Dynamic Insertion of Object Services

Ajay Mohindra and Murthy Devarakonda
IBM T. J. Watson Research Center, Hawthorne, NY

George Copeland
IBM Austin, Austin, TX

Dynamic Insertion of Object Services

Ajay Mohindra — *IBM T. J. Watson Research Center, Hawthorne, NY*, *ajay@watson.ibm.com*

George Copeland — *IBM Austin, 11400 Burnet Road, Austin, TX*, *copeland@austin.ibm.com*

Murthy Devarakonda — *IBM T. J. Watson Research Center, Hawthorne, NY*, *mdev@watson.ibm.com*

Abstract

This paper presents our experiments with *dynamic insertion* of object services, where dynamic insertion is defined as adding services to object instances at runtime. In contrast, the *static* approach is defined as adding services to objects at *class definition* time. Dynamic insertion allows class implementors to concentrate on the basic functionality for objects, freeing them from the chores of providing normal system services such as persistence, transactions, and concurrency. This paper compares the dynamic insertion approach to the static approach using two benchmarks. The two key contributions of the paper are: (1) It shows how to use the dynamic insertion of services in well known benchmarks; (2) It demonstrates that the dynamic insertion approach incurs low overhead.

1 Introduction

In the Common Object Service Specification document [1] and the Object Services Architecture document [2], Object Management Group (OMG) has specified several object services including persistence, concurrency, transactions, and security. Class implementors can use these services in developing new classes to meet their requirements. At the present, implementors incorporate necessary services to application objects at the class definition time. One disadvantage of this approach is that several classes have to be defined to provide different combinations of services. For example, for a base class called *PhoneDirectory*, a class implementor would have to write a new class called *PersistentPhoneDirectory* to support persistence (inheriting from *PhoneDirectory* and *PersistentObject* classes), a class called *TransactionPhoneDirectory* to support transactions (inheriting from *PhoneDirectory* and *TransactionObject* classes), and a class called *TransactionPersistentPhoneDirectory* to support both transactions and persistence (inheriting from *PhoneDirectory*, *PersistentObject* and *TransactionObject* classes). As can be seen, for every new service, all possible combinations of services doubles the number of classes that need to be written. A total of $2^{\mathcal{P}}$ classes are needed to provide all combinations of $\mathcal{P}$ services. For $\mathcal{N}$ different base classes, this results in a total of $\mathcal{N} * 2^{\mathcal{P}}$ classes–an unnecessary increase in complexity and size of the class implementor's class diagram and library.

Copeland [3] argues that a class implementor should be only concerned with writing code for the base class without worrying about system services such as persistence, concurrency, and transactions. At the time of object instantiation, a user of the class should be able to specify needed services for an object instance. The object runtime would then add requested services to the object instance. This approach of adding services to objects at the instantiation time is called *dynamic insertion* of object services. In contrast, incorporating services at the class definition time is referred to as the *static* approach in this paper. The main advantages of the dynamic insertion approach are:

- It reduces the number of classes to be written by a class implementor to support all combinations of services–For supporting $\mathcal{P}$ services, only $\mathcal{N}+\mathcal{P}$ classes are needed as compared to $\mathcal{N} * 2^{\mathcal{P}}$ classes in the static approach;
- It allows for rapid development of object systems as class implementors focus on writing code to provide only the basic functionality of the objects; and
- It improves programmer productivity by encouraging reuse of system provided services.

Being a novel approach, many people are not aware of how to use the dynamic approach for building object systems. In this paper, we present our experience in using the dynamic insertion of object services for two standard benchmarks. The two benchmarks are implemented in IBM's System Object Model (SOM) [4]. We use OMG's persistence service [1] as an example. We provide performance comparisons of the static and dynamic insertion approachs, and discuss the conditions under which one is better than the other.

The rest of the paper is organized in the following manner. Section 2 briefly describes implementation of the dynamic insertion approach in SOM. Section 3 describes the benchmarks used in the study. Section 4 presents performance measurements and Section 5 presents discussion of the results. Conclusions and future work are presented in Section 6.

2 Dynamic Insertion of Services

In the dynamic insertion approach, a user specifies the set of services to be added to an object instance. The set of services are specified as arguments to the `create_instance()` call. The system runtime, in turn, generates a new hybrid class with the set of services added and returns an instance of the new class. In SOM, dynamic insertion can be implemented using the following two runtime mechanisms:

- Dynamic Subclassing – A new class is dynamically created (at the runtime) by mixing the original class and a special class that provides the required service.
- BeforeAfter Metaclass – The BeforeAfter metaclass allows a *before* method and an *after* method to be inserted around each method of the original class. The before and after methods can be customized to do any service specific tasks to implement the desired functionality (such as locking and unlocking for providing concurrency). Further, multiple BeforeAfter metaclasses can be used to implement a combination of services. For details on the BeforeAfter metaclass mechanism in SOM, the reader is referred to [5].

An example will help illustrate the two mechanisms. Consider a *Directory* class that had been initially written by a class provider with no persistence in mind. The *Directory* class supports three operations: `lookup`, `insert`, and `remove`. A user wants to use the *Directory* class for an application but additionally wants the directory to be persistent. One approach the user can take is to rewrite the *Directory* class by subclassing it from a base class that supports persistence (see Figure 1a). The base persistent class provides necessary interfaces for saving and restoring the *Directory* class. The disadvantage of this hand-crafted approach is that any new combination of services would require the user to rewrite the *Directory* class taking into account all the needed services. This unnecessarily complicates and increases the size of the class diagram–every new service causes the number of classes to double.

Under SOM, similar effect can be achieved at runtime using the subclassing mechanism. Using the *Directory* class as an example, if the user wants to make the *Directory* object instance persistent then

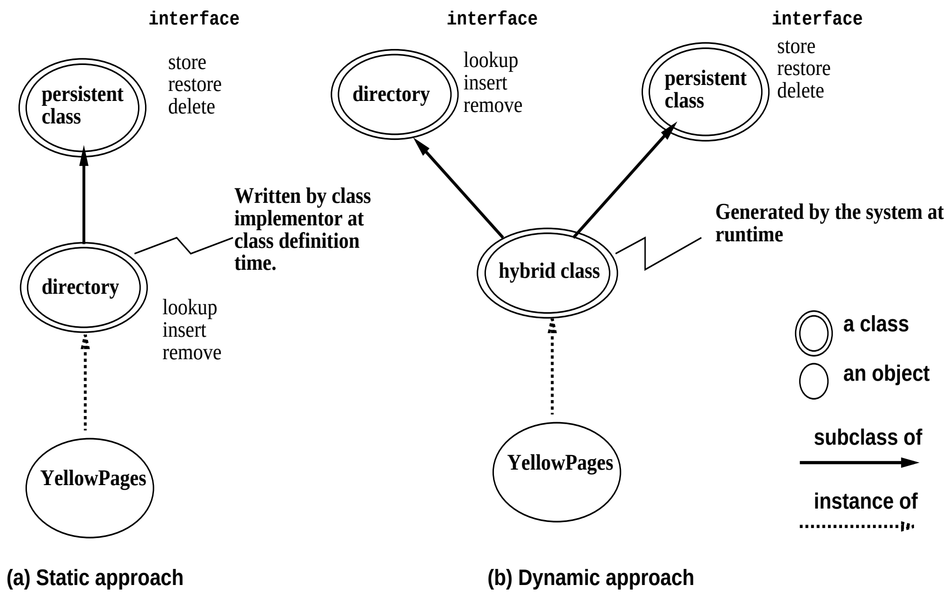

Figure 1: Two approaches for creating a persistent directory object.

the user would specify the need for the persistence service at the time of instantiating the object. The SOM runtime would then generate a new hybrid class from the *Directory* class by mixing in a special class that provides the persistence service (see Figure 1b). This approach automatically inserts the required set of services at runtime, thereby, reducing the amount of work that needs to be done by the user. Figure 2 shows the pseudo-code for the SOM runtime.

The SOM BeforeAfter metaclass mechanism can also be used for dynamically inserting services although the mechanics are slightly differently from the subclassing mechanism. The BeforeAfter metaclass allows the insertion of a *before* method and an *after* method around each method of the original class. This facilitates adding services that have a well-defined "before" and "after" effect. For example, if an object wants to implement concurrency control for all its methods then the *before* method can ensure that the object is properly locked prior to the execution of the method; and the *after* method can do the unlock subsequent to the completion of the method. A variety of such BeforeAfter metaclasses can be written *a priori* and used for adding the desired set of services to the objects.

In contrast to the two dynamic mechanisms described above, the static approach requires the services to be inserted at class definition time. The next section describes the two benchmarks used in performance comparison of the static and dynamic approaches.

3 Benchmarks

We used the OO1 benchmark [6] and TPCB benchmark [7] for the evaluation. They were chosen to measure the effectiveness of the two approaches but not to measure the performance of the underlying database subsystem.

3.1 OO1 benchmark

OO1 was the first benchmark to measure the performance of engineering databases. The benchmark operates on a database consisting of part objects and connections between them. Each part has five fields: a part-id, a part type, an (x, y) coordinate pair, and a build date. The connections capture the linkage between different parts. Each part has three outgoing connections to other parts, and a variable number of incoming connections from other parts. Each connection is represented by a connection type and a length. To provide a notion of locality, all parts

Application Program | **SOM Runtime**

```
Application Program                                    SOM Runtime

SOMObject *obj;
/* Specify the class name and
list of services to add */
obj = create_instance(ClassName, svce1, svce2, ...);
                                                       Lookup internal directory to see if a class
                                                       with name ClassName exists with services
                                                       svce1 and svce2 added

                                                       If such a class exists
                                                           then create and return an instance
                                                       else {
                                                           Create a new class which inherits from
                                                           ClassName and other classes that
                                                           provide services svce1 and svce2.

                                                           Initialize and Register the new class.

                                                           Return an instance of the new class.

                                                           Add the new class to the internal
                                                           directory for future use.
                                                       }
method1(obj, ...);        /* Invoke methods */
```

Figure 2: Pseudo-code for the dynamic insertion approach.

are logically ordered by part-id, and 90% of the outgoing connections are to 1% of parts with part-id values "closest", while the remaining 10% of connections are made to any randomly selected part.

The benchmark consists of three tasks. The first task randomly selects 100 parts, retrieves each part and performs an operation on the part. The second task performs forward traversal–one part is selected at random and all connections from it to other parts are traversed in a depth-first manner up to seven levels. An operation is performed on each part encountered during this traversal. A similar traversal is also performed in the reverse order, starting with a randomly selected part and retrieving all the parts connected to it, and so on. In the third task, 100 new parts are added to the database with 3 connections from each part to other parts, and then updates are committed.

The results were recorded for a "cold start", where the database existed only on disk, and a "warm start", where measurements are taken after ten iterations of each task. For our measurements, we used a database with 20,000 parts and 60,000 connections.

SOM implementation

In the SOM implementation, the data was stored in the IBM's UNIX relational database system, DB2/6000 [8]. The part and connection were writ-

```
interface Part : PersistentObject,
                 StreamableObject {
 attribute long      id;
 attribute string    type;
 attribute long      x;
 attribute long      y;
 attribute long      build;
 void set_values(in long id, in char *type,
                 in long x, in long y,
                 in long tm);
 void connect( in Part p, in Connection c);
 void print_part();
};
interface Connection : PersistentObject,
                       StreamableObject {
 attribute string    type;
 attribute long      id;
 attribute long      length;
 attribute long      source;
 attribute long      target;
 void set_values(in long index, in long length,
                 in char *type);
 void assign(in Connection c);
 void print_connection();
};
```

Figure 3: IDL for the Part and Connection classes.

ten as two separate classes with the individual fields as class attributes. In the static approach, the part and connection classes were subclassed from a *PersistentObject* class, while in the dynamic approach the part and connection classes were subclassed from the base *SOMObject* class. Figure 3 shows the IDL for the static approach. The *StreamableObject* class shown in the IDL is to ensure that OMG's Externalization service is available for the part and connection classes. The *StreamableObject* class provides interfaces to get data in and data out of an object. For the dynamic insertion approach, we used the same IDL except the two classes did not inherit from the *PersistentObject* class. The persistent service, however, was added to the object at the time of instantiation using the dynamic subclassing mechanism.

3.2 TPCB benchmark

The TPCB benchmark from the Transaction Processing Performance Council measures the OLTP system performance. It is based on the TPCA benchmark with the exception that no terminals and networking components are included. TPCB is designed only to exercise the basic components of a database system. TPCB represents transactions on the database belonging to a hypothetical bank that has one or more branches. Each branch has multiple tellers. The bank has many customers each with an account. The database maintains the cash position of the branch, teller, and account, and a history of recent transactions at the bank. The transaction represents the work done when a customer makes a deposit or a withdrawal against his or her account. Figure 4 shows the algorithm for the TPCB benchmark. For our measurements, we used a database with 1 branch, 10 tellers, and 100,000 accounts.

```
Randomly select an account Aid, a teller Tid,
a branch Bid and Delta.
BEGIN TRANSACTION
  Update Account where AccountID = Aid:
    Read AccountBalance from Account
    Set AccountBalance = AccountBalance + Delta
    Write AccountBalance to Account
  Write to History:
    Aid, Tid, Bid, Delta, TimeStamp
  Update Teller where TellerID = Tid:
    Set TellerBalance = TellerBalance + Delta
    Write TellerBalance to Teller
  Update Branch where BranchID = Bid:
    Set BranchBalance = BranchBalance + Delta
    Write BranchBalance to Branch
COMMIT TRANSACTION
```

Figure 4: Algorithm for the TPCB benchmark

SOM implementation

In the SOM implementation, the data was stored as four tables in IBM's DB2/6000. The account, history, teller, and branch were written as separate classes. Similar to the OO1 implementation, in the static approach, these four classes inherited from

Table 1: Measurements for creating and deleting an instance of the *part* object for object sizes 36 bytes and 32K bytes (The times are given in microseconds).

Obj Size	Ops.	Static	Dynamic	
			Naive	File-Away
36 bytes	Create	106	2630	146
	Destroy	32	38	38
32K bytes	Create	1566	4235	1610
	Destroy	394	415	409

the *PersistentObject* class while in the dynamic approach, persistence was added at the time of object instantiation using the dynamic subclassing mechanism. Figure 5 shows the IDL for the four classes for the static approach. Locking was implemented using the AIX operating system semaphores. The *StreamableObject* class shown in the IDL is to ensure that OMG's Externalization service is available for the objects.

4 Measurements

The measurements reported in this section were taken on an IBM RISCSystem/6000 with 256M bytes RAM. For the persistence storage, we used IBM's relational database, DB2/6000. The benchmarks were implemented in SOM version 2.1GA. We divided our measurements into two categories: the first category measures the cost of creating and deleting an object instance, while the second category measures the performance of the benchmark.

Table 1 shows the timings for creating and destroying an instance of an object using the two approaches. For the dynamic approach, we took two sets of measurements. Times under column labeled "naive" are for the strategy in which each `create_instance` operation results in a new hybrid class (see Figure 1) being generated, i.e. the system performs all the necessary operations for creating an instance. Column labeled "file-away" corresponds to

```
interface Account : PersistentObject,
                    StreamableObject {
  attribute long      accountid;
  attribute long      balance;
  attribute long      branchid;
  void update_balance(in long delta);
};
interface Branch : PersistentObject,
                   StreamableObject {
  attribute   long    branchid;
  attribute   long    balance;
  void set_values(in long branchid,
                  in long balance);
  void update_balance(in long delta);
};
interface History : PersistentObject,
                    StreamableObject {
  attribute       long    accountid;
  attribute       long    tellerid;
  attribute       long    branchid;
  attribute       long    delta;
  attribute       long    time;
 void set_values(in long accountid,
                 in long tellerid,
                 in long branchid,
                 in long delta,
                 in long time);
};
interface Teller : PersistentObject,
                   StreamableObject {
 attribute   long    tellerid;
 attribute   long    balance;
 attribute   long    branchid;
 void set_values(in long tellerid,
                 in long balance,
                 in long branchid);
 void update_balance(in long delta);
};
```

Figure 5: IDL for the Account, Teller, History and Branch classes.

Table 2: Performance of the OO1 benchmark.

Time in seconds				
Operation		Static	Dynamic	
			Naive	File-Away
lookup	Cold	7.0	9.6	6.8
	Warm	2.2	4.1	2.2
forward traversal	Cold	54.0	85.0	60.0
	Warm	44.0	59.0	45.0
reverse traversal	Cold	49.0	102.0	62.0
	Warm	39.0	79.0	40.0
insert	Cold	4.8	6.4	4.6
	Warm	3.4	4.2	3.4

Table 3: Performance of the TPCB benchmark.

Units are transactions per second		
Static	Dynamic	
	Naive	File-Away
13.73	13.91	13.91

the strategy in which the system does not treat each `create_instance` operation as a new one, but instead first looks through an internal directory to see if the hybrid class with the specified set of services had been created before. If such a class exists then that class is used for creating a new instance. If such a class does not exist then the system creates a new hybrid class and files it away for future use. As one can see, the file-away strategy optimizes the `create_instance` operation by reducing the amount of work in subsequent instantiations.

Indeed, Table 1 shows that it is more expensive to create or destroy an object instance using the dynamic approach. The additional cost is especially high when using the naive approach, and the additional cost is relatively small using the file-away strategy. The additional costs are due to the operations that are need to create the new hybrid class. These operations include generating a new meta-class, defining the new class data structure, and creating a new instance. The file-away strategy reduces most of the associated cost. When the object is large, the overhead is quite small as Table 1 shows (only about 3.5% for a 32K byte object). For the static approach, most of the work needed to create a new class is performed at the compile time. The performance characteristics of the destroy operation are similar.

Table 2 shows the performance of the three tasks for the OO1 benchmark. The lookup task involves looking up 1000 randomly chosen parts in the parts database. The forward and reverse traversals perform traversal seven levels deep into the parts database, and the insert task inserts 100 new parts into the parts database. Table 2 shows that the performance of the static approach is consistently better than the dynamic approach when compared to times for the naive strategy. When using the file-away strategy, however, the performance for the lookup and insert tasks in the static and dynamic approach is approximately the same. The performance is still poor in the dynamic approach for the traversal tasks for "cold start". The result was surprising at first because the only difference in the two approaches is that the dynamic approach has a higher cost for creating object instances. Further investigation revealed that the overhead associated with creating object instances during traversal was dominating the overall time for a "cold start". During the traversal tasks, more object instances were created than during the lookup and insert tasks. Once the memory was populated with newly created objects, the performance of the dynamic approach for the "warm start" was similar to the static approach.

Table 3 shows the performance of the TPCB benchmark. The benchmark was run on an account database consisting of 100,000 accounts using multiprogramming level of one. The benchmark creates one instance each of the account, branch, teller and history objects. These four instances are used to run all the transactions in the benchmark. Thus, the difference in creation costs contributes a negligible portion to the performance of the benchmark. As a result, the two approaches show similar performance. The main overhead for the TPCB benchmark lies in the heavyweight database operations that are performed.

5 Discussion

In the previous section, we presented performance measurements of the static and dynamic approaches using two standard benchmarks. The results indicate that the dynamic approach can potentially incur significant overhead for creating object classes on the fly. However, the overhead associated with the dynamic approach can be reduced to a small percentage by using a file-away strategy where newly created hybrid classes are reused internally. The file-away strategy, however, introduces overheads of its own: First, the storage overhead for storing the hybrid classes, and second, search overhead for searching through a set of hybrid classes. These overheads can be minimized by using LRU-based garbage collection, and efficient search strategies. For applications that create few new object instances or for applications that create large objects, the performance difference between the static and the dynamic approach is quite small. Thus, dynamic insertion of object services offers a promising approach for designing large object systems. Of course, the main advantage of the dynamic approach is that it reduces the number of classes that a programmer has to write; allows class implementors to focus on the core functionality of the objects; and increases programmer productivity by encouraging reuse of system provided services. The dynamic approach can, thus, be used for rapid prototyping of complex applications.

6 Summary

In this paper, we presented a dynamic approach to adding services to objects, showed how to use the approach for two standard benchmark programs, and demonstrated that the dynamic approach can be implemented while incurring only a small overhead. The main advantage of the dynamic approach is that it allows the class implementors to concentrate the basic functionality of the objects, freeing them from the chores of incorporating system services. The dynamic approach also reduces the number of classes a class implementor has to write to provide different combinations of $\mathcal{P}$ services for $\mathcal{N}$ different classes (a total of $(\mathcal{N}+\mathcal{P})$ classes for the dynamic approach as compared to $(\mathcal{N} * 2^{\mathcal{P}})$ classes for the static approach). The two approaches have been compared using the OO1 and TPCB benchmarks. The results demonstrate that a file-away strategy for the dynamic insertion incurs only a small additional performance penalty over the static approach. The key advantage of the dynamic approach is that it allows for rapid prototyping of object systems and improves programmer productivity. As part of the future work, we are investigating the performance of the BeforeAfter metaclass mechanism, and the effects on the dynamic approach when two or more services are simultaneously requested.

References

[1] Object Management Group. Common Object Services Specification Volume 1. Technical Report 94-1-1, Mar. 1994.

[2] Object Management Group. Object Services Architecture. Technical Report 94-11-12, Dec. 1994.

[3] George Copeland. System services for SOM objects. Technical Report, IBM Austin, Oct. 1994.

[4] IBM, 11400 Burnet Road, Austin, TX. *SOMObjects Developer Toolkit Users Guide*, Oct. 1994.

[5] I.R. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclass in SOM. In *Object Oriented Programming Systems, Languages, and Applications*, Oct. 1994.

[6] R.G.G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database systems*, 17(1):1–31, Mar. 1992.

[7] Jim Gray. *The benchmark handbook: for database and transaction processing systems.* Morgan Kaufman, 1991.

[8] IBM. *Database 2 AIX/6000 Programming Guide*, Oct. 1993.