Persistent Object-Oriented Patterns

by Adrian Marriott, an independent consultant at Logos Software.

This paper outlines a series of common OO patterns that have been encountered in the past decade working with ObjectStore based systems on major projects across all industry sectors.

Table of Contents

Persistent Singleton....3
Database Manager....9
Query Visitor....17
Bespoke Indexes....22
Head/Body....26
Compress Persistent Data....30
Small Object Pool Allocator....37
Transaction Memento....44
String Table....48
Persistent Mutex....52
Evolver....57
Persistent Queue....68
OO Anti-Patterns....79
Frame....79

Persistent Singleton

Intent

Encapsulate database entry-point objects as unique instances, and provide a global point of access thereto.

Motivation

Navigational access to object oriented databases (OODB) typically starts by finding a particular entry-point object using a well-known-name of some description. Programmers can use the object database API directly to find this entry point object each time, but this will litter the code with well-known-names and introduce implicit dependencies throughout the code, and depending on the database engine concerned, it might be slightly less efficient than finding this once and then maintaining a pointer to this object for subsequent use.

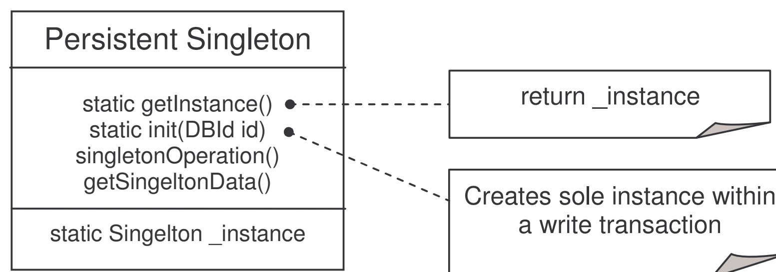
Encapsulating this code within a persistent singleton class, ensures that there is only one instance of this entry-point object, holds the well-known-name in a single place, and provides convenient global access using a variant of this well established pattern.

Applicability

Use a persistent singleton pattern when

- there are database entry points that must be found using a well-known-name
- when the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code

Structure



Participants

- Persistent Singleton
  - Defines a `getInstance()` operation which internally uses the well-known-name to find and return the persistent unique instance from within the OODB. A pointer to the persistent instance is held in the static data member `_instance` for subsequent use.

- May provide dedicated initialization method that takes a database identifier for creating the persistent instance within a write transaction if it is not found

Collaborations

- Clients access the persistent singleton solely through the Persistent Singleton's `getInstance()` method.

Consequences

The Persistent Singleton pattern has all the benefits of its transient relation.

1. *Controlled access to sole instance.* Because the Persistent Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

2. *Reduced name space.* The Persistent Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

3. *Permits refinement of operations and representation.* The Persistent Singleton class may be sub-classed, and it is easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the Persistent Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Persistent Singleton instance needs to change.

5. *More flexible than class operations.* Another way to package a persistent singleton's functionality is to use static member functions, but this makes it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ and Java are never virtual, so subclasses can't override them polymorphically.

However, with a persistent singleton there are further considerations

1. *Cross-transactional validity of _instance*. A transient singleton uses a pointer to reference its sole instance. With a persistent singleton it is necessary to ensure that the `_instance` data member is of a type such that it can be re-used from one transaction to another. With some OODBMs it may be technically possible to access the 'persistent singleton' outside a transaction; the programmer needs to decide whether such non-transactional access should be allowed.

2. *Persistent Singleton scope*. In systems with multiple databases, because these singletons are often used to encapsulate database entry-points, there can be the (strangely contradictory) issue with how many singletons there really are,

and/or which database actually contains the singleton instance relevant to the current context. One solution is to provide a static initialization method which takes some form of database identifier as an argument, and this determines which database is accessed to find or create the persistent instance.

3. *Database access*. On calling the `getInstance()` method, the latest transactionally consistent version of the persistent singleton instance should be returned, and this can require database access, but we don't want to burden the caller with the task of remembering the correct database identifier and passing this in every time. One solution is to provide a static initialization method which is called once at the beginning of the program and takes some form of database identifier as an argument. The persistent singleton can cache the reference to the database for subsequent use when `getInstance()` is called.

4. *Persistent object construction requires a write transaction*. Typical transient singleton implementations check whether `_instance` is null, and if so the sole instance is created before it is returned. Constructing a persistent singleton would require a write transaction, which means accessing the persistent singleton within a read-only transaction is technically unsafe. One solution is to provide a static initialization method which takes some form of database identifier as an argument. This is called once at the start of the program and it attempts to find the persistent singleton in the database passed, if not found the singleton is created and a write transaction is started as required. Thereafter, all calls to `getInstance()` are guaranteed to find the sole instance and can safely be called in a read-only transaction.

Sample Code

Here we describe an example of a persistent singleton written in C++ that holds a collection of `Foo` objects in the ObjectStore database.

```
// Persistent singleton example class
//
class Singleton
{
private:

        // Use soft ptr here so instance is valid across txns.
        static os_soft_pointer< Singleton > _instance;

        // A rootname for the singleton
        static char* _rootName;
        // A persistent extent of foo's
        os_Array<Foo*>* _fooExt;

        // Private ctor so callers cannot create instances hereof
        Singleton();

public:

        // Method that returns the singleton instance. It demands
        // that initialization has occurred and that the caller has
        // started a transaction.
        static Singleton* getInstance()
        {
```

```
        cout << "\nChecking initalization status";
        assert(_instance != 0);
        cout << "\nChecking that we're in a txn";
        assert(os_transaction::get_current());
        cout << "\nReturning persistent singleton";
        return _instance;
    }


    // Static factory function which requires the caller to
    // specify the database that will hold the Singleton and ensures
    // that at runtime there is an update transaction in progress
    static void create(os_database* db);

    // Static initialization function which requires the caller to
    // specify the database containing the Singleton and ensures
    // that at runtime there is a transaction in progress. Returns
    // true if correctly initialized, otherwise false
    static bool init(os_database* db);

    // Returns the persistent extent of Foo objects
    os_Array<Foo*>* getFooExtent()
    {
        cout << "\nReturning Foo extent";
        return _fooExt;
    }
};
```

Above is the C++ header code which declares the static `_instance` data member as an `os_soft_pointer`. This type behaves exactly like a standard C++ pointer pointing to an object allocated on the heap, except this can be used to refer to persistent objects within the database and will be valid from one transaction to the next. It also declares a static `char*` called `_rootName` to hold the well-known-name that labels the singleton object within the database, and an instance variable called `_fooExt` which is the persistent collection of `Foo` objects. There are also inline implementations of the `getInstance()` method, [1] and a method to return the extent of `Foo` objects, along with declarations of the `create()` and `init()` methods which construct and initialize the singleton respectively.

```
// Static initialization
os_soft_pointer<Singleton> Singleton::_instance = 0;
char* Singleton::_rootName = "Singleton";
```

Above are examples of static initializers for the `_instance` and `_rootName` data members, and below we see a constructor which takes care of creating the collection in which the extent of Foo objects will be stored.

```
// Private ctor
//
Singleton::Singleton()
{
    cout << "\nChecking that the singleton instance is persistent";
    assert(objectstore::is_persistent(this));

    cout << "\nCreating os_Array of Foo objects in their own cluster";
    os_segment* seg = os_segment::of(this);
    os_cluster* clr = seg->create_cluster();
```

[1] Note this method does not take any database reference as an argument which makes it easy to use throughout the code base because the database pointer does not have to be managed with respect to the singleton of interest.

```
	_fooExt = new (clr, ts< os_Array<Foo*> >())
			os_Array<Foo*>();
}
```

Below is the create method implementation. This method is passed the database in which the singleton should be created, and starts by checking to ensure that an update transaction is in progress, it then attempts to find any existing singleton object in the database, and will only create a new one if this is not found.

```
// Static factory function which requires the caller to
// specify the database that will hold the Singleton and ensures
// that at runtime there is an update transaction in progress
//
void Singleton::create(os_database* db)
{
	assert(db != 0);
	bool mustCommit=false;
	os_transaction* txn = os_transaction::get_current();
	if(!txn)
	{
		cout << "\nNo txn detected, so starting update txn";
		txn = os_transaction::begin();
		mustCommit=true;
	}
	else
	{
		cout << "\nChecking that current txn is an update txn";
		assert(txn->get_type() == os_transaction::update);
	}

	cout << "\nFinding root '" << _rootName << "'";
	os_database_root* root = db->find_root(_rootName);
	if(!root)
	{
		// Create the root object in its
		// own, commented segment
		char buf[500];
		sprintf(buf,"%s Segment", _rootName);
		cout << "\nCreating " << buf;
		os_segment* seg = db->create_segment();
		seg->set_comment(buf);

		cout << "\nCreating persistent singleton";
		Singleton* only = new (seg, ts<Singleton>())
			Singleton();

		cout << "\nCreating root called '" << _rootName << "'";
		root = db->create_root(_rootName);

		cout << "\nBinding persistent singleton to root";
		root->set_value(only);
	}

	cout << "\nInitializing persistent singleton from database";
	_instance = (Singleton*) root->get_value();
	assert(_instance != 0);

	if(mustCommit)
	{
		cout << "\nCommiting the update txn started during
		      singleton create";
		txn->commit();
	}
}
```

Here we have an initialization function for read-only clients. This is called by programs that will not or cannot start an update transaction, and is passed the database in which the singleton should be found. Again it starts by ensuring that a read-only transaction is in progress, before obtaining a reference to the persistent singleton object through the well-known-name and storing this in the `_instance` data member.

```
// Static initialization function which requires the caller to
// specify the database that contains the Singleton and ensures
// that at runtime there is a transaction in progress
//
bool Singleton::init(os_database* db)
{
	assert(db != 0);
	bool ret=false;
	bool mustCommit=false;
	os_transaction* txn = os_transaction::get_current();
	if(!txn)
	{
		cout << "\nNo txn detected, so starting read-only txn";
		txn = os_transaction::begin(os_transaction::read_only);
		mustCommit=true;
	}

	cout << "\nFinding root '" << _rootName << "'";
	os_database_root* root = db->find_root(_rootName);
	if(root)
	{
		cout << "\nInitializing persistent singleton from database";
		_instance = (Singleton*) root->get_value();
		assert(_instance != 0);
		ret=true;
	}

	if(mustCommit)
	{
		cout << "\nCommiting the txn started during singleton init";
		txn->commit();
	}

	cout << "\nInit returns " << ret;
	return ret;
}
```

After calling either `create()` or `init()` once at the beginning of the program clients can then go on to use the `getInstance()` method with impunity anywhere within their code base, assuming that they have started a database transaction of a type (either read or write) appropriate for their intended operations.

Example Uses

Every ObjectStore customer uses some form of this pattern.

Related Patterns

Database Manager

Database Manager

Intent

In programs which use multiple database instances this pattern simplifies the management of database references or connections, helps to ensure they are open in the correct mode (read or write), abstracts database references away from physical characteristics such as file names and IP addresses, and provides a global point of access to them.

Also Known As

Database Connection Singleton

Motivation

For some systems to achieve performance and scalability targets it is necessary to host a dataset in multiple databases distributed across several different machines. Distributed systems generally require multiple database client processes, and each of these programs might access the various databases in different open modes, reading from one or more databases and updating others. Each program will need to reliably identify these databases, and in such a way that this identification is independent from physical system properties such as file paths or IP addresses.

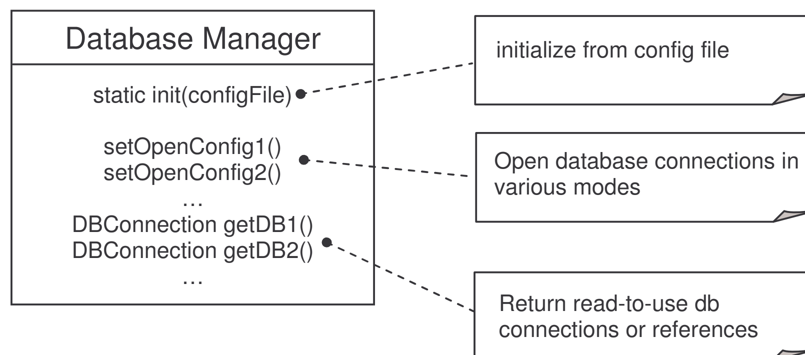
This pattern is a specialization of the singleton [2] so it provides a global access point to the database connections thus removing the burden of passing database connections/identifiers/references around the code.

Applicability

Use a Database Connection Singleton wherever

- there are multiple database connections to manage
- the system can be installed on a different set of machines, or a changing set of machines, during its lifetime

Structure



[2] See http://en.wikipedia.org/wiki/Singleton_pattern for a description

Participants

- Database Manager
  - Provides an initialization method which takes an argument or set of arguments which contain enough data to connect to and open the databases of interest. This is typically read from a configuration file of some description.
  - Different programs may need to open these databases in a variety of modes (for example update or read-only) and different use-cases might require different sets of open modes. Each valid open mode configuration can have a dedicated method which opens these databases in the expected configuration
  - If only a 'handful' of databases is being managed this class can provide dedicated methods to return a valid connection or reference to each of the databases. If there are many, or they are unknown at compile time, then the connection can be returned from a single method `getDBConnection(String)` which takes a database identifier such as a well-known-name.

Collaborations

- Clients get hold of database connections only through using one of the `getDB()` methods.
- In C++ template programming if the templates are written to take database functor or policy [3] objects which return the database connection of interest for use within the template, then the functor or policy objects can be written to adapt the template code to access the database manager as required.

Consequences

The Database Manager can be written as a transient singleton in which case it has all the advantages of a singleton, or the interface can be written entirely as static functions.

1. *De-coupled from physical DB characteristics*. This pattern decouples the physical location of the database in terms of file path, IP address and connection details, from the entire program by encapsulating this information in a single place. Clients of this class can use the dedicated `getDB()` methods or their own internal well-known-name scheme to identify the

[3] Policy-based class design is described by Andrei Alexandrescu in his book Modern C++ Design as, "…assembling a class with complex behaviour out of many little classes (called *policies*), each of which takes care of only one behavioural or structural aspect. …a policy establishes an interface pertaining to a specific issue. You can implement policies in various ways as long as you respect the policy interface."

databases of interest. This insulates the code from changes in these physical characteristics.

2. *Easy open-mode management.* The correct open mode configuration can be set or re-set easily and reliably.

3. *Global access to DB connections.* The database connections do not need to be passed around the code which simplifies programming. For example, C++ template code can utilize functor objects which work with the database manager, providing a very rich generic programming environment.

Sample Code

There are an infinite number of ways to implement database management of varying degrees of complexity and compile-time safety. A minimum requirement for the database manager is it provides a global access point for all databases. An important secondary is to encapsulate all open configurations in a single place so that callers can simply request that the databases are opened appropriately for a particular scenario. Otherwise this class could be implemented in many ways; for example: as a singleton, using an array of database pointers and a database numbering scheme, or STL maps which use well-known-names to manage databases so that specific database file paths do not appear directly in the code. Other DB related services can also be provided such as 'spawning' facilities so that related databases can be created on the fly. As a general guideline the database manager should be designed such that a programmer can always use it without the requirement that a database transaction be in progress.

Below is an example class which provides global access to named ObjectStore databases and ensures that each are opened in the correct mode for various scenarios.

```
class DBMgr
{
public:

	static void createDBsForInitialUseCase();
	// Open modes for clients that want to use
	// persistent queues
	static void openDBsForPQWriter();
	static void openDBsForPQReader();
	static void openDBsForReadOnly();
	static void openDBsForUpdate();
	static void closeAll();

	static os_database* getFirstDBPtr()
	{
		assert(_firstDBPtr && _firstDBPtr->is_open());
		return _firstDBPtr;
	}

	static os_database* getSecondDBPtr()
	{
		assert(_secondDBPtr && _secondDBPtr->is_open());
		return _secondDBPtr;
	}

	static os_database* getThirdDBPtr()
	{
		assert(_thirdDBPtr && _thirdDBPtr->is_open());
```

```
		return _thirdDBPtr;
	}

	static os_database* spawnDatabase();
	static os_database* getSpawnedDBPtr()
	{
		assert(_spawnDBPtr && _spawnDBPtr->is_open());
		return _spawnDBPtr;
	}

	// Allows the spawn series to be reset to zero.
	static void rewind()
	{
		_spawnDBNumber=0;
	}

private:

	// Could be implemented with arrays of pointers here
	static const char* _firstDBName;
	static os_database* _firstDBPtr;

	static const char* _secondDBName;
	static os_database* _secondDBPtr;

	static const char* _thirdDBName;
	static os_database* _thirdDBPtr;

	static const char* _spawnDBName;
	static int _spawnDBNumber;
	static os_database* _spawnDBPtr;
};
```

This class is not implemented as a singleton, but rather uses static data members and static methods everywhere. This manages three databases along with a series of ‘spawned’ databases – a process where databases can be created and filled on the fly. In this case the database names would be hard-coded within the `DBMgr.cpp` file.

There is a mixture of method declarations and method definitions. Most of the definitions simply return pointers to particular databases. Now we examine some of the other methods.

```
void DBMgr::createDBsForInitialUseCase()
{
	closeAll();
	_firstDBPtr = os_database::create(_firstDBName,0664,1);
	_secondDBPtr = os_database::create(_secondDBName,0664,1);
	_thirdDBPtr = os_database::create(_thirdDBName,0664,1);

	OS_BEGIN_TXN(affTxn, 0, os_transaction::update)
	{
		_firstDBPtr->affiliate(_secondDBPtr);
		_firstDBPtr->affiliate(_thirdDBPtr);
		_secondDBPtr->affiliate(_firstDBPtr);
		_secondDBPtr->affiliate(_thirdDBPtr);
		_thirdDBPtr->affiliate(_firstDBPtr);
		_thirdDBPtr->affiliate(_secondDBPtr);
	}
	OS_END_TXN(affTxn)
}
```

The method above closes all open databases so it can be called at any time, and then re-creates all three databases, and then runs a transaction to ‘affiliate’ the ObjectStore

databases one with another so that C++ pointers can target objects in remote databases.

```
void DBMgr::openDBsForPQWriter()
{
	closeAll();
	_firstDBPtr = os_database::open_mvcc(_firstDBName);
	_secondDBPtr = os_database::open(_secondDBName);
}
```

This method first closes all databases and then opens the first and second databases in a specific read-only/update configuration ready for use with a persistent queue writer.

```
void DBMgr::openDBsForPQReader()
{
	closeAll();
	_firstDBPtr = os_database::open(_firstDBName);
	_secondDBPtr = os_database::open_mvcc(_secondDBName);
}
```

Above is the corresponding method for a persistent queue reader; this time the database open modes are reversed.

```
void DBMgr::openDBsForReadOnly()
{
	closeAll();
	_firstDBPtr = os_database::open_mvcc(_firstDBName);
	_secondDBPtr = os_database::open_mvcc(_secondDBName);
	_thirdDBPtr = os_database::open_mvcc(_thirdDBName);
}

void DBMgr::openDBsForUpdate()
{
	closeAll();
	_firstDBPtr = os_database::open(_firstDBName);
	_secondDBPtr = os_database::open(_secondDBName);
	_thirdDBPtr = os_database::open(_thirdDBName);
}
```

Here are other examples of controlled open modes; for read-only and update respectively.

```
void DBMgr::closeAll()
{
	if(_firstDBPtr && _firstDBPtr->is_open())
	{
		_firstDBPtr->close();
	}
	if(_secondDBPtr && _secondDBPtr->is_open())
	{
		_secondDBPtr->close();
	}
	if(_thirdDBPtr && _thirdDBPtr->is_open())
	{
		_thirdDBPtr->close();
	}
}
```

The method above does the work of properly closing all databases used by this application.

```
os_database* DBMgr::spawnDatabase()
{
	char buf[500];
```

```
        sprintf(buf,"%s%d.odb", _spawnDBName, ++_spawnDBNumber);

        TIX_HANDLE(err_database_not_found)
        {
                _spawnDBPtr = os_database::open(buf);
        }
        TIX_EXCEPTION
        {
                _spawnDBPtr = os_database::create(buf);

                OS_BEGIN_TXN(affTxn, 0, os_transaction::update)
                {
                        _firstDBPtr->affiliate(_spawnDBPtr);
                        _spawnDBPtr->affiliate(_firstDBPtr);

                        _secondDBPtr->affiliate(_spawnDBPtr);
                        _spawnDBPtr->affiliate(_secondDBPtr);

                        _thirdDBPtr->affiliate(_spawnDBPtr);
                        _spawnDBPtr->affiliate(_thirdDBPtr);
                }
                OS_END_TXN(affTxn)
        }
        TIX_END_HANDLE

        return _spawnDBPtr;
}
```

Above is an example of code to spawn a database named with a numeric suffix, by incrementing the suffix, creating the new database, and then affiliating it with other databases that need to have direct C++ pointers reference objects it contains.

Having a global access point for databases and scoped control of their open modes means we can use policy-based programming techniques examples of which are shown below.

```
// A policy which always uses the same database.
//
struct SingleDBPolicy
{
        static os_database* getDB()
        {
                return DBMgr::getFirstDBPtr();
        }

        static os_database* getNextDB()
        {
                return DBMgr::getFirstDBPtr();
        }
};
```

A policy in this case essentially functions as an adapter, adapting the `DBMgr` interface to an interface expected by other classes or templates. The policy user is written in terms of the `getDB()` and `getNextDB()` methods; and this policy will ensure that only the first database is returned.

```
// Always returns the database in which the object that uses
// this is allocated.
//
class DatabaseOfThisPolicy
{
public:
        os_database* getDB()
```

```
	{
		return os_database::of(this);
	}

	os_database* getNextDB()
	{
		return os_database::of(this);
	}
};
```

Above is another variant which always returns the database of the object within which the object that uses this is allocated. [4]

```
// A policy where we cycle around a fixed set of databases.
// The number of databases is supplied at compile time.
//
template<int NumberOfDBs>
struct CyclicDBPolicy
{
	CyclicDBPolicy()
		:_currentDBNumber(0)
	{}

	os_database* getDB()
	{
		return getCurrentDatabase();
	}

	os_database* getNextDB()
	{
		_currentDBNumber += 1;
		_currentDBNumber %= NumberOfDBs;
		return getCurrentDatabase();
	}

private:

	os_database* getCurrentDatabase()
	{
		os_database* ret=0;

		switch(_currentDBNumber)
		{
		case 1:
			ret=DBMgr::getSecondDBPtr();
			break;
		case 2:
			ret=DBMgr::getThirdDBPtr();
			break;

		case 0:
		default:
			ret=DBMgr::getFirstDBPtr();
			break;
		}

		return ret;
	}

	int _currentDBNumber;
};
```

[4] For this to work correctly this policy would have to actually be part of a persistent object, and in this case it was designed to be passed in as a template parameter on a persistent template class.

Here we have a template policy which takes the maximum number of databases in the cycle set. When the policy user calls `getDB()` it simply returns the current database. As they call `getNextDB()` this policy changes the current database through the first three databases in turn, then spawns databases up to the maximum specified and then wraps around to the first three databases again.

Example Uses

Most ObjectStore customers who have to manage multiple database instances use some form of this pattern.

Related Patterns

Singleton, Persistent Singleton

Query Visitor

Intent

Represents a query to be performed on the elements of a persistent object structure. Query Visitor allows you to define new result set formats without changing the underlying persistent object model, and avoids polluting the persistent classes with rendering logic.

Motivation

This is a variant of the standard visitor as described in GoF [5] specifically dealing with OODB queries. Consider a persistent object model held in a database from which you need to render some subset into XML to load into another system. By providing an `accept(QueryVisitor v)` method on persistent objects, you prepare the way for many different query result formats without requiring the author of the query change any aspect of the persistent object model. Different query visitors can render different subsets of the data into a variety of formats. This can be useful for writing reports that scan the persistent object model, general query processing for example to produce HMTL directly from the persistent object model, or to produce comma delimited files ready to load into a relational database.

Less usual but still possible are visitors which update the data set, for example certain schema evolution strategies can utilize an update visitor.

Applicability

Use the Query Visitor pattern when

- A persistent object structure contains many classes of objects with differing interfaces, and you want to perform queries that depend on the concrete classes.

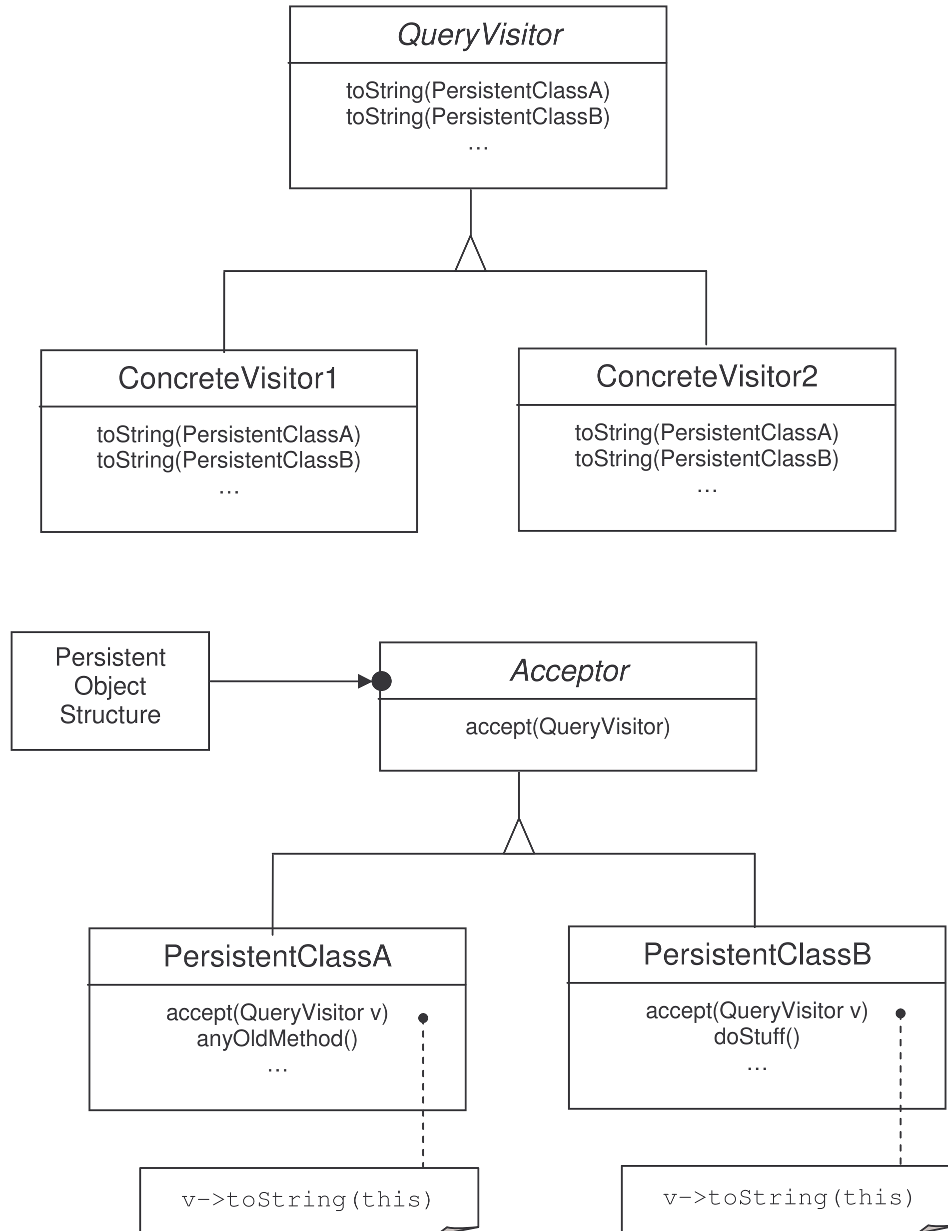
- There are several distinct and unrelated query result formats, such as XML, comma delimited tuples, HTML etc. and you want to avoid polluting the persistent classes with these rendering operations. Query Visitor lets you encapsulate all the query related code in a single class.

- The persistent model rarely changes, or should not change to avoid the expense of schema evolution on existing data sets, but you often need to produce new result set formats.

- The persistent object structure is shared by many applications but only a few need to render the model into the result set produced by the various visitors.

[5] See: http://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612/ref=sr_1_1?ie=UTF8&s=books&qid=1238501992&sr=8-1

Structure



Participants

- QueryVisitor
  - An interface that declares a `toString()` operation for each class in the persistent object model. The operation's signature determines

which persistent class invokes `toString()` as a call-back on the concrete visitor, which can then access the persistent object directly through its public interface.

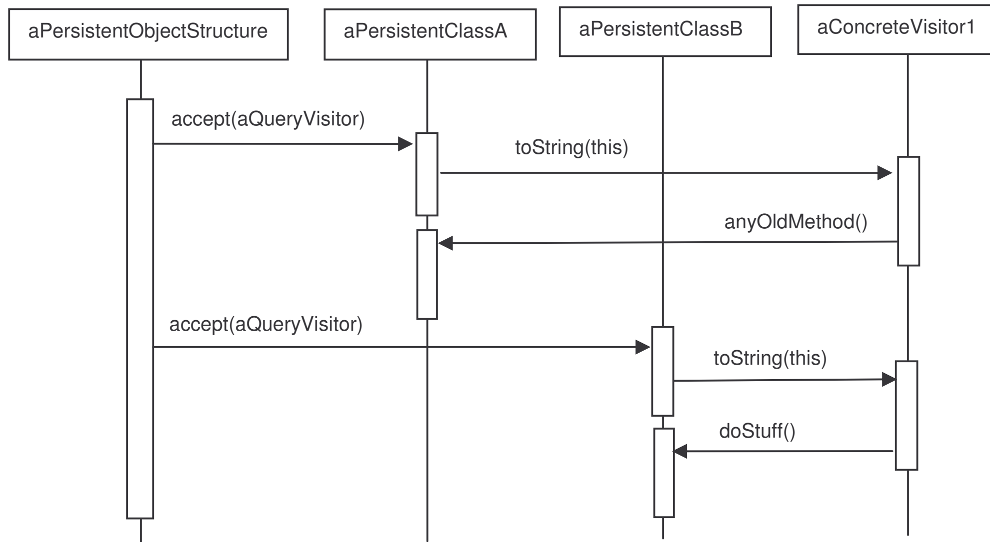
- ConcreteVisitor
  - Implements every overloading of the `toString()` operation declared by the QueryVisitor interface. Each of these overloaded methods encapsulates the rendering code for one particular persistent class. ConcreteVisitor provides context for the query and stores any local state that may need to be accumulated during the traversal of the persistent model.
- Acceptor
  - An interface that declares an `accept()` method that takes a QueryVisitor as an argument.
- PersistentClass
  - Implements the Acceptor interface by defining an `accept()` method that takes the interface QueryVisitor as an argument. This is usually implemented to call the `toString()` method on the passed QueryVisitor.
- PersistentObjectStructure
  - Can enumerate its elements
  - May provide a high-level interface to allow a QueryVisitor to visit some or all of the persistent objects in the database
  - Typically either a composite [6] of some description or a collection such as an array or list.

Collaborations

- A client that uses the QueryVisitor pattern must create a ConcreteVisitor object and then traverse the persistent object structure, visiting each persistent object with the visitor.
- When a persistent object is visited, it calls the `toString()` operation that corresponds to its class, by supplying itself as an argument to this operation. This also lets the visitor access its current state via its interface to render query results.

[6] See http://en.wikipedia.org/wiki/Composite_pattern for more details

The following interaction diagram illustrates the collaborations between a persistent object structure, a query visitor, and two persistent objects:



Consequences

Some of the benefits and liabilities of the QueryVisitor pattern are as follows:

1. *QueryVisitor make adding new query result formats easy*. Visitors in general make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor. In contrast, if you spread this functionality over many classes, then you must change each class to define a new query format.

2. *A QueryVisitor encapsulates query specific code.* Behaviour and state concerned with query formatting isn't distributed throughout the persistent object model; it is localized in a visitor. Also, unrelated query code can be partitioned in their own visitor subclasses. That simplifies both the persistent object model and the query algorithms defined in the visitors. Any query-specific data structures can be hidden in the visitor.

3. *Adding new persistent object classes is harder.* The QueryVisitor pattern makes it more difficult to add new persistent object classes that need to extend the Acceptor interface. Each new PersistentClass requires a new `toString()` method be declared in the QueryVistor interface, and a corresponding implementation in every ConcreteVisitor class. Sometimes a default implementation can be provided in QueryVisitor that can be inherited by most of the ConcreteVisitors, but this is the exception rather than the rule.

   So the key consideration in applying the QueryVisitor pattern is whether you are more likely to change the query result format applied, or introduce new query formats, rather than change the persistent object model itself. The

QueryVisitor class hierarchy can be difficult to maintain when new PersistentClass classes are added frequently, but this can be much better than triggering the need for schema evolution by defining query operations, that may need query specific state, within the persistent classes themselves.

4. *Visiting across class hierarchies.* An iterator can visit the objects in a structure as it traverses them by calling their operations, but an iterator can't work across object structures with different types of elements. By using a form of double-dispatch, [7] the visitor does not have this restriction. It can visit objects that don't have a common parent class. You can add any type of object to a Visitor interface and they do not have to be related through inheritance at all, and this is ideally suited to implementing query code against object databases.

5. *Accumulating state.* QueryVisitors can accumulate state as they visit each persistent object in the database. Without using a QueryVisitor, this state would be passed as extra arguments to the methods that perform the traversal, or they might appear as global variables.

6. *Breaking encapsulation.* The QueryVisitor pattern assumes that the public interface of each persistent object is powerful enough to allow the query visitor to access the necessary state to fulfil the query. As a result, the pattern often forces you to provide public methods that access a persistent object's internal state, which under some circumstances may compromise its encapsulation.

Example Uses

The government mapping agency of the UK, the Ordnance Survey, uses this pattern in its OS MasterMap system. Other ObjectStore customers use variants of this pattern for dumping database contents to ASCII files.

Related Patterns

Composite

[7] See http://en.wikipedia.org/wiki/Double_dispatch for more information

Bespoke Indexes

Also Known As

Don't Query for Every Object

Intent

For programs that require the absolute maximum of performance and scalability it is necessary to write programs that utilize novel data structures and new algorithms designed with detailed knowledge of the specific problem context. Support the most critical use-cases of your system directly with bespoke persistent index structures that optimize read and write operations across the objects used by those use-cases.

Motivation

Direct navigation between persistent objects can offer performance advantages over querying collections of objects for those that meet specific criteria. The possibility of direct navigation offered by OODBs is best exploited by designing object structures optimized to support particular use-cases; simply navigating an extent [8] of objects exhaustively searching for the ones of interest is almost certainly less than optimal, particularly if the selection criteria can be known in advance, for example at compile time.

This idea can be illustrated by a simple example. Imagine an OODB of vehicle data, where one of the primary use-cases is to find all the cars of a certain colour from all those currently in stock. In the relational domain, this would typically be implemented as a table of vehicle data, one column of which would be the colour code, and the relevant query would be an SQL statement with a WHERE clause that selects on the colour code column.

The equivalent OO model might use an extent, say an array or a list, of VehicleData objects, and the program uses an iterator to spin across this extent testing each element as it goes, building a result set in a heap allocated collection by inserting references to all those persistent objects found that meet the selection criteria. This exhaustive search visits every VehicleData object in the database.

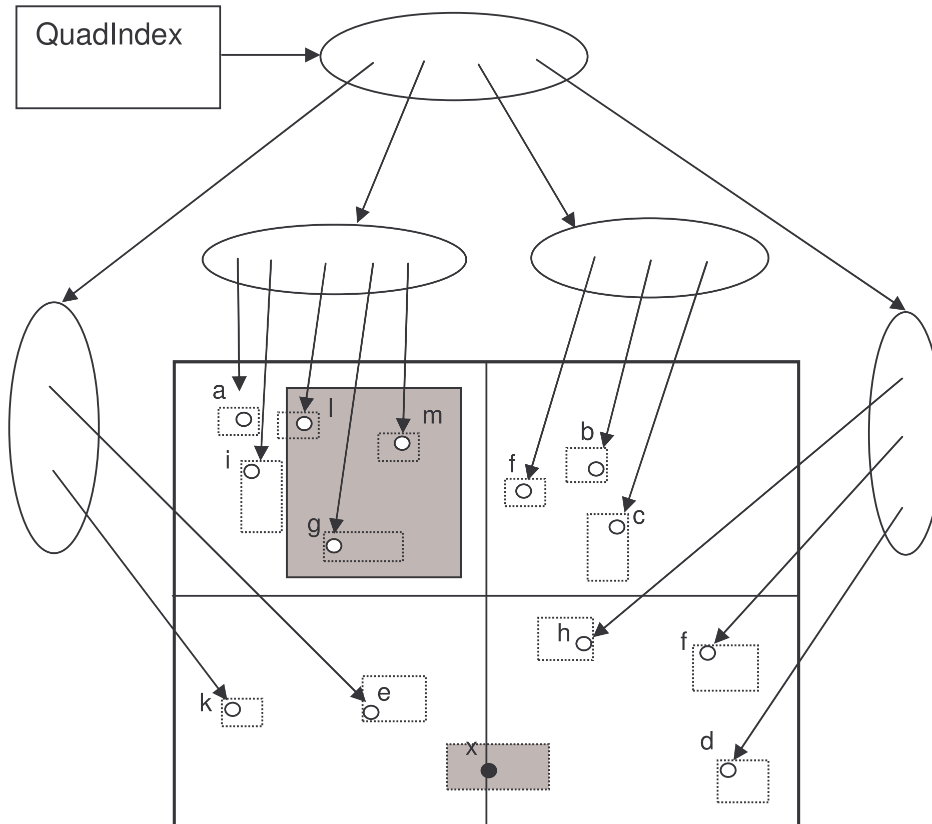
Improvements to this can be made by designing a bespoke index structure consisting of a persistent hash table keyed on the colour code, the entries of which reference collections of VehicleData objects containing only those with the designated colour code. This primary use-case can now be satisfied by using the colour code information passed in with the 'query' to perform a single hash table lookup and return the result collection directly; no collection is scanned for compliant objects, and no result set collection needs to be allocated and populated.

Of course relation tables with indexes on particular columns introduce precisely this type of hash structure, but there are examples where the OO designer can better exploit information specific to the context of use. Lets take another example, where

[8] An 'extent' in OO parlance is a collection of all the instances of a particular class or sub-class.

the property being indexed is not totally orderable, [9] and not well handled by standard relational database indexes.

Assume there is a rectangular map area containing various map elements as C++ objects. A quad-index divides this map area into four equal sized cells and holds the objects that are contained within the cell in a separate collection. Queries are presented to the index in the form of rectangular areas that partially cover the map. The index works by reducing the number of objects that need to be examined to fulfil a 'query'.



The diagram above shows a quad index with the map area divided into 4 cells and containing objects (a – m), each of which has an associated bounding box. Each cell is represented by a collection of pointers to the objects contained within that cell. The grey rectangle represents a query. To fulfil the query only 5 objects need to be examined (a, g, i, l, m).

First, an overlap test is done between all the cells and the query. In this case only one cell overlaps (the top left). Now the overlap test is re-applied to all the bounding boxes of the objects within that cell. This produces the query result set.

[9] See http://en.wikipedia.org/wiki/Total_order for an explanation

There are problems with Quad indexes but there is no need to detail these issues here, save to say that this structure can be refined and developed to provide optimal performance for the queries it is designed for.

In short, design your persistent object model much as you would one that is heap allocated, but with an understanding that it will actually reside on disc, and this can provide orders-of-magnitude performance advantages. System based on relational databases cannot better this type of optimization.

Applicability

Use the Bespoke Indexes pattern when:

- There are important well-defined use-cases which need to read objects from the database and the selection criteria can be known at compile time. The opposite of this condition is the ad-hoc query, where the selection criteria will be substantially defined at runtime. [10]

- Critical use-cases would benefit from specialized update behaviour to eliminate or remove deadlocks [11] or reduce transaction commit times. [12]

Structure, Participants & Collaborations

The structure, participants and collaborations at the object level are entirely context dependent because there is no sensible generalization of a bespoke index by definition – they are bespoke, and the point is that the performance improvements is dependent on their being designed with a particular use-case in mind.

For example, some types such as map coordinates cannot be arranged in a total order, and traditional indexing structures do not handle these well. Academics and industry practitioners concerned with geographical data have designed spatial indexes such as the Quad-tree [13] and the R-tree [14] that better handle such types, and these can be directly implemented in the persistent model – albeit with adaptations to account for the fact that disc reads will occur as this structure is navigated.

Consequences

The Bespoke Indexes pattern has the following consequences:

1. *Critical use-cases can run very quickly.* The use-cases for which the bespoke indexing structure was designed can run orders-of-magnitude faster than implementations which used generalized data structures.

---

[10] Selection criteria consist of the sort of data that appears in the WHERE clause of an SQL SELECT statement. Ad-hoc queries have no part or sub-clause which can be used to construct a bespoke data structure in advance of the query's execution to optimize the selection process. Sometimes only one part or sub-clause of the selection criteria needs to be identified in order to be able to define a suitable indexing structure.
[11] For example, see Persistent Mutex pattern below.
[12] For example, see Pool Allocator pattern below.
[13] See http://en.wikipedia.org/wiki/Quad_tree for more info
[14] See http://en.wikipedia.org/wiki/R_tree for a description

2. *Different use-cases can be supported by separate bespoke indexes*. If there are several critical use-cases, each can be accommodated by their own access structures.

   To take an example from the finance industry, trades in stocks and shares might have two critical use-cases: the first to find trades by stock symbol, the second to find trades between two points in time. The first might be best accommodated by a dedicated hash-structure keyed on symbol, and the second might require some form of skip list [15] or other ordered data structure.

3. *Access patterns can be aligned with disc location*. Further performance improvements are often possible with some database systems if the programmer has control of the relative location of persistent objects.[16] By aligning the design of the persistent data structures with the physical location of the objects on disc, very fast systems are possible. If the entire data set can be held memory resident 'locality of reference' is not an issue.

4. *Bespoke index structures can have implicit dependencies*. The separate index structures used for different use-cases can 'interact', particularly when optimal clustering is attempted for each, and what is optimal clustering for one is sub-optimal for others. Then it is necessary to examine how the use-cases interact and reconsider the combined behaviour of all the structures involved and attempt to resolve these contradictions by refining the data model.

5. *Bespoke indexing can be difficult to maintain*. If system requirements change in such a way as to contradict any assumptions underpinning the original specification, bespoke indexes can become less effective or even entirely useless. Either the index code needs to be changed to meet new demands, or new bespoke index structures can be introduced. In situations where databases are 'in production', this can force schema evolution on existing databases.

   This effort is comparable to a similar scenario with relational database systems where requirements change so assumptions underpinning a particular normalization are undermined, and the existing table schema needs to be reconfigured; bespoke indexes and relational normalization are both undertaken in the context of underlying assumptions, and if they change, then this affects the effectiveness of the database

Example Uses

Protek (Telco), BlueCrest, BNP Paribas, Ordnance Survey, Orange, nearly every ObjectStore customer.

Related Patterns

All

[15] See http://en.wikipedia.org/wiki/Skip_list for further details
[16] This is referred to as 'locality of reference' and/or 'clustering' in the literature.

Head/Body

Intent

Improve database read access by deferring the cost of reading large objects.

Also Known As

Persistent Proxy

Motivation

This is a variant of the standard proxy as described in GoF [5] specifically concerned with improving database read access by deferring the cost of accessing large objects.

Consider an image database which stores millions of pictures along with meta-data associated with each image, such as the subject, a date, the photographer etc. Users of the database might search for images by subject, and be given a list of candidate pictures, from which they choose to see the full image. This is a two stage process: the first might return some basic meta-information along with a small thumbnail from a potentially very large list of candidates; the second returns all the meta-data and the full sized image.

For efficiency we use a proxy (head) containing the summary meta-data and a thumbnail, which are the only objects that need to be indexed, searched and returned to fulfil the first use-case. These reference the body which contains the rest of the meta-data and the full sized image.

The idea can be extended to support different sub-classes of head/body pairs which all adhere to the same interface, so clients can use the different concrete sub-classes transparently. In our example, such a scheme could be used to support different image formats, and/or cases where some of the images are stored within the OODB and others are stored in external files or systems.

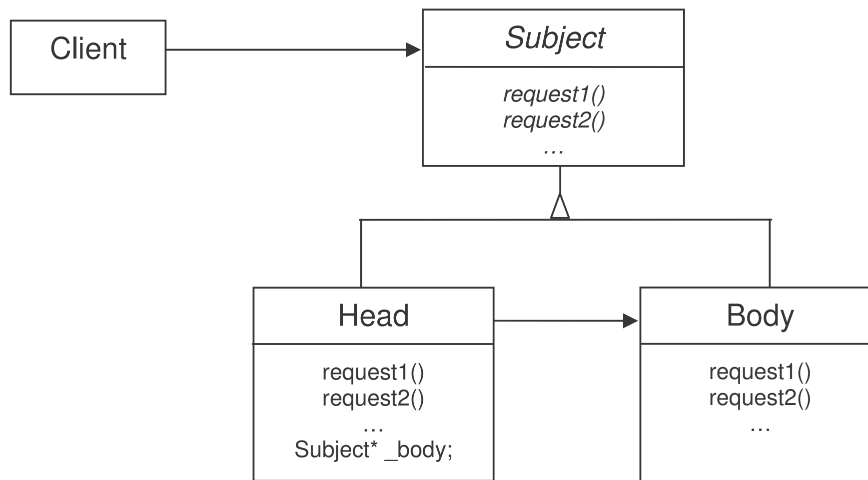
Applicability

Use a head/body whenever

- There is a two stage access to large objects, the first to retrieve summary information from a potentially large collection of these objects, and the second which chooses from this initial search result.

- Some of the data is held outside the OODB. In our example the proxy can reference a disc file containing the full sized image, rather than store this directly in the database. The first search only needs to read the small amount of summary data from the OODB and can postpone reading the external disc file until it is needed.

Structure



Participants

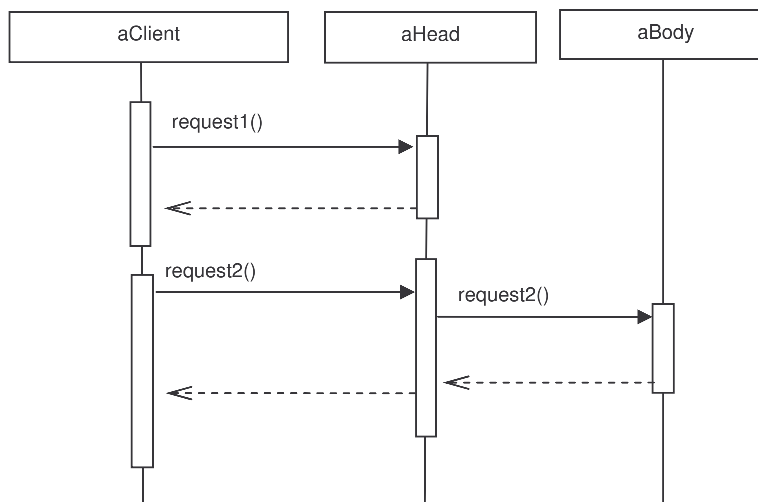
- Head
    - Maintains a pointer or reference to the body object. If the Body class also implements the Subject interface then the pointer can be a base-class pointer (as shown).
    - Caches additional information about the body so that access to the body can be postponed
    - Can provide an interface identical to Subject's so that a head can be substituted for the body.
    - Can control access to the body and may be responsible for creating, reinitializing and deleting it.
    - Can check that the caller has the access permissions required to perform a request
- Subject
    - Defines the common interface for Head and Body so that a Head can be used anywhere a Body is expected
- Body
    - Defines the real object containing the detailed information needed for the postponed access.
    - Can provide an interface identical to Subject's so that a body can be used wherever a head might be. This may entail having a back-pointer

or reference to the head, to correctly delegate those calls which require access to state held in the head.

Collaborations

- The Head services initial requests using only the state of its own data members, and only accesses the Body to service requests that require ‘detail data’.

The following interaction diagram illustrates the collaborations between a client, a head, and a body when an initial request for summary information is made, and then when more detailed information is required.



Consequences

The Head/Body pattern introduces a level of indirection when accessing large persistent objects.

1. *Effectiveness increased when combined with clustering*. The principle underpinning this pattern is to reduce unnecessary disc reads for the *first* access. The first access typically requires that many head objects are accessed to compile a list of ‘selection candidates’. If all the head objects are located on disc together with Body objects, it is doubtful whether disc reads will be reduced significantly. To be properly effective the Head objects must be located on separate disc areas to the Body objects, so the entire collection of heads can be read as efficiently as possible. Consequently, this pattern is most effective when used with OODBs that offer some degree of control of the physical location of persistent objects on disc.

2. *Multiple heads are sometimes used*. In situations where there are several different selection criteria, multiple different head objects can be used all referring to a single body object. All the instances of the different head object types can be located on separate disc areas, to minimize disc reads for searches across head objects of the same type.

Example Uses

Various ObjectStore customers particularly those which store large ‘blobs’ such as images in film libraries or image databases (DeventIt).

Related Patterns

Proxy

Compress Persistent Data

Intent

For large data sets where there are many millions or billions of objects, compressing data into as few bytes as possible can maximize the number of objects that can be held resident in memory ready for use. Reducing disc accesses can massively improve overall performance.

Motivation

Take a hypothetical GIS database holding all the mapping data for all of Europe down to the level of detail where every building, every garden, each wall or fence, even the kerb-stones along the side of the road are represented as individual objects. This data set would comprise billions of objects. Aside for the very complex problems concerning representation, for example whether to store or derive topology, one of the main problems with this system is the sheer volume of data; any strategy to reduce its size will almost certainly have a positive impact on every use-case for which the data is required.

Aside from using standard lossless compression techniques [17] such as run-length encoding [18] or zLib [19] there are several approaches that can be used which require careful analysis of the data to be stored. The standard algorithms were designed for general use and tend not to produce very high compression ratios on small blocks of data, and they can incur significant decompression overheads at runtime. As a general rule of information theory the more you understand of your data the better chances you have of significantly compressing it, and using bespoke techniques can also result in really insignificant runtime overheads. Examples of these come under various categories such as: not storing optional attributes, re-ordering data members, data type demotion, bit-field compression, string tables and the elimination of virtual functions in very small objects.

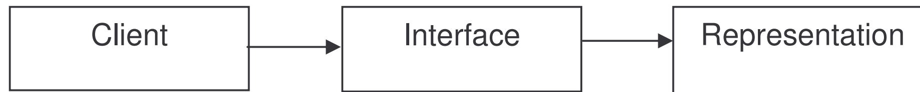
Applicability

Use the Compress Persistent Data pattern whenever:

- The data set is larger than can be accommodated in physical memory
- There are demanding performance requirements that would benefit from a memory resident data set

[17] See http://en.wikipedia.org/wiki/Data_compression#Lossless_data_compression
[18] See http://en.wikipedia.org/wiki/Run-length_encoding
[19] See http://en.wikipedia.org/wiki/Zlib

Structure



Participants

- Interface
    - This is identical to a naïve uncompressed implementation of the class.
    - Method signatures are identical to standard implementation. Updater methods that operate on compressed data members will take arguments of the full-width type and check for out-of-range values before updating the compressed representation.
    - Return types are uncompromised. Any compressed values are converted to the full-width type before they are returned.
- Client
    - Uses the class/object interface in exactly the same way as it would for a standard uncompressed implementation
- Representation
    - The holds the data values in a few bits as possible
    - This can be entirely local to the object or can be augmented with remote data structures such as string tables.
    - Optional attributes can be stored externally to the class and referenced with a pointer. If the pointer is null, then the optional attribute is absent, and the cost of allocation is saved.
    - Re-order data members such that compiler alignment behaviour does not waste space by making data members of the same type adjacent, progressing from shortest to longest data types (or vice versa). If necessary replace `structs` with their primitive members, to avoid bad `struct` alignment, and any accessor functions can still return a newly created `struct` with the correct data values as necessary.
    - Analyse the possible range of values that each data member can take at runtime and limit the size of the data members to the smallest native type that can accommodate all these values. For example, a person's age is unlikely to require a 32-bit integer.

- In large classes with many data members this idea can be extended to whole groups of data members, where each data member can be allotted a number of bits sufficient for any value it might take on, and together space can be saved by implementing these using C++ bit fields.

- String tables can store common repeated strings once in a dedicated hash or table structure, and instances of these strings can be internally represented as an integer or a pointer into this structure. If the strings are large than 4 characters, this will save space.

- Eliminate the virtual function pointer in small objects whose size is such that said pointer forms a significant proportion of the objects total size.

Collaborations

Clients access the compressed data solely through the class/object interface exactly as they would do for a normal class where effort had not been expended to crush the data into as few bits as possible.

Behind the scenes there may be collaborations with other classes depending on how the compression has been achieved.

Consequences

Some compression techniques are only effective when a certain ratio of persistent objects have particular properties, and some techniques can incur runtime overhead.

1. *Only compress numerous objects*. Compressing a singleton object will have an insignificant effect on total database size. Generally it is only worth attempting to compress the most common objects in the data set.

2. *Optional attribute pointers only save space under certain conditions*. Using a pointer to refer to optional attributes externally only saves space if the size of optional attribute is greater than that of a pointer; the greater the proportion of objects that have the optional attribute unset, (the pointer is null) the greater the space saving. As time progresses, this ratio can change such that the space saving is no longer significant.

3. *Re-ordering data members is never a disadvantage.[20]* This is literally free; there is no negative side effect from eliminating unused space arising from compiler data type alignment. However, it can only be used for languages that have a well-defined object layout (i.e. C/C++).

4. *Demoted data types can cause maintenance issues*. Again this is free from negative runtime side effects but code maintenance can be affected; if the requirements change and the range of values increases, then a data member

[20] If there is deployed database, reordering data members can mean that the existing database must undergo schema evolution.

can require type promotion. For example, if an `integer` data type was sufficient to record the position of a map feature to the nearest centimetre and the survey accuracy subsequently increases to millimetre precision, the range of values increase by a factor of ten, and a `long` data type may be required. This might force a schema evolution of existing data sets.

5. *C++ Bit-Fields can be implemented using integer arrays*. Although C++ offers bit-fields to elegantly support the processes of squeezing whole collections of data member values into as small a memory space as possible, similar can be achieved in other languages with bit-shift operators and bit-wise masks against integer data members or integer arrays.[21]

6. *Bit-fields are more effective on large classes*. Bit-fields are most effective on classes with large numbers of data members that all have a limited range of values. This enables each of the different data members occupy adjacent bits on disc with little or no wastage.

7. *Check all updates for out-of-range values*. When updating a compressed data member, where the signature of the update method has a wider type than the data member it is important to check incoming arguments for out-of-range values; either `assert()`, write to a log file, or both.

8. *Virtual function pointer elimination rarely useful*. This technique is only worthwhile on *very* numerous small objects, because to make a difference the object has to be small – around the same size as a pointer – and because the objects are small there has to be a huge number to appreciable impact the overall database size. Examples include hand-rolled international character sets, where each character has a virtual function pointer, and point objects in GIS databases; in both cases there can be billions of these objects.

Sample Code

Here is a contrived example which demonstrates the idea of using bit-fields clearly.

```
class Foo
{
public:
	// Ctors
	Foo()
		:_a(false), _b(0), _c(0), _d(0)
	{}

	Foo(bool a, char b, int c, int d)
		:_a(a), _b(b), _c(c), _d(d)
	{}

	// Accessors
	bool getA() const { return _a;}
	char getB() const { return _b;}
	int getC() const { return _c;}
	int getD() const { return _d;}

	// Updaters
```

[21] See http://java.sun.com/javase/6/docs/api/java/util/EnumSet.html for a description of Java `EnumSet` which is specifically adapted for this sort of work.

```
	void setA(bool a){
		_a = a;
	}

	void setB(char b){
		assert(b >= 0 && b < 8);
		_b = b;
	}

	void setC(int c){
		assert(c >= 0 && c < 1024);
		_c = c;
	}

	void setD(int d) {
		assert(d >= 0 && d < 131072);
		_d = d;
	}

private:

	// *** Implementation Section ***

	bool _a;
	char _b;
	int _c;
	int _d;

};
```

Above we have the direct implementation of a class `Foo` which has four data members: a boolean, a character and a couple of integers. Analysis of the hypothetical use-cases for which `Foo` is used means that the values of its data members are limited to a range of values, and the `asserts()` in the updater methods check for these ranges. Running `sizeof(Foo)` on this class returns a size of 12 bytes; the `bool _a` and the `char _b` are at offset 0 and 1 in the object each occupy 1 byte. The `integer _c` and `_d` members are at offsets 4 and 8 respectively and occupy 4 bytes each.

Now we examine the same class but implemented using bit-fields. The code is shown below.

```
class Foo
{
public:
	// Ctors
	Foo()
	{}

	Foo(bool a, char b, int c, int d)
		:_bits(a,b,c,d)
	{}

	// Accessors
	bool getA() const { return _bits.getA();}
	char getB() const { return _bits.getB();}
	int getC() const { return _bits.getC();}
	int getD() const { return _bits.getD();}

	// Updaters
	void setA(bool a) {_bits.setA(a);}
	void setB(char b) {_bits.setB(b);}
	void setC(int c) {_bits.setC(c);}
```

```
        void setD(int d) {_bits.setD(d);}

private:

        // *** Implementation Section ***

        // Private nested bitfield that is only visible within
        // the Foo class
        //
        class Bits
        {
        public:

                Bits()
                        :_a(0), _b(0), _c(0), _d(0)
                {}

                Bits(bool a, char b, int c, int d)
                        :_a((a==true) ? 1 : 0), _b(b), _c(c), _d(d)
                {}

                // *** Accessors
                bool getA() const { return ((_a==1) ? true : false);}
                char getB() const { return (char) _b;}
                int getC() const { return ((int) _c);}
                int getD() const { return ((int) _d);}

                // *** Updaters
                void setA(bool a){
                        _a = ((a==true) ? 1 : 0);
                }

                void setB(char b){
                        assert(b >= 0 && b < 8);
                        _b = b;
                }

                void setC(int c){
                        assert(c >= 0 && c < 1024);
                        _c = c;
                }

                void setD(int d){
                        assert(d >= 0 && d < 131072);
                        _d = d;
                }

        private:

                // The implementations implementation!!!
                unsigned _a : 1;    // 0 - 1
                unsigned _b : 3;    // 0 - 7
                unsigned _c : 10;   // 0 - 1023
                unsigned _d : 17;   // 0 - 131071

        };

        // A data-member of Foo.
        Bits _bits;
};
```

The first thing to note is the interface to the new implementation of `Foo` is identical to the original; the accessor methods have the same signatures and return the same types as before, and the updaters all take the same argument types. Their implementation now delegates to an instance of a nested bit-field called `_bits`. It is here that data type demotion and promotion occurs, and the ranges of the various data are checked

with `assert()`. Running `sizeof(Foo)` on the new implementation of `Foo` returns a size of 4 bytes – one third of the size of the original.

Example Uses

The UK mapping agency Ordnance Survey has a ObjectStore database in the OS MasterMap system containing billions of map coordinates and point objects, and Matra data vision, a French engineering firm, implemented its own international character sets.

Related Patterns

String Table

Small Object Pool Allocator

Intent

Reduce the runtime cost of small object construction by amortizing this cost across a single array construction of many objects, and then manage these as they are requested. A pool allocator can increase database write performance by orders of magnitude.

Motivation

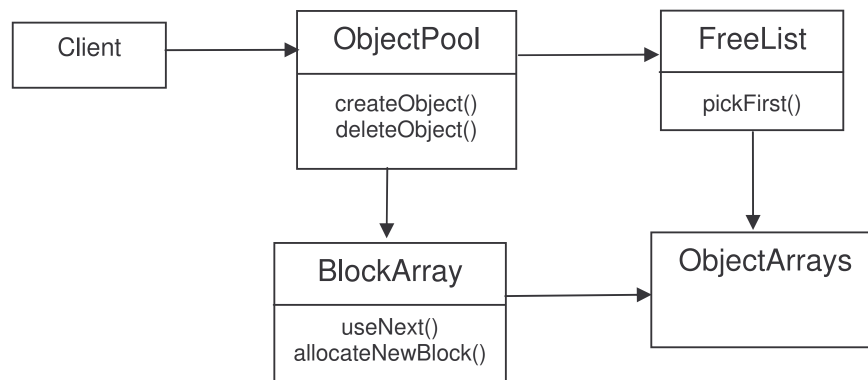
Imagine a system which collects alarm events off a telephone network or trade events on a stock trading system, and we need to collect these events into a list ordered by their time-of-arrival. If we attempt to allocate each individually within their own transaction, the throughput of the system will be relatively modest. The first performance improvement would be to batch these events into single larger transactions, and this would massively improve the achievable write rate. However, as the batch size is increased and the transactions get longer, the limiting factor becomes the time it takes to run individual object constructors. Utilizing a pool allocator at this point will further increase throughput in excess of tens of thousands of events per second [22] up to the limits of disc write performance.

Applicability

Use a pool allocator for persistent objects whenever

- There are many small objects that need to be created and written quickly to disc.

Structure



[22] The number of events per second achievable is a function of the size of the events and the disc speed; bigger events take longer to write.

Participants

- **ObjectPool**
  - Provides an interface for clients to construct and delete objects from the pool
  - Can provide a global point of access to the pool, or there can be different object pools managing the same object type in different persistence contexts.
  - Usually templated so each pool only manages objects of a single type. This provides compile-time type safety.

- **FreeList**
  - Manages object deletions within the pool usually by maintaining a list of references to free 'slots' within the object arrays
  - A free list is not always required for example if objects within the pool are never deleted.

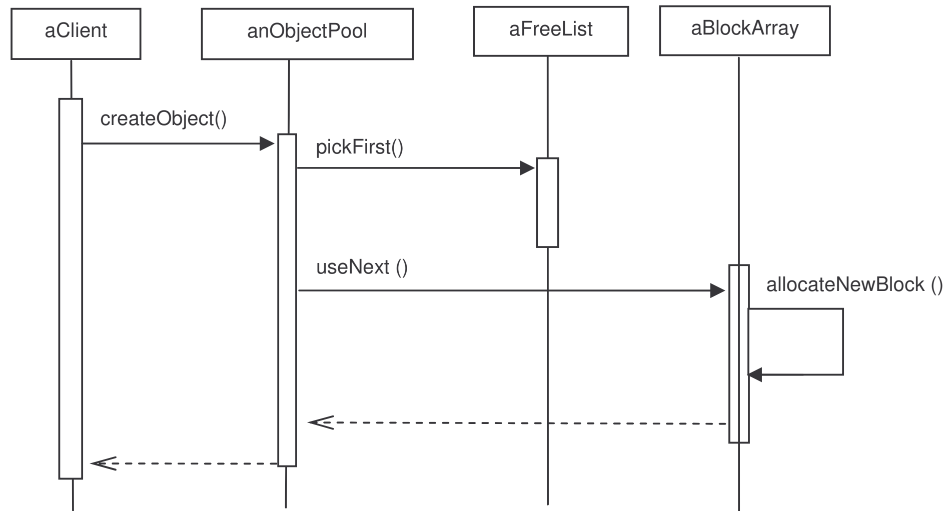
- **BlockArray**
  - Manages and tracks the allocation of new object arrays (each called a Block). This collection typically grows monotonically and never shrinks so is usually implemented as an array.

- **ObjectArrays**
  - The actual arrays of small objects.

- **Client**
  - Replaces all calls to `new` and `delete` with calls using the ObjectPool interface `createObject()` and `deleteObject()`. Must call the `init()` method on the small object returned from the pool.

Collaborations

The ObjectPool provides the interface for clients to request new objects, but it delegates the work of re-cycling empty slots to the FreeList, and the construction and management of the pool to the BlockArray. The following interaction diagram illustrates the sequence for a request when the free list is empty and the current pool has no space.



Consequences

Using a pool allocator in the right context will significantly increase the speed of persistent small object allocation.

1. *FreeList not always required*. If the small objects are never deleted the pool implementation can be simplified by omitting the free list structure and all associated code. As a result the persistent storage used will be slightly smaller and the code will be faster.

2. *Compromises locality of reference*. Objects that reference small objects in the pool, where before the pool was used the small object could be allocated near the referencing object, now the reference has to be on to a remote area of the database in the pool. This can negatively impact read performance.

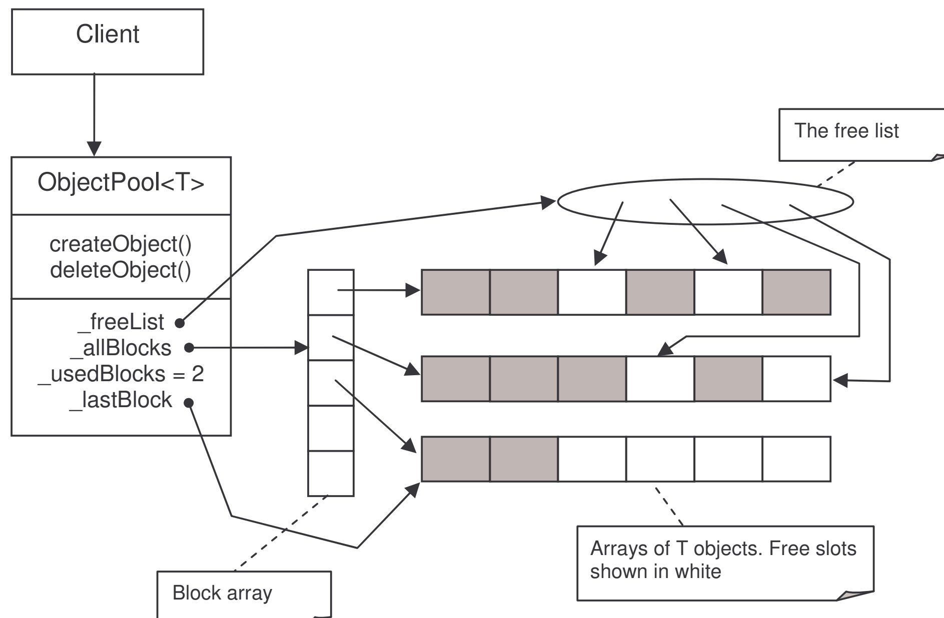
3. *Small objects must have a no argument constructor*. In C++ array allocations of objects requires a no-arg ctor, and the same is true in other languages. Clients must initialize the small object returned using a pseudo-ctor function which is typically called something like `init(),` whose arguments will be identical to one of the small object constructors.

4. *Cost to read performance*. The pool allocator improves write performance but this can sometimes introduce an associated cost for read performance because locality of reference is compromised. When a critical read use-case accesses the small objects in the pool these objects can be widely separated on disc because it is not easy to control their location in the pool with respect to how they are used; some may be allocated from the free list, others from the current block.

   It is possible to preserve locality of reference by more complex management schemes which copy related ranges of small objects whenever their number grows to ensure they remain allocated closely together. Examples of this are not covered here.

Sample Code

Below we outline an example of a type-safe small-object pool written using C++ templates suitable for use with the ObjectStore database.



The ObjectPool class is templated on the type of the small objects in the pool and provides an interface for the client to create and delete objects within the pool

```
template <class T >
class ObjectPool
{
public:
        ObjectPool(int blockSize = NUM_FIT_IN_ADDRESS_CHUNK(T));
        ~ObjectPool();
        // Returns either recycled slot in pool or
        // one from the end of the array
        static T* createObject(){ return _instance->getObject(); }

        // Frees up the passed object for reuse.
        static void deleteObject(T*& p) { _instance->freeObject(p); }
};
```

Now we examine a constructor for this simple object pool. It takes care to allocate the array of blocks-of-T and the `freeList` in their own clusters within the same segment as the `ObjectPool` instance. [23] It also initializes the first block-of-T's ready for use.

The macro `NUM_FIT_IN_ADDRESS_CHUNK(T)` is used to provide a sensible default size for the blocks. This is held in the data member `_blockSize` which is used when new blocks-of-T's are allocated as the pool needs to expand. The assumption here is that the T instances are significantly smaller than an address space chunk (64k). This is reasonable since pool allocators are only usually used on very

[23] *Segments* and *clusters* are physical entities within the ObjectStore database into which a programmer can locate C++ objects when they are constructed.

small objects. Just in case, there is an `assert` which catches horrible errors in debug versions.

The code for the macro is something like:

```
#define NUM_FIT_IN_ADDRESS_CHUNK(X) (((64*1024)/sizeof(X))-2)
```

There are also references to the `ts<>()` template function. This function simply efficiently returns an `os_typespec` object for the type passed in. [24] It does this with a consistent syntax so that templates can be utilized with arrays, other templates, classes and native types.

```
ObjectPool(int blockSize = NUM_FIT_IN_ADDRESS_CHUNK(T))
	:_blockSize(blockSize), _usedSlots(0), _lastBlock(0),
	_allBlocks(0), _freeList(0)
{
	// ObjectPool needs positive block size.
	assert(_blockSize > 0);

	// Get the cluster where we are constructed
	os_cluster* clr = os_cluster::of(this);
	os_segment* seg = clr->segment_of();

	// Create the free-list in a separate cluster
	os_cluster* freeClr = seg->create_cluster();
	_freeList = new(freeClr, ts< os_List<T*> >()) os_List<T*>();

	//...and the array of all blocks
	os_cluster* arrClr = seg->create_cluster();
	_allBlocks = new(arrClr, ts< os_Array<T*> >()) os_Array<T*>();

	// T object must have a no arg ctor so it can be array allocated.
	_lastBlock = new(clr, ts< T >(), _blockSize) T[_blockSize];
	_allBlocks->insert(_lastBlock);

}
```

Here we show the destructor which uses the collection of all the blocks to correctly clean up after itself, even though this code is unlikely to ever execute. It is unusual for object pools to be entirely deleted.

```
~ObjectPool()
{
	// Spin through and destroy all allocated blocks
	os_Cursor<T*> cur(*_allBlocks);
	for(T* block=cur.first(); cur.more(); block=cur.next())
	{
		delete [] block;
		block = 0;
	}

	// Delete the two ObjectStore collections
	delete _freeList;
	_freeList = 0;
	delete _allBlocks;
	_allBlocks = 0;
}
```

[24] ObjectStore uses the class `os_typespec` to indicate a persistent object's class. The reader is referred to the ObjectStore documentation for further details.

If the object pool is the only object in the current segment, then this deletion could be optimized by simply destroying the entire segment in one go. So now we show how the pool 'creates' a new object.

```
T* getObject()
{
	// Try  the free list first
	T* ret=_free->pick();

	if(ret){
		free->remove(ret);
	}
	else if(_usedSlots < _blockSize){
		// Try the newest block
		ret = (_lastBlock + _usedSlots);
		usedSlots += 1;
	}
	else {
		// Allocate a new block in our own cluster
		os_cluster* clr = os_cluster::of(this);
		lastBlock = new(clr, ts<T>(), _blockSize) T[_blockSize];
		allBlocks->insert(_lastBlock);
		ret = _lastBlock;
		usedSlots = 1;
	}
	return ret;
}
```

This method either finds or creates an object as appropriate. First it tries the free list. If this fails it then tries the newest block, otherwise it allocates a whole new block and returns the first slot from that. This means that most allocations will involve:

- A `pick()` on an `os_List` – a very fast call. [25]

Then either:

- Remove on the `os_List` – this is more efficient than on `os_Array` which explains the choice here. [26] A few pointer assignments.

Or:

- A test for a null pointer
- An integer inequality comparison
- A pointer value assignment
- An integer increment

All these operations are incredibly efficient and would compile to very few instructions in most compilers. Assuming there are no deletes, then the only time `T` object constructors are run is every '`blockSize`' requests. If objects are deleted from the pool, then performance is superior to that, in proportion to the number of deletes.

An object is 'deleted' simply by being placed in the free list. The caller is forced to pass a reference to a pointer so we can set the callers pointer to zero so they get horrible errors if they subsequently try to use it. Extra error checking can be inserted into this method to ensure that `p` refers to a persistent object or even that it points into

[25] `os_List` is a list class that forms part of the ObjectStore collections library.
[26] `os_Array` is an array class that is also part of the ObjectStore collections library.

one of the blocks. However, these checks would of course compromise runtime performance so would need to be compiled into the debug image only.

```
void freeObject(T*& p)
{
       _free->insert(p);
       p = 0;
}
```

Example Uses

Many ObjectStore customers use pool allocators of some description for very rapid updates. Examples include ObjectStore based systems in major banks collecting trade data from the market, and in the Telco sector collecting events off networks.

Related Patterns

Persistent Queue

Transaction Memento

Intent

Save heap and/or stack allocated object state to one side prior to the start of a transaction in the event of a transaction retry. The objects can then be restored to their original state cleanly and the transaction retry will start from the same state as the original.

Also Known As

Rollback Memento

Motivation

This is a variant of the standard memento pattern a la GoF. If for some reason there is an error in a database transaction and the transaction re-executes, any changes in persistent object state are rolled-back by the database engine, but heap or stack allocated objects are the programmer's responsibility. So if prior to the start of a transaction, or up to the point that a nested transaction is started, heap or stack allocated state is accumulated, the programmer must ensure that in the event of transaction retry, this state remains correct.

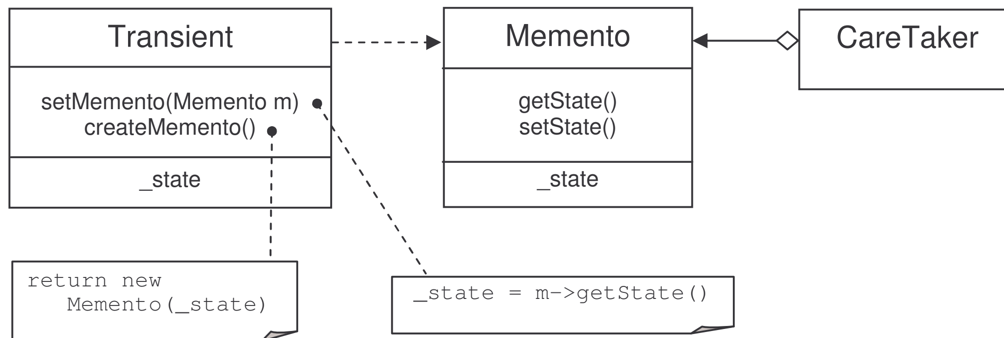
There are several possible general approaches here; you could attempt to undo all the accumulated state from the previous transaction execution, you can contrive to start the transaction from a simpler state which is easily specified at the start of the transaction or you can record the internal state of the objects involved. In the last case you must save state information somewhere so that you can restore objects to their previous states. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) a transiently allocated object's state must be saved prior to a the start of a (nested) transaction so that it can be restored to that state later, *and*
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure

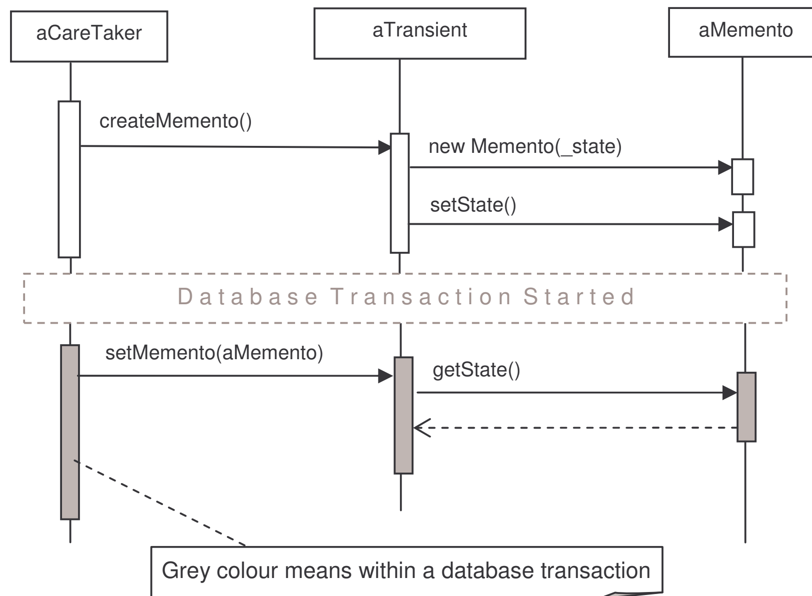


Participants

- **Memento**
  - Stores internal state of the `Transient` object. The memento may store as much or as little of the transient object's internal state as necessary at the programmer's discretion.
  - Protects against access by objects other than the `Transient` object. Mementos have effectively two interfaces. `Caretaker` sees a *narrow* interface to the `Memento`—it can only pass the memento to other objects. `Transient`, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the transient instance that produced the memento would be permitted to access the memento's internal state.
- **Transient**
  - Creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.
- **Caretaker**
  - Is responsible for the memento's safekeeping.
  - Never operates on or examines the contents of a memento.

Collaborations

Here a caretaker requests a memento from a `Transient` object, holds it for a time, and passes it back to the originator when the transaction starts, as the following interaction diagram illustrates:



The new memento object is created and retained to one side by the care taker object before the transaction starts, and `setMemento()` is called as the first call within the transaction, thus if the transaction retries, the first thing that happens is the transient objects state is restored to its original state.

Consequences

The Transaction Memento pattern has several consequences:

1. *Preserving encapsulation boundaries.* Transaction Memento avoids exposing information that only the transient object should manage but that must be stored nevertheless outside the transient object in order that the transaction rollback can start from the correct state. The pattern shields other objects from potentially complex transient object internals, thereby preserving encapsulation boundaries.

2. *It simplifies Transient.* In other encapsulation-preserving designs, the transient object keeps the versions of internal state that clients have requested. That puts the entire state management burden on the transient object. Having clients manage the state they ask for simplifies the transient class and keeps clients from having to notify transient objects when they're done.

3. *Using transaction mementos might be expensive.* Transaction Mementos might incur considerable overhead if the transient object must copy large amounts of information to store in the memento, or if clients create and return mementos

in large numbers. [27] Unless encapsulating and restoring the transient object's state is cheap, the pattern might not be appropriate because even on the first run through the transaction, the memento is accessed 'unnecessarily' and is essentially a NOP [28] to set the transient object's state to its current state.

4. *Defining narrow and wide interfaces.* It may be difficult in some languages to ensure that only the transient object can access the transaction memento's state.

5. *Hidden costs in caring for transaction mementos.* A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento. Hence an otherwise lightweight caretaker might incur large heap storage costs when it stores mementos. One idea is to have the caretaker object use the Guard idiom [29] to ensure that all its mementos are released when the transaction ends, and the possibility of a retry is zero.

Example Uses

Many ObjectStore customers use this memento pattern to handle heap allocated object during transaction rollback.

Related Patterns

Memento

[27] If there are a huge number of mementos this implies an equally large number of transaction retries which would probably point to some other problem in the system!
[28] NOP is an assembly language instruction that effectively does nothing.
[29] This is related to the idea of Resource Acquisition is Initialization (RAII) See: http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Scope_Guard

String Table

Intent

Store the character representation of repeated strings once in a persistent associative container and then reference this character representation from persistent string objects using either integers or pointers. Depending on the properties of the data set being stored this can produce significant space savings within the database.

Motivation

Strings often form a huge proportion of the useful business level information within any persistent data set; therefore it makes sense to attempt to compress this information as effectively as possible to make efficiency gains at runtime. The best way to compress this data, to achieve a significant reduction in data set size without incurring unacceptable runtime overheads strongly depends on the particular data set, the prevalence of repeated character sequences within the data set, how and when these occur, the life-cycle of the data concerned, and the operations that the string data must undergo and which of these operations have critical time and performance constraints.

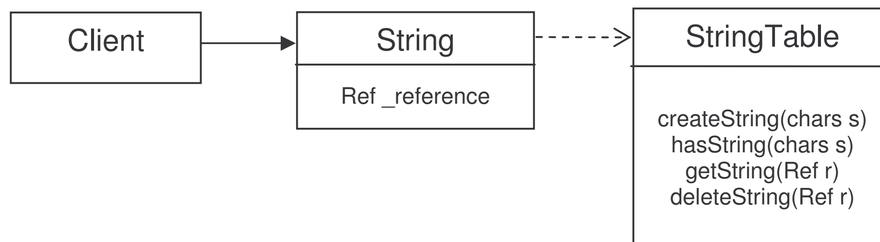
Applicability

Use a String Table whenever:

- A data set contains large numbers of repeated strings greater than four characters long

- Particularly if these strings do not get deleted, or do not need to be deleted, because this can simplify the implementation considerably

Structure



Participants

- **Client**

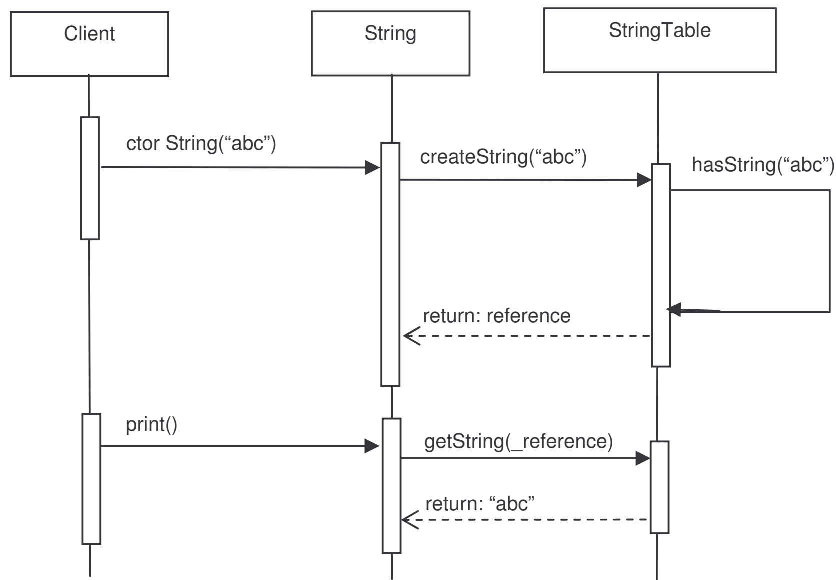
    - The client allocates and uses persistent strings exactly as they would any other type of string. Whether the string is persistent or transient is

immaterial; any persistent representation is opaque from their perspective.

- **String**
  - Holds a reference to the character representation of its contents within the String table data structure.
  - Can access its character representation from within this data structure whenever it is required to do so.
- **StringTable**
  - Holds associative data structures which afford efficient two-way access between a string objects' characters and its reference.
  - Provides an interface which supports the creation of new strings, finding existing strings both by reference and as a string of characters and optionally string deletion.

Collaborations

Here we show a client constructing a new `String` object containing the characters `"abc"` , and the `StringTable` checking for the pre-existence of this string and then finally returning a reference for the String object to use.



The client then invokes a `print()` method on the `String` object which then uses its internal _reference to get the correct character representation from the `StringTable` object.

Consequences

Using a `StringTable` has several consequences

- *Reduced dataset size*. Data compression is achieved by storing the character contents of repeated strings once and then referencing them with an integer or pointer. The degree of compression is a function of the number of repeats and the length of the strings compared to the size of the reference. On 32-bit platforms the critical point is 4 characters, but with 64-bits this goes up to 8.

- *String table representation can be optimized*. The best data structure for the `StringTable` can be tailored to suit particular data sets. Strings typically form a significant part of most databases, and their relative frequency, their average length, their character frequency profiles, and the algorithms used to manage and manipulate them all affect how best to represent them in the string table.

- *The long string problem*. The more characters there are in a string, the less likely that sequence is to be repeated in the data set, but the better the compression ratio is if the string is repeated. In some data sets it will make sense to break strings into common words or phrases and hold multiple references to each, encoded in a set of bits. Prior analysis of these characteristics in your particular data set can help to decide on the best implementation.

- *Cost of mutable strings*. With the characters stored in the `StringTable` if the persistent strings are highly volatile, it can become expensive to continuously change them, because each time the existing character sequence must be copied onto the heap, modified, and then set as the value of the string; the costs being running the constructor and updating random persistent pages within the string table structure.

- *Efficient string comparison*. Normal string comparison operations are reduced to a single integer/pointer comparison which is significantly faster than comparing an array of characters one at a time.

- *Efficient global string changes*. Changes to the character representation within the `StringTable` will in effect change every `String` object that references it. This can be exploited in certain contexts to provide very efficient updates.

- *Real deletes are difficult*. It is difficult to really delete entries in the `StringTable` because of the cost of finding all the `String` objects that reference them. There are several approaches to this problem:

    1. Have a separate 'purge string table' process which runs at specific times and visits the entire data set and deletes only those entries that have no references

    2. Use reference counting, but this is complex, error prone and increases the size of the data set

3. Do nothing, and simply allow the `StringTable` to grow without limit. This is surprisingly common and very effective particularly where the proportion of repeated strings is high.

4. In some contexts it makes sense to treat certain accesses to non-existent strings in a special way, such that the 'delete' slowly propagates throughout the data set. Having been 'deleted' from the `StringTable`, as each `String` object requests this deleted entry it sets itself into the deleted state i.e. null.

- *Use multiple string tables*. Where there are large disjoint data subsets, it is sometimes more efficient to have separate string tables for each subset, maybe with different implementations optimized for each.

- *Compromised locality of reference*. Every string object is likely to be held in database sections remote from their character contents. This can affect the efficiency of persistent page fetch and update.

- *StringTable can become less effective with time*. As the data set changes through time, the proportion of repeated strings, or the occurrence of long strings can change, and if this change is to such a degree that it contradicts the assumptions underlying the original `StringTable` implementation then the system can become less efficient.

Example Uses

Most ObjectStore customers where repeated strings form a significant proportion of the total size of the data set use this pattern, for example Areva who provide software to the nuclear power industry and Ordnance Survey, a UK based mapping agency in their OS MasterMap software.

Related Patterns

Compress Persistent Data

Persistent Mutex

Intent

Provide a clean way to acquire an early write lock on a section of the database and avoid those deadlocks that are caused by out of order prior reads. Separate database clients avoid deadlock by competing for global process-wide persistent mutexes.

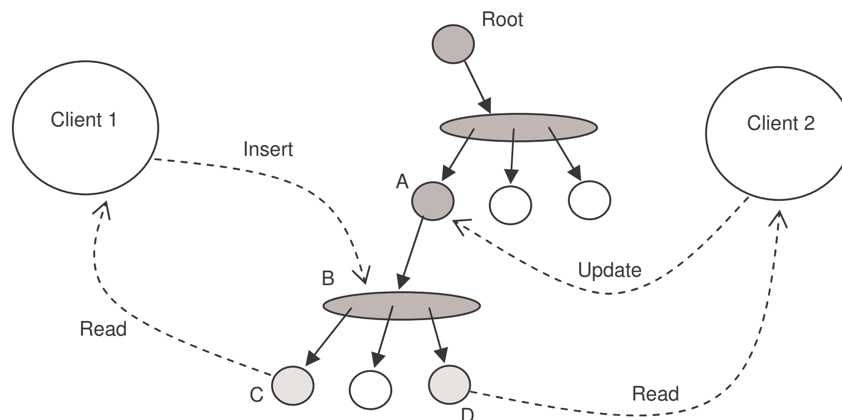
Also Known As

Write Locker

Motivation

Imagine a persistent tree-structure which has several client programs each navigating from one node to another searching for a particular object or set of objects with a view to updating them, deleting them, or to insert new nodes into the tree. Such a tree structure is very common in persistent databases, and use-cases involving such un-coordinated multiple updaters are also common, and these almost invariably lead to database deadlocks.

The problem is that as the clients navigate through the tree-nodes they progressively acquire read locks before acquiring write locks on the particular nodes of interest, and the order of these lock acquisitions can be different for each client. This means that it is very easy for a cycle of dependencies to appear in the locking order as shown in the example below.



Here instances are represented as circles, and collections are shown as ovals containing pointers (arrows) to the objects they contain. Client 1 follows pointers from the root until it finds object C which contains information requiring client 1 to insert an object into the collection B. Meanwhile client 2 has navigated from the root to find object D which requires an update to the state of object A visited earlier.

Both clients acquire read locks as they navigate the tree from the root object; the common read locks are indicated by the shaded icons. Client 1 also has a read lock on object C, while client 2 has an additional read lock on object D. These read locks can be acquired simultaneously by both clients because two simultaneous reads of the same object does not compromise transactional integrity.

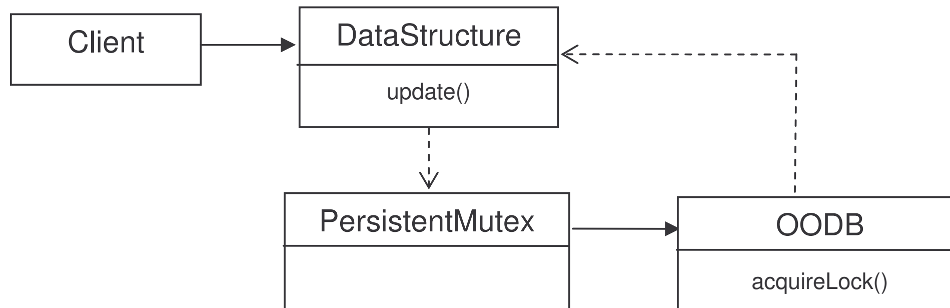
When Client 1 attempts to insert into collection B this will require a write lock, which being incompatible with a read lock requires that client 1 wait until the read lock on this collection held by client 2 is released. Client 2 cannot release this lock until its transaction has completed, but before this time client 2 attempts to acquire a write lock on object A, an object previously read locked by client 1. This attempt is the final step in the cycle of lock dependencies; now client 1 is waiting for client 2 to release its read lock on collection B, and client 2 is waiting for client 1 to release its read lock on object A. This deadlock is typically detected by the OODB server or runtime and resolved by aborting the transaction in one of these clients.

Applicability

Use the Persistent Mutex when:

- There are multiple updaters on a single data structure that are experiencing a high number of writer-writer deadlocks.

Structure



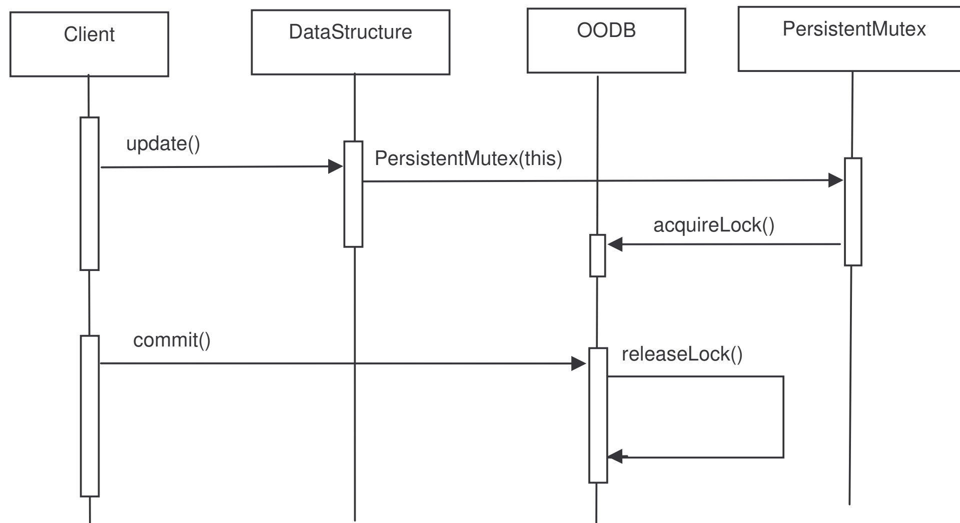
Participants

- **Client**
  - The client(s) uses the persistent data structure within an update transaction exactly as normal.
- **DataStructure**
  - The data structure provides an interface for clients to read and update its contents according to normal OO practice
  - Its methods lock the persistent mutex at appropriate points in their processing to ensure proper serialization on to the data structure.
- **PersistentMutex**
  - Locks the data structure using the OODB API. This is typically done from within its constructor, and is an instance of the guard idiom.

- **OODB**
    - Write locks an object or page in the database such that other clients are prohibited from reading or writing the protected data structure.
    - Ensures that the lock is released correctly should the transaction commit or abort or if any other error occurs.

Collaborations

Here we see a client making an update call on the `DataStructure` object. Within this method a *transient* `PersistentMutex` is created on the stack. From within its constructor the `PersistentMutex` object acquires a write lock on the `DataStructure` object using the OODB API. Later the transaction is committed by the client and the database releases the write lock on the `DataStructure` instance.



Consequences

The `PersistentMutex` pattern has several consequences:

- *It is a voluntary locking scheme*. If clients always access data structures in the same order, deadlock cannot occur. The `PersistentMutex` is an extreme case of 'always access in the same order' and by using the `PersistentMutex` pattern the actual lock acquisition is instigated by the methods on the `DataStructure` object; this helps to ensure that all clients adhere to the protocol.

- *Convert deadlocks into waits*. In general handling waits is much cheaper than handling deadlocks, because if a deadlock occurs, transactions get retried, and all the work of the transaction has to be re-executed. This can be very expensive.

- *Guaranteed to eliminate 'prior-read' deadlocks*. Acquiring the write locks early without a prior read lock on the target object guarantees the elimination of all deadlocks between clients (or threads) that follow the protocol. However, there is still a potential for deadlocks in scenarios where there are multiple data structures involved, each protected by their own persistent mutex, and clients lock these mutexes in different orders.

- *Can over-serialize access*. This can be a major draw-back because if a write lock is taken on the root of a tree-structure, this effectively stops any other clients accessing the tree, even for read. Some OODBs provide special read-only modes that do not block other readers or writers, and this can in certain cases be used to mitigate this problem. Even so, over-serialization is sometimes far less costly overall than handling very frequent deadlocks.

Sample Code

Here we discuss code for a PersistentMutex written in C++ that is suitable for use with the ObjectStore database.

```
class PersistentMutex
{
public:

   PersistentMutex()
   :_exp(0)
   {}

   PersistentMutex(void* p)
   {
       if(objectstore::is_persistent(p))
       {
          // Wait forever to get a write lock
          _exp = objectstore::aquire_lock(p, os_write_lock, -1);
       }
   }

   ~PersistentMutex()
   {
     delete _exp;
     _exp=0;
   }

private:

   os_lock_timeout_exception* _exp;
};
```

Here we see a simple persistent mutex with a constructor that takes a `void*` pointer, which can point to any C++ object. Within the constructor it confirms that the pointer targets a persistent object, and then uses the ObjectStore API to acquire a write lock on the page containing that object. In the case of ObjectStore this call returns a transient object allocated on the heap, which we ensure gets cleaned up in the persistent mutexes' destructor.

```
class DataStructure
{
public:
```

```
   void addElement(const Foo& f)
   {
      // Wait indefinitely for a write lock
      // on the page containing 'this'
      PersistentMutex(this);
      _allFoos.push_back(f);
   }

private:

   typedef list<Foo, OTLAlloc<Foo> > FooObjList;
   typedef FooObjList::iterator FooObjListItr;
   FooObjList _allFoos;
   FooObjListItr _itr;
};
```

Above we show a simple persistent `DataStructure` class which uses a persistent STL list internally. The `addElement` method allocates a `PersistentMutex` on the stack before attempting to insert an element into the STL list; if every other updater method in the `DataStructure` class also allocates a `PersistentMutex` that locks `'this'` in a similar fashion then different clients will be totally serialized and no deadlocks will occur.[30]

Example Uses

Versions of this pattern are ubiquitous throughout the ObjectStore customer base.

Related Patterns

None.

[30] Given the caveats mentioned in the 'consequences' section above.

Evolver

Intent

Hide the implementation details of a class from callers to aid rapid *in-situ* schema evolution, without compromising compile-time type-safety or runtime performance.

Motivation

The Virtual Memory Mapping Architecture (VMMA) used by ObjectStore has many advantages, such as the ability to store real C++ arrays and pointers, which result in significant performance improvements on other approaches to storage. ObjectStore stores the memory pages containing the bytes comprising the C++ objects as laid out by the compiler.

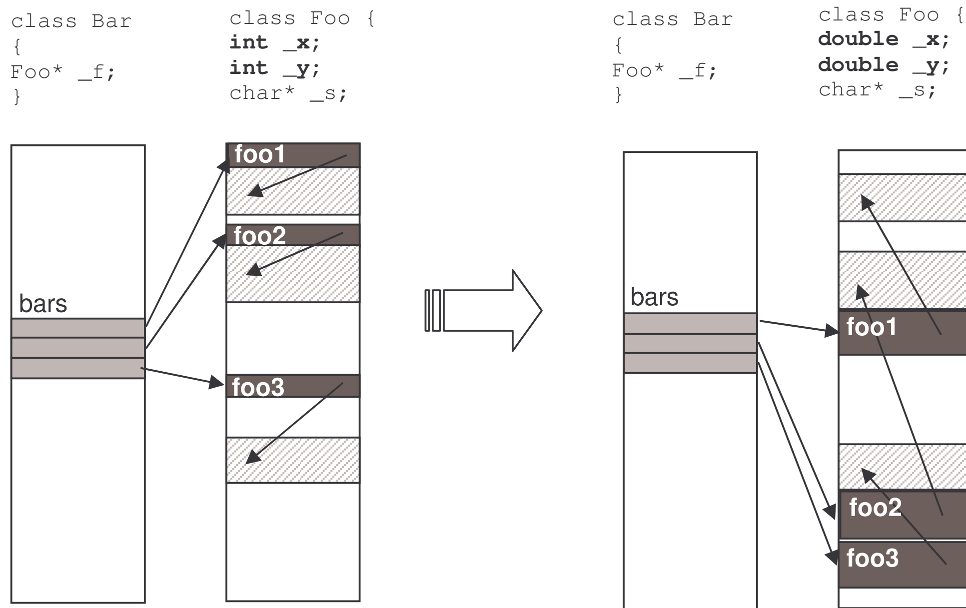
The C++ language has evolved over time to solve many real-world programming problems which have constrained which features are included within the language. In C++ every object has an explicit class, the layout of which is determined at compile time. At runtime the layout of classes cannot change. The only way to change the layout of a class is to stop the program, edit the class definition, recompile, and re-run the program. In C++ any instances of a class that are encountered at runtime will have been constructed either on the heap or stack with the `new` operator. Thus, C++ guarantees all runtime objects will be laid out in memory consistent with the compile-time definitions. All this assumes that objects in a C++ program *do not persist* and are not subsequently re-accessed after the process exits.

ObjectStore clients break this assumption. For example, persistent objects can be encountered by programs that never run the appropriate constructors and the source code of classes can be edited such that the layout of the classes used by the program diverge from those stored on disc. This means that in theory it would be possible for an ObjectStore client to access objects with an unknown memory layout. ObjectStore guards against that possibility by using *schema validation* to guarantee that the layout of classes within the database are identical to those expected by the program at compile time. This protects the database from unintended corruption.

However, as programming projects progress and enter the maintenance phases changes to class memory layout are inevitable; addition of new data members, changes to data member types, renaming of data members etc. To access the current data set requires that the stored instances are “re-laid out” so that their byte configuration corresponds to the new class definition. This process is called *schema evolution.* Schema evolution within the context of the VMMA results in several issues that need resolution.

Pointer Fix-up Issue

The most common change is the addition of data members. C++ compilers will use particular rules about how to layout objects in memory. If you add a data member to a class the size of class instances will increase, take one away and the size will decrease. In the general case, changing the definition of a class will change its size.



Object Movement and Pointer Fix-up

If a persistent object increases in size and is bounded by neighbouring objects allocated nearby there will not be enough byte space to contain the new layout. Therefore the object must be relocated within the database. Any objects that reference these with a pointer (or C++ reference) must have their pointers re-targeted. This is illustrated above. The black `Foo` objects have increased in size because the `_x` and `_y` data members have changed to `double`, so they need to move within the database. The `Foo` pointers within the grey `Bar` objects must be found and re-targeted but the strings (shown as grey hatched rectangles) referenced within the `Foo` objects themselves can remain constant.

It is important to note that because these `Foo` instances could in principle be targeted by a pointer from anywhere within the database, even a `void*`, any general purpose schema evolution algorithm must check every pointer on every page to ensure that they are all re-targeted appropriately. Needless to say, this is computationally expensive.

Class Name Issue

The program that runs the schema evolution must access the old objects so it can use the values of their data members to initialize the new objects. This requires that we have a C++ program with two different layouts for the same class simultaneously. This is not possible in C++.

Applicability

Use the evolver pattern when

- Part of a persistent schema is predicted to change regularly and existing deployed databases will need to be schema evolved

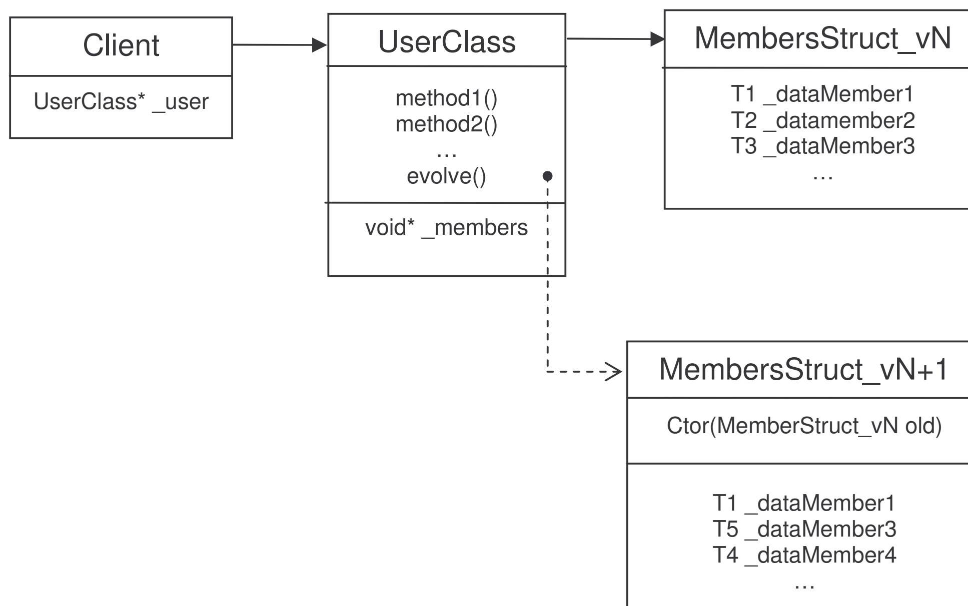
- It is impractical to recreate existing deployed databases for whatever reason i.e. they are too big/numerous or they are the database of record.

- There is an opportunity to compile and release a new version of the code to accommodate class changes

- Maximum runtime performance and compile-time type-safety are paramount

- The schema changes are relatively simple, for example the entire inheritance structure or virtualization of classes does not change.

Structure



Participants

- **UserClass**

  - Has a single data member which is an un-typed references (i.e. a `void*` pointer) to a data record (i.e. a c-style `struct`) containing all the other data members.

  - Methods are declared in the interface in the normal way, but they are implemented to use the data held in the struct by first casting the un-typed reference to the correct type, and then accessing the public data members as its own.

- Defines an `evolve()` method which can be used to change the content of existing instances to the new one by constructing an instance of `MemberStruct_vN+1` from the existing `MemberStruct_vN`

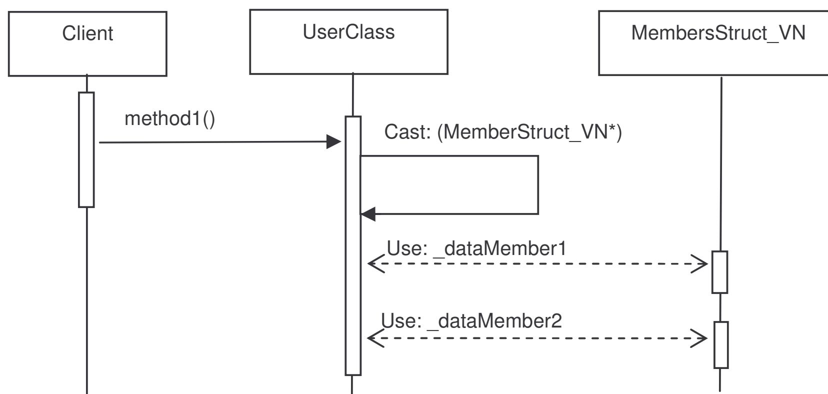
- **MembersStruct_vN**
  - Simple plain-old-data (POD) class which only contains public data members
  - Named with a version number `_vN` so the schema version is clear from examining the persistent data.

- **MembersStruct_vN+1**
  - Simple plain-old-data (POD) class which contains the data members for the new version of this class.
  - Defines a constructor that takes an instance of `MemberStruct_vN` as (one of) its input argument(s). It can then set its own state based on the existing state very flexibly.

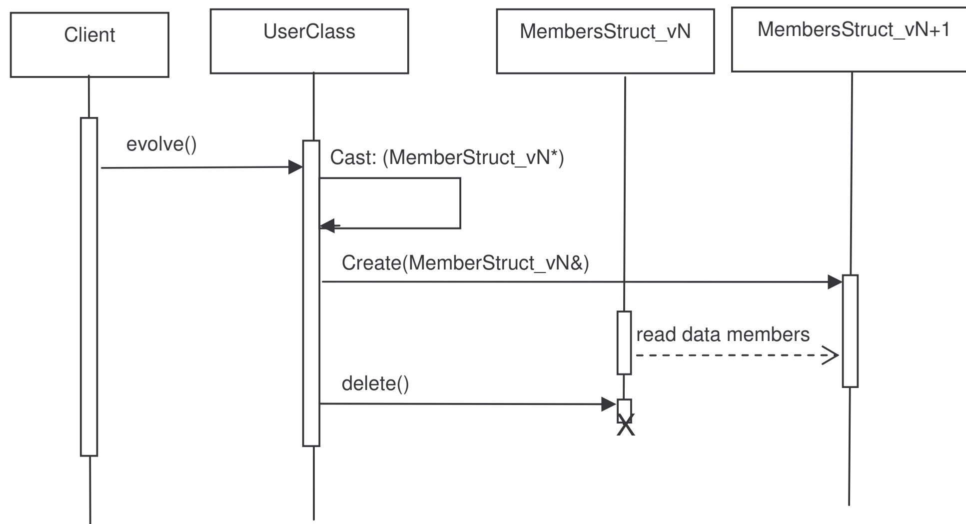
- **Client**
  - Uses a correctly typed reference to access instances of the `UserClass` exactly as normal.

Collaborations

First we see the `Client` class calling a method on the `UserClass`. This shows the sequence of events during normal usage at runtime. Then we see a second scenario which shows the behaviour during schema evolution.



Here the client calls `method1()` on an instance of the `UserClass`. In this method `_members` is cast to a `MemberStruct_VN*` and then the public data members within the struct, which hold the state of this instance, can be accessed directly.[31]



Now we see the pattern of interactions involved in the schema evolution of a single instance. The `evolve()` method is called on the `UserClass` instance and it starts by casting its un-typed reference to a `MembersStruct_vN*`. It then constructs a `MemberStruct_vN+1` instance passing in a reference to the previous version. The `MemberStruct_vN+1` constructor then reads the existing state from the old version to set its own data members as appropriate. After the constructor has completed, the `evolve()` method simply deletes the old `MemberStruct_vN` instance.

Consequences

The following consequences hold for the evolver pattern.

1. *Evolution can run in-situ*. It is not necessary to dump and reload an entire database, or render its contents in ASCII, or write complex code to fix-up pointers. Using the evolver means that evolution can be run directly against the database; in some contexts it can even be run with the database online!

2. *Evolution can run lazily*. Variants of this pattern postpone evolving all instances until they are accessed by some other use-case.

3. *Compile-time type safety is retained*. Unlike some other approaches to 'evolvable schema', such as those based around Frames and Slots, [32] this pattern allows clients of the `UserClass` to work with strongly typed references within the language type system. All potentially unsafe casts are encapsulated within the `UserClass` code itself.

---

[31] In this diagram we have taken liberties with the syntax of the normal UML Interaction Diagrams by showing type casts and direct data member access.
[32] See: http://en.wikipedia.org/wiki/Frames_(artificial_ingellience) and http://en.wikipedia.org/wiki/CLOS

4. *Evolution is fast*. The evolution process does not need to go via an intermediate representation; it simply creates the new versions directly from the old. It is only necessary to search the entire database for the instances of `UserClass` that need to be evolved if there is no extent thereof.

5. *Low runtime impact*. Introducing the extra pointer and 'externalizing' the data members of the `UserClass` has a very low impact at runtime, i.e. one type-cast and an extra pointer de-reference each time a data member is accessed. However, after evolution the `UserClass` and the `MemberStruct` are not necessary collocated within the database, and this can introduce additional page fetch overhead. [33]

6. *Low memory impact*. The space impact of adding 4 bytes for the pointer will depend on the overall size of the object. It may become significant for very small numerous objects whose size is around the size of a pointer.

7. *Based on the Pimpl Idiom*. This pattern is closely related to the pimpl idiom [34] so it 'inherits' all its advantages for free, i.e. it hides the implementation details of the class from the clients, so that the implementation can be changed without the need to recompile the modules using it. This is ideal in the context of schema evolution.

8. *Supports most schema changes*. This pattern can work with the most common types of schema changes including adding, removing, re-naming and re-ordering data members or changing their type. It can be used when the `UserClass` inherits from either an interface or by extension. However, it does not support schema changes to this inheritance hierarchy. It also does not work if the virtualization of the `UserClass` is changed (i.e. changes such that a virtual function pointer within the `UserClass` is added or removed.)

9. *Old code can be deleted*. It is not necessary to keep the old versions of the `MembersStruct_vN` around. These definitions can be deleted along with all the code the uses them. This helps to limit the growth of unused code.

Sample Code

Here we look at some example C++ code which shows how a user class called `Foo` which contains an `int` and two `char*` data members can be evolved such that it has a new `long` data member added, a data member name change, and a type change. This is summarized below:

[33] Such overhead has not been measured as significant in the field.
[34] As described by Herb Sutter in http://www.amazon.co.uk/Exceptional-C-Herb-Sutter/dp/0201615622/ref=sr_1_1?ie=UTF8&s=books&qid=1240492966&sr=1-1

```
class Foo
{
   private:
       int _id;
       char* _creationDate;
       char* _text;
};
```

Before

```
class Foo
{
   private:
       int _id;
       long _version;
       char* _description;
       Date _creationDate;
};
```

After

Here we have a Foo class which initially contains three data members: an integer object id, a creation date held as a C-style string and some text. This Foo class needs to be evolved such that a new 'version' data member is added and initialized to 1, the creation date changes its type into an embedded instance of a bespoke class 'Date' and must be initialized from the existing date string, and the text member is renamed to 'description' and must still contain the old text. Some of the data members are also re-ordered.

```
class Foo
{
public:

       // Ctors.
       Foo();

       // Destructors
       ~Foo();

       // Accessors
       int getId() const;
       long getNumber() const;
       const char* getDescription() const;
       const Date& getDate() const;

       // Updaters
       void setNumber(long number);
       void setDescription(const char* description);
       void setDate(const Date& date);

       // A temporary method that is used to evolve this object
       // to the new version
       void evolve();

       friend std::ostream& operator<< (std::ostream& out, const Foo& obj);

private:

       // Pointer to the data members of this class.
       // It is void* so we can point to any-little-ol-thang we want.
       void* _members;
};
```

Above we see the declaration of the class `Foo` as it appears in the `Foo.h` file. Note the interface declarations are exactly as they would be for a normal class, but the only data member is a `void*` called `_members`. The `evolve()` method is temporary because after the schema evolution has been completed, this method can be deleted from the code.

Now we examine the contents of the `Foo.cpp` file.

```
struct FooMembers_V1
{
public:

	// Init the id because it is const.
	// Leave the Foo ctor to allocate the char*'s
	FooMembers_V1()
		:_id(++_nextId), _creationDate(0), _text(0)
	{}

	// A value which NEVER changes after construction.
	const int _id;

	// Creation date held as a string of dd/mm/yyyy
	char* _creationDate;

	// Some text which describes the object
	char* _text;

	// Used to init the Id's in default ctor.
	static int _nextId;
};

int FooMembers_V1::_nextId = 0;
```

Above is the FooMembers_V1 struct which is simply a POD with three data members and a no-arg constructor. It is not visible outside the scope of the `Foo.cpp` file.

```
struct FooMembers_V2
{
public:

	// Init the id because it is const.
	// Leave the Foo ctor to allocate the char*'s and
	// initialise the date.
	FooMembers_V2()
		:_id(++_nextId), _number(0),  _description(0)
	{}


	// Ctor that initialises itself from the old version
	FooMembers_V2(FooMembers_V1& obj)
		:_id(obj._id), _number(1), _description(obj._text),
		 _creationDate(obj._creationDate)
	{}


	// A value which NEVER changes after construction.
	const int _id;

	// A new data member
	long _number;

	// Renamed text member
	char* _description;

	// Creation date no longer a string
	Date _creationDate;

	// Used to init the Id's in default ctor.
	static int _nextId;
};
```

```
int FooMembers_V2::_nextId = 0;

typedef FooMembers_V2 FooMembers;
```

Above is the new version of the `FooMembers` struct. Again it is a POD with a constructor but this time it has an additional constructor which takes an instance of the `FooMembers_V1` as an argument, and it uses its contents to initialize its own data members.[35] This constructor can be written to initialize the new object very flexibly, and can utilize any data from within the database, or any state accumulated during evolution. Note also the judicious use of the `typedef` above. This enables the rest of the code in the `Foo.cpp` file to be written in terms of '`FooMembers`' and ignore the current version, which makes changing the code easier and less error prone. Again the `typedef` and the struct are scoped to the `Foo.cpp` file.

```
void Foo::evolve()
{
	// Get a pointer of the correct type...
	FooMembers_V1* oldPtr = (FooMembers_V1*)_members;

	// ...and construct the new member data therefrom
	_members = new (os_cluster::of(this), ts<FooMembers_V2>())
		FooMembers_V2(*oldPtr);

	// Delete those members of the old struct that are no longer used
	delete [] oldPtr->_creationDate;

	// ...and then delete the old struct itself
	delete oldPtr;
}
```

Here we have the temporary `evolve()` method. As described earlier this casts the `void*` to the correct type, constructs the new `FooMembers` object passing in the old object so it can initialize itself from the old data.

Finally we show the code implementing the constructor, destructor and the other simple methods in this example.

```
// Ctors.
Foo::Foo()
:_members(new (os_cluster::of(this), ts<FooMembers_V2>()) FooMembers_V2())
{
	// Get a pointer of the correct type
	FooMembers* mPtr = (FooMembers*)_members;
	// Set to a random number;
	mPtr->_number = rand()%10;
	// Allocate space for the description
	mPtr->_description = new (os_cluster::of(this), ts<char>(), 30)
		char[30];
	sprintf(mPtr->_description, "New object number: %-10d", mPtr->_id);
	// Init creationDate to today
	mPtr->_creationDate.now();
}

// Destructors
Foo::~Foo()
{
	// Get a pointer of the correct type
	FooMembers* mPtr = (FooMembers*)_members;
```

[35] Here we must assume that the bespoke `Date` class has a constructor which takes a `char*` containing a well-formatted-date which it can parse and use appropriately.

```
        // We allocated the char arrays, so we must delete them.
        delete [] mPtr->_description;
}

// Accessors
int Foo::getId() const
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        return mPtr->_id;
}

long Foo::getNumber() const
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        return mPtr->_number;
}



const char* Foo::getDescription() const
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        return mPtr->_description;
}

const Date& Foo::getDate() const
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        return mPtr->_creationDate;
}

// Updaters
void Foo::setNumber(long number)
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        mPtr->_number = number;
}

void Foo::setDate(const Date& date)
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        // Note: now does date validation here
        mPtr->_creationDate = date;
}

void Foo::setDescription(const char* description)
{
        // Get a pointer of the correct type
        FooMembers* mPtr = (FooMembers*)_members;
        int len = strlen(description);
        // Only reallocate the array if necessary...
        if(len > strlen(mPtr->_description))
        {
                delete [] mPtr->_description;
                mPtr->_description = 0;

                mPtr->_description = new
                   (os_cluster::of(this), ts<char>(), len)
                       char[len];
        }
        strcpy(mPtr->_description, description);
}
```

```
std::ostream& operator<< (std::ostream& out, const Foo& obj)
{
	// Get a pointer of the correct type
	FooMembers* mPtr = (FooMembers*)obj._members;
	out << "(id=" << mPtr->_id
		<< ", number=" << mPtr->_number
		<< ", description=" << mPtr->_description
		<< ", creation date=" << mPtr->_creationDate
		<< ")\n";
	return out;
}
```

They are all written such that they cast the `_members` pointer to the correct type, and then use this to access the data members directly. All these casts are encapsulated within this `Foo.cpp` file.

Example Uses

Early versions of the Ordnance Survey mapping system OS MasterMap used this pattern.

Related Patterns

The Frame Anti-Pattern.

Persistent Queue

Intent

Enables the asynchronous production and consumption of information or objects. Activity can be suspended and resumed on either side without the loss of any data.

Also Known As

Reliable Producer-Consumer

Motivation

In network management software alarms and other events are raised asynchronously from hardware on the network, and these need to be collected very quickly and written to the database. Parallel dispatcher processes will read these events off the queue, examine them, and before forwarding them to the person or process best able to handle them. These events can arrive very quickly and from multiple pieces of equipment simultaneously. Typically these systems are configured with multiple dispatcher processes and as each event arrives the next idle dispatcher will immediately read it off the queue, processes it, and forward them appropriately. Event objects can stay safely in the queue until a dispatcher can deal with it.

There can be a huge range of different requirements around this pattern; everything from the event size, the event arrival rate, the number of queues and the queue topology, the throughput and latency requirements, the numbers of reader and writer processes or threads, the maximum size of the queue and whether this is bounded or 'unbounded', and the idle time available for reader catch-up, among many others.

Applicability

Use persistent queue pattern when:

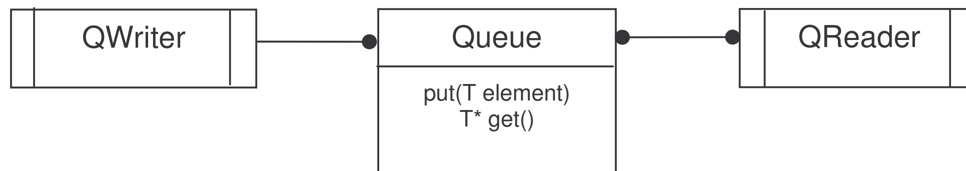
- Objects must be written to the database by multiple writers in parallel and performance requirements demand that the writers must not block each other, or any readers

- There is a data 'processing pipeline' architecture where objects move through a series of states before they reach a final destination state.

- Message objects or events must be stored persistently before being processed to avoid losing any.

- There is a requirement for interruptible queues where either the writers or readers can be suspended, and then resume from where they left off.

Structure

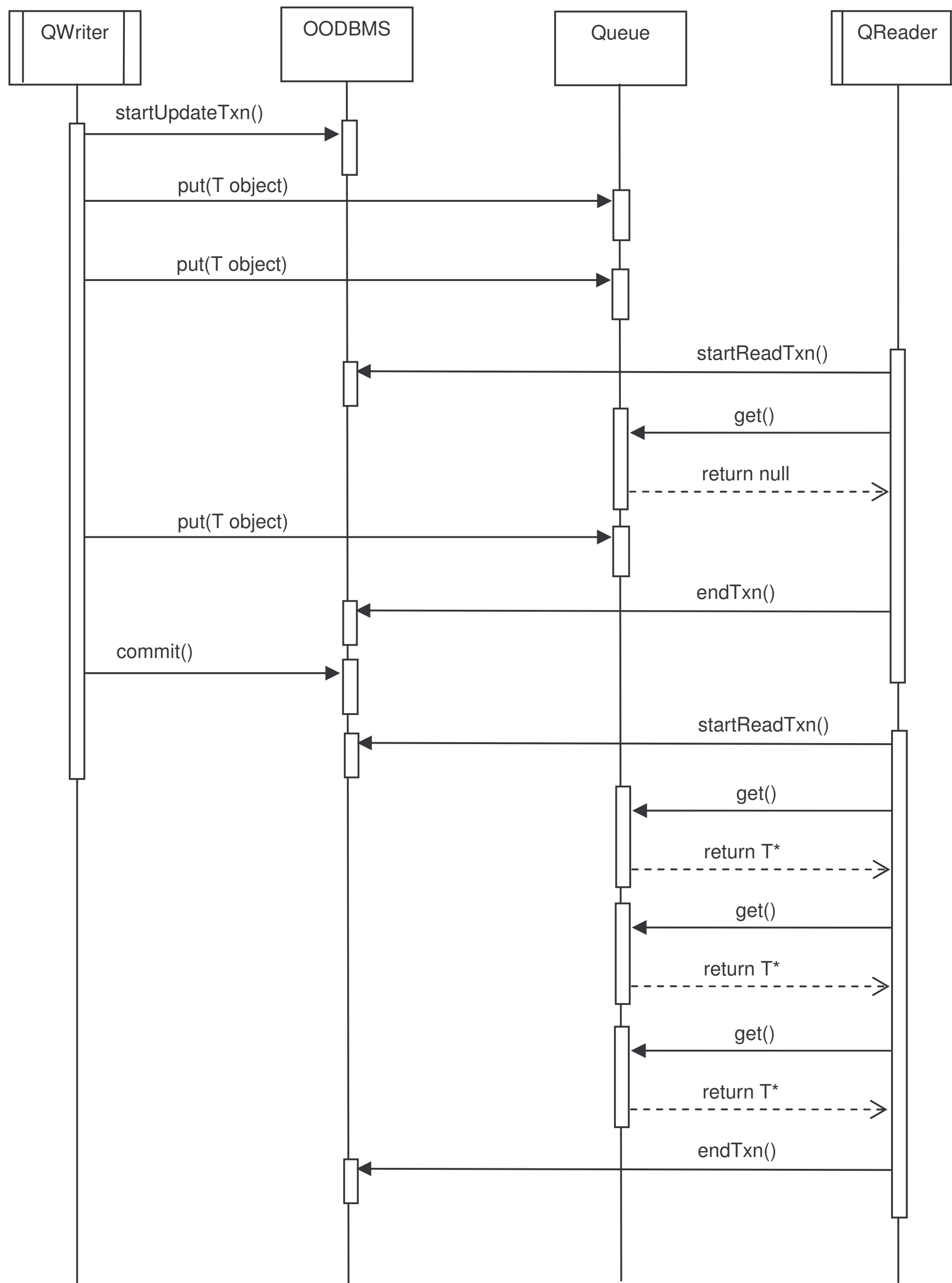


Participants

- **Queue**
  - Holds a collection of objects written into the queue by the writer
  - Each queue is written into by a single writer
  - Objects remain in the queue until all readers have read the objects off the queue
  - Queues can be named
- **QWriter**
  - Instances of this class supply objects that are used by the queue readers
  - Works asynchronously from the readers, and from one another if there are multiple queue writers
  - If all readers are busy, the writer can still put an object onto the queue and continue with other tasks.
  - Can write to multiple persistent queues
- **QReader**
  - Readers consume objects produced by writers
  - Readers read objects off the queues in order of arrival.
  - If the queue object is empty then the reader can do other work, or pause, until there is an element in the queue. The reader periodically polls for objects on the queue.
  - Readers can read multiple queues.
  - Queues can have read priorities

Collaborations

Here we show a series of objects being written onto a queue by the writer within a write transaction, and a reader reading off the queue within read transactions.



There are two simultaneous threads of control shown here on a single collaboration diagram, so the actual concurrent behaviour is not well represented. These two threads of control are initiated by the `QWriter` and the `QReader` each running in a separate database client program. The `Queue` is initially empty. The `QWriter` starts an update transaction by calling the appropriate OODBMS API, calls `put()` three times to push objects on the persistent queue in quick succession, and finally commits this update transaction.

Meanwhile in another process a `QReader` starts a non-blocking read-only transaction and attempts to `get()` an object from the queue. This read attempt return a `null` so the `QReader` ends its read transaction. The `null` is returned because the `QWriter` did not commit its update transaction before the `QReader` starts its read-only transaction. After a short pause `QReader` restarts a new read-only transaction, and succeeds in reading the contents of the persistent queue with three `get()` calls because this transaction started after the `QWriter` committed its update transaction.

Consequences

The following consequences hold for the persistent queue pattern

1. *Reading from or writing to a persistent queue requires disc access*. The need for disc access can compromise throughput compared to queues implemented purely in memory. The seriousness of this depends on the details of the particular context and can often be alleviated by using bespoke data structures and transaction batching.

2. *Throughput can be increased by batching reads and writes*. By batching multiple writes and reads within a single high level database transaction very high object read and write rates can be achieved.

3. *Throughput can be increased by using a pool allocator*. If the implementation uses a pool allocator for the objects in the queue, then this can hugely improve performance, especially if the objects on the queue are small.

4. *Queues can be very large*. By holding the queue in a database there is no limit on the queue size imposed by the memory available. Typically systems have far more disc space than memory, so persistent queues can grow virtually without limit.

5. *Writers and readers are interruptible*. Holding the queue in a database means that if a reader is stopped and restarted, it can carry on reading the queue from where it left off. If a writer needs to write into a queue where the objects are not being consumed by a reader, the writer can safely write into the queue because the queue can grow without bounds.

6. *Object delivery is guaranteed*. A persistent queue will maintain its state between invocations of readers and writers such that every object written into the queue is virtually guaranteed to reach every reader assigned to that queue. The contract usually holds from the point that the reader subscribes to receive objects from the queue; all objects written after the time the reader subscribes are guaranteed to be delivered. The strength of the guarantee is obviously dependent on the other high availability (HA) characteristics of the database and the hardware it runs on.

7. *Writer and reader clients are never blocked*. This depends on the implementation and the facilities offered by the underlying OODBMS. If the OODBMS offers a non-blocking transactionally consistent read mode, and the queue data structures have been properly implemented, it is possible to ensure

that clients writing into the queue never have to wait for or block reader clients, and likewise for readers; they never wait for or block writers.

8. *Each writer can write into a single queue without blocking*. If multiple writers write into a single queue although this will work, they will compete for database access, so they will block each other. In situations where there are multiple writers it is often better to arrange for each to write into their own queue.

9. *Multiple readers can read from a single queue without blocking*. It is possible to arrange the internal persistent data structures comprising the queue such that multiple readers can read from a single queue without blocking each other or the writer. There are broadly two different classes of implementations; those where all readers are guaranteed to receive all the objects written into the queue, and those where each object is read by exactly one reader client, i.e. the reader clients share the contents of the queue between themselves. This latter arrangement is useful in situations where having read the queue the reader clients undertake significant processing before being able to read the queue again.

10. *Queues preserve arrival order*. The order of arrival in a single queue is preserved, but this order cannot easily be guaranteed between multiple queues.

11. *Complex queue topologies can be implemented*. Database client programs can read from and/or write to many different queues to set up complex chains or webs of object processing. All these can be non-blocking and can be implemented across one or more database instances.

12. *Queues can be compile-time type-safe*. Compile-time type safety in queues is a great advantage because when a reader reads from the queue they have no need to determine the type of the object read. They can use it with confidence directly to perform whatever processing is required. If there are a variety of object subtypes in the same queue, there are two strategies; use a different queue for each subtype eliminating the requirement for any runtime-type identification (RTTI), or provide an efficient `getType()` method on each object so RTTI is as fast as possible.

13. *Work best when objects are small PODs*. This pattern really works well when the queues contain objects that are small (i.e. no larger than a 1000 bytes), and are plain-old-data in no inheritance hierarchy and that themselves do not allocate memory outside themselves (i.e. simple C-structs with no pointers to other persistent objects including variable length strings). This is for three reasons: they can be easily pool allocated when written, there is no requirement to use any form of RTTI when reading them, and being small there is less disc activity per object for both reads and writes.

14. *Can arrange for priority queues*. Readers can be implemented to read sets of queues in a particular order, such that the queues containing the most important objects always get read and processed first. This means that high priority data can get special attention if required. However, when reading

multiple queues in a fixed order, there is a danger of the low priority queues never being serviced, which can severely skew the latency characteristics of each queue.

15. *Can have named and anonymous queues*. By identifying queues by name, and by creating and setting up queue structures on-the-fly, it is possible to have queue end points visible in a system via globally well-known names – somewhat like JMS topics. It is also possible to use queues as part of a library in such a way that they are anonymous, and buried within a service infrastructure.

16. *Test queue implementations for long periods*. This is important especially when there are many thousands of objects arriving per second, as unexpected behaviour can emerge. Some of these are listed below.

17. *Read starvation*. A reader is reading many queues, but does not get time to read events off some of the queues, which start to fill with events pending. The database starts to get very large, because the writer is always creating queue nodes, and the latency goes up - events are waiting in the queue to be read. This is more likely to happen if the multiple queues are always read in the same order.

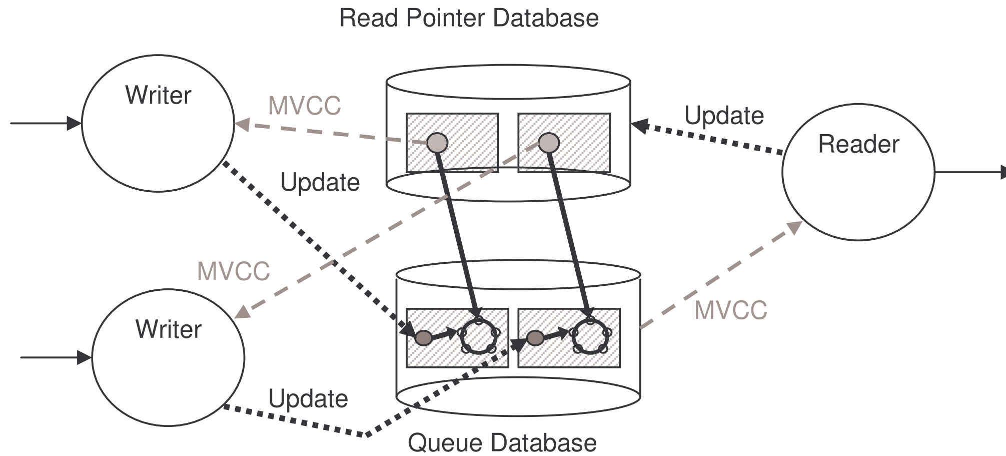
18. *Read/Write rate imbalance*. The writers are writing faster than the readers can empty the queues. This results in the queues expanding and the averages latency increasing. Even though the readers do read off every queue, they never read enough events to empty the queue, so they progressively fall behind the writers; the queue/database grows without bound.

19. *Interrupted reader fails to catch up*. Stopping and restarting a reader should result in the reader 'catching up' with the writer if the read and write batch sizes are set correctly. The number of events you attempt to read should be larger than the write size by some percentage, for example around 20% - 50%, to ensure that this ‘catch up’ behaviour will occur.

20. *Latency problems in high throughput situations*. When the throughput is in the order of thousands of events per second if the batch sizes are too large, both for the reader and writer, then you will increase the latency - i.e. some of the events that you read will have been waiting in the database for the duration of the transaction that wrote them, and others for most of the duration of the read transaction. It is necessary to balance throughput with latency when there are high event volumes.

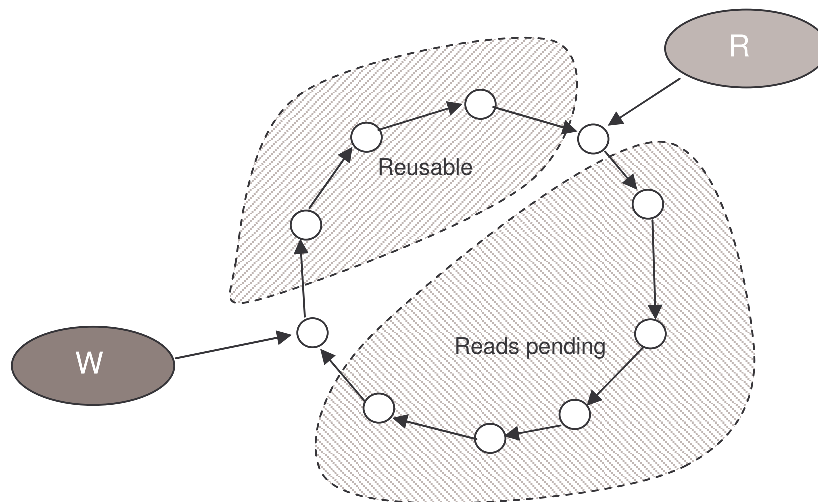
Sample Code

Here we describe a non-blocking persistent queue implementation that uses C++ templates to ensure compile-time type-safety, and uses the ObjectStore database. The structure of the queue is illustrated below.



Here we see two queue writers writing into separate queues held across two ObjectStore databases, and a single reader consuming objects from both queues. The writers open the Queue Database (QDB) for update, and the Read Pointer Database (RPDB) in a non-blocking read-only mode called MVCC,[36] whereas the reader does the opposite; it open the QDB in MVCC and the RPDB for update. These open modes and the data flows are indicated by the dashed arrows on the diagram.

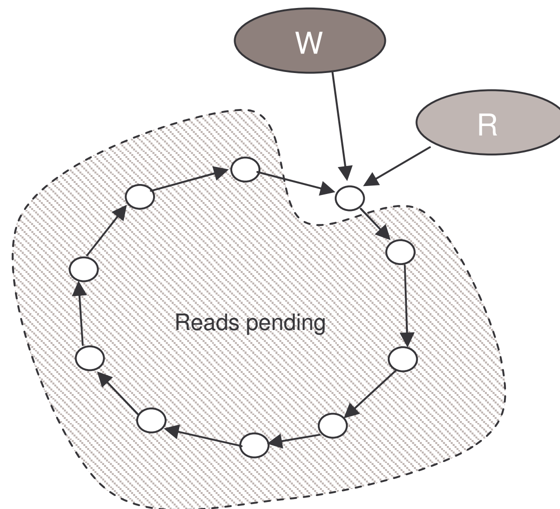
The QDB at the bottom of the diagram contains two queue data structures, one for each writer. Each is a 'ring', which is essentially an intrusive singlely-linked list of objects, where the tail points back to the head, along with a write pointer (shown as a dark grey oval) pointing to the current update position within this ring. The RPDB above contains two read pointers (shown as light grey ovals) pointing to the current read position in the ring data structure held in the other database. A magnified version of these ring data structures is illustrated further below.



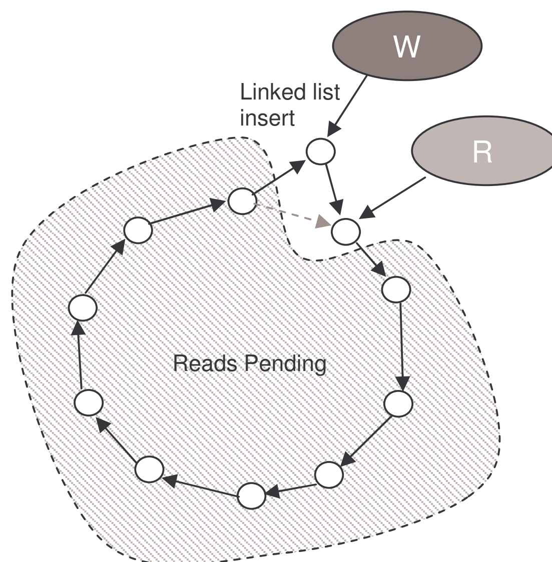
[36] MVCC means multi-version concurrency control. This is a special open mode for an ObjectStore database which allows multiple readers on any particular page *and* a single writer, such that all parties are guaranteed a transactionally consistent view of the database and will not block each other.

Above we see the intrusive singley-linked list as a ring of objects where each element in the ring is pointing to the element in front. The write pointer and the read pointer are shown as dark and light ovals respectively, and together they divide the ring into two sections; *Reads Pending*, containing those objects in the list which have yet to be consumed by the reader, and *Reusable*, containing objects that have been read and whose data values can be overwritten with updates.

To put an object on the queue, the writer moves to the next element and, if the value of the write pointer is not equal to that of the read pointer it overwrites the element. This process can continue until the write pointer ‘catches up’ with the read pointer as shown below.



When the write pointer is equal to the read pointer it can simply insert into the list instead of reusing existing elements, as illustrated below.



Meanwhile the reader client reads from the queue (or queues) until the read pointer equals the write pointer, in which case there are no more pending reads. So now we show some example code that implements this structure.

```
template <class T>
class Ele
{
public:

	// Ctors
	Ele()
		:_value(0), _next(this)
	{}

	Ele(T& v, Ele* e)
		:_value(v), _next(e)
	{}

	// *** Accessors

	T* getValue() { return &_value; }
	Ele<T>* getNext() { return _next; }

	// *** Updaters

	void setValue(T v) {_value = v; }
	void setNext(Ele<T>* e) { _next = e; }

private:

	T _value;
	Ele<T>* _next;

};
```

This is a template for elements within a type-safe persistent queue, so the queue can only contain elements of a single C++ type. There are two data members, the value, which is the type of the template, and a pointer to another instance of this `Ele<T>` template. Embedding a `T` for the value (rather than a `T*`) obviates the need to have an extra pointer to the value, so saving space, and means that objects being inserted into the queue do not themselves have to be allocated persistently, but will be copied either bitwise or via the assignment operator into the persistent space. This makes the code easier to use. The second data member forms the basis of the intrusive linked list and means the objects that are to be inserted into the queue do not have to implement this code.

There is a no-arg constructor so we can pool allocate these elements should we need to, and there are methods to get and set values, and get and set the next element in the list. The `getValue()` method returns a pointer because we can use the `null` to signify when the list is empty, it is more efficient for large objects as no copy ctor is involved, it works for classes and primitive types such as `int` or `double`, and we can guarantee that the value will not be overwritten while the read pointer is targeted at '`this`'. The `setValue()` method requires that either type `T` is bitwise copyable or that an assignment operator has been defined for that type.

```
template <class T>
class PQ
{
public:
```

```
        // The only ctor
        PQ(Ele<T>*& rPtr)
               :_head(0), _wp(0), _rp(rPtr)
        {
               _head = new(os_cluster::of(this), ts< Ele<T> >()) Ele<T>;
               _wp = _head;
               _rp = _head;

        }

        // ..and the dtor
        ~PQ()
        {
               // A persistent queue owns all its elements.
               // Must go through and delete them all.
               Ele<T>* ptr = _head->getNext();

               while(ptr != _head)
               {
                      Ele<T>* reaper = ptr;
                      ptr = ptr->getNext();
                      delete reaper;
               }

               delete _head;

        }

        // Put always succeeds. Resources are allocated as required.
        void put(T& x)
        {
               Ele<T>* prev = _wp;
               _wp = _wp->getNext();

               if(_wp == _rp)
               {
                      // We must grow the queue
                      _wp = new (os_cluster::of(this), ts< Ele<T> >())
                             Ele<T>(x, _rp);
                      prev->setNext(_wp);
               }
               else
               {
                      _wp->setValue(x);
               }
        }

        // Reading an empty queue returns a null pointer.
        T* get()
        {
               // Assume queue is empty, so prepare to return null.
               T* ret = 0;

               // Ensure that we don't increment past the write pointer.
               if(_rp != _wp)
               {
                      _rp = _rp->getNext();
                      ret = _rp->getValue();
               }

               return ret;
        }

        // Used by factory class when creating the queue.
        Ele<T>* getHead() const { return _head; }

private:
```

```
        // Can't copy or assign these queues so hide 'em.
        PQ(const PQ& obj);
        PQ& operator= (const PQ& obj);

        Ele<T>* _head;
        Ele<T>* _wp;

        // Reference to ptr because it must be
        // allocated within a separate database.
        Ele<T>*& _rp;
};
```

Above we have a light-weight persistent queue template called `PQ<T>` with three data members; a pointer to the head of the queue `_head`, a write pointer `_wp`, and a reference to the read pointer `_rp`. This is a reference to the read pointer because the read pointer itself is a pointer in another database that refers back into this queue structure.

There is a single ctor which takes a reference to this remote read pointer as its only argument, and after creating an `Ele<T>` persistently it targets the read and write pointers onto `_head`. Then there is a destructor which is responsible for deleting the entire contents of the queue.

The `put()` method takes a reference to an object to be inserted into the queue, and starts by incrementing the write pointer after saving its current value on the stack should it be needed. If it has caught up with the read pointer, a new element is allocated [37] and inserted into the list, with the pointers being fixed up as necessary. Otherwise the write pointer targets a reusable element, so its state can be simply overwritten.

The `get()` method assumes that the queue is empty so prepares to return a null value, then if the read pointer does not equal the value of the write pointer it returns a pointer to the next element in the list.

Example Uses

The Swiss investment bank Lombard Odier uses a similar arrangement to handle certain types of events in their investment portfolio management system G2.

Related Patterns

Pool Allocator

[37] Replacing the persistent new here with a call into pool allocator can improve write performance.

OO Anti-Patterns

Here we present a common anti-pattern that has been encountered in the past decade working with ObjectStore based systems.

Frame

Intent

Do not define a single meta-type within the system as collections of name/value pairs or type/name/value tuples, and then implement all persistent entities in terms of this meta-type. Use the language types known to the compiler and benefit from compile-time type-safety and better runtime performance.

Also Known As

Meta-Type System, Name-Value Pairs

Motivation

It is surprisingly common for designers to deconstruct a system into a set of types implemented as hash tables of name value pairs. Designers implement a single base class, called a Frame, which is essentially a hash table with a number of optional slots; each slot has a name, corresponding to a data member name, a type, such as a `string` or an `int`, and a value. The designer then goes on to represent every persistent entity as an instance of one of these frames, so that that data members can be added or removed easily as requirements change.

It is useful to contrast ‘explicit’ and ‘implicit’ type systems here. Explicit type systems determine an object’s class by explicitly classifying any object as a member of a particular class. In practice this is usually done at the moment of object creation. By being a member of a particular class the object then acquires a set of attributes that all members of that class are defined to have, and this set of attributes does not alter throughout the life-time of the object.

With implicit type systems objects are assigned a set of attributes and, by virtue of the current set of attributes they have, are said to be a member or not of any particular class. Any instance could at any time be a member of several different classes simultaneously and the set of classes to which it belongs will change during the lifetime of the object as attributes are added or deleted.

Explicit type systems, such as C++, tend to define classes at compile time. One of the advantages of this is type-safety. Many type mismatches (aside from casting) can be caught at compile time and can contribute to robust program behaviour. Explicit type systems will tend to catch errors caused by objects being of the wrong type at compile time.

Implicit type system objects have their ‘classes’ defined at run-time, which means that objects can easily change their class membership during processing. Consequently compile-time type checking, in terms of any implicit types makes very little sense. Implicit type systems will tend to catch errors caused by objects being of the wrong type (i.e. having the wrong set of attributes) only at run-time.

Using Frames to express every persistent entity introduces an implicit type system which can be very flexible, and conflates the ideas of class membership, schema evolution, and object versioning; because any object is only a member of a 'class' by virtue of its current slot list, slots can be added or removed at runtime so the 'class' of the object can change, and if a history of these changes is maintained, not only can the values of slots change between different 'version's but the 'class' can change also. A perceived requirement for this sort of flexibility is often the motivation for a designer to adopt this approach.

Applicability

Beware the Frame anti-pattern when:

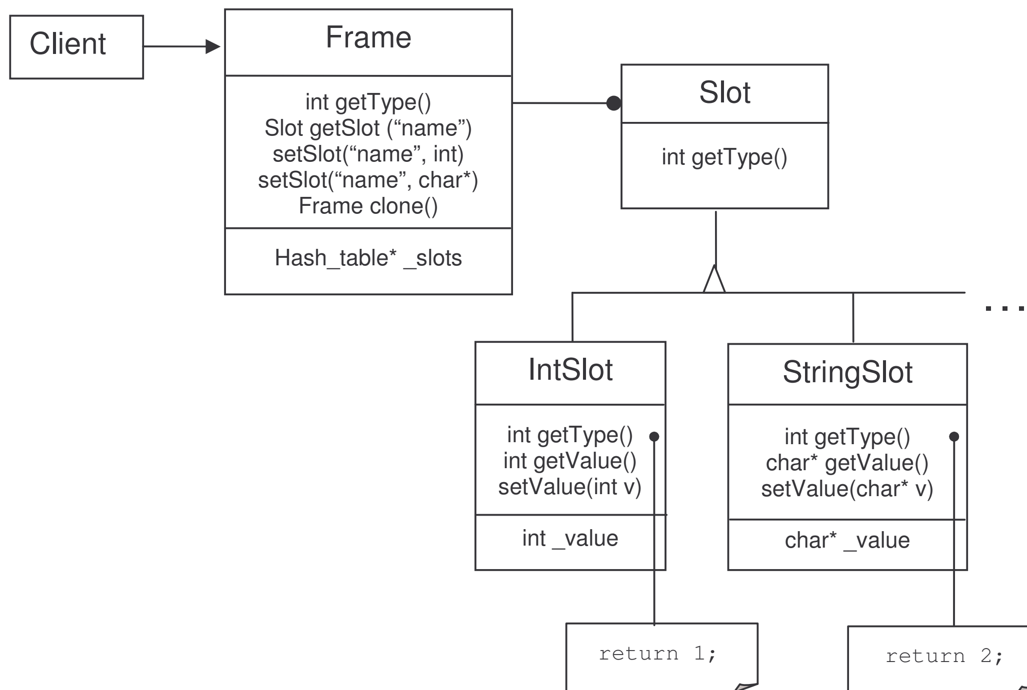
- There is a requirement for a flexible type system, where instances have to change their data members by way of schema evolution.

- Flexible object versioning schemes where not only snap-shots of data-member values must be recorded, but changes to data-members must be accommodated.

- Underspecified designs where points of change within the class system have been poorly identified, or badly localized.

Structure

Below we show a typical structure, but this can vary radically depending on the precise implementation



Participants

- **Frame**

  - Owns a collection `_slots` of slot objects of various sub-types. Depending on the implementation, the name of the slot can be held within the slot object itself, or more typically, the slot objects are entries in a hash structure keyed on the member name.

  - Provides an interface to `get` and `set` slot objects which return `null`, or throw exceptions, if not present. Sometimes there is a single `setSlot()` method that takes a name and a slot object; more common are a series of overloadings that take a name and a value of the correct type. Sometimes there are type specific get methods, called something like `getIntSlotValue()` which return the values themselves rather than the slot objects.

  - Optional method that returns a 'type' which is often a simple integer, but can be a more complex type particularly if attempts are made to support multiple inheritance. The type returned can give the caller an indication as to what slots are currently present, although this type information cannot be enforced by the compiler.

  - Optional `clone()` method which returns a deep-copy of the frame and all its slots. This circumnavigates the problem of construction of instances with optional attributes and is an example of the prototype pattern.

- **Slot**

  - Provides a super-type to unify all the slot sub-types so they can be referenced by a single collection within the frame object.

  - Optionally provides a virtual `getType()` method which must therefore be implemented by every slot subclass. Designers should be careful to distinguish these from any Frame type identifier.

  - Some implementations attempt to provide slot value getters and setters in this interface; however the language rules on method signature formation in C++ and Java mean this is neither useful nor elegant, as explained below.

- **IntSlot**

  - Example slot sub-class that holds integer values, and provides methods to get and set integer slot values.

    - Optionally implements a `getType()` method that is usually implemented to return a hard-coded type identifier such as an `enum` or an `int` to provide an efficient way for clients to identify the slot type prior to casting. In the absence of this method users will need to use some form of RTTI [38] built into the language.

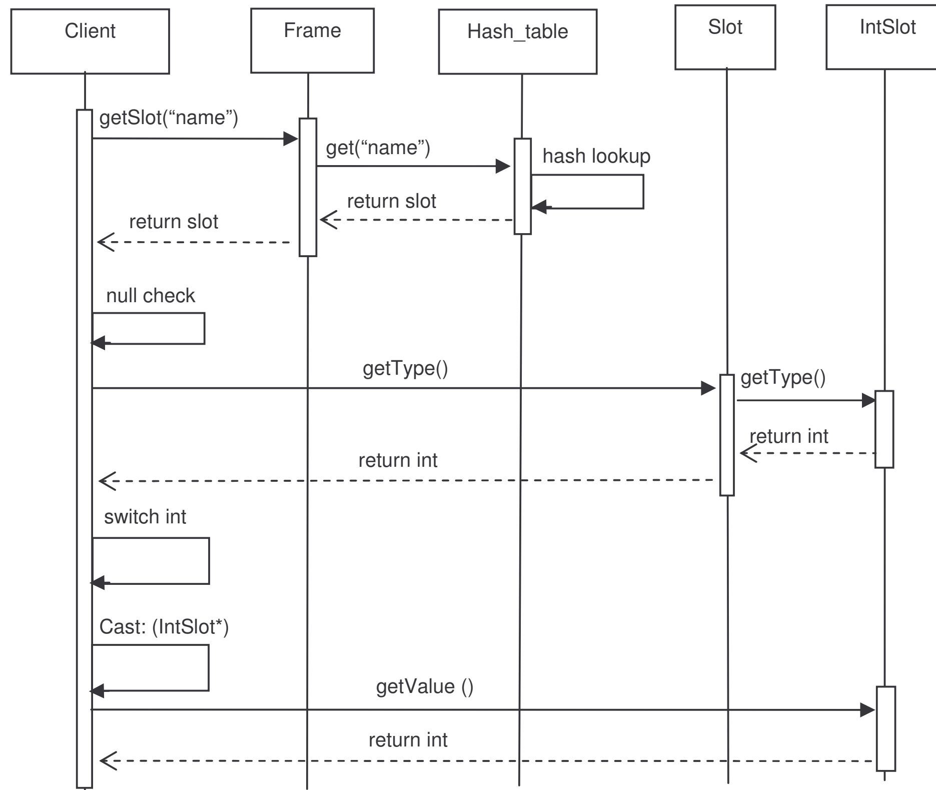
- **StringSlot**
    - Another example slot sub-type that deals with strings, with the same methods and attributes designed to deal with strings rather than integers.

- **Client**
    - Constructs new empty frame objects and fills them with slot objects as needed, or clones existing ones to create new instances.
    - Creates or updates slots using a frames `setSlot()` methods often passing in hard-coded member names.
    - Accesses slots using hard-coded member names, and then discovers at runtime whether the slot exists.
    - Discovers the type of the slot returned using the slot `getType()` method or RTTI and then casts the slot to the correct slot sub-type before accessing its value.
    - Uses the frame `getType()` method and then 'expects' slots of certain name, type and values to be present and simply accesses the slots without adequate error or type checking.
    - Implements code to actually work with the slot values returned. Most of the code that operates on the slot values is, from the compiler's perspective, external to the Frame, and as such is implemented in the client.

[38] RTTI is run-time type identification

Collaborations

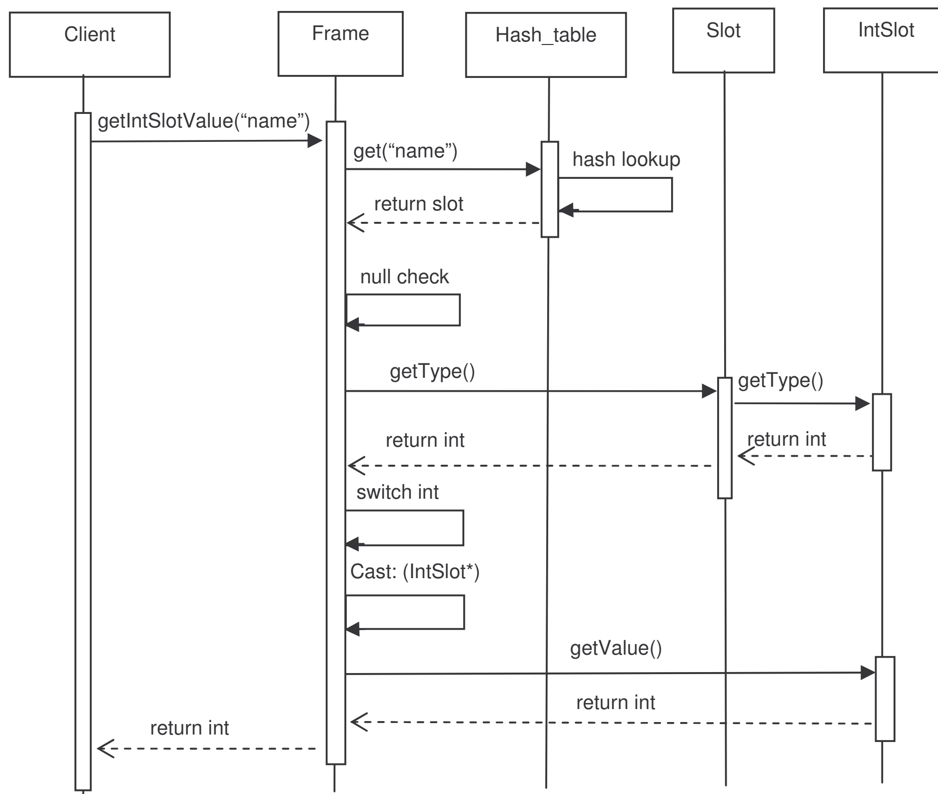
Here we show three scenarios; first a client getting the value of an integer slot, then a client adding a new slot, and then updating the value of an existing slot.



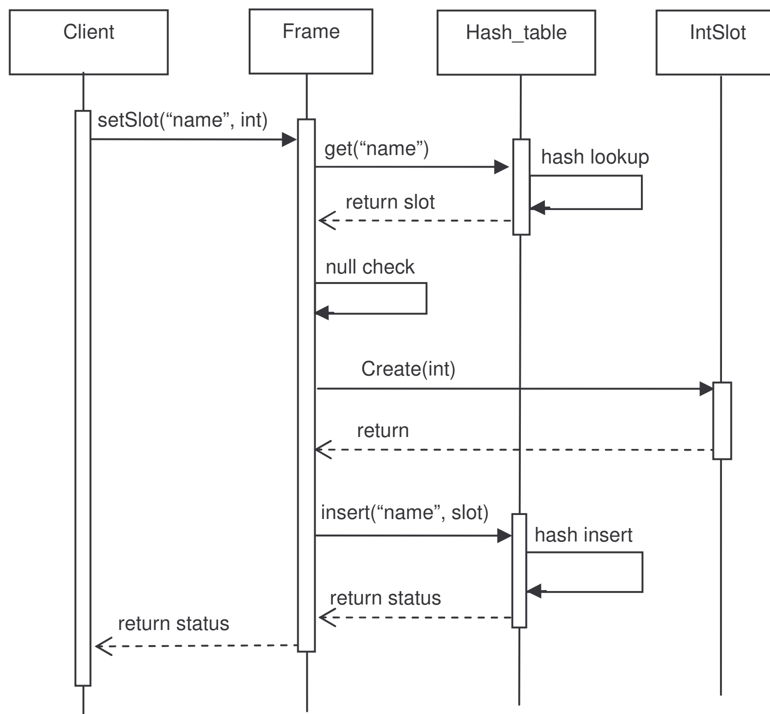
The client starts by calling the `getSlot()` method passing in the name of the slot. The frame object then uses the passed string to do a hash lookup on its hash table; if found, the slot object is returned. The client must check for a `null` return value otherwise there is a danger of attempting to de-reference a null pointer. The client then calls the `getType()` method which is virtually dispatched to the `IntSlot` sub-type which then returns a hard-coded integer uniquely identifying the sub-type. The client must then check the type returned before casting to the slot object appropriately. The client can now call the `getValue()` method which returns an integer.

So to return the value of an integer attribute requires, 3 function calls, 1 virtual function call, a hash lookup, 3 return values, 2 conditionals and a type cast.

As mentioned above it is quite common to extend the Frame interface and locate this code in 'getter' methods which are specific to particular slot types, so the getter method can encapsulate the type checking and the casting. These methods are named something like `getIntSlotValue()` and return the actual value of the slot. This variant is shown below.

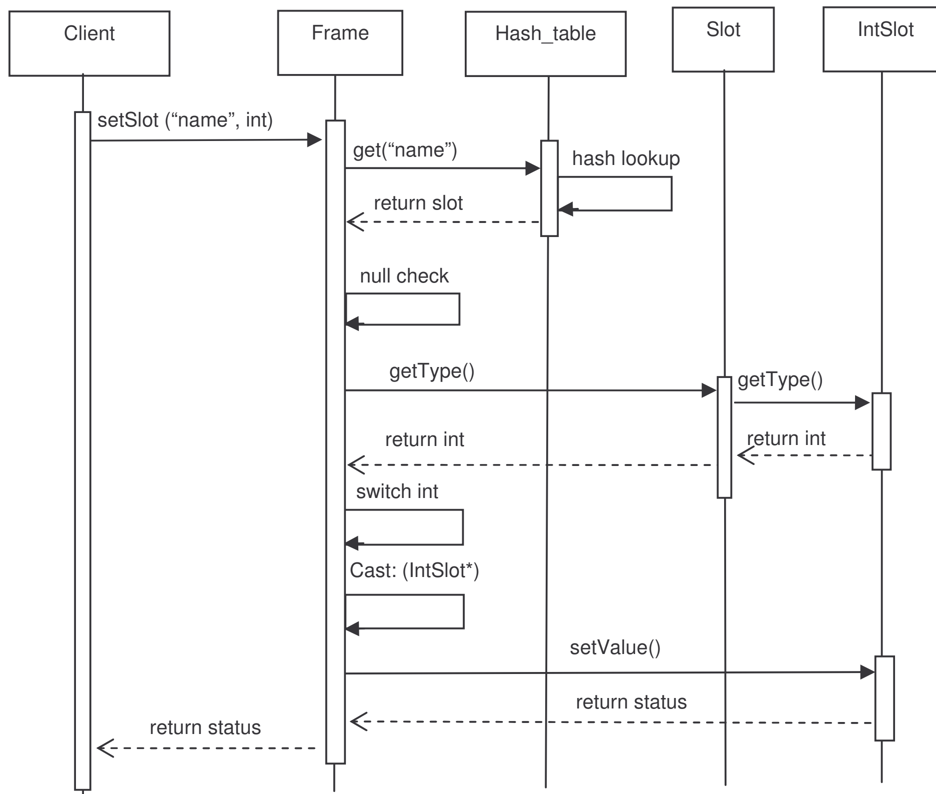


However error conditions must now be indicated with either exceptions or contrived return values. Now we look at setting the value of a non-existent slot.



The client calls the `setSlot()` method passing in the name of the new attribute and its integer value. Internally the frame now attempts to get the slot object from the hash structure and checks the return value is null.[39] If a slot object had been returned, its value could have been updated directly, but in this scenario a null was returned, so a new `IntSlot` object is constructed with the correct value and this is inserted into the hash structure. The status of this operation is then returned.

To add a new integer attribute requires 3 function calls, 2 or 3 return values, a hash lookup, a hash insert, one conditional and one object construction. Now we look at setting the value of a slot that already exists.



To update the value of an existing slot the client calls `setSlot()` passing in the ‘member’ name and the value as before, and the `Frame` object performs a hash lookup to get a reference to the slot object, which it must check for `null`. It must then check the type at runtime,[40] before casting to the correct slot subclass and then call the `setValue()` method on the `IntSlot` object. The status of this operation is then returned.

[39] The initial attempt to read an object from the hash table is to avoid creating and inserting new slot objects, and having to destroy the old slot objects, unnecessarily.
[40] Even though an attribute by this name has been returned, there is no guarantee that it is of the correct type, therefore the type must be checked to ensure the type cast does not fail.

To update an existing slot requires 3 function calls, 1 virtual function call, a hash lookup, 2 conditionals, 1 type cast and 2 or 3 return values.

Consequences

The following consequences hold for the Frame anti-pattern

1. *Compile-time type-safety is compromised*. Code that uses the Frame class, and even methods within the Frame class itself, cannot rely on any particular slot being present; and even if it is, they cannot rely on its type. The compiler cannot guarantee the type of any Frame object in terms of which 'data members' (i.e. slots) are present, consequently every slot access, both read and write, has to be checked at runtime.

2. *Encapsulation is compromised*. All slots are public and can be changed by any code anywhere in the system. The compiler cannot help with data hiding because slots and/or their values are directly available to any caller globally. [41]

3. *Component coupling is increased*. Every call point that uses a Frame must have knowledge of what attributes are expected, which slots, their type and knowledge of the pre-conditions, post-conditions and invariants. The names of slots are typically hard-coded as strings throughout the code, and also error handling code is ubiquitous. These hard-coded names, the expected types, the contracts and the error handling must be coordinated throughout the code base, and means that the code of systems based around Frames tend to become highly coupled.

4. *Frames methods have no privileged access to slots.* Business level code that manipulates the values of slots cannot usefully be located within the Frame class, because at the point this code is written there is no guarantee that the slots required even exist, or are of the correct type. Every method in the Frame class must check for the existence and type of each slot at runtime just like external client code.

---

[41] Established OO principles, such as encapsulation, mandate the use of private data members within language-level classes. There are two distinct phases with explicit systems: *class design* where candidate attributes for any class are chosen during design and implementation, and *run-time* where the set of agreed classes is utilised to create object instances within a running program. Class attributes are chosen in the design phase and then the resulting classes are simply used at runtime. Objects thus acquire their attributes at a prior time and in a completely different context compared to when they are instantiated. As a result, at run-time there is never any need to expose the internals of a class to outsiders, although this may be done for reasons of simplicity or optimisation.

Objects in implicit type systems can be created tabula-rasa: they have no need of attributes at creation time since they can accumulate them during the run-time process. Where do the attributes come from? Throughout the lifetime of any object it may have attributes added or deleted and these attributes could be instances of other objects but the question remains: Where does the first attribute come from?

To avoid the problems of infinite regress we need prior definition of at least some attributes because there are no prior classes defined in purely implicit systems. This is necessary in-order to bootstrap the aggregation process. Since some of these attributes have prior existence to object instances then it follows they must be in some sense external to the object. By combining this observation with arguments from symmetry, universality and simplicity we can posit a tendency for attributes in implicit type systems to be public data members rather than private. Objects with implicit type are essentially frames with variable slots. The attributes are specified by some client process, external to the object. So even as the object comes into existence some external client already has information about some sub-set of its attributes. This is true for every attribute therefore every attribute must have at some point been publicly accessible at runtime.

5. *Programming to a contract [42] more onerous.* Asserting pre-conditions, post-conditions and invariants is more complex because slot existence, type and value are all so flexible, and globally exposed. Any code attempting to assert truths about slot values must contend with the fact that such slots might not exist, or be of the wrong data type, before they handle considerations of value, and because the slots can be changed from anywhere, this contract checking tends to be written more frequently.

6. *Error handling is more complex.* Code that uses Frame objects tends to require more explicit error checking at every call site, than functionally comparable code that uses simple language level classes. This can entail checking for special return values such as null, or impose the overhead of exception handling. This error handling code is difficult to encapsulate because the slots and their values are publicly available, and must be run everywhere they are used.

7. *Code complexity increases*. In the general case every Frame client must check for missing slots or slots with unexpected types, and at every call site, more conditionals are run for existence and type checks, and/or there is more than the average number of lines of exception handling code. If the unexpected does occur, then remedial action itself can be complex. If a slot of an unexpected type is encountered, some other code must have created it. If remedial action replaces the slot, it is not easy to determine where the other code is or how it will behave

8. *Code maintenance is more difficult*. Industry research shows that the major business cost of code-ownership is the cost of code maintenance. Typically, the code base for long-term software projects is written once by a single programmer and read nine times by maintenance programmers during the lifetime of the software. The more complex the code base, the harder it is for maintenance programmers to understand the code. This is particularly acute if the code base is badly encapsulated, and highly coupled. Code heavily based on the Frame pattern is very badly encapsulated and very highly coupled and as a result is unnecessarily difficult and costly to maintain.

9. *Software testing is more difficult*. The lack of compile-time type safety means that the compiler cannot find many common programming errors; errors that would be exposed by a C++ or Java compiler with a straight-forward class implementation need to be caught at runtime. This means that before any new version of a system using Frames can be released the correctness of the code must be tested thoroughly and this test suite has many more code paths to check – not only because of all the possible runtime exceptions that can be generated – but because of all the pre-conditions, post-conditions and invariants that need to be checked.

10. *Large memory overhead*. The memory overhead for a Frame can be large compared to a standard language-level class. The Frame will need a hash

[42] Design by Contract or programming by contract are ideas introduced by Dr Bertrand Meyer in the design of the Eiffel language. See: http://en.wikipedia.org/wiki/Design_by_contract for an introduction.

structure, the size of which is strongly implementation dependent, maybe a pointer thereto, unless the hash structure is directly embedded in the frame, and at least one pointer/reference for each slot object. For Frames with a small number of slots this problem can be more acute, and if there are many instances, it can compromise any in-memory caching offered by the OODBMS.

11. *Increased runtime overhead*. There are significant runtime overheads associated with every aspect of Frame usage; simply reading or updating a value involves three function calls, one virtual function call, a hash lookup, two conditionals, a type cast and two or three return values. This ignores the overheads introduced by any exception handling code that may surround the client call site specifically to deal with unexpected slot states.[43] For a simple language-level class this is reduced to one function call and a return value and no mandate to supply extra exception handling just for these calls. The Frame pattern can also impact the chances of having the object memory resident, which means that performance can degrade as a result of the increased disc access required.

12. *Slot base-class `get` and `set` methods are awkward*. If virtual `set` method declarations are provided in the `Slot` base-class such that all `Slot` sub-classes must implement them, most of these implementations would simply throw a runtime exception; unless input arguments can be converted to other sub-types,[44] but this is not generally the case. Get methods have a more serious problem. In C++ and Java, functions cannot be overloaded by their return type alone, so declaring virtual methods such as `int getValue()` and `char* getValue()` in the Slot base class is impossible. Even if it were possible, by differentiating them with type specific names such as `int getIntValue()` and `char* getStringValue()`, again most of these would do nothing, return a special value, such as null, or throw an exception. In every instance the client code still has to deal with the runtime failures.

13. *Frames allow great flexibility*. The class(es) that an object is an instance of can change at runtime, because essentially every slot is an optional attribute. This means that schema evolution is just another update use-case, that can run in a lazy way across instances as they are encountered in other use-cases, or in one go across all instances in the database.

14. *Frames are best created using factories and/or prototypes*. To enable practical object creation it is common to utilize Factory classes, which return Frames with well-known slots, and for ad-hoc creation an existing Frame is simply cloned using the prototype pattern.

15. *Pool-allocators for frame and slot objects can increase performance*. Some of the overheads that result from using Frames are the cost of frame and slot object construction. This can be reduced by using pool allocation techniques, particularly effective if the frame/slot objects are small.

---

[43] Meyers, S. (1996). More Effective C++. Pub. Addison Wesley. P.78
[44] By promotion from `int` to `double` for example.

16. *Reduce the cost of RTTI by providing a frame meta-type system*. If it is possible to run a single type check on a frame and from that guarantee certain well-defined slots then we can dispense with individual slot type checks. This will reduce much of the code complexity and runtime overhead described above, however because the slots are still only defined at runtime, this approach usually only works in situations where multiple frames of the same meta-type are used in succession.

17. *Reduce the size of frames by replacing the hash-structure with a C++ array.* Hold hash-structure on a per-frame-meta-type basis, and implement the Frame object's collection of slots as a C++ array. Hold the slot objects directly if all the slots are of the same slot sub-type, or otherwise by reference. To access a slot by name first determine the type of the frame, then resolve the name to an array offset using the meta-type-wide hash-structure and then access the slot using the offset returned. Again this approach is usually more successful in situations where multiple frames of the same meta-type are used in succession because the costs of converting slot names to array offsets can be done once and then reused.

18. *Reduce the cost of RTTI by using homogeneous frame extents*. Much of the cost of using frames comes from having to check for the existence and type of each slot. However, if collections of frames can be created with well-defined slots, for example using a factory method, then algorithms that need to visit every frame in the collection can test the type of the first frame in the collection, and then rapidly and confidently access every slot of every frame in the collection without any further RTTI. This can result in a significant performance improvement.

Example Uses

Bad examples of the frame pattern are too numerous to mention. There have been some excellent examples of successful frame implementation using ObjectStore within the banking sector for example in risk aggregation.

Related Patterns

Evolver