Progress ObjectStore

PROGRESS SOFTWARE

Persistent OO Patterns

Adrian Marriott
Independent Consultant

Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

Persistent OO Pattern Sources

- The patterns presented here arose from ten years working with ObjectStore
  - On various projects
  - Across many industry sectors
  - Predominantly on the C++ platform

Pattern Environment

- OO Patterns are relative to particular technologies
  - i.e. pattern concerning memory management in C++ might be irrelevant in Java
  - We discuss some which are C++/Java agnostic
  - Some can be applied to other OODBMs
- ObjectStore offers features which support persistent implementation of GoF patterns
  - Can store real C++ pointers and arrays
  - Can store template classes
  - Transparent C++ language support makes it easy to using GoF style patterns in the database

OO Patterns Reduce Risk

- OO Patterns reduce risk when writing bespoke code

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."
>
> Christopher Alexander

- Novelty can be combined with established knowledge in a controlled way
  - New ideas in the context of one or more patterns that have well understood characteristics

Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

Bespoke Indexes (meta-pattern)

- 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

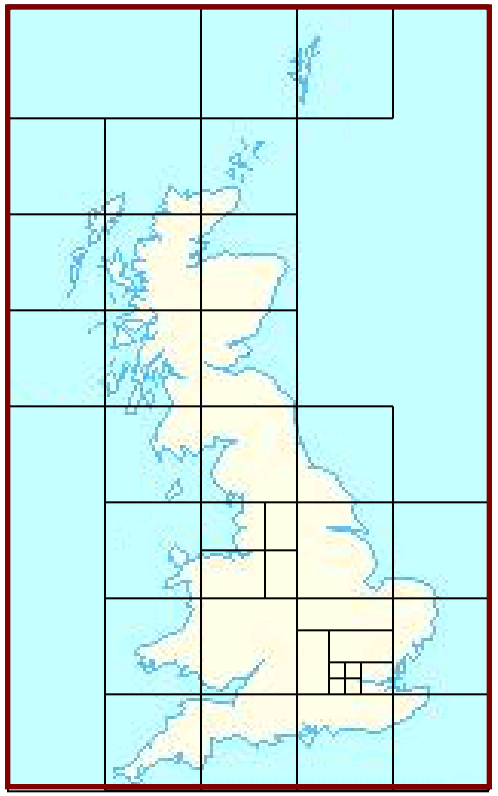
- Motivation

  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 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.

Bespoke Index: motivation

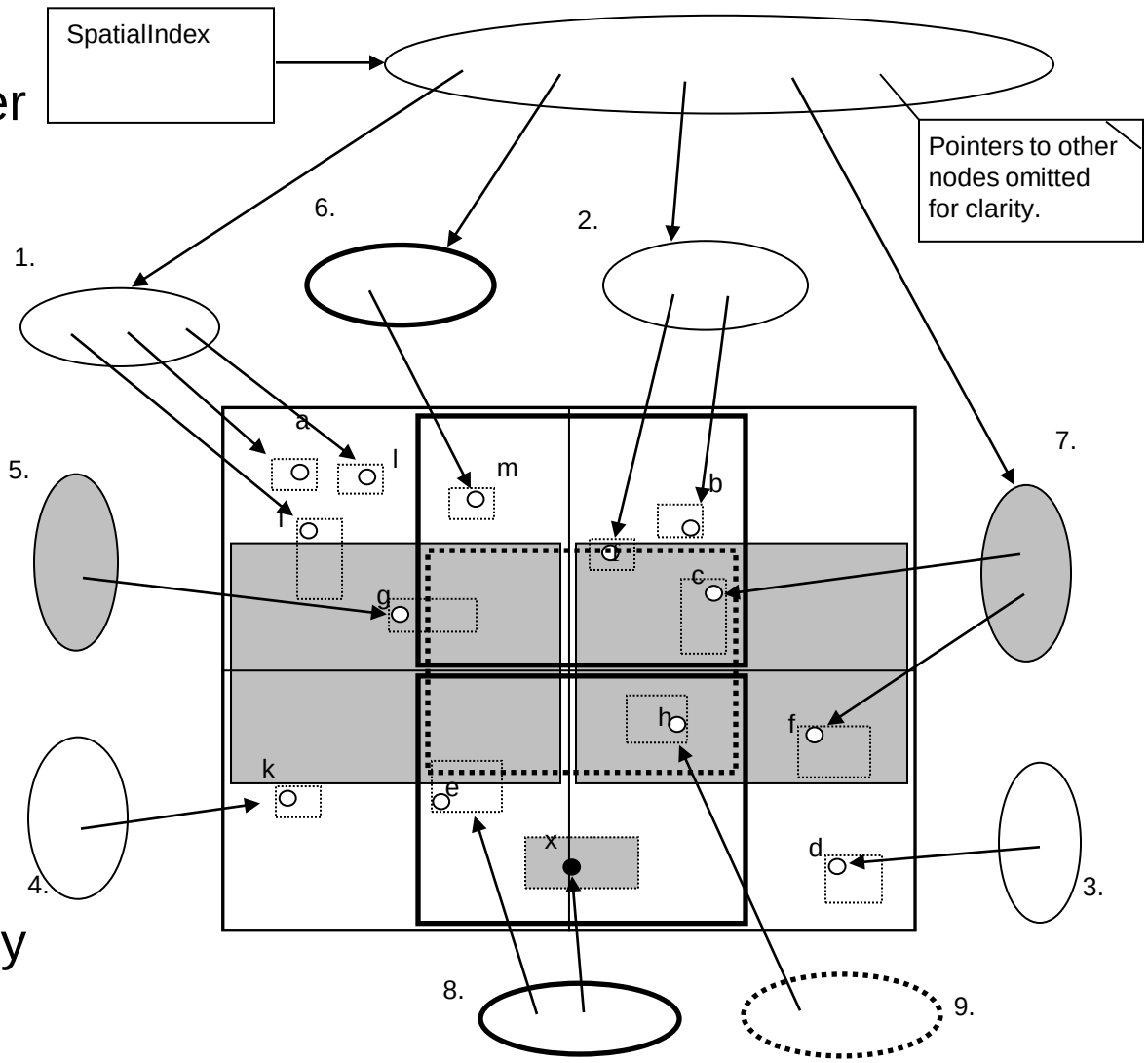
- Example

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’.



Bespoke Indexes: motivation

- Based on Quad-index
- Uses 9 cells in each layer
  - Solves ‘object overlap’ problem
- Early cell fissioning exploits ObjectStore clustering
  - No density threshold issues
  - Index ‘resolution’ determines smallest cell
- Objects stored in smallest cell that can fully contain them



Bespoke Indexes: applicability

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

Bespoke Indexes: structure

- Totally context dependent…

  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

- …Bespoke Indexes is a ‘meta-pattern’

Bespoke Indexes: consequences

- Critical use-cases can run very quickly
  - Can be orders of magnitude faster than generalized data structures
- Different use-cases use different indexes
  - i.e. one is supported by a hash-structure and another is supported by an ordered list
- Align access patterns with disc location
  - locality of reference can reduce page fetch
  - With memory resident data sets this is not an issue

Bespoke Indexes: consequences

- Designing and implementing bespoke indexes takes time and effort
  - Getting them correct takes skill
- Bespoke indexes can have implicit dependencies
  - Optimal ‘locality-of-reference’ for one might be sub-optimal for others
- Bespoke indexes can be difficult to maintain
  - System changes contradict underlying assumptions underpinning the original specification

Bespoke Indexes

- Summary
  - Don’t query for everything
  - Extend OO design techniques from the heap into the database
  - Potential to improve performance by orders of magnitude

Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

Persistent Singleton

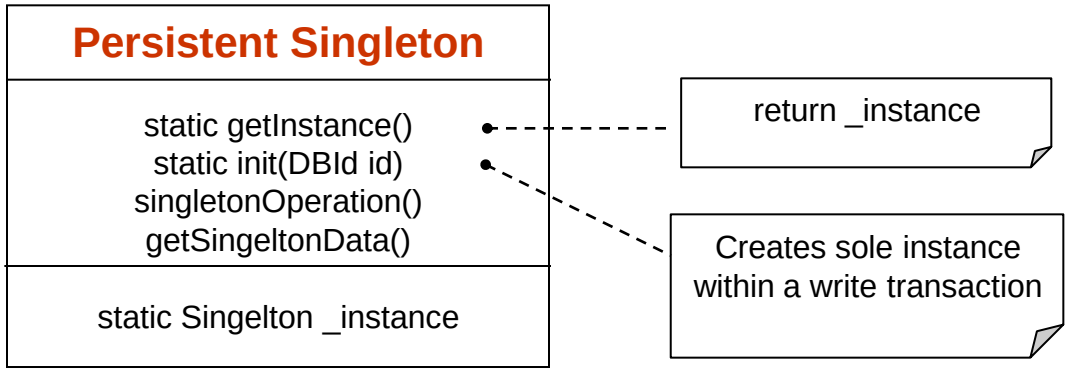
- Intent

  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

Persistent Singleton: structure



- Participants
  - **Persistent Singleton**
    - Defines a `getInstance()` method to return sole instance from within the database
    - May provide `init()` method for creating the sole instance in a write transaction

Persistent Singleton: consequences

- **Consequences**
  - Cross-transactional validity of _instance
    - Is the C++ pointer valid across transactions?
    - Can the singleton be access outside a txn?
  - Database reference access
    - Does the caller pass in a DB reference to `getInstance()` everytime?
  - Singleton creation requires write transaction
    - Is correct operation in read transactions guaranteed?
  - Persistent singleton scope
    - Which database contains ‘the’ singleton?

Persistent Singleton: 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()
        {
                // Checking initalization status
                assert(_instance != 0);
                // Checking that we're in a txn
                assert(os_transaction::get_current());
                return _instance;
        }
...
```

No DB pointer passed in here

Any txn works here either read or write

Persistent Singleton: class

```
...

    // 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()
    {
        return _fooExt;
    }
};
```

Persistent Singleton: constructor

```
Singleton::Singleton()
{
        // Checking that the singleton instance is persistent
        assert(objectstore::is_persistent(this));

        // Creating payload objects in their own cluster
        os_segment* seg = os_segment::of(this);
        os_cluster* clr = seg->create_cluster();
        _fooExt = new (clr, ts< os_Array<Foo*> >())
                              os_Array<Foo*>();
}
```

Persistent Singleton: create()

```
void Singleton::create(os_database* db)
{
        assert(db != 0);
        bool mustCommit=false;
        os_transaction* txn = os_transaction::get_current();
        if(!txn){
                // No txn detected, so starting update txn
                txn = os_transaction::begin();
                mustCommit=true;
        }
        else{
                // Checking that current txn is an update txn
                assert(txn->get_type() == os_transaction::update);
        }
        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);
                os_segment* seg = db->create_segment();
                seg->set_comment(buf);
                // Creating persistent singleton, and attach to root
                Singleton* only = new (seg, ts<Singleton>())
                        Singleton();
                root = db->create_root(_rootName);
                root->set_value(only);
        }

...
```

Update txn required so object can be committed to the database

Persistent Singleton: create()

```
        ...

        // Initializing persistent singleton from database
        _instance = (Singleton*) root->get_value();
        assert(_instance != 0);
        if(mustCommit)
        {
             // Commiting the update txn started during singleton create
             txn->commit();
        }

}
```

Persistent Singleton: init()

```
bool Singleton::init(os_database* db)
{
        assert(db != 0);
        bool ret=false;
        bool mustCommit=false;
        os_transaction* txn = os_transaction::get_current();
        if(!txn){
                // No txn detected, so starting read-only txn
                txn = os_transaction::begin(os_transaction::read_only);
                mustCommit=true;
        }
        os_database_root* root = db->find_root(_rootName);
        if(root)
        {
                // Initializing persistent singleton from database
                _instance = (Singleton*) root->get_value();
                assert(_instance != 0);
                ret=true;
        }
        if(mustCommit)
        {
                // Committing the read-only txn started
                // during singleton init
                txn->commit();
        }
        return ret;
}
```

Persistent Singleton

- Summary
  - Update clients initialize the singleton with the `create()` method
  - Read-only clients initialize the singleton with the `init()` method
  - Thereafter all clients can call `getInstance()` from inside any txn
    - no database reference required for this call
  - This makes the singleton very reliable and easy to use

Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

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

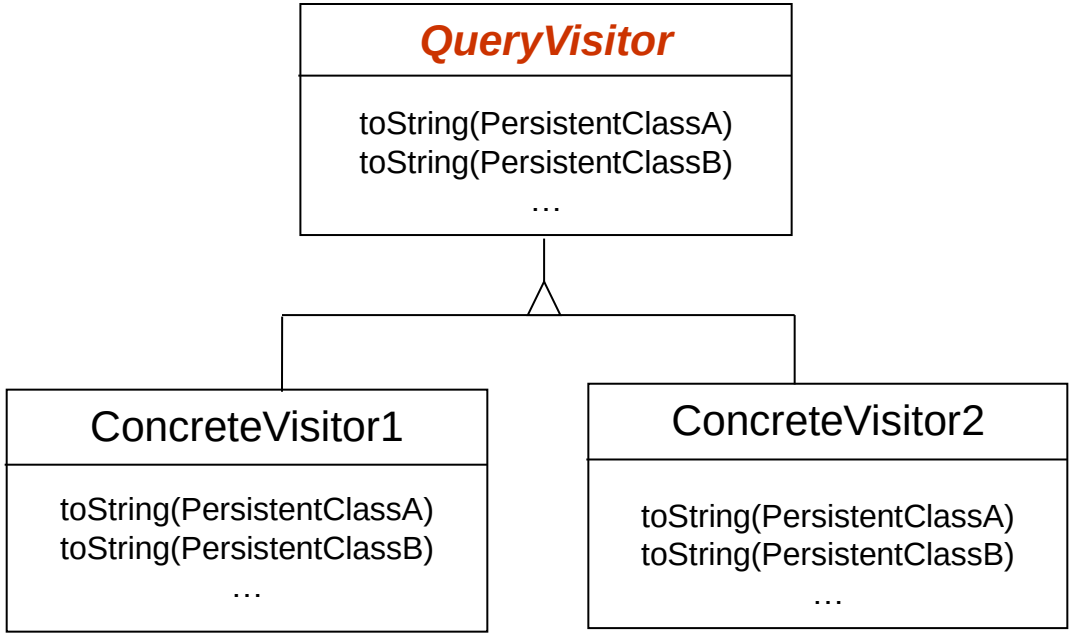
  This is a variant of the standard visitor as described in GoF specifically dealing with OODB queries. Consider a persistent object model held in a database from which you need to render some subset into XML to load into another system. By providing an `accept(QueryVisitor v)` method on persistent objects, 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.

Query Visitor: applicability

Applicability

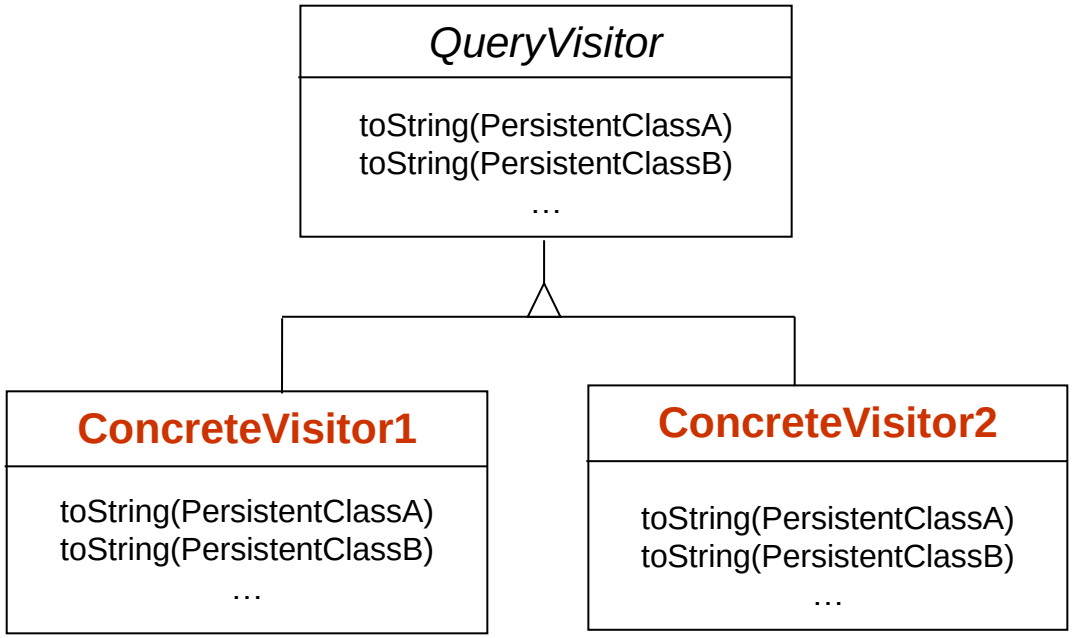
- A persistent object structure contains many classes of objects with differing interfaces, and you want to perform queries that depend on the concrete classes.
- There are several distinct and unrelated query result formats, such as XML, comma delimited tuples, HTML etc. and you want to avoid polluting 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.

Query Visitor: structure



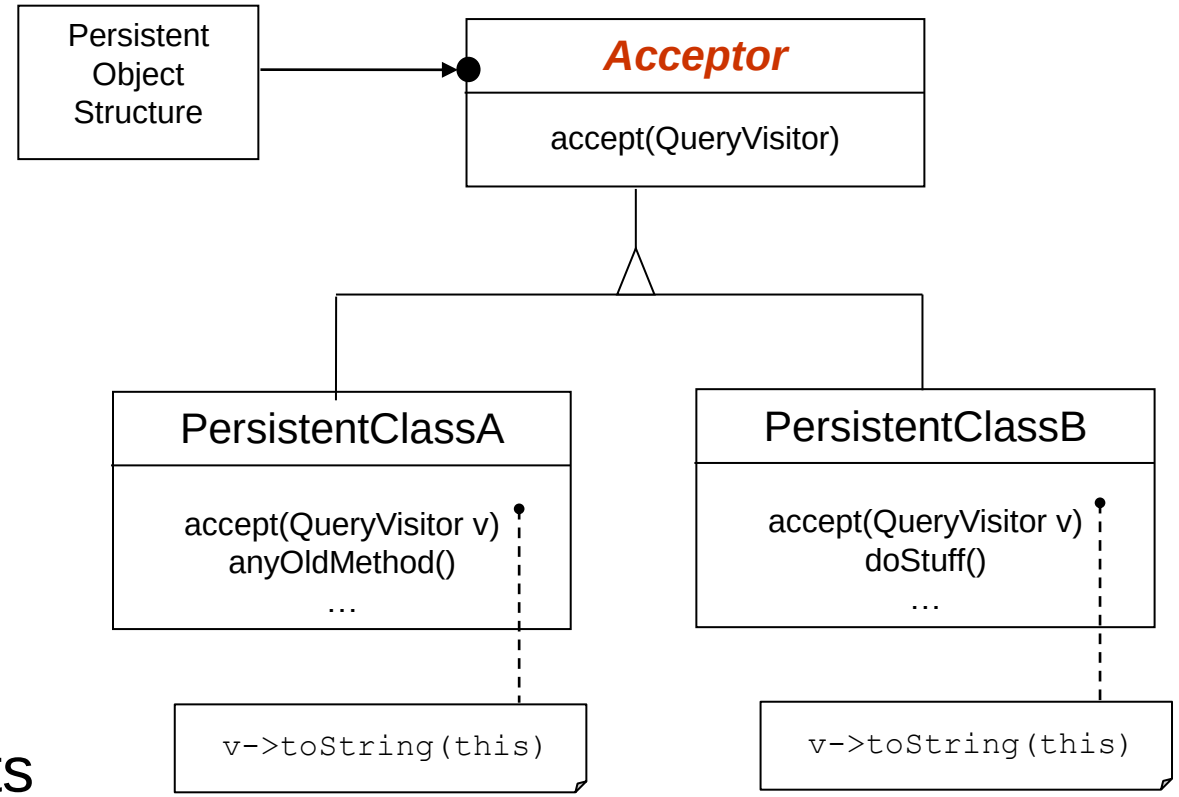
- Participants
  - **QueryVisitor**
    - Declares `toString()` for each persistent class
    - Signature determines how call-backs are dispatched

Query Visitor: structure



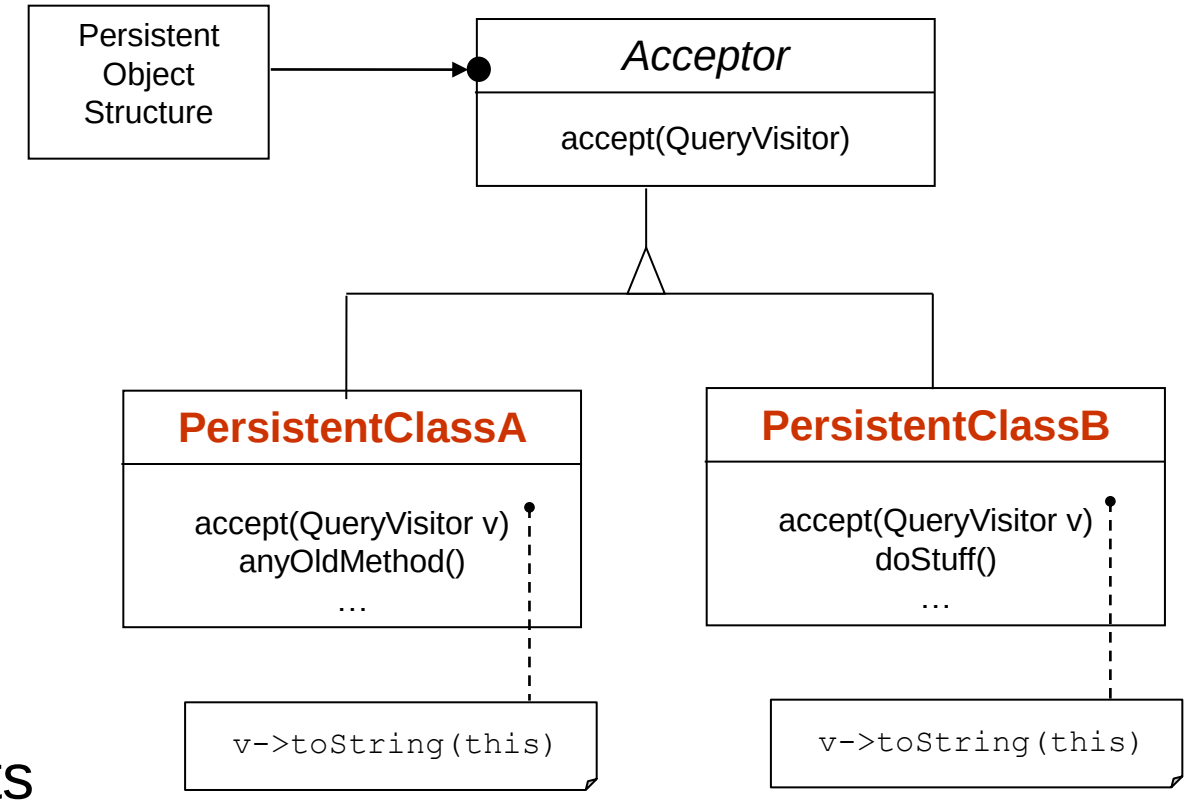
- Participants
  - **ConcreteVisitor**
    - Implements all overloadings of `toString()`
    - Encapsulates rendering code in one class
    - Can accumulate state during traversal

Query Visitor: structure



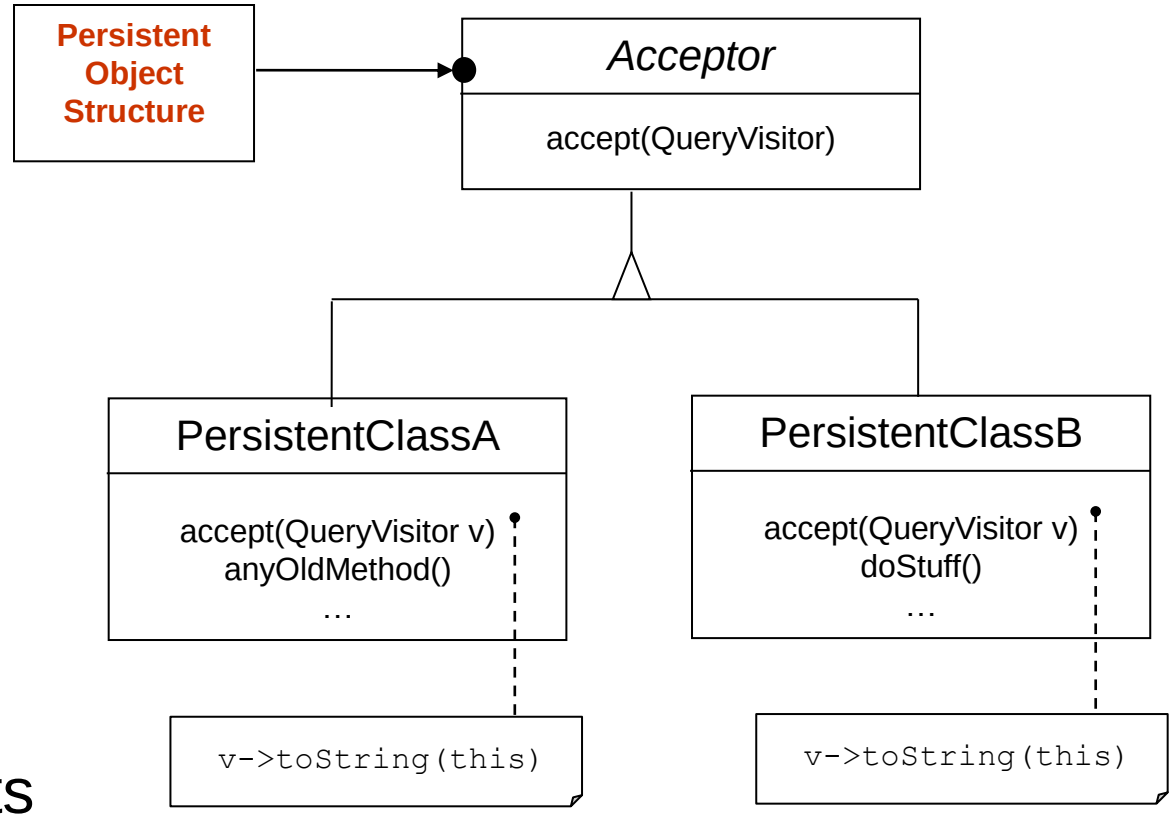
- Participants
  - **Acceptor**
    - Declares `accept()` that takes a QueryVisitor as an argument

Query Visitor: structure



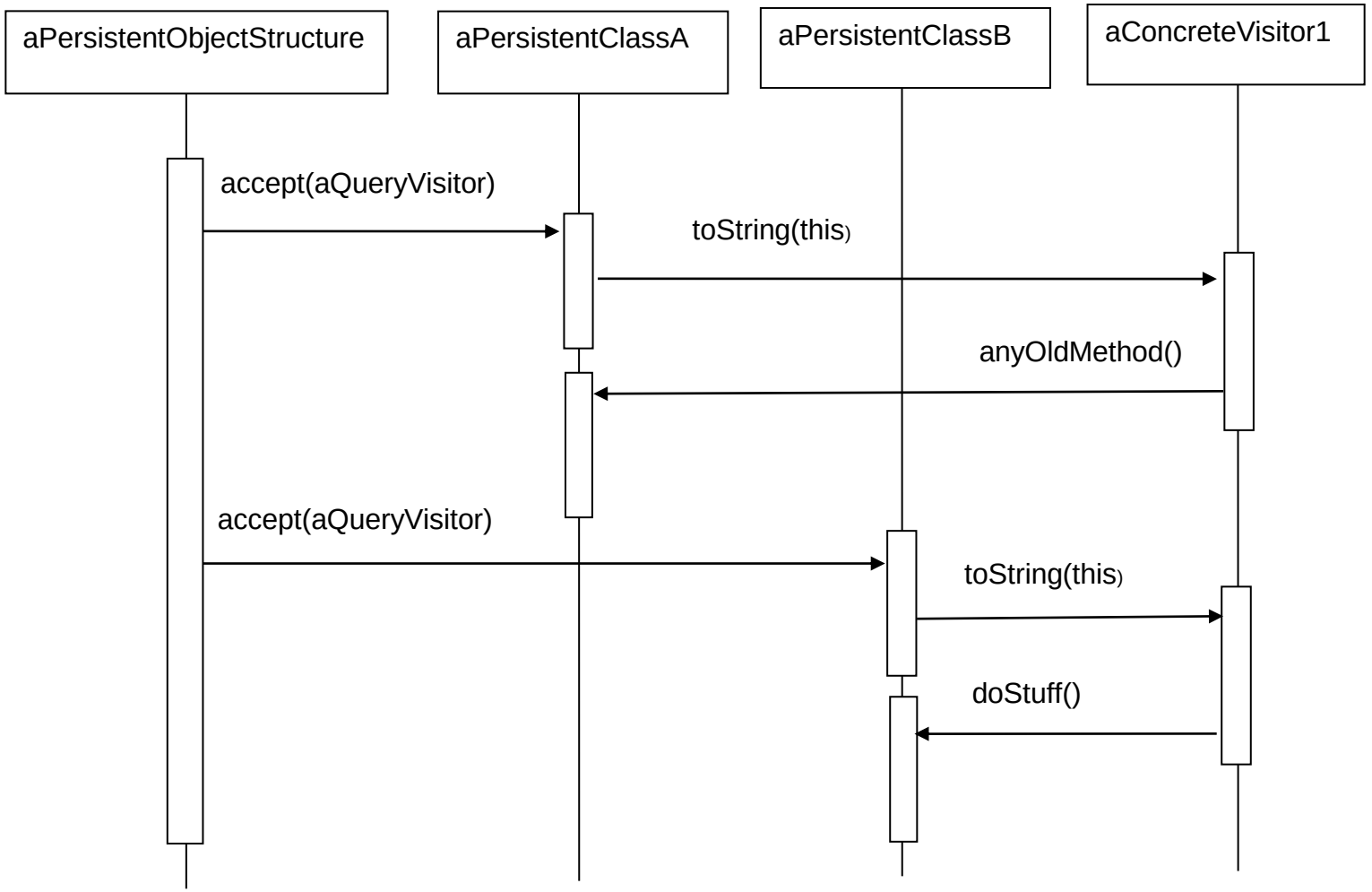
- Participants
  - **PersistentClass**
    - Implements `accept()` method
    - Usually calls toString() method on passed QueryVisitor

Query Visitor: structure



- Participants
  - **Persistent Object Structure**
    - Can enumerate its elements
    - May provide high-level iteration interface

Query Visitor: collaborations



Query Visitor: consequences

- **Consequences**
  - Adding new result formats is easy
    - Define new query format by adding a new visitor
  - Encapsulates query specific code
    - Behaviour and state concerned with result formats isn't distributed throughout the persistent model
  - Adding new persistent objects to model is harder
    - They extend the acceptor interface
      - Visitor must overload toString() for every new class
      - Every ConcreteVisitor must implement it

Query Visitor: consequences

- **Consequences**
  - Can visit across hierarchies
    - Iterator cannot work across heterogeneous collections
    - Visitor's double-dispatch gets round this limit
  - Accumulate State during traversal
    - Better than using globals for accumulation
  - Breaking encapsulation
    - Assumes public interface of persistent classes is sufficient for visitor's purposes
    - Sometimes persistent class needs to expose internal state to visitor
      - This can compromise encapsulation

Query Visitor

- **Summary**
  - Key question is:
    - Are you more likely to change the query result format, or introduce new query formats, rather than change the persistent object model itself?
  - If the formats change more often use the Query Visitor

Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

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.

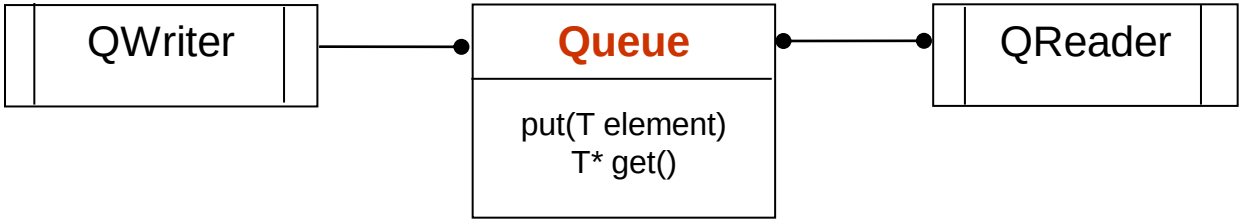
Motivation

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 persistent queue until a dispatcher can deal with it.

Persistent Queue: applicability

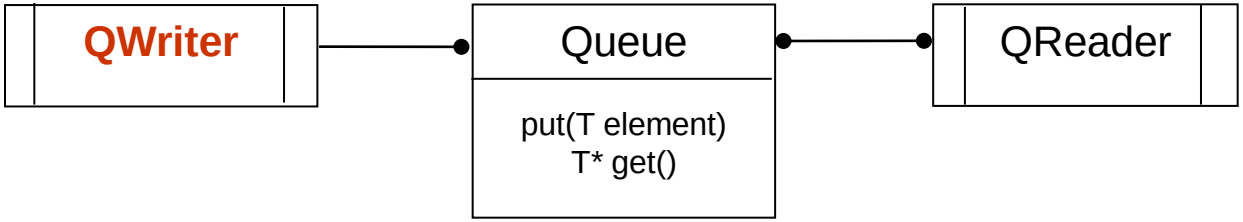
- Applicability
  - Objects must be written to the database by multiple writers in parallel and performance requirements demand that the writers must not block each other, or any readers
  - There is a data ‘processing pipeline’ architecture where objects move through a series of states before they reach 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.

Persistent Queue: structure



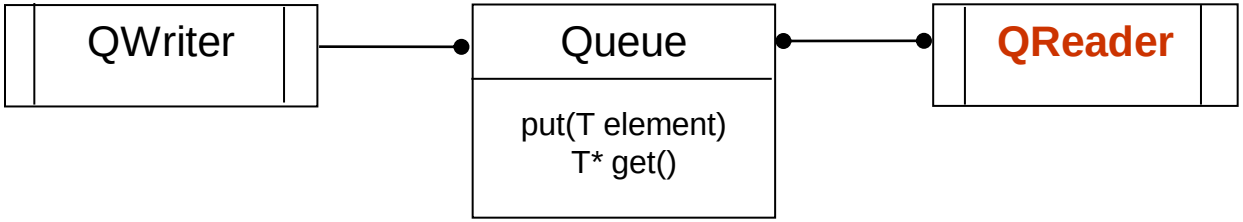
- Participants
  - **Queue**
    - Holds objects written into the queue by the writer
    - Each queue is written into by a single writer
    - Objects remain in the queue until all readers have read the objects off the queue
    - Queues can be named

Persistent Queue: structure



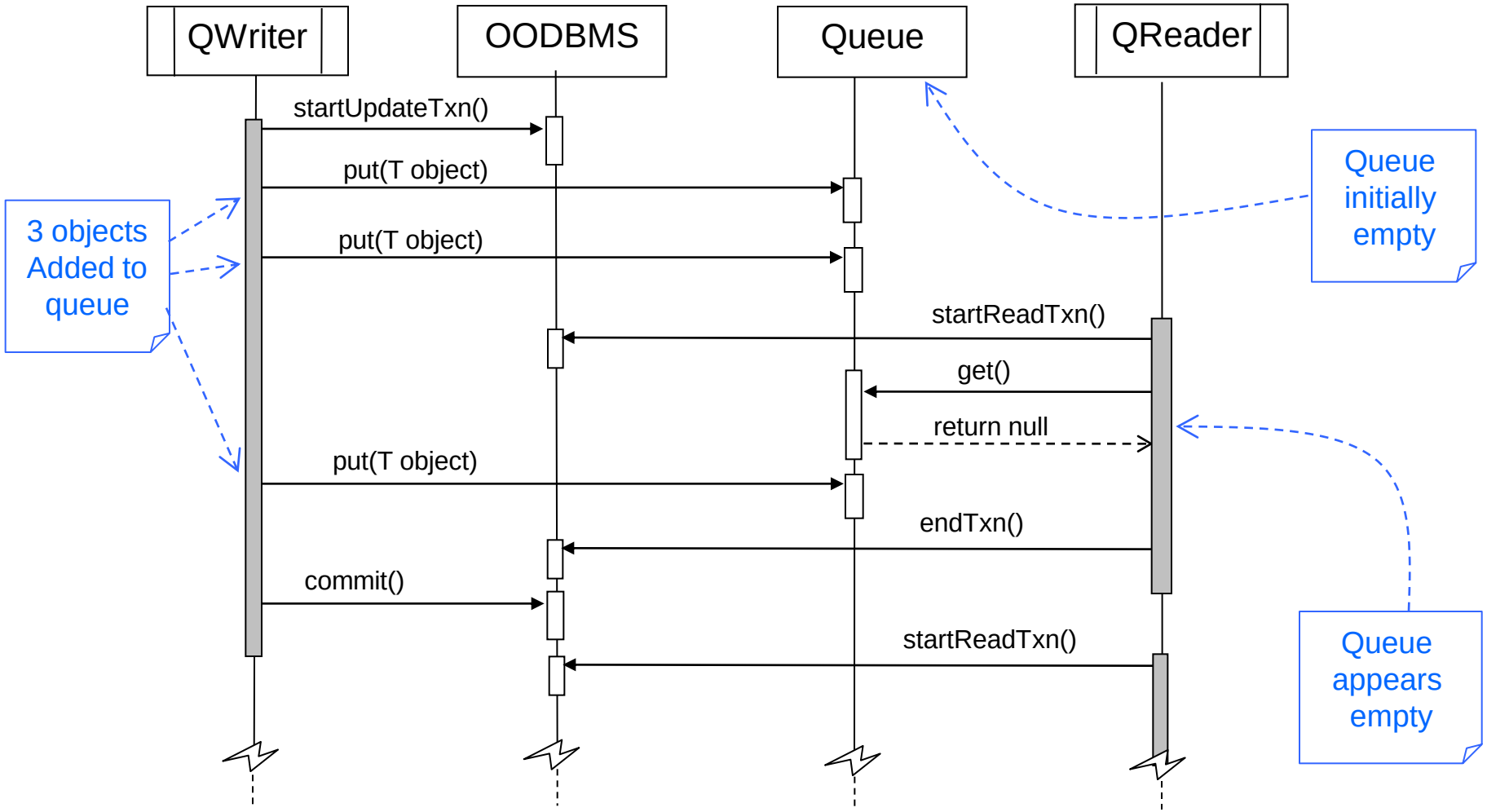
- Participants
  - **QWriter**
    - Supply objects that are used by the queue readers
    - Works asynchronously from the readers, and from one another if there are multiple queue writers
    - If all readers are busy, the writer can still put an object onto the queue
    - Can write to multiple persistent queues

Persistent Queue: structure

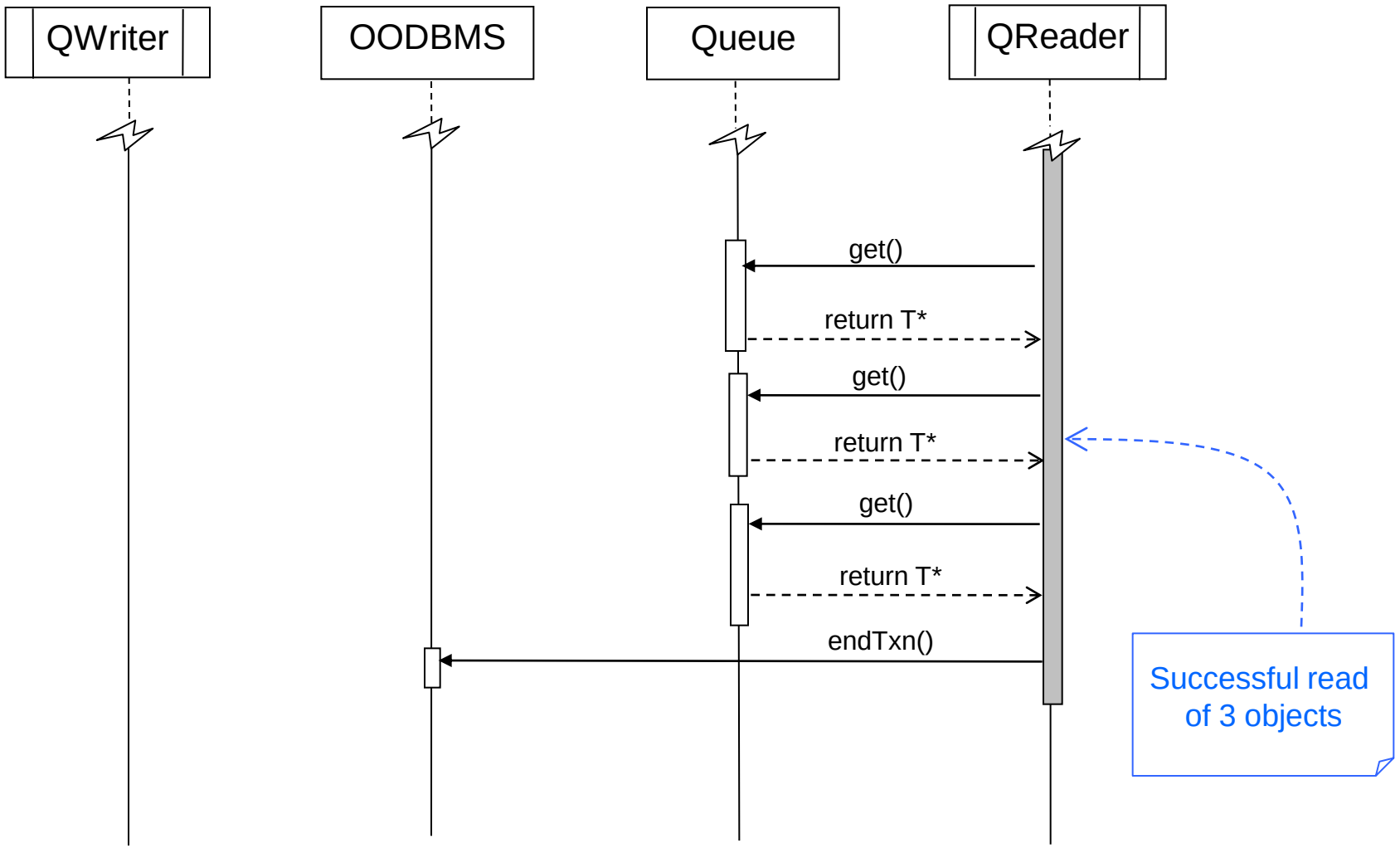


- Participants
  - **QReader**
    - Readers consume objects produced by writers
    - Read objects off the queues in order of arrival
    - The reader periodically polls for objects on the queue; If queue is empty the reader can do other work, or pause, until there an element arrives
    - Readers can read multiple queues
    - Queues can have read priorities

Persistent Queue: collaborations



Persistent Queue: collaborations



Persistent Queue: consequences

- **Consequences**
  - Queue reads and writes require disc access
    - Compromises performance compared to transient queues
  - Throughput increased by batched reads/writes
    - Single OODB txn per batch
    - Further increased using a pool allocator
  - Queues can be very large
    - Only bounded by disc space
  - Readers and writers are interruptible
    - will continue from where they left off when restarted

Persistent Queue: consequences

- **Consequences**
  - Object delivery is guaranteed
    - All objects written after a reader attaches to a queue will be delivered
  - Readers and writers need never block
    - Multiple readers can read from a single queue without blocking
    - A single writer can write to a queue without blocking
  - Queues preserve arrival order
    - Cannot easily be preserved between multiple queues

Persistent Queue: consequences

- **Consequences**
  - Complex queue topologies can be implemented
    - OODB clients can read/write to many queues arranged in chains or webs of processing
    - They can all be non-blocking
  - Queues can be compile-time type-safe
    - No RTTI; reader knows exactly what object has arrived
    - For heterogeneous elements either:
      - Use several queues
      - Provide efficient `getType()` method
  - Best when elements are small PODs
    - Pool allocator works well

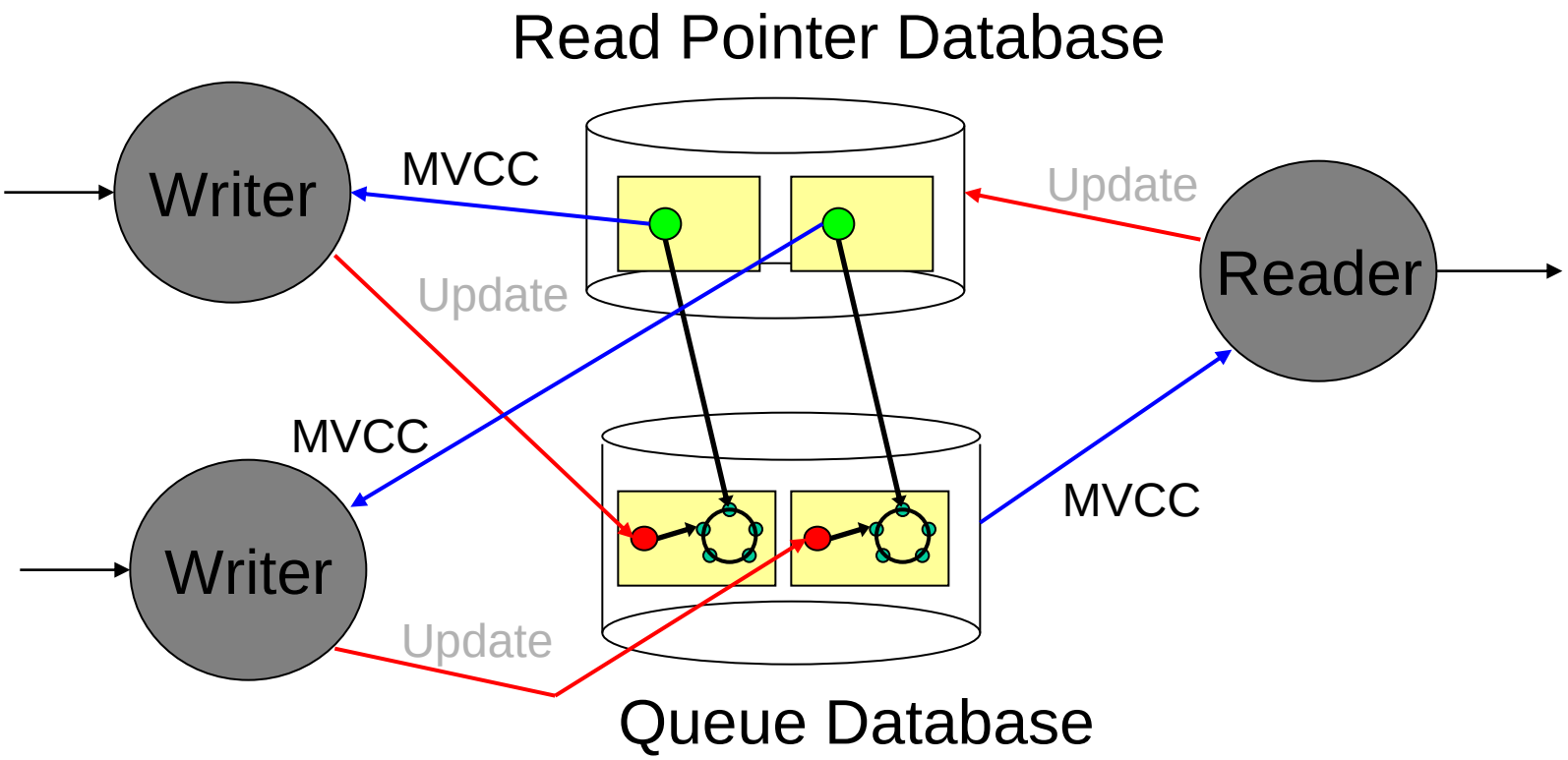
Persistent Queue: consequences

- **Consequences**
  - Can arrange for priority queues
    - Readers can read important queues first
    - High-priority elements get read first
    - Danger of low-priority queue read starvation
  - Queues can be named or anonymous
    - Named queues can be globally visible using a well-known-name
    - Can be anonymous and buried within a service infrastructure

Persistent Queue: consequences

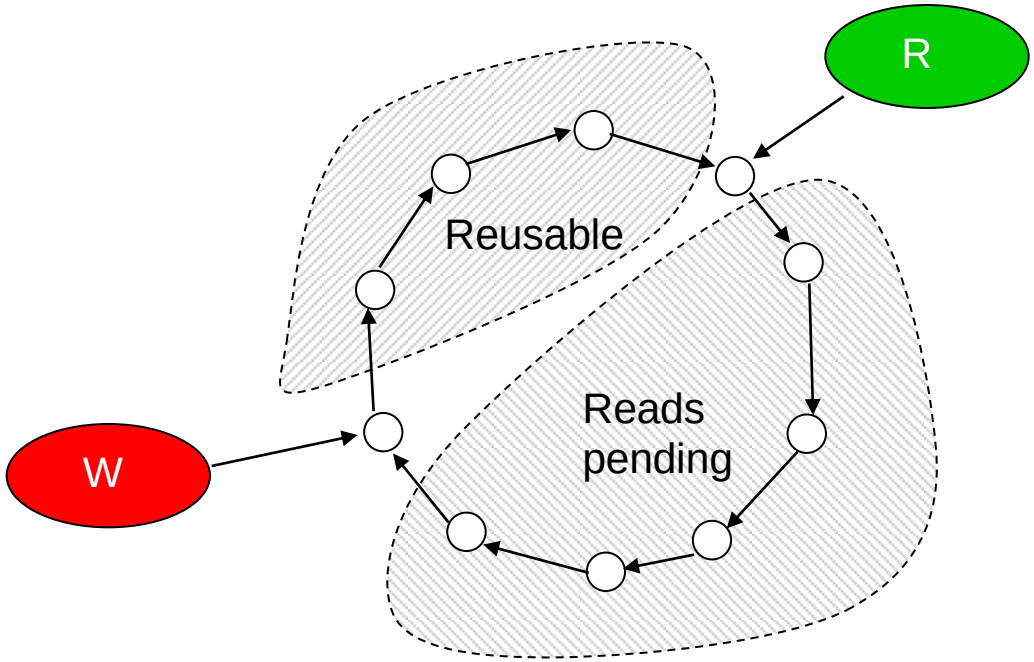
- **Consequences**
  - Must test queues for long periods
    - Read starvation
      - Reading multiple queues in the same order, some get neglected
      - Writer must create nodes in these queues so DB size increases
    - Read/write imbalance
      - Writers are writing faster than the readers are reading
      - Queues expand and average latency increases
    - Interrupted reader fails to catch up
      - If read/write batch sizes are not set correctly
      - Reader batch size should be larger than writer by some percentage

Persistent Queue: implementation



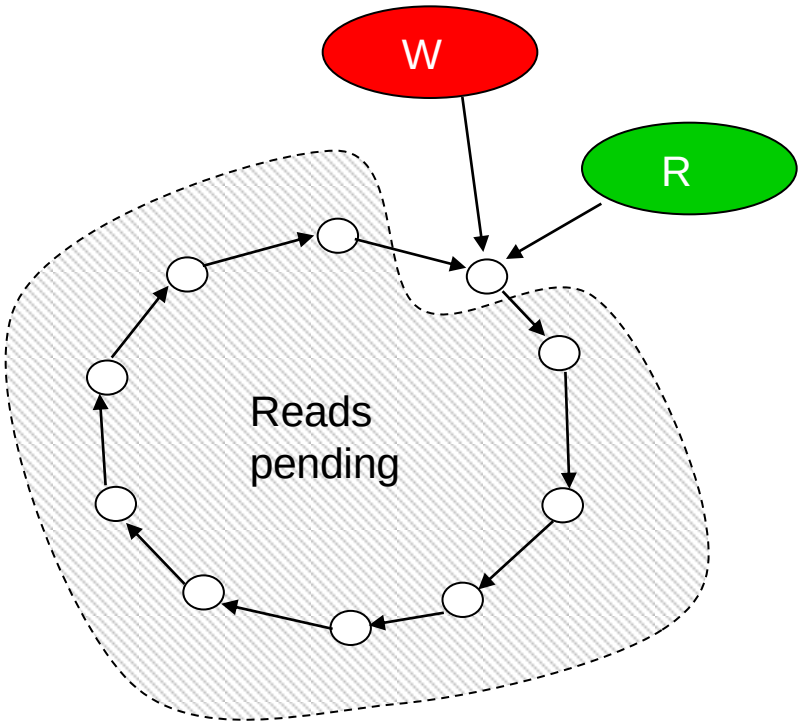
- Persistent non-blocking queue requires two ObjectStore databases

Persistent Queue: implementation



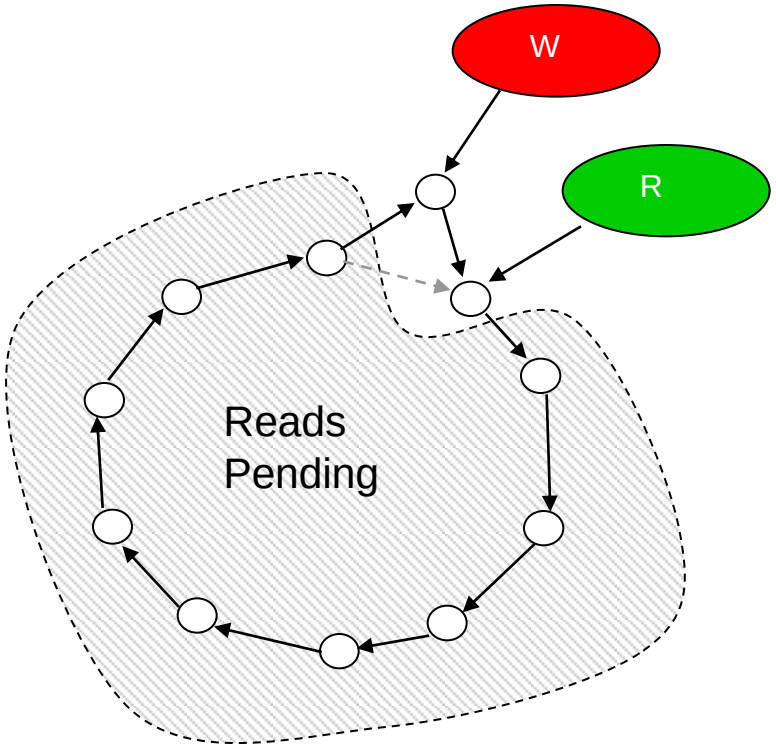
- The queue ‘ring’ data structure
  - Intrusive singly-linked list
  - Ring has two sections:
    - Objects waiting to be read
    - Objects that have been read

Persistent Queue: implementation



- Writing into the queue
  - Move to next element and over-write…
  - …until write pointer equals the read pointer

Persistent Queue: implementation



- When write pointer equals the read pointer
  - Insert by creating new list nodes
  - Meanwhile reader can read objects pending

Persistent Queue: element template

```
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;
};
```

Use C++ template for type-safe queue

Embed T rather than use T*
Save space of extra pointer.
Transient objects will be copied into queue bitwise or via assignment operator.

The intrusive linked-list is implemented here so the elements themselves have no need to

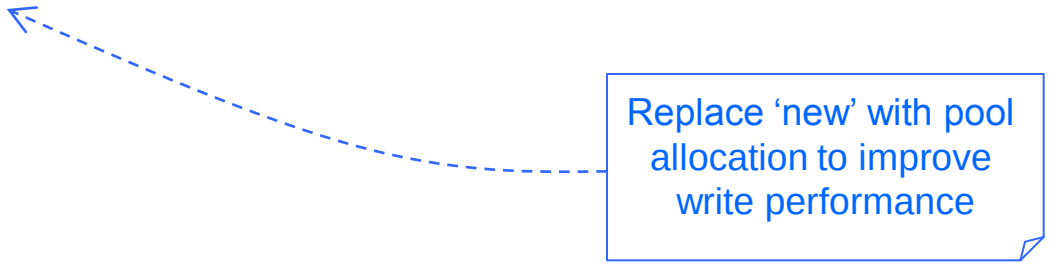
Persistent Queue: PQ template

```
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;
          }

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

Takes reference to the remote read pointer as its only argument

Persistent Queue: PQ template

```
// 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;
}
```



Agenda

- Introductory Comments
- Bespoke Indexes
- Persistent Singleton
- Query Visitor
- Persistent Queue
- Conclusion

Conclusion

- Using an OODBMs allows GoF style patterns to be applied directly in the database
- Persistent OO patterns can help programmers to:
  - Implement bespoke index structures to optimally support critical use-cases
  - Exploit particular OODBMs features to the full
  - Avoid pitfalls
- ObjectStore features, such as *really* transparent support of the C++ language, enable programmers to be very creative in this area.

For More Information

- Papers
  - Under **web.progress.com/docs/media-coverage/**
    - **bespokeprog.pdf**
  - Under **web.progress.com/docs/whitepapers/public/**
    - **/objectstore/persistent_oodb_patterns.pdf**
    - **/objectstore/objectstore_db_architecture.pdf**
- Contact Us
  - Adrian Marriott, Independent Consultant
    - adrian@logos-software.co.uk
  - Jeff Wagner, Product Manager
    - jwagner@progress.com

www.progress.com/objectstore

Questions ?