Patterns for Program Query Optimisation

David J. Pearce and **James Noble**
Computer Science
Victoria University of Wellington
New Zealand
{djp,kjx}@mcs.vuw.ac.nz

Abstract

Operations on data can be classified as either *queries* or *updates*. Modern object-oriented programming languages require classes/interfaces to support a predefined set of queries. This presents a challenge for software designers, since a fixed interface can severely restrict the opportunities for optimisation. In this paper, we present two common patterns for optimising queries. The first requires specific knowledge of which query to optimise beforehand, whilst the latter provides more leeway in this regard. These patterns are commonly occurring in software, and we find numerous instances of them within the Java standard libraries.

Introduction

Abstraction — separating the "what" from the "how" — is a central theme of computer science [3, 13]. Take for example, abstract data types. These have long been studied (see e.g. [9, 5, 2]) and most modern languages come with libraries providing numerous ADTs and their implementations. Well-known examples include the Standard Template Library [18], the Java Collections Library [19] and the Boost Library [1]. Typical ADTs include `Map`s, `Set`s and `Sequence`s, and are backed by implementations such as `HashMap`, `TreeMap`, `HashSet`, etc to name but a few. These provide a good degree of separation between the "what" and the "how". In this paper, we present two common patterns for optimisation which arise from separating these two things. Perhaps unsurprisingly then, several known uses of these patterns can be found within common ADT libraries.

We take the view, as others have [10], that all operations on data can be classified as either *queries* or *updates*. According to this viewpoint, Abstract Data Types provide a fixed set of queries (the interface) optimised for some specific purpose. As an example, consider a `Map` ADT. A `Map` can be viewed as a `Set` of pairs which has been optimised for a specific query: the `get()` operation. Thus, the `Map` interface is similar to that of `Set`, but includes an additional `get()` method. This allows for implementations (e.g. `HashMap`) which can perform the query more efficiently than simply enumerating every pair and selecting a match. In this we see a common pattern; namely, that an interface (i.e. `Set`) has been taken and optimised for a specific query (i.e. `get()`), resulting in a new interface (i.e. `Map`). We refer to this pattern as **Restricted Query Optimisation (1)**. Here, restricted indicates that a specific query has been optimised, rather than a more general class of queries. Thus, optimising additional queries requires changing the interface again. For example, to optimise a query giving the reverse of `get()` (i.e. to find the keys that map to an object), we must extend the interface with another method, such as `reverseGet()`.

Forces

Each of the patterns trades-off different forces. The primary force being resolved is *performance*. These patterns all increase speed — *time performance* — by increasing the *space usage* of the program. More specifically, time performance is resolved in situations when the *query/update ratio* is high, since the benefits from optimising a query in this case are far greater. A second benefit is that these patterns (especially **Generalised Query Optimisation (2)**) also provide *flexibility*, in that designs become more resilient to future performance requirements. However, the patterns introduce *complexity* into the program — particularly, into the ADT implementations themselves — reducing *readability* and *maintainability*. Furthermore, the patterns are difficult to *apply*, since the expected *query/update ratio* is often hard to know concretely and may vary between program runs and/or clients (for frameworks).

1 Restricted Query Optimisation

How do you design an interface optimised for a specific query?

Suppose we are implementing a data structure on which there are *several queries that can be performed* and, furthermore, *we know a priori which of these queries we wish to optimise*. For example, in a drawing program we might have a canvas containing the shapes that it draws:

```
class Canvas {
 private List<Shape> shapes;
 ...
 public List<Shape> getShapes() { return shapes; }
}
```

Now, when our program receives a mouse click, it searches the list returned by `getShapes()` looking for those whose bounding box contains the click. If we have many shapes in the list, this operation will be a performance bottleneck as the total number of shapes will dwarf the number we're interested in.

Forces

- You need to increase the *time performance* of the program
- You can afford to trade increased *space* to reduce *time*
- You wish move complexity from client code to server objects.
- You know which queries the program will perform

Therefore: *Add a new method optimised for the particular query in question.*

Since we have identified a specific query to optimise, it is sensible to add a specific method for that query. Doing so enables its optimisation, even if this is not implemented immediately. In this case, a method such as `getEnclosingShapes()` would be appropriate.

Example

To illustrate, we present an example from [20] which is based on a real-world application called *Robocode* [11]. The Robocode game is written in Java and pits user-created simulated robots against each other in a 2D arena. The game has a serious side as it has been used to develop and teach ideas from Artificial Intelligence [6, 7, 12].

A Robocode Battle object maintains a private list of Robots, with an accessor method that returns all Robots in the battle:

```
class Battle {
  private List<Robot> robots;
  ...
  public List<Robot> getRobots() { return robots; }
}
```

During each turn of the game, robots scan their field-of-view within the battle arena to locate other Robots to attack. The code implementing this iterates the list of robots, selecting those which are alive and within the robots field-of-view as follows:

```
class Robot {
  public int state = STATE_ACTIVE;
  public boolean isDead() { return state == STATE_DEAD; }
  public void die() { state = STATE_DEAD; }
  ...
  private void scan() {   // Scan field-of-view to find robots
    for(Robot r : battle.getRobots()) {
      if(r!=null && r!=this && !r.isDead() && r.intersects(...)) {
        ....
}}}}
```

To optimise programs like Robocode, programmers focus on methods like `scan()` that are called repeatedly. A common and effective approach is to cache intermediate results which, in this case, are the sub-collection(s) being frequently traversed. For example, the programmer might know that, on average, there are a large number of dead robots. To avoid repetitively and needlessly iterating many dead robots in `scan()`, he/she might maintain a cache — a list of just the "alive" robots — as follows:

```
class Battle {
  private List<Robot> robots;           // master list of all robots
  private List<Robot> aliveRobots; // cached list of alive robots
  ...
  public List<Robot> getRobots() { return robots; }
  public List<Robot> getAliveRobots() {
    return aliveRobots;
}}
```

Then, each `Robot` can iterate the list of alive robots, without needing to check whether each is alive or dead:

```
class Robot {
  ...
  private void scan() {
    for(Robot r : battle.getAliveRobots()) {
      if(r!=null && r!=this && r.intersects(...)) { ... }
}}}
```

Here, `aliveRobots` is a sub-collection of `robots` containing only those where `!isDead()` holds. Thus, the for-loop in `scan()` no longer needlessly iterates over dead robots. Since (after the game has been running a while) more Robots are typically dead than alive, this reduces the time taken for the loop at the cost of extra memory (the cache).

When the source collection(s) of a query are updated (e.g. by adding or removing elements), or an element of a source collection is itself updated, any cached result sets may become invalidated. Traditionally, encapsulation is used to prevent this situation from arising, by requiring all updates go via a controlled interface. Thus, updates to a collection can be intercepted to ensure any cached result sets are updated appropriately. To illustrate, consider a simple `addRobot()` method for adding a new robot to the arena, where a cache is being maintained explicitly:

```
class Battle {
  private List<Robot> robots, aliveRobots;
  ...
  public List<Robot> getRobots() { return robots; }
  public List<Robot> getAliveRobots() {
    return aliveRobots;
  }

  public void addRobot(Robot r) {
    robots.add(r);
    if(!r.isDead()) { aliveRobots.add(r); }
  }

  public void robotDied(Robot r) {
    aliveRobots.remove(r);
 }}
```

Here we see that, when a robot is added via `addRobot()`, the `aliveRobots` list is *incrementally updated* to ensure it remains consistent with the `robots` collection.

There are several issues to consider when implementing this pattern:

- **Dependency Tracking.** An important issue regarding *incrementalisation* (i.e. incremental updating of cached result sets) is that updates affecting cache consistency must be intercepted. While encapsulation does help in this regard, it is not always sufficient. For example, the `aliveRobots` cache may become inconsistent if a `Robot` instance is mutated outside of the `Battle` class's knowledge. To alleviate this issue, we provided a `robotDied()` method which must be explicitly called when a robot dies. Another solution would be, where possible, to restrict the `Robot` interface such that a `Robot` may only die via the `Battle.robotDied()` method.

- **Query/Update Ratio.** An important issue in deciding whether or not to optimise a specific query is its *query/update ratio*. This is because there is typically a cost associated with incrementally maintaining a cached result set: when the number of updates affecting a result set is high, compared with how often it is actually used, it becomes uneconomical to cache that result set. In cases when the expected query/update ratio is not known, or is known to vary greatly, one can also use a mechanism based on *caching heuristics*. For example, by dynamically monitoring the query/update ratio and using this data to decide when to begin caching, and when to stop.

Consequences

- **Benefits.** *Program performance* can be significantly improved, often by several orders of magnitude or more. Furthermore, the resulting system is more *flexible*. That is, even if the query under consideration is not time-critical, it may become so in the future. Thus, by providing an extended interface, the implementation can be optimised at a later date as needed.

- **Liabilities.** Explicitly maintaining extra collections has several drawbacks. Clearly, caching increases the *space* requirements for the program. Furthermore, it can be difficult to introduce caches when the interface of the providing object (i.e. `Battle`) is fixed (e.g. it's part of a third-party library, and/or the source is not available, etc). The optimisation also reduces *readability* and *maintainability* as the source becomes more cluttered. Maintaining cached collections is also rather tedious, since they need to be updated whenever the underlying collection or the objects in those collections are updated — whenever a new Robot "spawns" into the game, or whenever an alive Robot dies. Finally, code to maintain these optimised collections must be written anew for each collection. For example, Robocode's `Battle` class also maintains a list of `Bullets` and employs a loop similar to `scan()` for collision detection. Programmers can introduce a sub-collection to cache live bullets, but only by duplicating much of the code necessary for the sub-collection of live Robots.

Known Uses

This pattern encompasses a very common optimisation that is particularly prevalent in the design of object-oriented collection libraries. In the Java Collections Library [19], a good example is `java.util.List`. This provides a `get(int)` query for accessing the i^{th} element in the list. This query is not strictly necessary from a functionality perspective, since the existing `Collection` interface is sufficient for performing this operation (via explicit enumeration of elements). Thus, the sole purpose of including this query is to enable optimisation; in particular, the `ArrayList` implementation is able to provide constant time access to elements via this query.

Another common example found in collection libraries is the `Map` interface. By regarding a `Map` as a `Collection` of pairs (as is done, for example, in the C++ Standard Template Library [18]), it becomes apparent that, again, the `get(Key)` query is provided purely to enable optimisation. This is because this query can be implemented by explicit enumeration of elements using the existing `Collection` interface.

A similar situation arises in libraries for manipulating graphs, such as the Boost Graph Library [17] and JGraphT [8]. Such libraries provide some kind of `Graph` interface, typically backed by `AdjacencyList` and `AdjacencyMatrix` implementations. As with `Map`, the `Graph` interface is essentially a `Collection` of pairs with additional queries for enabling optimisation. A good example of such is the `edges(Node)` query which returns those edges adjacent to the given node; again, this information can be determined by explicit enumeration of the elements using the existing `Collection` interface. Thus, the `edges(Node)` query is provided purely for the purposes of optimisation.

The Relationship Aspect Library [15, 14], provides another interesting example. This library is designed for representing the *relationships* between objects in an object-oriented program. Since the web of relationships is essentially a graph, the `Relationship` interfaces provided by this library are, in fact, similar to those found in common graph libraries. As such, the `Relationship` interface provides `from(Object)`/`to(Object)` queries to enable efficient navigation of the object graph.

Finally, numerous real-world applications such as Robocode [11] embody this pattern as it provides a fundamental optimisation technique.

2 Generalised Query Optimisation

How do you design an interface optimised for an unknown set of queries?

Suppose we are implementing a data structure *on which there are several queries that can be performed*, but *we don't know specifically which ones to optimise*. This may arise, for example, if the set of queries requiring optimisation varies between program runs, or between clients (if we are implementing a library or framework).

Forces

- You need to increase the *time performance* of the program
- You can afford to trade increased *space* to reduce *time*
- You wish move complexity from client code to server objects.
- You do not know which queries the program will perform

Therefore: *Provide a single method that supports all possible queries.*

Since we are unsure what query should be optimised, it makes sense to leave our options open. That is, to design an interface for which optimised implementations can be provided at a later date.

Example

For example, suppose we want to design a collection API supporting different kinds of query optimisation, such as the following:

```
Collection<String> col = ...;
...
for(String s : col) {
  if(s.equals(‘‘Dave’’)) {
    ...
} }
```

Optimising this code with the Java collections API is essentially impossible since the `Collection` interface provides no appropriate method. To optimise such code, we need to include a method supporting a wide-range of queries. The design of our collections API might look like this:

```
interface UnaryFun<T> { boolean select(T x); }

interface MyCollection<T> {
  ...
  MyCollection<T> filter(UnaryFun<T> f);
}
```

Here, the `filter` function is provided to capture the set of queries we wish to optimise. This accepts a unary function that selects the required elements of the collection. Thus, our client code now looks like this:

```
MyCollection<String> col = ...;
...
UnaryFun<String> f = new UnaryFun<String> {
  public boolean select(String x) { return x.equals(‘‘Dave’’); }
};
for(String s : col.filter(f)) {
  ...
}
```

The advantage of this design is that we can now provide optimised implementations of the `filter` method. Such an implementation might look like this:

```
class CachingCollection<T> implements MyCollection<T> {
  private HashMap<UnaryFun<T>,MyCollection<T>> cache = ...;
  private MyCollection<T> base = ...;
  ...
  public MyCollection<T> filter(UnaryFun<T> f) {
    MyCollection<T> result = cache.get(f);
    if(result == null) {
      result = base.filter(f);
      cache.put(f,result);
    }
    return result;
} }
```

Here, we see that a cache is maintained for the results of a specific query. Thus, when the `filter` function is called again with the same `UnaryFun`, the results can be quickly recalled. As with the *restricted query optimisation*, our implementation should incrementally update its cached result sets, which we refer to as *incrementalisation*. This is done by intercepting all operations which may mutate the underlying collection. For example, the `add` method could be incrementalised as follows:

```
class CachingCollection<T> implements MyCollection<T> {
  ...
  public boolean add(T item) {
    if(base.add(item)) {
      for(Map.Entry<UnaryFun<T>,MyCollection<T>> e : cache.entrySet()){
        if(e.getKey().select(item)) {
          e.getValue().add(item);
      } }
      return true;
    }
   return false;
} }
```

Here, the code iterates the caches available for the different `UnaryFun` objects, and adds the `item` to those which match.

There are several important issues to consider when implementing this pattern:

- **Dependecy Tracking.** A key issue is being certain that *all operations affecting the results of a particular query are intercepted*. Starting with the `base` collection, encapsulation can be used to ensure operations for adding/removing elements are intercepted (as above). However, if the unary functions supplied to `filter` rely on state contained in elements held by the collection, then changes to these objects could clearly put the cache in an inconsistent state. Dealing with this problem is not as easy: one option is to simply require that elements held in the collection do not change state — this is essentially the contract already required by many of Java's `Collection` classes (e.g. `HashSet`); another option is to require the unary functions do not access mutable state of elements held in the collection; yet another option is provide a method on `MyCollection` for indicating that a particular `Object` held in the collection has changed state.
- **Query/Update Ratio.** Knowing when to optimise an individual query is key to successful application of this pattern. This issue becomes more challenging here, since the individual queries (i.e. instances of `UnaryFun`) are unknown a head of time. Memory consumption becomes important, since the number of queries is unbounded; this contrasts with **Restricted Query Optimisation (1)**, where memory (and other resource) utilisation can be easily bounded. To alleviate this issue, a *cache-replacement strategy* can be employed to evict cached results which, for example, are not used frequently. Furthermore, in cases where the expected query/update ratio varies widely between individual queries, additional mechanism may be required identify those which should be optimised (static or dynamic profiling can typically be used for this).

Consequences

- **Benefits.** *Program performance* can be significantly improved, often by several orders of magnitude. The resulting system is also more *flexible*, as optimised implementations can be provided at a later date. This effect is more pronounced here, compared with **Restricted Query Optimisation (1)**, since the range of queries is far greater; one could even imagine implementations which categorise the range of query parameters and optimise them differently.

- **Liabilities.** This pattern trades *time* for *space*, increasing the space required by the program. This optimisation clearly reduces *readability* and *maintainability* for the sake of *performance*. In fact, it may also increase the chance of *program error*, especially when strict rules must be enforced to ensure proper dependency tracking. Furthermore, the benefits are only realised in situations where the *query/update ratio* is actually favourable. However, whether or not this is the case depends upon the programs' usage patterns, which can be difficult to determine ahead of time.

Known Uses

This pattern occurs less frequently in practice than **Restricted Query Optimisation (1)**. This is primarily because the pattern introduces additional *complexity* and many problems are not sufficiently performance critical to warrant its use. Typically it is found buried in the core algorithm(s) of an application that have key performance requirements.

One such example is found in `javax.swing.tree.AbstractLayoutCache` and its subclasses, `FixedHeightLayoutCache` and `VariableHeightLayoutCache`, which are part of the Java standard library. These classes provide parts of the implementation for an expandable *tree view*, as used in many GUIs. The key interface is:

```
abstract class AbstractLayoutCache {
  ...
  // Returns a rectangle giving the bounds needed to draw path.
  public abstract Rectangle getBounds(TreePath path,Rectangle placeIn);

  // Instructs the layout cache that the bounds for path are invalid, and need to be updated.
  public abstract void invalidatePathBounds();
}
```

The `getBounds()` method returns the bounding box for a particular subtree, and this depends upon whether the subtree is expanded, or partially expanded. The `TreePath` identifies the path to the component in question. Thus, calling `getBounds()` on the top-level `TreePath` returns the bounds of the whole tree (which swing needs in calculating layout information). The two subclasses of `AbstractLayoutCache`, are `FixedHeightLayoutCache` and `VariableHeightLayoutCache`. The former does not need to calculate its bounding box as it uses a fixed bounding box, whilst the latter does. The implementation of the latter uses an internal cache to store the bounding boxes of different `TreePaths`, thus implementing **Generalised Query Optimisation (2)** pattern. The interface addresses cache coherency issues in two ways: firstly, `TreePath` is immutable; secondly, the `invalidatePathBounds()` method is provided to signal that the bounds for a particular `TreePath` are now invalidated.

Many instances of this pattern are found in the Java `Collection` interface. A good example is the `ContainsAll` method:

```
interface Collection<T> {
  ...
  // Returns true if the collection contains all of the elements in the specified collection
  boolean containsAll(Collection<?> c);
}
```

The `containsAll()` method is very general, supporting comparisons against all implementations of `Collection`. Furthermore, this method is not strictly necessary — it can be easily implemented using the explicit enumeration facility of the `Collection` interface. Presumably, in this instance, `containsAll()` was provided for convenience. Nevertheless, it could be used for various performance optimisations. For example, since the common implementation of `containsAll()` has linear time-complexity in the size of the parameter, we might like to reduce this. One approach is to exploit the sortedness of collections such as `TreeMap`; that is, if we know the maximum element of this collection, and that the parameter collection is in sorted order, we can often stop the `containsAll()` comparison early. Another approach is to cache those collections which have already been tested, thus giving constant-time performance on repeat queries. One problem with this latter approach is, of course, tracking dependency changes. The problem is particularly acute here, since the specification of `containsAll()` makes no restrictions which might help us. Nevertheless, such an optimisation can be applied in certain situations; in particular, when the parameter collection is known to be immutable.

Other related examples include collection APIs, such as those of Smalltalk [4] and Python [16]. These typically provide a `filter(UnaryFun)` method, similar to that discussed above. However, these any collection implementations do not perform query optimisation, despite the interface enabling this possibility.

References

[1] Boost C++ libraries, http://www.boost.org.

[2] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the Workshop on Foundations of Object-Oriented Languages*, pages 151–178. Springer-Verlag, 1991.

[3] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[4] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[5] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[6] Ken Hartness. Robocode: using games to teach artificial intelligence. *J. Comput. Small Coll.*, 19(4):287–291, 2004.

[7] Jin-Hyuk Hong and Sung-Bae Cho. Evolution of emergent behaviors for shooting game characters in robocode. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 634–638. IEEE Press, 2004.

[8] Jgrapht, http://jgrapht.sourceforge.net/.

[9] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM Symposium on Very high level languages*, pages 50–59. ACM Press, 1974.

[10] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 473–486. ACM Press, 2005.

[11] Mathew Nelson. Robocode, http://robocode.sourceforge.net, 2007.

[12] Jackie O'Kelly and J. Paul Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *Proceedings of the SIGCSE conference on Innovation and technology in computer science education*, pages 217–221, New York, NY, USA, 2006. ACM.

[13] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[14] David J. Pearce and James Noble. Relationship aspect patterns. In *Proceedings of European Conference on Pattern Languages of Programs (EuroPLOP)*, pages 531–546. ACM Press, 2006.

[15] David J. Pearce and James Noble. Relationship aspects. In *Proceedings of the ACM conference on Aspect-Oriented Software Development (AOSD)*, pages 75–86. ACM Press, 2006.

[16] Python. http://www.python.org.

[17] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[18] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[19] Sun Microsystems Inc. *The Java Collection Framework*, 1995–1999. `http://java.sun.com/j2se/1.5/docs/guide/collections/`.

[20] Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation for the Java Query Language. In *Proceedings of OOPSLA*, page to appear, 2008.