, 0 }
};
```

[4303] There is a class MultiObjectFetch that performs the fetch and associates all of the related objects that have been fetched so that when these related objects are accessed there is no further access to the database. The MultiObjectFetch class uses the MultiObjectFetchSpec to determine how the objects fetched relate to each other. This implementation assumes that the persistent framework being used can fill in the relationship given a source object, a set of target objects and the name of the relationship.

```
class MultiObjectFetch
{
    public:
    MultiObjectFetch *MultiObjectFetch(
        MultiObjectFetchSpec *mcfSpec);
    PersistentObject *fetch( );
    RWOrdered *fetchRows( );
};
```

[4304] There is an assumption that the Household, Hobbyist, Hobby and Interest business objects inherits from a

common base class, PersistentObject. If the restriction on the household is to bring back one Household, fetch( ) would used. If the restriction on the Household will bring more than one Household, fetchRows( ) would be used. The fetch and the fetchRows brings back the Household objects(s) and the related Hobbyists, Hobbies and Interests.

[4305] Static Approach (Using Stored Procedures)

[4306] A stored procedure would be written that would retrieve the Household object(s), Hobbyist objects related to the Household objects, Hobby objects related to the Hobbyist objects and the Interest objects also related to the Hobbyist objects. It is important that the stored procedure fetch the objects in this order since the multi object fetch spec declared that this is the expected order. These fetched objects would then be passed to the MultiObjectFetch object which would fill in all the relationships of the returned object using the fetch specification.

[4307] Dynamic Approach (Dependent Requests/Pending Actions)

[4308] The multi-object fetch can be pre-processed before sending the request to the database. Any objects that can be fetched from the cache will be fetched from the cache. The remaining requests that cannot be satisfied from the cache will then be sent as a batch request to the database. This requires complex processing to determine if the cache can be used. This dynamic processing assumes that dynamic SQL will be used since it is not known at design time what objects will be found in the cache and what objects still need to be fetched from the database.

[4309] Dependent Request—used in dynamic approach

[4310] Request Batcher—used in dynamic approach

[4311] Batching Update—similar to batching of fetches but used to batch updates

[4312] Relationship Stereotype—The setAssociation method is called to fill in the relationships

[4313] Object Identifiers as Objects

[4314] **FIG. 170** illustrates a flowchart for a method **17000** for implementing an association of business objects without retrieving the business objects from a database on which the business objects are stored. In operations **17002** and **17004**, a business object is provided and an instance of an associated object is stored on a database. An association of the business object with the instance of the associated object is determined in operation **17006**. In operation **17008**, an object identifier is generated containing information including the determination association which is necessary to retrieve the instance of the associated object from the database. The object identifier is loaded when the business object starts in operation **17010**. A location of the instance of the associated object on the database is determined in operation **17012** from the object identifier and the instance of the associated object is retrieved from the database in operation **17014**.

[4315] The object identifier may be used to provide a unique identity that is required for implementing caching and identity management. Also, the object identifier may include a unique row identifier generated by the database, an identifier generated by a utility, and/or a unique string generated from one or more attributes. As an option, different types of business objects may be provided. In such a case, a different class of object identifier may be generated for each type of business object. As another option, the determination of a location of the instance and the retrieval of the instance of the associated object may also include the taking the object identifier as an argument and returning the instance of the associated object.

[4316] Most useful business objects have a relationship, or association, with other business objects. How should this association be implemented without having to read the associated object's state from the database?

[4317] Most useful business objects have a relationship, or association, with another business object instance. Traditionally, this relationship is expressed as a pointer or reference to the object. However, pointers (and references) are memory constructs valid only so long as the object state exists in memory. When storing the object to a persistent storage medium (such as a relational database) these associations need to be expressed in some other way. Likewise, when the object is restored from persistent storage, the associations need to be reinitialized since it is unlikely that the associated object will reside in the same memory location. If the associated object is stored in the persistent medium, this usually involves restoring it as well. This can become a long and expensive process if the association graph corresponding to the restored object is large or complex. It is particularly undesirable if the associations are not even traversed by the application.

[4318] Therefore, implement the associations using object identifiers that contain the necessary information to retrieve the object if it is needed. These objects can then be loaded when the object is restored, eliminating the need to restore the entire associated object. In addition, since these object identifiers uniquely identify an object instance, they can be used/passed in place of memory pointers. When the object is needed, simply restore the instance using the object identifier.

[4319] Benefits

[4320] Performance. Objects are retrieved only when they are needed.

[4321] Caching and identity management. Object Identifiers can be used to provide the unique id needed to implement caching and identity management.

[4322] The object identifier (or OID) must contain enough information to uniquely identify the instance. This identifier could be a unique row id generated by a database, a UUID generated by a utility or a unique string generated from one or more attributes. It is generally desirable to have a different class of OID for each type of object, thereby creating a more type-safe environment. It should also be noted that OID's should have value semantics.

[4323] In addition, a mechanism must be available to retrieve objects given their OID. This can be as simple as a static (or class) method such as getById that takes the OID as an argument and returns the instance of the object. A more sophisticated approach would be to implement this and other persistence related methods in a generalized utility class. Below is a simple example that illustrates the relationship between two classes using object identifiers. Please note that

this example is an extreme simplification. A useful implementation of this pattern would exist as part of a persistence framework that would include many additional methods and abstractions.

```
class FooId
public:
    // accessors, constructors and destructors
private
    long _id;
};
class Foo
{
public:
    // accessors, constructors and destructors
    static Foo* getById(FooId& id);
};
class Bar
{
public:
    Foo* getFoo( )
    {
        return Foo::getById(FooId);
            // The caller now owns the instance of Foo. Use of an
            // auto_ptr here is highly recommended.
    }
private:
    FooId _fooId;
};
```

[4324] Collaborations

[4325] Identity Manager—Uses Object Identifiers as unique key's for storing persistent state objects.

[4326] Persistent State Separate from Persistent Object—Uses Object Identifiers embedded in persistent state objects to eliminate coupling.

[4327] Object Identity Cache

[4328] **FIG. 171** illustrates a flowchart for a method **17100** for mapping of retrieved data into objects. An object is retrieved from a data store and cached in operations **17102** and **17104**. A unique object identifier is assigned to the object in operation **17106**. The object identifier is mapped to a representation of the object in a dictionary in operation **17108**. When a request for a reference to the object is received in operation **17110**, the object identifier of the object is retrieved from the dictionary in operation **17112**. The requested reference is associated with the representation of the object stored in the dictionary in operation **17114**.

[4329] In one embodiment, a data store may be accessed if the object identifier is not found in the dictionary to retrieve the object so that the process may be repeated with the retrieved object. In another embodiment, a query may be performed to retrieve multiple objects. A check may be performed to verify that each object is already cached so that objects not already cached may be cached.

[4330] Also, if an object in the cache has changed since read, an error may be raised if the object retrieved has changed since read and the object retrieved may be ignored if the object retrieved has not changed since read. If an object in the cache has not changed since read, the object in the cache may also be replaced with the object retrieved if the object retrieved has changed since read and the object retrieved may be ignored if the object retrieved has not changed since read. Further, if a query is guaranteed to return at most a single object, the cache may be used prior to going to the data store by sequentially applying the function to each object in the cache and implementing a predicate function which answers whether or not a given object satisfies the query.

[4331] Within a client context (e.g., a logical unit of work), the same object may be referenced more than once. How can object identity be preserved and redundant accesses to persistent store be avoided?

[4332] (Although this pattern is not specific to relational databases, we will, for the sake of concreteness and clarity, describe the pattern in terms of an object-to-relational mapping.)

[4333] Objects can be stored in and retrieved from a relational database. A retrieval strategy that simply translates relational data into objects in memory will almost certainly result in the instantiation of multiple copies of the same object. Furthermore, such a strategy is inefficient as the same data may be repeatedly and unnecessarily read from the database. This violates object identity and contributes to performance problems.

[4334] Suppose the class Account has an association with the class Customer. Suppose the instance of Account ABC is associated with the instance of Customer 123. The following demonstrates multiple requests to Customer 123.

[4335] Example

[4336] customer123=getCustomer(123)

[4337] accountABC=getAccount(ABC)

[4338] secondReferenceToCustomer123 accountABC.getCustomer( )

[4339] Note that customer123 and secondReferenceToCustomer123 are the same customer. In this scenario, the desired behavior is to have the data store accessed once for customer123. Also there should only be one instance of customer123 in memory. Customer123 and secondReferenceToCustomer123 should reference this instance of the customer123.

[4340] Therefore, the mapping of retrieved data into objects should be mediated by an Identity Cache. **FIG. 172** illustrates an Object Identity Cache.

[4341] Logically, an Identity Cache is a dictionary which, for each cached object, maps a unique object identifier to a representation of the object. Each object must be assigned an object identifier (OID) which uniquely identifies the object over the life of the system.

[4342] The mediation mechanism works as follows: When a reference to an object is requested, the identity cache is consulted. If the object's OID is found in the cache, the requested reference is associated with the representation of the object stored in the cache. If the OID is not found in the cache, the data store is accessed. The object representation that is retrieved from the data store is inserted into the cache and the requested reference is associated with it.

[4343] Benefits

[4344] Performance. Performance is improved for frequently accessed objects since they are only fetched from the database once.

[4345] Identity Preserved. Object identity is preserved since objects are cached based on the objects OID.

[4346] A dictionary can be used to implement the Object Identity Cache. The following points should be considering when implementing an Object Identity Cache.

[4347] A query could be performed that returns multiple objects. Each object that is retrieved must be checked to see if it is already in the cache. If it is not in the cache it must be inserted. The following shows what should be done when an object is retrieved that is already in the cache to get is correctly inserted into the cache.

	Object retrieved has changed since read	Object retrieved has not changed since read
Object in cache has changed since read	Raise an error. At commit transaction there will be an optimistic lock failure so it is better to raise it now.	Ignore the retrieved object, the object in cache is newer and the changes should not be lost.
Object in cache has not changed since read	Replace the object in cache with the object retrieved since the retrieved object is newer.	Ignore the retrieved object. It is already in the cache.

[4348] If the object is not in the cache when the object is retrieved, a simple insertion into the cache can be done.

[4349] If a query is guaranteed to return at most a single object, the cache may be used prior to going to the database by:

[4350] implementing (for a given class) a predicate function which answers whether or not a given object satisfies the query

[4351] sequentially applying the function to each object in the cache.

[4352] A multiple row query must go to the database unless there is a mechanism to indicate that the class is completely cached. This is applicable to static reference data.

[4353] Life Time of cached objects can affect Cache size. If life time of the cache is associated with a transaction there will be no problem. If the life time of the cache is associated with a longer lived entity such as a thread or a process, removal of objects from the cache must be actively managed. Two commonly used strategies are reference counting and LRU purging.

[4354] Collaborates With

[4355] Object Relational Query pattern describes a mechanism for storing objects in a relation database.

[4356] Object Identity

[4357] Persistent State Separate from Persistent Object

[4358] Used by

[4359] Context Management

[4360] Each Context (e.g., transaction, thread, etc.) owns an Identity cache which holds all of the objects in that context.

[4361] Persistent State Separate from Persistant Object

[4362] **FIG. 173** illustrates a flowchart for a method **17300** for separating logic and data access concerns during development of a persistent object for insulating development of business logic from development of data access routine. A persistent object being developed is accessed and a state of the persistent object is detached into a separate state class in operations **17302** and **17304**. The state class serves as a contract between a logic development team and a data access development team. Logic development is limited by the logic development team to developing business logic in operation **17306**. In operation **17308**, data access development is restricted by the data access development team to providing data creation, retrieval, updating, and deletion capabilities.

[4363] The business logic team may develop persistent objects that manipulate the state of the persistent object in accordance with business requirements. In one embodiment, the state may be implemented as a structure of values of basic data types. In another embodiment, the state class may contain member variables of lower data types including basic data types, extended basic data types with value semantics, other state classes, and/or vectors of the basic data types, the extended basic data types with value semantics and other state classes.

[4364] Optionally, the state class may support data structures of arbitrary shapes. Supporting classes may manipulate the state in a polymorphic fashion. As another option, the state may be further implemented as a class that contains key-value attribute pairs. The state class may also contain a keyed data structure containing attribute names and attribute values. Additionally, the state can also be asked to write data to a stream.

[4365] When designing and implementing persistent objects, how do we effectively insulate the development of business logic from the development of database access routine?

[4366] Given the use of a relational database as the persistent store, the scope of a persistence architecture can range across the following levels of transparency and automation:

[4367] Heavyweight, fully-automated persistence architecture. Including the mapping of the object model to the database schema and generation of all the database access code

[4368] Variants of the above scheme allowing the customization of database access code

[4369] Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se.

[4370] Minimal persistence approach in which each business object is directly responsible for database access

[4371] From a persistence perspective, no matter which of these approaches is used, development of the system and architecture presents two distinct challenges to the development team. The first challenge is to accurately represent the business logic as a collection of business objects that include interfaces for performing the correct set of functionality. The second challenge is to be able to create, retrieve, update and delete (CRUD) records that represent the state of these business objects from the database in an efficient fashion.

[4372] Data access and business logic are significantly different tasks both in their goals and development approaches. Consequently, except when a fully automated persistence architecture is used, it is often the case that two separate teams are responsible for the development of business logic and data access. If both teams work directly with the business objects, serious contention may result. Problems encountered in practice include:

[4373] Changes to business logic that impacts the development (e.g., requires recompilation) of database access code even when there is no change to the attributes of business objects. (Note: recompilation can be a problem even if a fully automated persistence framework is used.)

[4374] Changes to the database schema can impact the development of business objects.

[4375] The data access team may unduly influence the design of the business objects, leading to a data-centric model and design.

[4376] The two teams' development schedules need to be in sync; slippage on one team can adversely impact the other team's progress.

[4377] Therefore, detach the state of the business object into a separate state class that can be agreed upon and completed prior to the start of significant development by either the data access team or the business object team. This class should be nothing more than the raw data (preferably basic data types) used to represent the state of the object. The business object contains all business logic and a reference/pointer to an instance of the respective state class. This allows the development of business logic and data access to proceed in parallel with the state class serving as a contract between the two teams.

[4378] Using this approach, it is important to limit data access focus to providing CRUD operations (i.e., no business logic provided by stored procedures). The purpose of the data access portion of the application is to provide essential access to the data used by the business objects, and deliver this data in the most efficient way possible. Likewise, the purpose of the business object team is to develop business objects that manipulate the state of the object in accordance with the business requirements. Maintaining this separation insures there is no overlap between the development teams.

[4379] Benefits

[4380] Development Time. Data access and business logic can be developed in parallel reducing overall development time.

[4381] Separation of concern. Data access remains separate from business logic, improving the understandability of the design.

[4382] Testability. Business objects can be more easily developed and tested based on data access stubs, thereby relieving the business object development teams from dependencies on the data access classes and the database libraries.

[4383] Caching and identity management. Separating the persistent state from the persistent object can be leveraged to aid in managing multiple class instances that represent the same entity (see the Object Identity Cache pattern).

[4384] Object distribution. Separating the persistent state from the persistent object can aid in passing state in a distributed system. In cases where in is necessary to pass an object as an argument to a distributed method, it is more desirable from a performance perspective to pass the object's state as opposed to a remote reference. A new instance can then be created from the state, manipulated locally and then returned to the caller.

[4385] The persistent state can be implemented in a variety of ways depending on the requirements of the system. They can roughly be described by the following.

[4386] Implement the State as a Structure of Values of Basic Data Types

[4387] Each business object class has an associated struct. This is the most straightforward approach, although also the least flexible. With this approach, the business logic may need to manipulate lower level data types contained in the state object.

[4388] Supporting classes which need to manipulate the state object (in order to retrieve data from the database or pass the value between processes) need to be knowledgeable about the struct data type to manipulate it. In addition, copying the struct may be non-trivial if it contains dynamically allocated memory.

```
struct State_Data
{
    long        id;
    char        code[8];
    string value;
};
```

[4389] Implement the State as a Class Containing Member Variables of Basic Data Types

[4390] 1. Implementation using a "developer-friendly" state class: A state class can contain basic data types (except char*), extended basic data types with value semantics (e.g., currency, String), other state classes, and vectors (not arrays) of all of the above. This does not, however, imply that the class is a flexible (dictionary-like) data structure. These state classes could/should all inherit from a common abstract base class which defines (but does not implement, at least in C++) a serialization protocol (Java is a different story than C++ because everything is serializable almost by default).

```
class StateClass
public:
    virtual void read(Stream inStream) = 0;
    virtual void write(Stream outStream) = 0;
};
class StateData : public StateClass
{
public:
    virtual void read(Stream inStream);
        // implementation reads state data from stream.
    virtual void write(Stream outStream);
        // implementation writes state data to stream.
Private:
    long        id;
    char        model[8];
    string value;
};
```

[4391] 2. Implementation using an enhanced kind of the class described in (A) but which also happens to be a flexible data structure (in the sense that the same class can, similar to a dictionary, support data structures of arbitrary shapes).

[4392] This approach provides more flexibility since some common behavior can be abstracted into a base class. In addition, supporting classes which need to manipulate the state object (in order to retrieve data from the database or pass the value between processes) can do so in a polymorphic fashion. Also, copy constructors and destructors can be used to handle dynamically allocated memory.

[4393] Implement the State as a Class that Contains Key-Value Attribute Pairs

[4394] This is an alternative approach to the one listed above. Using this technique the state class would contain a keyed data structure (e.g. a dictionary) containing keys (the attribute name) and values (the attribute value). In cases where you want to copy the state or pass it to another process, the supporting code does not need to know the type of the state object it is working with. State objects can simply be asked to read or write their data to and from a stream or string. While this offers a more dynamic solution, it should be noted that with this solution additional logic would need to be included in the persistent object to insure the validity of the associated state class.

[4395] In all of these approaches, another important concept is the implementation of associations between objects. In general, the best approach is to store these as Soft References to the other objects as opposed to actual pointers. This illuminates the need to load a large graph of objects when only one is needed, as well as easing the question of whether to implement a deep or shallow copy.

[4396] Identity Manager: Manages a collection of persistent state objects for a given class.

[4397] Context Manager: Used in conjunction with Identity Manager to maintain separate collections of persistent state objects for separate application contexts.

[4398] Lazy Instantiation: Restores persistent state object for a given object instance on-demand.

[4399] Object Identity Cache: Caches persistent state objects referenced by persistent objects.

[4400] Piecemeal Retrieval

[4401] **FIG. 174** illustrates a flowchart for a method **17400** for providing a warning upon retrieval of objects that are incomplete. An object is provided with at least one missing attribute in operation **17402**. Upon receipt of a request from an application for the object access to the attributes of the object is allowed by the application in operations **17404** and **17406**. A warning is provided upon an attempt to access the attribute of the object that is missing in operation **17408**.

[4402] The warning may include information on how to handle the missing attribute. An implicit transaction may also be called by the object upon the attempt to access the attribute of the object that is missing. In addition, an explicit transaction may be called by the object upon the attempt to access the attribute of the object that is missing. Also, the transaction may also include finding the attribute that is missing.

[4403] When legacy transactions don't allow object or entity based retrieval, how do we retrieve useful objects?

[4404] Object and component based projects designed and built from the ground up will likely have a well thought out component model and architecture where GUI widgets are linked or bound to domain objects. Data access (and retrieval) for these objects is organized around the business entity, rather than a transaction, and so data is packaged into cross-functional objects, rather than transaction-specific data structures. Each business object manages the retrieval of its data items.

[4405] These domain objects provide public methods for accessing, comparing, displaying, and mutating data. Therefore, developers will only access data through these encapsulated, self-requesting domain objects. (See Individual Persistence).

[4406] For example a billing application that accepts bill payment over the phone might use the account number, customer name, amount due, date due, and credit card number. With an Entity-Based Data Access architecture, the account **17500**, customer **17502**, monthly bill **17504**, and bill payment objects **17506** will all be retrieved. **FIG. 175** illustrates an Entity-Based Data Access System.

[4407] This architecture is preferred if conditions allow, however legacy programs usually retrieve data through transaction or message based systems. These transactions often have two notable characteristics. One, a business unit of work or business transaction often acts on multiple business entities, and two, for each individual entity, the transaction might only retrieve and update a few data attributes.

[4408] Using the same bill payment example, a legacy system might utilize a 'accept bill payment' transaction. This transaction would require only a small portion of the attributes for the account, customer, or monthly bill entities and so the data would be retrieved piecemeal only as required by the transaction.

[4409] **FIG. 176** illustrates a Retrieving Data Piecemeal System.

[4410] In this case, the transaction would allocate a single record (an 'accountPaymentData' **17600** structure as shown

in the Figure) to fetch and store the required data items. This structure would then be used to populate the business entities.

[4411] As a result, domain objects are left incomplete and therefore unsuitable for use by all services. This forces the developers of services to know and understand the use of transactions to retrieve objects.

[4412] Therefore, when legacy circumstances dictate, retrieve data piecemeal, on a transaction basis as necessary rather than as complete business entities and develop mechanisms to handle access and updates to missing attributes of piecemeal objects.

[4413] By default, objects should return a warning when a missing attribute is accessed with no other means to retrieve it. This warning would simply let calling applications know that an attribute is missing or unavailable. The calling application would then have to determine how to handle these missing attributes.

[4414] A preferred approach to handling missing attributes would be to use multiple overlapping transactions. While, each transaction might only populate a part of the object, the transactions taken together assemble complete objects.

[4415] This use of overlapping transactions could either be implicit or explicit to the objects. An implicit transaction would be called by the object when a missing attribute is accessed. This method of lazy retrieval may be preferable because it is transparent to the calling application.

[4416] An explicit transaction would be called by a task independent of the object. Note however, that explicit transactions would either require that the task holds the object or that an object identity mechanism is used to find the existing object rather than create a new one.

[4417] In some cases, new transactions can be created for the explicit purpose of retrieving full or partial business entities. This approach requires a thorough knowledge of the legacy system.

[4418] In other cases, the legacy code can be opened and the transactions modified to bring back additional data. Care should used when doing so as legacy code is often fragile and poorly documented.

[4419] Benefits

[4420] Legacy Reuse. The overwhelming benefit of piecemeal retrieval is that it enables reuse of legacy systems. Clients generally have a substantial investment in their existing systems and they will be hard pressed to convert them to component based systems built on objects. This pattern allows new systems to reuse existing business logic through their transactions.

[4421] Performance and Impedance. Objects based on transactions typically bring back only that data which is needed. The transaction is typically mapped directly to change that the user is trying to make. This improves performance by reducing the number of network calls and only bringing back data that is needed.

[4422] Individual Persistence organizes data access around business entities rather than transactions.

[4423] Object Streaming handles the conversion of data from the structures received from transactions into business objects.

[4424] Transaction Service Patterns (1012)

[4425] Transactions and LUWS

[4426] A transaction is a set of business events that, coupled together, accomplish a particular business function, such as turning on gas service. Because the events are logically related, their data changes are logically related as well. Taken together, these data changes create a new, consistent state for the business model.

[4427] While a transaction is in process, the state of the business model may not be consistent, so it is necessary to manage the entire transaction from its point of origin to its point of completion. Whether the transaction is successful or not, the point of completion will always result in a steady, consistent state for the business model. For successful transactions, data changes will be committed and the business model will reflect all new business data associated with the transaction. For failed transactions, data changes will be rolled back and the business model will appear as it did prior to the start of the transaction.

[4428] To help manage the transaction from point of origin to point of completion, each transaction is organized through a single Logical Unit of Work (LUW). This LUW manages the business model and any of its subsequent data changes. While both users and internal exceptions can determine the success of a transaction, the LUW handles the commit and rollback operations.

[4429] FIG. 177 illustrates a Commit and Rollback routine 17700.

[4430] As transactions become more complex and require a greater scale of changes to the business model, the LUW trying to manage these changes becomes large and unwieldy. To simplify these transactions, the LUW is broken down into nested, more granular, logically related units of work called Secondary LUWs. Secondary LUWs are identical to LUWs except that their commit and rollback operations affect only the business model of their parent LUW and are not persisted to a data store. Consequently, a secondary LUW must manage its data changes differently than its parent.

[4431] FIG. 178 illustrates Nested Logical Units of Work.

[4432] One method for managing changes to the business model involves copying the model into the secondary LUW. Another often simpler approach is to store both old and new (or potential) values for all objects in the business model.

[4433] Transaction Patterns

[4434] In the process of managing its business model, a LUW will often have to send messages to all business objects within the LUW. Examples of such messages include saveDataChanges, retrieve, or isDirty. Rather than hardcoding a call to each object in the model, the pattern LUW Context suggests using a bag (or collection) to hold all objects referenced by the LUW. Then, a single message can be sent to the bag which will forward it to all objects within it.

[4435] Support for user multi-tasking can also present problems for LUWs. Through multi-tasking, multiple

LUWs will be running concurrently. Problems occur if the business models of these concurrent LUWs overlap and the transactions attempt to write to the same business object. A call center representative trying to solve two customer problems during the course of one call is one example of this scenario. The Separate Models pattern helps solve this issue by assigning each LUW independent copies of their portion of the business model. This keeps one transaction from interfering with another.

[4436] There are also several patterns that address problems related to sending transactions across the network. These patterns typically focus on minimizing network messaging.

[4437] When an LUW is called to commit, the transaction will assemble the necessary objects from the business model to send their data changes to the information services layer. This group of objects will include all new and dirty objects as well as any objects marked for deletion. For each business object, the transaction will likely have a corresponding request. If each of these requests were then sent to information services independently, a large number of network messages would result. To solve this problem, the Request Batcher pattern batches all requests associated with a transaction together into one network message. On the other end of the network, Information services would unbatch the transaction requests and persist the data changes.

[4438] Another problem that may arise when multiple requests are sent for a given transaction is deadlock. Deadlock occurs when two requests are trying to lock the same pair of objects. Each request locks one object and waits to commit until it can lock the other. Therefore, each request will wait for the other to complete while neither is able to do so. The Request Sorter pattern works with Request Batcher to handle this problem by sorting requests as they are being unbatched by Information Services. A request is not allowed to proceed until any dependent requests are completed.

[4439] During retrieval, one request may depend on the response data for another request. For example, a business transaction that tries to retrieve a customer when given a customer ID will probably also want to retrieve the customer's address. However, the transaction won't have the address ID until the customer is retrieve. Thus multiple network messages are required when one request is dependent on another. The Dependent Request pattern solves this problem by allowing a batched request to indicate that it depends on another request.

[4440] As these patterns demonstrate, there is a high degree of correlation between Transaction Services and Information Services. Many of these transaction patterns require an understanding of Information Services patterns. Two such examples are Individual Persistence and Piecemeal Retrieval. It is recommended that these patterns be read and understood prior to using Request Batcher, Request Sorter and Dependent Request.

[4441] Dependent Request

[4442] FIG. **179** illustrates a flowchart for a method **17900** for allowing a batched request to indicate that it depends on the response to another request. A group of business objects necessary for a transaction are provided in operation **17902**. Logically-related requests received from the business objects are batched into a single network message in operation **17904**. One of the requests is a parent request. Received from the parent request in operation **17906** is a register that at least one of the requests is dependent upon the response data. The network message is sent across a network and the requests are unbundled from the network message in operations **17908** and **17910**. Upon receipt of a response to the parent request in operation **17912**, data is directed from the response to the parent request to the dependent request in operation **17914**. Received from the response to the parent request is a response to the dependent request based on the received data in operation **17916**.

[4443] The dependent request may not have a primary key. In such situation, the response to the parent request may include the primary key for allowing the dependent request to be responded to. As another option, the dependent request may also include a pointer indicating that the dependent request is dependent on the parent request. In this situation, the pointer may be passed to the parent request during the step of batching the requests into the network message.

[4444] Additionally, the pointer may be a configurable field of the dependent request. The requests may also be issued independently of each other. In event another aspect of the present invention, the dependent request may wait for the parent response at the server for minimizing network traffic.

[4445] During retrieval, one request may depend on the response data for another request. Nevertheless, ensure that a single network message contains both requests, while using Request Batcher.

[4446] A business transaction typically acts on multiple business entities. Consider an account maintenance window, which edits information from account, customer, credit profile, and home and work addresses. Given a unique account identifier, the business transaction can retrieve all five objects.

[4447] Once the account retrieves all of its data, it will know its unique customer identifier. The customer can then retrieve its own data, which includes the identifiers for the credit profile and both addresses. Finally, the profile and addresses can retrieve their data. In this case, profile and address retrieval depends on customer data; customer retrieval in turn depends on account data.

[4448] However, Individual Persistence requires that each object have its own, independent request module. That is, customers do not always need accounts to be retrieved. After a customer's unique identifier has been filled in—regardless of by whom—it retrieves its data independently.

[4449] In theory, this independence is not an issue. The account could first get its data. After the customer's identifier was filled, the customer could send its own request.

[4450] In practice, however, sending multiple messages, in series like this, degrades network performance. Request Batcher **18000** provides a solution which bundles up requests into one network message, thereby minimizing traffic. FIG. **180** illustrates a Batching Retrievals and Dependency.

[4451] This batching framework applies not only to update messages. For retrieval as well, one overall, batch request receives one batch response. Yet an individual batched

request may depend on the response data from another batched request. The serial nature of the two requests must be preserved, even while the requests actually go in the same batch, in parallel.

[4452] Therefore, additional mechanisms should allow a batched request to indicate that it depends on another request. If a business object does not have its primary key—or other attributes guaranteeing a unique match—it becomes a Dependent Request. Because a dependent cannot fetch itself, some other business object will inevitably have the necessary foreign key. The dependent request will therefore register its dependency on this other object.

[4453] This object, the parent request, can have multiple dependents. The aforementioned customer had credit profile and address dependencies. This implies the need for an arbitrarily large collection of dependent requests. A dependency collection can even include requests for multiple instances of the same class. This was the case with two address requests depending on the same customer.

[4454] FIG. 181 illustrates the Dynamically Setting Dependency.

[4455] Dependencies should not be hard-coded. Any business object can register as dependent on any other object which can provide the necessary data.

[4456] In this manner, dependencies can be set at run-time. This could happen while building the model, or as requests register with Request Batcher.

[4457] Benefits

[4458] Performance. This solution supports request batching for retrieving interdependent models.

[4459] Reuse. Because dependencies are not hard-coded, business objects can be reused independently of each other.

[4460] Loose Coupling. When a request dynamically registers its dependency, it need not know anything about its parent request. The dependent effectively says, "I don't know who you are, but I know that your response data contains my identifier."

[4461] Dependent Request would be irrelevant were it not for the "transaction impedance mismatch." This mismatch means that transactions no longer map one-to-one to access modules. Individual Persistence is the approach which dictates that each business entity have its own independent access module.

[4462] Request Batcher solves the performance problems of sending multiple, small-grained request messages over a network. But the resultant single message must support dependencies, with Dependent Request or a similar mechanism.

[4463] LUW Context

[4464] FIG. 182 illustrates a flowchart for a method 18200 for sending a single message to all objects in a logical unit of work. A group of business objects necessary for a transaction are provided and managed in a logical unit of work in operations 18202 and 18204. A receiver is created which communicates with the business objects in the logical unit of work in operation 18206. Upon receiving a message for the business objects in the logical unit of work in operation 18208, the message is directed to the receiver in operation 18210. The receiver also forwards the message to each of the business objects in the logical unit of work.

[4465] Several groups of business objects necessary for a transaction may also be provided with each group of business objects being managed in a separate logical unit of work. Also, a separate receiver may communicate with each group of objects. As another option, a request batcher in communication with the receiver may also be provided for batching requests from the business objects for delivery. In such an embodiment, the request batcher intercepts the requests from the business objects and holds the requests until told to deliver the requests by an activity associated with the logical unit of work.

[4466] Optionally, the receiver may hide technical details including details of persistence and garbage collection from business developers. As a further option, the business objects may be distributed across a network. Also, the receiver may distribute the message to each of the business objects across the network. Additionally, the logical unit of work may optionally be modeled as an object in software.

[4467] Applications often need to send technical messages, like saveDataChanges( ) or release( ), to all business objects in an LUW. Do this in a consistent manner and hide technical details from business developers.

[4468] Consider an Account Payment window, which displays information about an Account, Customer, Monthly Bill, and Payment. This window occasionally needs to send generic, technical messages to all business objects within its LUW. This messaging has nothing to do with the window's application-specific behavior. In fact, the other windows in the system need to send the same, generic messages to their LUW business objects. Although the business objects receiving these messages differ from window to window, the messages remain the same.

[4469] In addition, this messaging typically has an arbitrary order. All that matters is that all business objects eventually receive the same message.

[4470] For example, the window might use Individual Persistence. Then, when the user decides to save the window, all business objects receive a message like saveDataChanges( ). The resultant pseudo-code would look like:

```
AccountPaymentActiveInterfaceDataChanges( )
{
    // Propagate along the save message to all business
    // objects in my LUW.
    this.getAccount( ).saveDataChanges( );
    this.getCustomer( ).saveDataChanges( );
    this.getMonthlyBill( ).saveDataChanges( );
    this.getBillPayment( ).saveDataChanges( );
}
```

[4471] A retrieveData( ) message might also be required, if the object model pre-instantiates objects before retrieving them. Similarly, refresh( ) could be used: the business object, if dirty, replaces any changes with data originally from the data store.

[4472] Even without Individual Persistence, there are other common messages the window may want to send. For

example, distributed objects typically need to be told when their memory can be reclaimed. COM+ uses the well-known method releaseRef( ), whereas some implementations of CORBA use release( ). Regardless, this is a common message that would also need to be sent to the business objects, similar to the saveDataChanges( ) propagation above.

[4473] Dirty Flag provides another example. Here, the window accumulates the results of dirty checking, as follows:

```
AccountPaymentActivity::isDirty () {
  // Return true if any single business object in the LUW is dirty.
  return
    (this.getAccount().isDirty() or
    this.getCustomer().isDirty() or
    this.getMonthlyBill() isDirty() or
    this.getBillPayment().isDirty());
}
```

[4474] This hard-coded approach, although straightforward, is both tedious and error-prone. It is tedious because business developers shouldn't have to deal with technical issues like dirty checking or distributed garbage collection. They should focus instead on business-specific processing.

[4475] Moreover, this is error-prone because it can be difficult to detect if the developer makes a mistake. For example, a new requirement could make the window display address information. In addition to repainting the window, the developer would also need to modify their hard-coded methods. But the developer might forget to update the isDirty( ) or release( ) methods. Such errors can be difficult to locate. (Readers who have debugged memory leaks will certainly agree.)

[4476] Instead, an architecture mechanism should encapsulate the propagation of these technical messages. When a message needs to be forwarded generically, to all objects in an LUW, the architecture should handle it. Such a capability would free business developers from worrying about these technical details. **FIG. 183** illustrates a Hand-crafted Message Forwarding scheme.

[4477] Therefore, an architecture "bag" will represent the business objects in a particular LUW. This bag, or collection, will hold onto each business object. Then, when the bag receives a message like saveDataChanges( ) or release( ), it simply forwards the message to each member business object.

[4478] **FIG. 184** illustrates a Generic Message Forwarding feature.

[4479] Each LUW must have its own bag. This enforces the Isolation property (of ACID) for LUWs. That is, one LUW should not affect another LUW. For example, if the Account Payment and Account Services activities have separate LUWs, they will correspondingly have their own bags. Then, calling saveDataChanges( ) on the Account Payment activity will forward saveDataChanges( ) to only those business objects owned by Account Payment.

[4480] The bag also helps ensure the Atomicity property (also of ACID) for LUWs. It provides a single, atomic interface into the multiple business objects of the LUW. By design, it ensures that all business objects receive the same architecture messages.

[4481] Thus, the scope of a bag is an LUW. In addition, a bag provides contextual information for the LUW—i.e., which business objects that LUW uses. The architecture bag therefore models the LUW Context, and will be named as such.

[4482] Benefits

[4483] Encapsulation. LUW Context hides technical details of persistence, garbage collection, etc., from business developers. Some of the Known Uses have managed to hide this framework, in entirety, from business logic.

[4484] Robustness. This approach guarantees that each business object in the LUW receives forwarded messages. There is no longer the chance of a developer forgetting to include a particular business object in a group message.

[4485] Application Maintainability. As the application requirements change, the set of business objects in an LUW can change without impacting the generic, LUW code. For example, a future version of the Account Payment window could also display Address information. This introduces a new business object into the LUW. Yet it would not require updating saveDataChanges( ), or release( ) methods, as it would have previously.

[4486] Performance. LUW Context can dramatically improve performance in a distributed environment. By nature, it batches up messages for a group. This can reduce network messaging.

[4487] For example, consider a search window which has instantiated 30 business objects. Releasing those objects, if the messages were sent independently, would require 30 network messages. However, with LUW Context, a single message can go from the client to the server. Then, within the server executable, the LUW Context forwards release( ) to the 30 member objects. This is far less costly than using the network for that messaging. Because of this message batching, some readers may confuse LUW Context with Request Batcher. It is true that both reduce the number of network messages. However, the former is concerned with supporting a family of generic, architecture messages, like isDirty( ) and refresh( ), on a single atomic object. The latter is concerned with grouping database requests into a physical package, for un-batching at the server. Although both have similar principles and characteristics, they solve different problems and are implemented differently.

[4488] Architecture Extensibility. LUW Context models the LUW as an actual object in the software. Any other architecture processing which executes on a per-LUW basis can also be coded there. (See the Related Patterns section for examples.)

[4489] This pattern seeks to hide the message propagation from business logic. In fact, messaging to an LUW Context can be hidden completely in an architecture superclass. Previously, saveDataChanges( ) would've been coded specifically in each concrete activity class. LUW Context allows it to be abstracted, as in:

```
AbstractUOWAcrossSaveDataChanges() {
  // Propagate along the save message to all business
  // objects in my LUW. Subclasses don't even need to
  // know about this method
    this.getLUWContext().saveDataChanges();
}
```

[4490] This assumes that business objects were put into the LUW Context in the first place. The context object can be passed in when instantiating an object, transparently by the persistence and streaming frameworks, etc.

[4491] An LUW Context can collaborate with a Request Batcher, if requests are batched for transmission to the data store. Rather than storing the batcher globally, each context and hence model can have its own manager. This allows multiple domain models, in multiple contexts, to send transactions simultaneously but independently. Then, whenever a business object requests an access or update, its request will be intercepted by the model's particular Request Batcher. The batcher then holds these requests until the activity—which owns the LUW—tells the batcher to send them.

[4492] The LUW Context holds onto all domain objects in a particular model. It can therefore collaborate with an Identity Registration Manager, to enforce object uniqueness within the particular context.

[4493] The Potential Variables pattern, which provides local undo, is discussed in the first version of the Object Solutions Handbook. If local LUWs use this approach, then LUW Context is a natural location to store the LUW phase variable.

[4494] In that pattern, every time a business object sets an attribute, the variable must be checked. The LUW Context, as intermediary, can provide a simple public interface which supports setting and querying the phase variable.

[4495] Request Batcher

[4496] FIG. **185** illustrates a flowchart for a method **18500** for batching logical requests for reducing network traffic. A group of business objects necessary for a transaction are provided and managed in a logical unit of work in operations **18502** and **18504**. In operations **18506** and **18508**, logically-related requests received from the logical unit of work are grouped into a single network message which is then stored. The message is sent in operation **18510** upon receiving an order to send the message.

[4497] Optionally, update and retrieval transactions may be grouped into a single network message which is stored. The message may be sent upon receiving an order to send the message. As another option, the requests from the message may be unpackaged at a point across a network and data changes may be persisted. In a further optional embodiment, responses to the requests may be received and the responses may be bundled into a reply. In one embodiment, the requests in the message may be sorted. In such an embodiment, the requests in the messages may also be separated into submessages.

[4498] When domain objects request themselves, minimize the impact of network traffic.

[4499] Individual Persistence assigns responsibility for data access to individual business objects. Then, each business object can retrieve, update, insert, and delete its data from a persistent store independently of other objects. This promotes encapsulation and reuse across business transactions.

[4500] FIG. **186** illustrates the manner in which the present invention sends requests **18600** independently.

[4501] Thus, an LUW which uses multiple business objects will correspondingly have multiple requests. This might suggest that each independent request communicate independently with data access services. Then, each logical request would translate into its own physical, network message.

[4502] Every network message imposes a certain amount of overhead, independently of its contents or length. This implies that multiple, small-grained messages have more overhead than a single, large-grained message. Many networked-constrained environments cannot tolerate this additional overhead. Such environments should minimize the impact of this network traffic.

[4503] Therefore, a high-performance transaction should batch its logical requests into a single network message. Moreover, a framework should handle this packaging, transparently to application logic.

[4504] FIG. **187** illustrates a manner in which the present invention registers requests.

[4505] A Request Batcher **18700** object will group logically-related requests. All requests will register with this coordinating object, rather than sending themselves immediately and independently to their server or database. The batcher will then store these requests together, until told to send them as a unit. This batching applies equally well to update and retrieval transactions.

[4506] A corresponding Request "Unbatcher"**18702** on the server will unpackage the bundled network message. Finally, this Unbatcher will bundle the network response and send it back to the Request Batcher.

[4507] Benefits

[4508] Performance. Sending a single message of multiple requests, as opposed to multiple messages of single requests, improves communication performance.

[4509] Dynamic. Batching and sorting requests is transparent to the requests themselves. Requests do not know that a particular transaction contains them. This dynamic relationship allows any type of request to be part of any transaction at run-time.

[4510] Scaleability. In an asynchronous or multi-threaded environment, an application could use multiple Request Batchers. For example, each LUW could have its own batcher. A batcher needs to store state while building the batch, as requests register. Using multiple instances facilitates registration for, and sending of, multiple batches simultaneously. Users can then multi-task while other, time-consuming requests process in the background.

[4511] This simultaneity can also be supported with one, multi-threaded batcher. In this case, each request registers along with its unique transaction id.

[4512] Centralization. The batcher has visibility over all requests in the LUW. This provides a centralized point to sort these requests, thereby supporting referential integrity and deadlock avoidance.

[4513] Request Sorter

[4514] **FIG. 188** illustrates a flowchart for a method **18800** for sorting requests that are being unbatched from a batched message. A group of business objects necessary for a transaction are provided in operation **18802**. Logically-related requests received from the business objects are grouped in operation **18804**. Sorting rules and/or sort weights are obtained in operation **18806** and, in operation **18808**, the requests in the message are sorted and placed in a specific order determined from the sorting rules and/or the sort weights. The sorted requests are batched into a single message which is sent to a data server where the requests are unbundled from the message in the specific order (see operations **18810**, **18812**, and **18814**).

[4515] A request may also not be allowed to proceed until all dependent requests are completed. A plurality of transactions may each use the same sorting rules for preventing deadlocks. Optionally, the class represented by each request may be determined so that the sorting rules may be based on a class type. As another option, the sorting rules may include referential integrity rules which ensure that references between two relational tables are valid. In such a situation, a linear ordering of requests may also be created based on the referential integrity rules. The numbering of the position of the request in the linear ordering may also be the weight of that request so that requests with lower weights are processed before requests with higher weights.

[4516] In an update transaction, order requests for referential integrity and deadlock avoidance.

[4517] Referential Integrity

[4518] Referential Integrity (RI) ensures that references between two relational tables are valid. That is, foreign keys in one table must refer to existing primary keys in another table. For example, RI rules could require that all accounts have a customer. Then, values in account.cust_id would need matching values in customer.cust_id.

[4519] Mission-critical RDBMSs can enforce RI at runtime. Then, if a modified foreign key does not match an existing primary key, the database prevents the update.

[4520] Continuing the example, a transaction may insert a new customer and its new account. If the transaction inserts the account first, account.cust_id will refer to a non-existent customer. The RDBMS will raise an error, thereby failing the transaction. Instead, the account request should run after the customer request.

[4521] Deadlock Avoidance

[4522] Even without RI, request ordering remains an issue.

[4523] Imagine a transaction A orders customer before account. Conversely, a concurrent transaction B orders account before customer. A will request a lock on the customer table, while B will request a lock on the account table. A must wait for B to complete and release its account lock. Yet B cannot complete until A releases its customer lock.

[4524] Thus, both transactions will deadlock. Many transaction-processing systems would simply fail both transactions, after a time-out. Yet both transactions may otherwise have been valid.

[4525] Traditional Approach

[4526] Traditionally, transactions have hard-coded deadlock avoidance and RI. Each transaction has called its own update module. Each hand-crafted module has ordered multiple SQL statements, according to these rules.

[4527] However, with Individual Persistence, a transaction no longer maps to a centralized module. Instead, independent requests register for the transaction in an ad-hoc manner. **FIG. 189** illustrates an Ad Hoc Registration feature.

[4528] Moreover, an account has no hard-coded "knowledge" that it should persist after a customer. This independence provides flexibility any business object can request an update without concern for other business objects. A framework which constrains the request order must support this flexibility.

[4529] Therefore, an update transaction should sort its requests before sending them to the data server. The sorted result will conform to RI rules. Then, across update transactions, all customer requests can appear before all account requests. In addition, every transaction will use the same sort algorithm. That will prevent deadlocks.

[4530] Multiple requests can no longer send themselves directly to the server, in an ad hoc fashion. Instead, they must register with a centralized object, which can sort them first. A centralized Request Sorter will order multiple requests before finally sending the transaction.

[4531] **FIG. 190** illustrates a manner in which the present invention sorts requests by weight.

[4532] The sorter **19000** will have visibility to sorting rules, or even weights, to determine this order. The rules can typically be based on the class type. Before sending the transaction, the sorter can ask each request which class it represents. In this manner, the sorter can re-order the requests appropriately.

[4533] Benefits

[4534] Separation of Concern. This sorting pattern hides the technical details and complexity of RI from business logic. Applications avoid hard-wiring customized RI rules for its transactions.

[4535] Maintainability. RI rules can easily be changed without impacting application code. Granted, this does not happen frequently in production.

[4536] Reusability. The generic Request Sorter uses universal sorting rules, or weights. These rules are global across business processes. Moreover, the rules are based on existing, reusable business objects. Therefore, new applications can reuse the sorter, as well.

[4537] Visibility. If RI enforcement is distributed across application logic, it can be difficult to get a complete picture of the referential rules. Request Sorter centralises those rules (i.e. weights) in one, visible place.

[4538] A complete, linear ordering of all domain classes can be created, based on the RI rules. Each class will have a unique position in the ordering. This position is the class' weight for the sorting algorithm. Requests for domain objects with lower weights will always appear before requests with higher weights.

[4539] For example, consider the ordering:

[4540] 27. . .

[4541] 28. Customer

[4542] 29. MeterRead

[4543] 30. Account

[4544] 31. Meter

[4545] 32. MonthlyBill

[4546] 33. . .

[4547] This satisfies the RI rule mentioned earlier, because Customers have a lower weight (28) than Accounts (30). Thus, requests for customers will appear in any transaction before requests for accounts. As long as the order satisfies every RI rule, the request sorter can use such a linear ordering.

[4548] This sort ordering can be created programmatically. A sort generator can convert pairwise relationships into linearly-ordered weights. Then, the Request Sorter could use an algorithm like QuickSort to do the actual sorting. (Alternatively, object requests could be sorted as they are registered, a la Insertion Sort.)

[4549] A centralized store-and-forward site must hold requests, before they send themselves to the server. Otherwise they cannot be sorted as a group. Request Batcher provides a centralized place to attach a sorter.

[4550] Separate Models

[4551] **FIG. 191** illustrates a flowchart for a method **19100** for assigning independent copies of business data to concurrent logical units of work for helping prevent the logical units of work from interfering with each other. In operation **19102**, multiple logical units of work operating concurrently are provided. Each of the logical units of work manipulate at least one common business object. In operation **19104**, a copy of the common business object is created for each of the logical units of work such that the copy of the business object for one logical unit of work becomes a separate instance from the copy of the business object for another logical unit of work. Each copy of the business object knows the context of that copy of the business object in relation to the associated logical unit of work. Upon receiving a request to make changes to a copy of the business object of one of the logical units of work in operation **19106**, that particular copy of the business object is changed while the other copies of the business object are not changed. It is then verified in operation **19108** that only one copy of the business object has been changed and the common business object is updated in operation **19110** based on the change to the copy of the business object.

[4552] A business object may optionally be passed as a parameter from one context to another. In such case, a context copy of the business object may be created which includes a duplicate of the original data and excludes a context variable. As another option, an exception may be thrown when an attempt is made to create a copy of a business object being altered by one logical unit of work for another logical unit of work.

[4553] The business object may also be sent to another context as at least one of a single focus of a window that is being created and a parameter in an explicit parameter-passing mechanism. Additionally, the copies of the business objects may be created from a same retrieved data stream. As a further option, receiving a request to make changes to a copy of the business object of one of the logical units of work and changing that copy of the business object may further include the broadcasting of the change to the other logical units of work.

[4554] Support multiple business LUWs within an MVC-based architecture. Manage these LUWs concurrently yet separately, thereby preserving the Isolation property of ACID.

[4555] Multi-tasking allows the user to complete several different business functions independently of each other. These functions which are business LUWs must process concurrently yet separately. For example, a user could establish a new customer account while separately verifying bill details for another customer.

[4556] Providing for these multiple LUWs demands mechanisms which ensure integrity. Specifically, as with any LUW, a primary LUW must isolate its own changes. It is an independent workspace which prevents its changes from affecting other LUW.

[4557] However, an MVC-based OO architecture does not naturally support this requirement. With MVC, the domain model stores all data changes. Windows are merely a view into this model, and they have little business data of their own. In addition, MVC model objects have no idea which views are using them. Instead, the model anonymously broadcasts its data changes, and all views on the model respond by updating themselves. This synchronizes windows with their business data. Thus, MVC allows multiple views to simultaneously display, and be refreshed by, a single copy of the model data.

[4558] **FIG. 192** illustrates the MVC Implementation with Global Model.

[4559] Unfortunately, this benefit of MVC introduces a problem. A globally-shared domain model does not naturally separate concurrent LUWs. It puts a burden on business "activity" objects, which coordinate the high-level business processing across their domain models. Each activity has to either avoid overlap or know specifically how it affects the model.

[4560] Consider a telecommunications system, with two separate business LUWs for paying bills and adding new services, like call waiting. An end user might launch windows for these two LUWs simultaneously. This would allow the user to multi-task while conversing with the customer.

[4561] Both windows display Account and Customer information. In addition, the Account Services window actually modifies the Account object, whereas the Account Payment window does not. Making a payment only modifies the Bill Payment object. Both windows, using MVC, could share the same Account **101** instance.

[4562] It is not atypical for custom architectures to have generic mechanism for persistence and transactions. For example, the architecture could use a straightforward mechanism which automatically saves all business objects within an LUW. Then, when the user saves the Account Payment window, the changes to Account **101** would be accidentally saved as well. The user would then not be able to later cancel changes on the Account Services window. This violates the *isolation* of the two LUWs.

[4563] A similar problem might arise with a garbage collection framework, which explicitly destroys all instances once the LUW has completed. In this case, Account Payment would need to ensure it did not explicitly free the memory for Account, while Account Services was still using it.

[4564] Therefore, using a global, MVC model may preclude using other architecture mechanisms. To avoid the problems of overlapping saves or releasing memory prematurely, the windows could have additional code to ensure the LUWs remain separate. However, adding application-specific code in this manner, to handle a *global technical requirement*, is undesirable.

[4565] Instead, business LUWs should be able to modify domain data independently of each other, transparently in the architecture. In addition, each data change should unambiguously belong to a single, originating LUW.

[4566] Modern Object Transaction Monitors promise to provide this capability. These products will handle locking, tracking which LUW has made changes to which piece of data, etc. However, in the absence of an OTM, a custom architecture needs a different approach.

[4567] Therefore, separate business LUWs by giving each LUW separate copies of business data.

[4568] Rather than using a globally-shared model, each business LUW will own a private, scratchpad copy of its *domain model*. This satisfies the *independence requirement*. A business object in one model will automatically be a separate instance from a business object in *another model*, even if they share the same functional identity. For example, simultaneously opened payment and services windows would have separate copies of Account **101**.

[4569] Then, changes made to a particular instance will only be reflected in the LUW which created and points to that instance. This contrasts with a single, globally-shared model. The latter would simultaneously reflect changes across multiple LUWs.

[4570] **FIG. 193** illustrates the Separate Models for Separate Business LUWs **19300**,**19302**.

[4571] The aforementioned telecommunications example had two separate business LUWs for the account payment and account services functions. Although both activities may be related by the same logical account, this pattern gives each a different context copy. Then, when the customer representative cancels the addition of call waiting, she can still save the payment details.

[4572] **FIG. 194** illustrates the Canceling of one LUW **19400** Independently of Another LUW **19402**.

[4573] Thus, using Separate Models preserves the integrity of business LUWs. It allows each LUW to easily save or cancel independently.

[4574] This pattern is not intended to allow different LUWs to simultaneously change their different physical copies of the same logical entity. In fact, if both windows modified their Account **101** copy, one of the LUWs would fail. (Mechanisms like optimistic locking would detect the data integrity conflict.)

[4575] Precisely for this reason, a good UI design doesn't typically allow simultaneous but separate LUWs to update the same data. And this was not an issue in the example above. Updates to the Account object occur on the Account Services window but not the Account Payment window.

[4576] Benefits

[4577] *Isolation*. Most fundamentally, this pattern solves the Isolation requirement of ACID. It ensures that each LUW has its own "working storage" copies of business data.

[4578] Transparency. Separating models can be done in an architected fashion, as outlined in the implementation section. The separation of LUWs—which is a technical issue—can be hidden completely from business logic.

[4579] Imagine instead that LUWs didn't have their own copies. Then, each operation might need an additional argument: the LUW owning the data change. This would pollute application code with an extra "transaction ID" argument, as in setBalance(newBalance, transactionId). As previously mentioned, this is only required in the absence of an Object Transaction Monitor. An OTM can transparently manage the transaction Id with the thread, without including it as an explicit argument.

[4580] Uniformity. Application developers don't need to know *down* which objects may or may not be used by other, concurrent LUWs.

[4581] The following implementation assumes that the LUW Context pattern is used to help separate the LUWs.

[4582] Each instance of a business object knows which LUW owns it. That is, each instance knows its context. By definition, context gives something a scope, a frame of reference, a relationship to other things. To provide this relationship, an actual LUW Context object will hold onto business objects which share a business LUW.

[4583] In addition, each business object can point to its context. In that manner, business objects know their LUW. This could be useful, for example, while building a domain model. Then, the parent object could propagate its context to a linked, child object.

[4584] Business objects owned by the same business LUW share the same LUW Context, whereas different LUWs have different contexts. Each context therefore con-

tains its own "working-storage" copy of the model. This delineates an individual workspace, or scratchpad, for each LUW.

[4585] At a higher level, each activity object which represents a business LUW has its own context object. That context remains with the activity throughout its entire lifecycle. For initialization, creating a new LUW activity also creates a new context instance for that activity. This context will then be passed downwards, to all business objects, as part of navigation.

[4586] Eventually, when the activity closes, it releases its LUW Context. This correspondingly releases all business objects. They can then be garbage collected, because the only LUW using those objects just closed. A context's lifecycle therefore corresponds directly to its activity's lifecycle.

[4587] Preserving Context Boundaries

[4588] Every context has a scope which limits the business LUW. This context boundary cannot be violated with objects from other LUW Contexts. For, the same instance of a business object cannot live in two different contexts. Otherwise, changes made to that instance would affect two different LUWs.

[4589] It is often necessary, however, to pass a business object as a parameter from one context to another. For example, a user may open up a customer details window based on a selection from a search window. The selected customer becomes the focus of the new window, but it was instantiated in the search context. It is the responsibility of the details window to take the passed-in customer and make a context copy of it. A context copy duplicates the original's data, excluding the context variable, which is re-set to the new context. The copied customer can then be safely used and modified within the details context.

[4590] A business object can be passed as parameter to another context as:

[4591] the single focus of a window that is being created

[4592] a parameter in an explicit parameter-passing mechanism

[4593] For example, when a business object is the focus of a new activity, the launching activity could instantiate the new activity as follows:

[4594] MeterMaintenanceActivity::prorateMeterRead(MeterRead aMeterRead)

```
{
  // Creates a new activity to prorate aMeterReads. This will essentially adjust the
  // read changes, based on corrections from the location, the office, etc.
  // Pseudo-code below
  // Create the new activity instance by reflection, based on the class name, and
// give it a new context and aMeterReads as focus.
newProrateActivity = this.newActivity(
      this.prorateMeterReadClassName(),
      aMeterRead);
  // Other initialization here ...
  newProrateActivity.startup();
}
```

[4595] The newActivity( ) architecture method instantiates a new activity, instantiates a new context, and creates a context copy of aMeterRead that the new activity can use.

[4596] Sometimes an activity cannot get enough information simply by navigating from the focus. Non-focus information that must be passed as an additional parameter could be handled in the following manner:

```
MeterMaintenanceActivity::prorateMeterReadWithCorrection(
  MeterRead originalMeterRead, MeterRead correctedMeterRead)
{
  // Creates a new activity to prorate originalMeterReads based on
measurements
  // in correctedMeterReads. Pseudo-code below. (Duplicates some
code shown
  // for clarification.)
  // Create the new activity instance by reflection, based on the class
name, and
  // give it a new context and originalMeterReads as focus.
  newProrateActivity = this.newActivity(
      this.prorateMeterReadClassName(),
      originalMeterRead);
  // Pass along the corrected read, as well. This will create a context
copy and
  // then use reflection to call the right public setter on the activity.
  newProrateActivity.receive(aCorrectedMeterRead,
"setCorrectedRead");
  // Other initialization here ...
  newProrateActivity.startup();
}
```

[4597] Here, the receive( ) framework method allows any business object to be passed across the context boundary. The receiving activity will automatically create a context copy and then call the specified setter method, with the copy as argument. The setter is application-specific, and it allows the activity to handle and store the context copy wherever it wants.

[4598] FIG. 195 illustrates the Context Copying Protects Context Boundaries.

[4599] A dirty object should not be safely copied into a new LUW context. Otherwise, the second LUW would begin using information that was half-completed in the first LUW. Again, this violates the isolation requirement. The second LUW could save its changes before the first LUW. This means the first LUW couldn't undo any changes it had made to the dirty object. Instead, to avoid this problem, an exception should be thrown when trying to copy dirty objects across contexts. This disallows users from beginning a new LUW based on half-entered data.

[4600] Thus, context copying allows LUW contexts to share parameter information while preserving context boundaries.

[4601] Persistence Caching

[4602] Although LUW contexts manipulate separate copies of business objects, they can often share the same retrieved data stream. For example, when a workstation retrieves data for Customer ABCD, the returned stream can be stored in a global cache. If another context wants to later instantiate its own copy of Customer ABCD, it can reuse the details stored in the stream cache. This improves performance, by avoiding a redundant request to the remote data store.

[4603] Context "Refresh"

[4604] Each LUW, while working on its data, is independent of the other LUWs. From that perspective, each LUW context manipulates data that, to its knowledge, is the most current information from the data store. One instance's changes remain invisible to another copy of the same business entity, during the course of normal processing.

[4605] However, when an LUW context successfully commits changes, it will have more current data than other contexts which it intersects. This up-to-date data can be broadcast and shared with the other contexts. These contexts can then decide to transparently incorporate the changes or not.

[4606] This refresh mechanism can be complex to build, and it requires an understanding of locking issues. For example, does the window have any changed data which might conflict with the new data? This would make the changes which hadn't yet been committed invalid, and the user would need to be notified.

[4607] Although only a few embodiments of the present invention have been described in detail herein, it should be understood that the present invention may be embodied in many other specific forms without departing from the spirit or scope of the invention. Therefore, the present examples and embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope of the appended claims.

What is claimed is:

1. A method for assigning a view to an activity, comprising the steps of:

(a) receiving notification that a startup event of an activity has occurred;

(b) receiving a reference to a first instance of an object created by the startup event of the activity;

(c) determining a view to launch in response to the receipt of the notification and the reference, wherein the view is based on predetermined criteria;

(d) associating the view with the activity; and

(e) displaying the view.

2. A method as recited in claim 1, wherein the predetermined criteria includes at least one of user preferences, an experience level of a user, security profiles, and workflow settings.

3. A method as recited in claim 1, wherein the activity is allowed to run without a corresponding view.

4. A method as recited in claim 1, wherein the activity operates on a machine separate from a machine of an end user.

5. A method as recited in claim 1, further comprising the step of sending a request to be notified when a new instance of an object is created.

6. A method as recited in claim 1, further comprising the step of reading from a configuration file for obtaining configuration information.

7. A computer program embodied on a computer readable medium for assigning a view to an activity, comprising:

(a) a code segment that receives notification that a startup event of an activity has occurred;

(b) a code segment that receives a reference to a first instance of an object created by the startup event of the activity;

(c) a code segment that determines a view to launch in response to the receipt of the notification and the reference, wherein the view is based on predetermined criteria;

(d) a code segment that associates the view with the activity; and

(e) a code segment that displays the view.

8. A computer program as recited in claim 7, wherein the predetermined criteria includes at least one of user preferences, an experience level of a user, security profiles, and workflow settings.

9. A computer program as recited in claim 7, wherein the activity is allowed to run without a corresponding view.

10. A computer program as recited in claim 7, wherein the activity operates on a machine separate from a machine of an end user.

11. A computer program as recited in claim 7, further comprising a code segment that sends a request to be notified when a new instance of an object is created.

12. A computer program as recited in claim 7, further comprising a code segment that reads from a configuration file for obtaining configuration information.

13. A system for assigning a view to an activity, comprising:

(a) logic that receives notification that a startup event of an activity has occurred;

(b) logic that receives a reference to a first instance of an object created by the startup event of the activity;

(c) logic that determines a view to launch in response to the receipt of the notification and the reference, wherein the view is based on predetermined criteria;

(d) logic that associates the view with the activity; and

(e) logic that displays the view.

14. A system as recited in claim 13, wherein the predetermined criteria includes at least one of user preferences, an experience level of a user, security profiles, and workflow settings.

15. A system as recited in claim 13, wherein the activity is allowed to run without a corresponding view.

16. A system as recited in claim 13, wherein the activity operates on a machine separate from a machine of an end user.

17. A system as recited in claim 13, further comprising logic that sends a request to be notified when a new instance of an object is created.

18. A system as recited in claim 13, further comprising logic that reads from a configuration file for obtaining configuration information.

* * * * *