



Chapter 8: Software Construction for Performance

8.3 Code Tuning for Performance Optimization

Ming Liu

May 16, 2018

Outline

- **1 Code Tuning Strategies and Process**
- **2 Common Sources of Inefficiency**
- **3 Code Tuning for Object Creation and Reuse**
 - (Creational) Prototype pattern creates objects by cloning an existing object.
 - (Creational) Singleton pattern restricts object creation for a class to only one instance.
 - (Structural) Flyweight reduces the cost of creating and manipulating a large number of similar objects.
 - Object Pool Pattern
 - Avoiding Garbage Collection
 - Object Initialization



1 Code-Tuning Strategies and Process



What is code tuning ?

- **Code tuning** is the practice of modifying correct code in ways that make it run more efficiently.
- “**Tuning**” refers to **small-scale changes** that affect a single class, a single routine, or, more commonly, a few lines of code.
- “**Tuning**” does not refer to large-scale design changes, or other higher-level means of improving performance.
- **Performance: time and space complexity.**

Consider these options first before tuning

- **Once you've chosen efficiency as a priority, whether its emphasis is on speed or on size, you should consider several options before choosing to improve either speed or size at the code level.**
 - **Program requirements:** Asking for more than is needed leads to trouble (e.g., return 1ms, is it always? On average? 99%?)
 - **System design:** High-level architecture design
 - **Class and function design:** algorithm performance
 - **Operating-system interactions:** Hidden OS calls within libraries
 - **Code compilation:** “Automatic” optimization
 - **Hardware:** update HW would be much easier
 - **Code tuning**



The Pareto Principle of program optimization

- **The Pareto Principle (80/20 rule) states that you can get 80 percent of the result with 20 percent of the effort.**
 - Barry Boehm reports that 20 percent of a program's routines consume 80 percent of its execution time.
 - Donald Knuth found that less than 4 percent of a program usually accounts for more than 50 percent of its run time.
- **The target of code-tuning is to find hotspots (the 20%) and optimize them.**
- **“The best is the enemy of the good”. Working toward perfection may prevent completion. Complete it first, and then perfect it. The part that needs to be perfect is usually small.**

(1) LoC vs. Efficiency

- **Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code – false!**

- This certainly doesn't imply the conclusion that increasing the number of lines of high-level language code always improves speed or reduces size.
- It does imply that regardless of the aesthetic appeal of writing something with the fewest lines of code, there's no predictable relationship between the number of lines of code in a high-level language and a program's ultimate size and speed.

```
for(i=0;i<5;i++)  
    A[i] = i;
```

A[0]=0
A[1]=1
A[2]=2
A[3]=3
A[4]=4

(2) Tuning by Profiling and Measuring

- Profiling helps determine where code is spending time (**hotspots**)
- Profiling provides basis for **measurement**: determine whether “improvement” really improved anything.
- **Code tuning is based on precise measurements.**
 - You must always measure performance to know whether your changes helped or hurt your program. The rules of the game change every time you change languages, compilers, versions of compilers, libraries, versions of libraries, processor, amount of memory on the machine.
 - When you tune code, you’re implicitly signing up to reprofile each optimization every time you change your compiler brand, compiler version, library version, and so.
 - If you don’t reprofile, an optimization that improves performance under one version of a compiler or library might well degrade performance when you change the build environment.

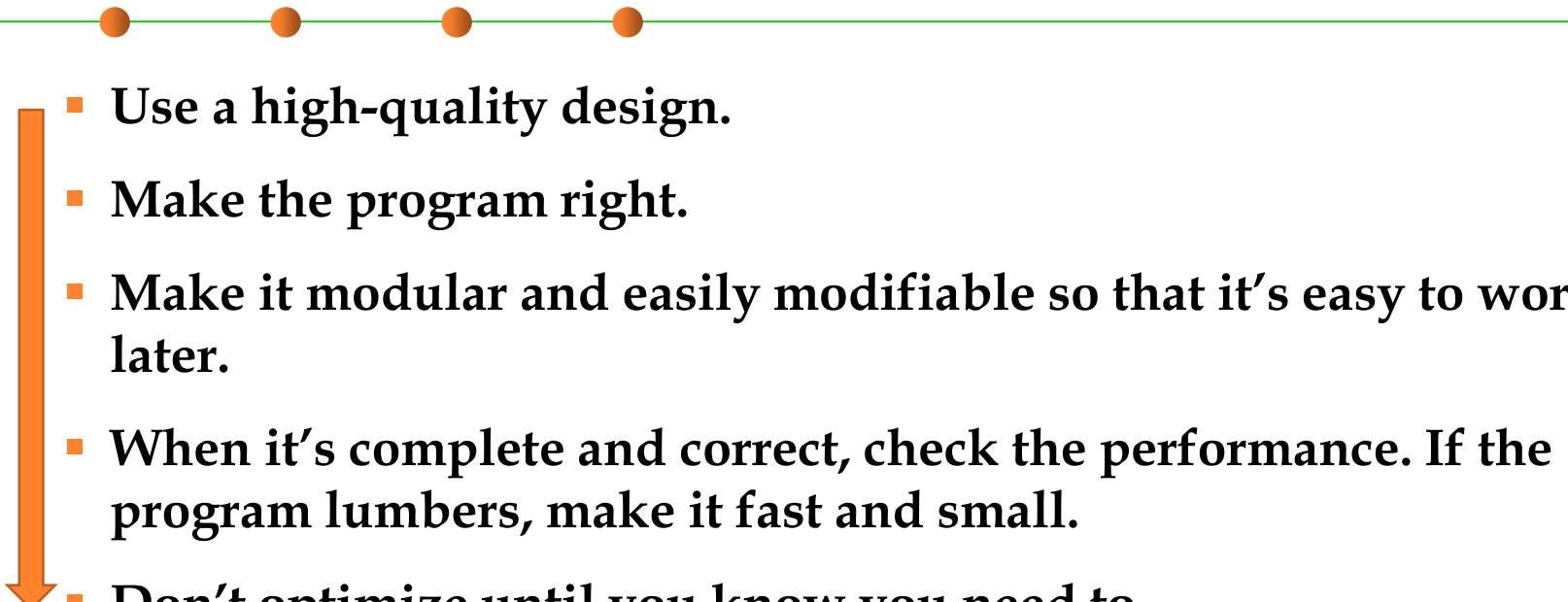
(3) Tuning until code complete

- You should optimize as you go – false!
- One theory is that if you strive to write the fastest and smallest possible code as you write each function, your program will be fast and small. This approach creates a forest-for-the-trees situation in which programmers ignore significant global optimizations because they're too busy with micro-optimizations.
- Here are the main problems with optimizing as you go along:
 - It's almost impossible to identify performance bottlenecks before a program is working completely.
 - In the rare case in which developers identify the bottlenecks correctly, they overkill the bottlenecks they've identified and allow others to become critical.
 - Focusing on optimization during initial development detracts from achieving other maybe more important program objectives.

(4) Fast vs. Correct

- **A fast program is just as important as a correct one — false!**
- **It's hardly ever true that programs need to be fast or small before they need to be correct.**
 - For a certain class of projects, speed or size is a major concern. This class is the minority, is much smaller than most people think, and is getting smaller all the time. For these projects, the performance risks must be addressed by up-front design.
 - For other projects, early optimization poses a significant threat to overall software quality, including performance.
- **Code correctness/robustness/readability/maintainability/etc. is usually more important than efficiency.**
- **Always start with well-written code, and only tune at the end**

When to Tune

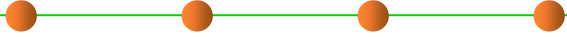
- 
- Use a high-quality design.
 - Make the program right.
 - Make it modular and easily modifiable so that it's easy to work on later.
 - When it's complete and correct, check the performance. If the program lumbers, make it fast and small.
 - Don't optimize until you know you need to.
-
- Jackson's Rules of Optimization:
 - Rule 1. Don't do it.
 - Rule 2 (for experts only). Don't do it yet— that is, not until you have a perfectly clear and un-optimized solution.



2 Common Sources of Inefficiency



(1) I/O

- 
- One of the most significant sources of inefficiency is unnecessary I/O.
 - If you have a choice of working with a file in memory vs. on disk, in a database, or across a network, use an in-memory data unless space is critical.

(2) Paging

- An operation that causes the operating system to swap pages of memory is much slower than an operation that works on only one page of memory.
- Sometimes a simple change makes a huge difference.
- E.g., one programmer wrote an initialization loop that produced many page faults on a system that used 4K pages.

Java Example of an Initialization Loop That Causes Many Page Faults

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {  
    for ( row = 0; row < MAX_ROWS; row++ ) {  
        table[ row ][ column ] = BlankTableElement();  
    }  
}
```

Paging

- The programmer restructured the loop this way:

Java Example of an Initialization Loop That Causes Few Page Faults

```
for ( row = 0; row < MAX_ROWS; row++ ) {  
    for ( column = 0; column < MAX_COLUMNS; column++ ) {  
        table[ row ][ column ] = BlankTableElement();  
    }  
}
```

- This code still causes a page fault every time it switches rows, but it switches rows only MAX_ROWS times instead of MAX_ROWS * MAX_COLUMNS times.
- The specific performance penalty varies significantly. On a machine with limited memory, the second code sample is about 1000 times faster than the first code sample. On machines with more memory, the difference is as small as a factor of 2, and it doesn't show up at all except for very large values of MAX_ROWS and MAX_COLUMNS.

(3) System Calls

- **Calls to system routines are often expensive.**
 - System routines include input/output operations to disk, keyboard, screen, printer, or other device; memory-management routines; and certain utility routines.
- **If they're expensive, consider these options:**
 - Write your own services. Sometimes you need only a small part of the functionality offered by a system routine and can build your own from lower level system routines. Writing your own replacement gives you something that's faster, smaller, and better suited to your needs.
 - Avoid going to the system.
 - Work with the system vendor to make the call faster. Most vendors want to improve their products and are glad to learn about parts of their systems with weak performance. (They may seem a little grouchy about it at first, but they really are interested.)

(4) Relative Performance Costs of Operations

Language	Type of Language	Execution Time Relative to C++	
C++	Compiled	1:1	
Visual Basic	Compiled	1:1	
C#	Compiled	1:1	
Java	Byte code	1.5:1	
PHP	Interpreted	>100:1	
Python		Relative Time Consumed	
Operation	Example	C++	Java
Baseline (integer assignment)	<i>i = j</i>	1	1
Routine Calls			
Call routine with no parameters	<i>foo()</i>	1	n/a
Call private routine with no parameters	<i>this.foo()</i>	1	0.5
Call private routine with 1 parameter	<i>this.foo(i)</i>	1.5	0.5
Call private routine with 2 parameters	<i>this.foo(i, j)</i>	1.7	0.5
Object routine call	<i>bar.foo()</i>	2	1
Derived routine call	<i>derivedBar.foo()</i>	2	1
Polymorphic routine call	<i>abstractBar.foo()</i>	2.5	2
Object References			
Level 1 object dereference	<i>i = obj.num</i>	1	1

Relative Performance Costs of Common Operations

Operation	Example	Relative Time Consumed	
		C++	Java
Level 2 object dereference	$i = \text{obj1.obj2.num}$	1	1
Each additional dereference	$i = \text{obj1.obj2.obj3...}$	not measurable	not measurable
Integer Operations			
Integer assignment (local)	$i = j$	1	1
Integer assignment (inherited)	$i = j$	1	1
Integer addition	$i = j + k$	1	1
Integer subtraction	$i = j + k$	1	1
Integer multiplication	$i = j * k$	1	1
Integer division	$i = j \% k$	5	1.5
Floating Point Operations			
Floating-point assignment	$x = y$	1	1
Floating-point addition	$x = y + z$	1	1
Floating-point subtraction	$x = y - z$	1	1
Floating-point multiplication	$x = y * z$	1	1
Floating-point division	$x = y / z$	4	1
Transcendental Functions			
Floating-point square root	$x = \text{sqrt}(y)$	15	4
Floating-point sine	$x = \text{sin}(y)$	25	20
Floating-point logarithm	$x = \text{log}(y)$	25	20
Floating-point e^x	$x = \text{exp}(y)$	50	20
Arrays			
Access integer array with constant subscript	$i = a[5]$	1	1
Access integer array with variable subscript	$i = a[j]$	1	1
Access two-dimensional integer array with constant subscripts	$i = a[3, 5]$	1	1
Access two-dimensional integer array with variable subscripts	$i = a[j, k]$	1	1
Access floating-point array with constant subscript	$x = z[5]$	1	1

Operation	Example	Relative Time Consumed	
		C++	Java
Access floating-point array with integer-variable subscript	$x = z[j]$	1	1
Access two-dimensional floating-point array with constant subscripts	$x = z[3, 5]$	1	1
Access two-dimensional floating-point array with integer-variable subscripts	$x = z[j, k]$	1	1

Most of the common operations are about the same price—routine calls, assignments, integer arithmetic, and floating-point arithmetic are all roughly equal. Transcendental math functions are extremely expensive. Polymorphic routine calls are a bit more expensive than other kinds of routine calls.

In every case, improving speed comes from replacing an expensive operation with a cheaper one.



3 Code Tuning for Object Creation and reuse

- (1) Singleton Pattern
- (2) Flyweight Pattern
- (3) Object Pool Pattern
- (4) Avoiding Garbage Collection
- (5) Object Initialization



(1) Singleton Pattern



Singleton objects

- **Some classes have conceptually one instance**
 - Many printers, but only one print spooler
 - One file system
 - One window manager
 - An accounting system will be dedicated to serving one company.
- **Naïve: create many objects that represent the same conceptual instance**
- **Better: only create one object and reuse it**
 - Encapsulate the code that manages the reuse

Singleton pattern



■ Intent

- Ensure a class only has one instance, and provide a global point of access to it.

■ Applicability

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

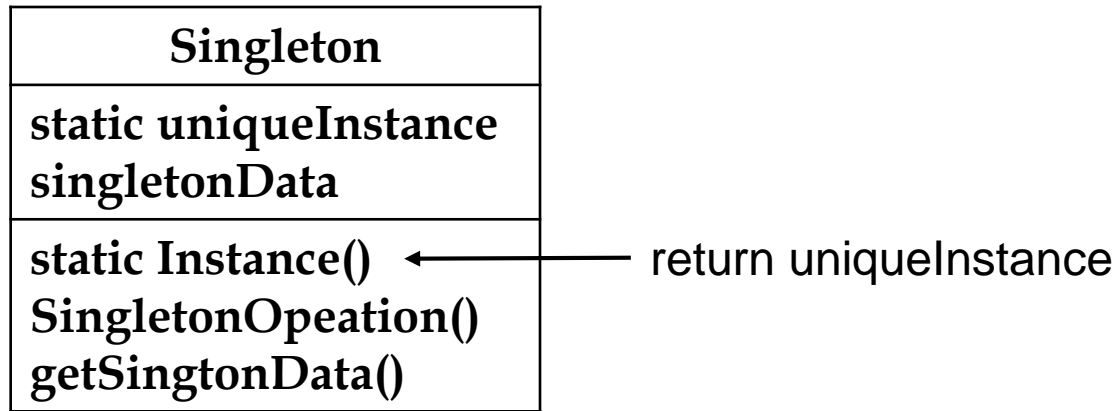
Singleton pattern

■ Benefit:

- Controlled access to sole instance.
- Reduced name space.(The Singleton pattern is an improvement over global variables).
- Class encapsulates code to ensure reuse of the object; no need to burden client
- **Reuse implies better performance**
 - For frequently used objects, you can save the time it takes to create a new object;
 - As the number of *new* operations decreased, the use of system memory can be reduced.
 - **Therefore, for the core components of the system and frequently used objects, the use of singleton pattern can effectively improve system performance.**

Singleton pattern

- **Class is responsible for tracking its sole instance**
 - Make constructor private
 - Provide static method/field to allow access to the only instance of the class
- **Structure**



Implementing the Singleton method

- **In Java, just define a final static field**

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton() {...}  
  
    public static Singleton getInstance() { return instance; }  
    // other operations and data  
}
```

- **Java semantics guarantee object is created immediately before first use**

Implementing the Singleton method

- To further improve performance, *lazy load* technique can be used to avoid the situation that it may take much time to create the sole instance at startup and the instance will be created when it is first used.

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton() {...}  
    public static Singleton getInstance() {  
        if (instance == null)  
            instance = new Singleton();  
        return instance;  
    }  
    // other operations and data  
}
```

- Pay attention to the usage in multi-threaded program

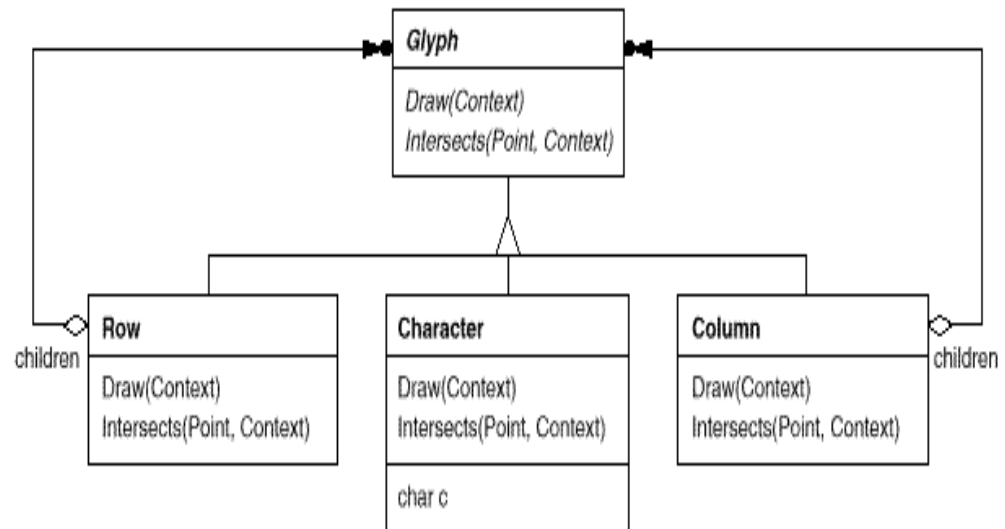
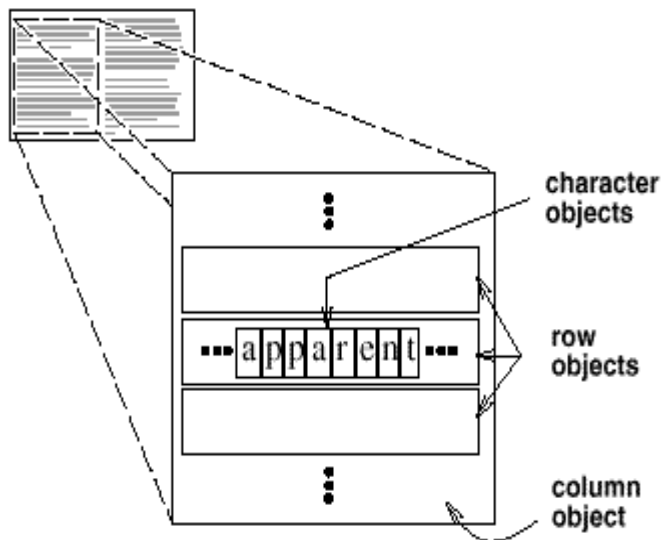


(2) Flyweight Pattern



Flyweight Pattern - Motivation

- How can a document editor use objects to represent characters?
- The drawback of such a design is its cost.
- Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.



Flyweight Pattern

- The Flyweight pattern describes **how to share objects to allow their use at fine granularities without prohibitive cost.**
 - A flyweight is a shared object that can be used in multiple contexts simultaneously.
- Use sharing to support large numbers of fine-grained objects efficiently.
- In the flyweight pattern, there is the concept of **Intrinsic and Extrinsic state.**
 - **Intrinsic states** are things that are constant and are stored in the memory.
 - **Extrinsic states** are things that are not constant and need to be calculated on the fly, and are therefore not stored in the memory.
 - E.g., in a game, the shapes of the aliens are all the same, but their color will change based on how mad each are. The shapes of the aliens will be Intrinsic, and the color of the alien will be Extrinsic.

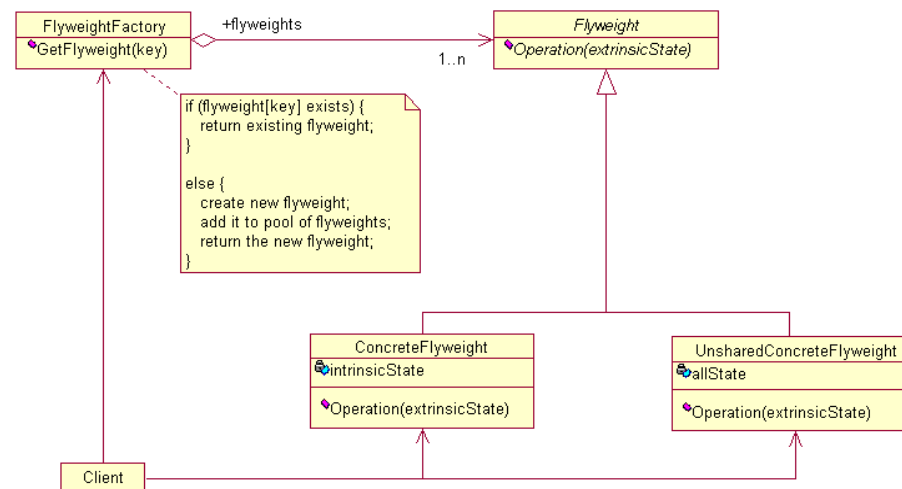
Flyweight Pattern

■ Flyweight

- Declares an interface through which flyweights can receive and act on extrinsic state.

■ ConcreteFlyweight (E.g.,Character)

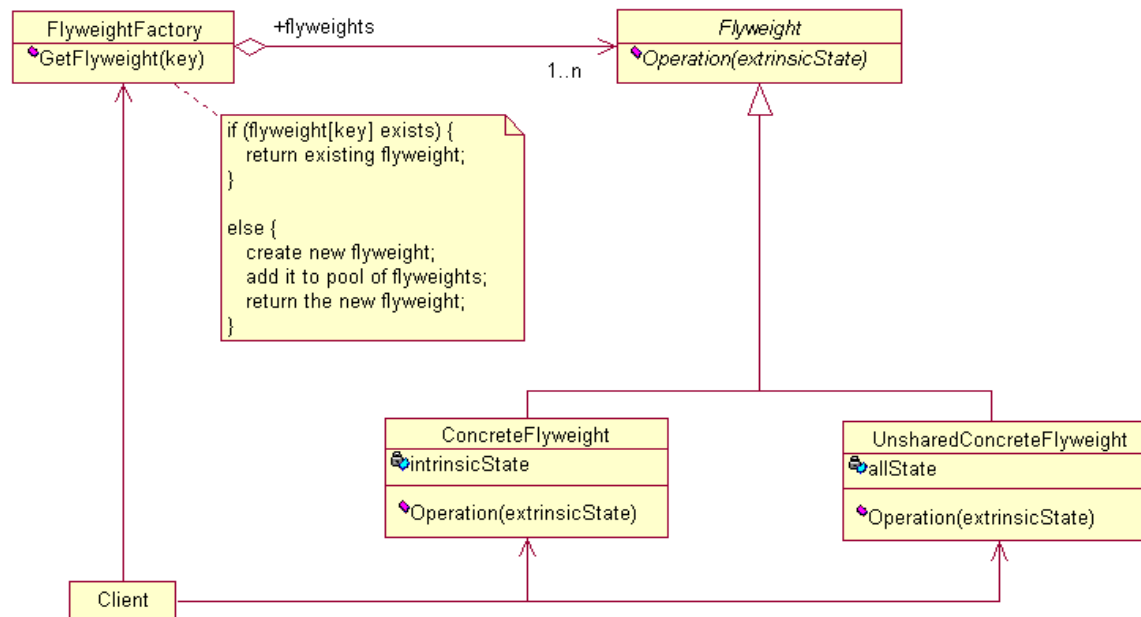
- Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.



Flyweight Pattern

■ UnsharedConcreteFlyweight (Row, Column)

- Not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).



Flyweight Pattern

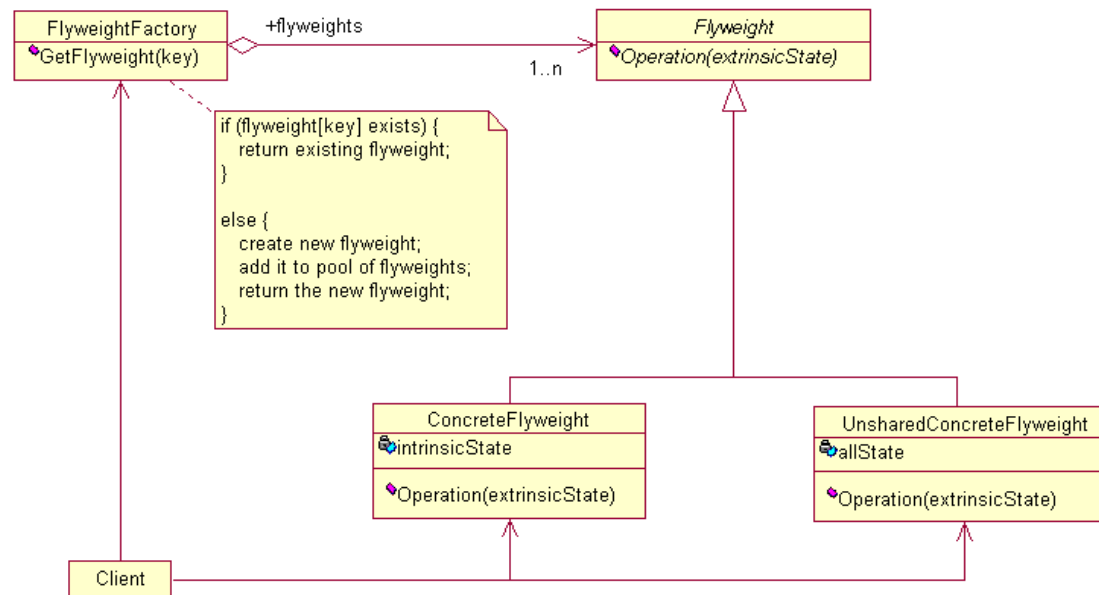
■ FlyweightFactory

- Creates and manages flyweight objects.
- Ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- In general, FlyweightFactory class is just an indexer that allows you to retrieve the flyweight object when given an index number, since you may have many flyweight objects in your application.

Flyweight Pattern

■ Client

- Maintains a reference to flyweight(s).
- Computes or stores the extrinsic state of flyweight(s).



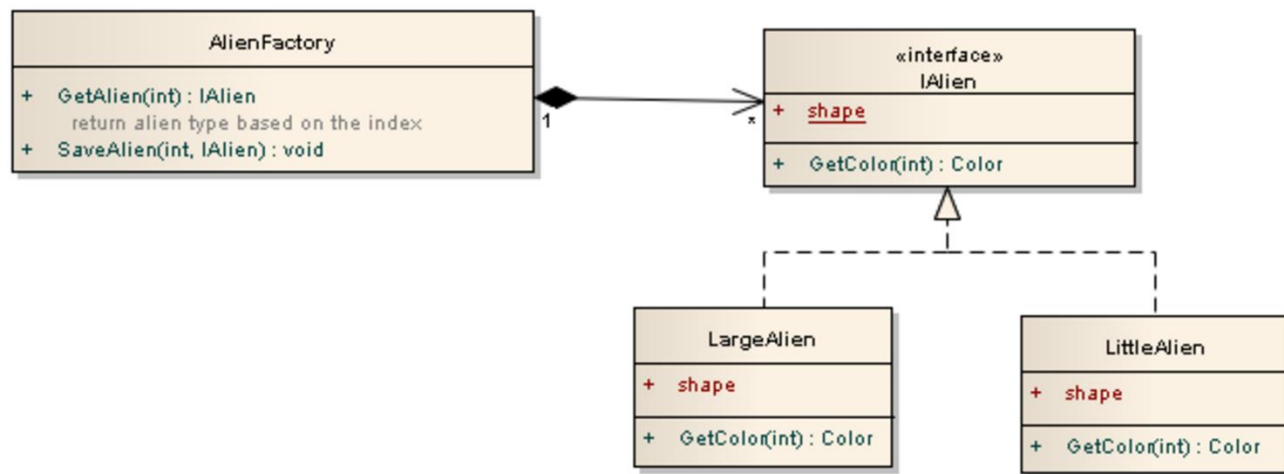
Flyweight Pattern

- **The Flyweight pattern's effectiveness depends heavily on how and where it's used.**
- **Applicability**
 - An application uses a large number of objects.
 - Storage costs are high because of the sheer quantity of objects.
 - Most object state can be made extrinsic.
 - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
 - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Example

■ Creating aliens of different shapes and colors

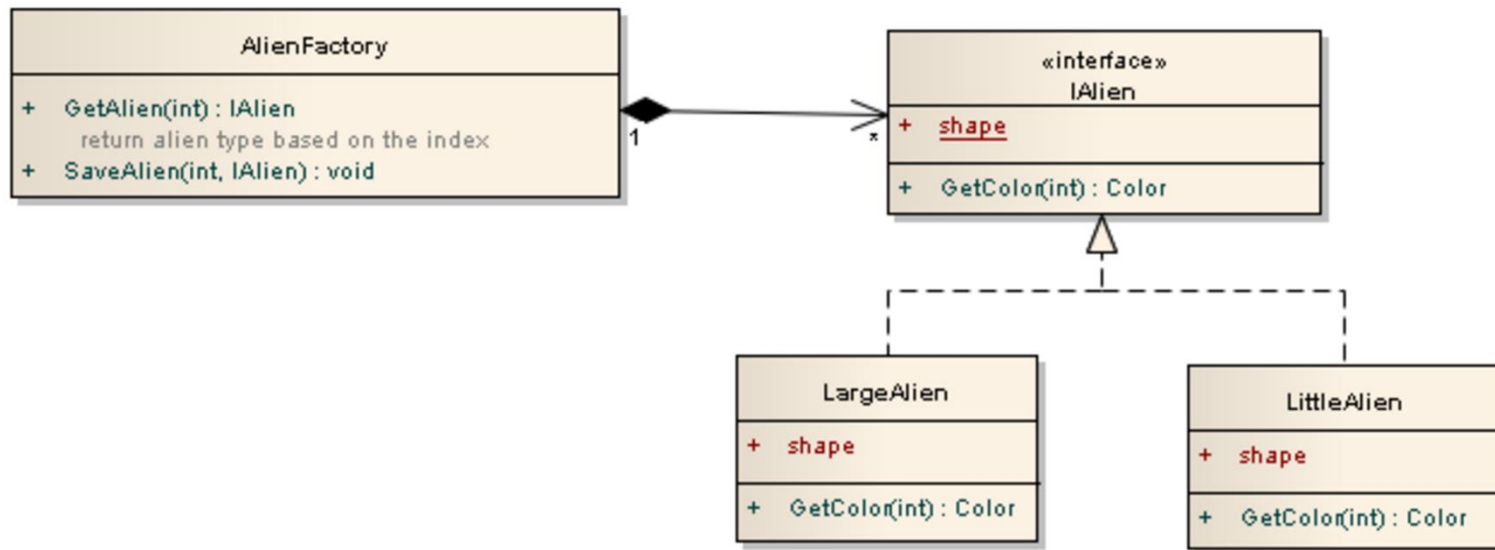
- The AlienFactory class stores and retrieves different types of aliens using its GetAlien and SaveAlien methods.
- The IAlien interface defines the shape property as the intrinsic state and the GetColor method as the extrinsic state.
- The LargeAlien and the LittleAlien classes are the flyweight objects where each has its own shape intrinsic state and ways of calculating the GetColor extrinsic state.



Example

```
public enum Color {Red, Green, Blank, Blue, Yellow}
```

```
public interface IAlien {  
    String Shape=null; //intrinsic state  
    String getShape();  
    Color getColor(int madLevel); //extrinsic state  
}
```



Example

```
public class LargeAlien implements IAlien{
    private String shape = "Large Shape"; //intrinsic state
```

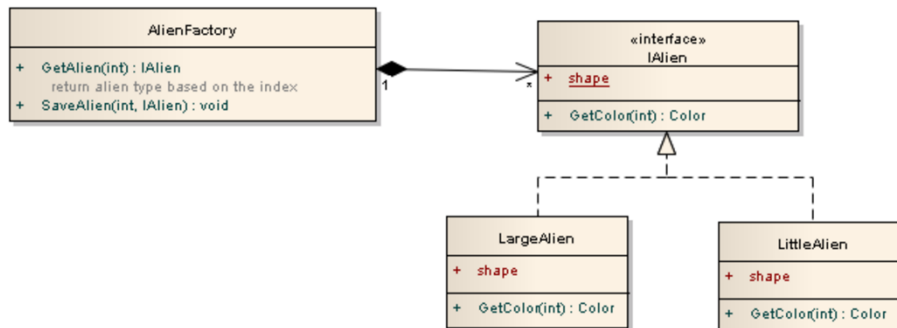
```
    public String getShape()
    {
        return shape;
    }
```

```
    public Color getColor(int madLevel) //extrinsic state
    {
        if (madLevel == 0)
            return Color.Green;
        else if (madLevel == 1)
            return Color.Red;
        else
            return Color.Blue;
    }
}
```

```
public class LittleAlien implements IAlien {
    private String shape = "Little Shape";
```

```
    public String getShape()
    {
        return shape;
    }
```

```
    public Color getColor(int madLevel) {
        if (madLevel == 0)
            return Color.Red;
        else if (madLevel == 1)
            return Color.Blue;
        else
            return Color.Green;
    }
}
```



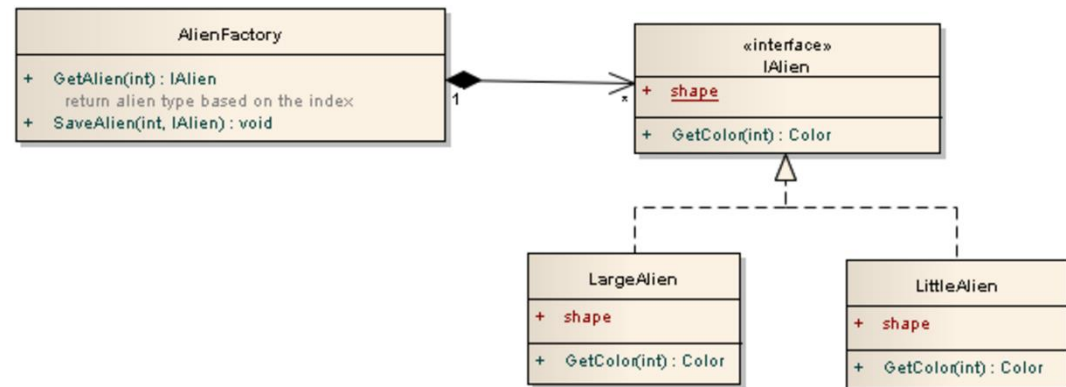
Example

```
import java.util.*;

public class AlienFactory {
    private Map<String,IAlien> list = new HashMap<>();

    public void SaveAlien(String index, IAlien alien)
    {
        list.put(index,alien);
    }

    public IAlien GetAlien(String index)
    {
        return list.get(index);
    }
}
```



Example

```
public class program {
    public static void main(String[] args)
```

```
    //create Aliens and store in factory
    AlienFactory factory = new AlienFactory();
    factory.SaveAlien("LargeAlien", new LargeAlien());
    factory.SaveAlien("LittleAlien", new LittleAlien());
```

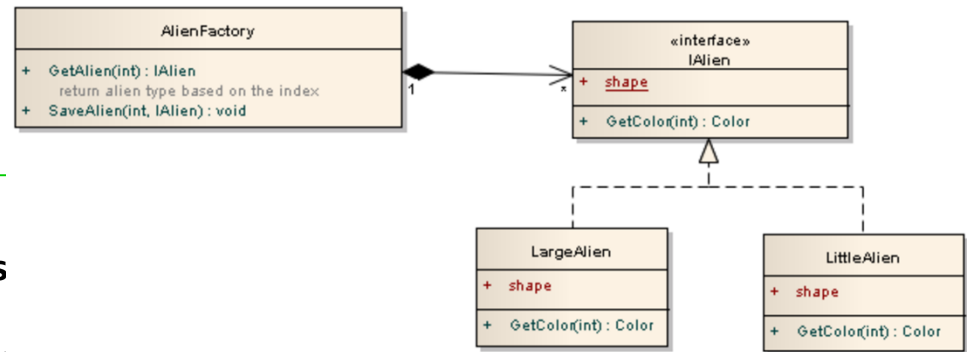
```
    //now access the flyweight objects
    IAlien a = factory.GetAlien("LargeAlien");
    IAlien b = factory.GetAlien("LittleAlien");
```

```
    //show intrinsic states, all accessed in memory without calculations
    System.out.println("Showing intrinsic states...");
    System.out.println("Alien of type LargeAlien is " + a.getShape());
    System.out.println("Alien of type LittleAlien is " + b.getShape());
```

```
    //show extrinsic states, need calculations
    System.out.println("Showing extrinsic states...");
    System.out.println("Alien of type LargeAlien is " + a.getColor(0).toString());
    System.out.println("Alien of type LargeAlien is " + a.getColor(1).toString());
    System.out.println("Alien of type LittleAlien is " + b.getColor(0).toString());
    System.out.println("Alien of type LittleAlien is " + b.getColor(1).toString());
```

```
}
```

```
}
```



```
Showing intrinsic states...
Alien of type LargeAlien is Large Shape
Alien of type LittleAlien is Little Shape
Showing extrinsic states...
Alien of type LargeAlien is Green
Alien of type LargeAlien is Red
Alien of type LittleAlien is Red
Alien of type LittleAlien is Blue
```



(3) Object Pool Pattern



Object reuse

- **Objects are expensive to create.**
 - Where it is reasonable to reuse the same object, you should do so. You need to be aware of when not to call new.
- One fairly obvious situation is when you have already used an object and can discard it before you are about to create another object of the same class.
 - You should look at the object and consider whether it is possible to reset the fields and then reuse the object, rather than throw it away and create another.
 - This can be particularly important for objects that are constantly used and discarded: for example, in graphics processing, objects such as Rectangles, Points, Colors, and Fonts are used and discarded all the time.
 - Recycling these types of objects can certainly improve performance.

Object Pool Pattern

- The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand.
- A client of the pool will request an object from the pool and perform operations on the returned object.
- When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Reusable containers / collections

- **Most container objects (e.g., Vectors, Hashtables) can be reused rather than created and thrown away.**
 - While you are not using the retained objects, you are holding onto more memory than if you simply discarded those objects, and this reduces the memory available to create other objects.
 - You need to balance the need to have some free memory available against the need to improve performance by reusing objects.
 - But generally, the space taken by retaining objects for later reuse is significant only for very large collections, and you should certainly know which ones these are in your application.

Reusable containers / collections



```
//An instance of the vector pool manager.
```

```
public static VectorPoolMgr vectorPoolMgr = new VectorPoolMgr(25);
```

```
...
```

```
public void someMethod( ) {
```

```
    //Get a new Vector. We only use the vector to do some stuff
```

```
    //within this method, and then we dump the vector (i.e., it
```

```
    //is not returned or assigned to a state variable)
```

```
    //so this is a perfect candidate for reusing Vectors.
```

```
    //Use a factory method instead of 'new Vector( )'
```

```
    Vector v = vectorPoolMgr( );
```

```
    ... //do vector manipulation stuff
```

```
    //and the extra work is that we have to explicitly tell the
```

```
    //pool manager that we have finished with the vector
```

```
    vectorPoolMgr.returnVector(v);
```

```
}
```

Canonicalizing Objects

- **Canonicalizing objects:** replacing multiple copies of an object with just a few objects.

- For example:

```
Boolean t1 = new Boolean(true);
```

```
System.out.println(t1 == Boolean.TRUE);    ➔ false
```

```
System.out.println(t1.equals(Boolean.TRUE)); ➔ true
```

- If Booleans had been canonicalized, all Boolean comparisons could be done by identity: **comparison by identity is always faster than comparison by equality**, because identity comparisons are simply pointer comparisons.

Example: Integer canonicalization

- If you use just a few Integer objects in some defined way, you repeatedly create the Integer objects with values 1, 2, 3, etc., and also access the `intValue()` to compare them.
- Otherwise, you can canonicalize a few integer objects, improving performance in several ways: eliminating the extra Integer creations and the garbage collections of these objects when they are discarded, and allowing comparison by identity.

```
public class IntegerManager {  
    public static final Integer ZERO = new Integer(0);  
    public static final Integer ONE = new Integer(1);  
    public static final Integer TWO = new Integer(2);  
    .....  
    public static final Integer NINE = new Integer(9);  
    public static final Integer TEN = new Integer(10);  
}
```

Example: Enumerating constants

- Another canonicalization technique often used is **replacing constant objects with integers**.
- For example, rather than use the strings "female" and "male", you should use a constant defined in an interface:

```
public interface GENDER {  
    public static final int FEMALE=1;  
    public static final int MALE=2;  
}
```

- **Used consistently, this enumeration can provide both speed and memory advantages.**
 - The enumeration requires less memory than the equivalent strings and makes network transfers faster.
 - Comparisons are faster too, as the identity comparison can be used instead of the equality comparison.



(4) Avoiding Garbage Collection



Various ways of reducing GC

- The canonicalization techniques are one way to avoid garbage collection: fewer objects means less to garbage-collect.
- Similarly, the pooling technique in that section also tends to reduce garbage-collection requirements, partly because you are creating fewer objects by reusing them, and partly because you deallocate memory less often by holding onto the objects you have allocated.
- Another technique for reducing garbage-collection impact is to avoid using objects where they are not needed.
- When a class does not provide a static method, you can sometimes use a dummy instance to execute instance methods repeatedly, thus avoiding the need to create extra objects.

```
String string = "55";  
int theInt = new Integer(string).intValue();  
int theInt = Integer.parseInt(string);
```

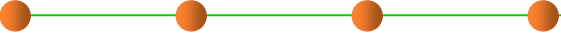
Primitive data types induce less GC

- The primitive data types in Java use memory space that also needs reclaiming, but the overhead in reclaiming data-type storage is smaller: it is reclaimed at the same time as its holding object and so has a smaller impact.
 - Temporary primitive data types exist only on the stack and do not need to be garbage-collected at all
- For example, an object with just one instance variable holding an int is reclaimed in one object reclaim.
- If it holds an Integer object, the garbage collector needs to reclaim two objects.

Primitive data types induce less GC

- Reducing garbage collection by using primitive data types also applies when you can hold an object in a primitive data-type format rather than another format.
 - For example, if you have a large number of objects, each with a String instance variable holding a number (e.g., "1492", "1997"), it is better to make that instance variable an int data type and store the numbers as ints, provided that conversion overhead does not swamp the benefits of holding the values in this alternative format.
- Similarly, you can use an int (or long) to represent a Date object, providing appropriate calculations to access and update the values, thus saving an object and the associated garbage overhead.
 - Hint: you have a different runtime overhead instead, as those conversion calculations may take up more time.

Primitive data types induce less GC



```
class MyClass {  
    int x;  
    boolean y;  
}
```

- To define a Collection of MyClass, there are two ways:

```
class MyClassCollection {  
    int[ ] xs;  
    boolean[ ] ys;  
    public int getXForElement(int i) {return xs[i];}  
    public boolean getYForElement(int i) {return ys[i];}  
}
```

```
List<MyClass> myClassCollection = new ArrayList<MyClass>;
```

Other general recommendations

- Reduce the number of temporary objects being used, especially in loops.
- Use `StringBuffer` in preference to the `String` concatenation operator (+). This is really a special case of the previous point.
- Be aware of which methods alter objects directly without making copies and which ones return a copy of an object.
 - Any `String` method that changes the string (such as `String.trim()`) returns a new `String` object, whereas a method like `Vector.setSize()` does not return a copy. If you do not need a copy, use (or create) methods that do not return a copy of the object being operated on.
- Avoid using generic classes that handle `Object` types when you are dealing with basic data types.
 - For example, there is no need to use `Vector` to store `ints` by wrapping them in `Integers`. Instead, implement an `IntVector` class that holds the `ints` directly.



(5) Object Initialization



Construction and Initialization of Objects

- All chained constructors are automatically called when creating an object with new.
- Chaining more constructors for a particular object causes extra overhead at object creation, as does initializing instance variables more than once.
- You should check the constructor hierarchy to eliminate any multiple initializations to instance variables.
- A common way to avoid constructors when creating objects, namely by creating a clone() of an object.
 - You can create new instances of classes that implement the Cloneable interface using the clone() method.
 - These new instances do not call any class constructor, thus allowing you to avoid the constructor initializations.

Preallocating Objects

- There may be situations in which you cannot avoid creating particular objects in significant amounts.
- You can still create the objects, but move the creation time to a part of the application when more spare cycles are available or there is more flexibility in response times.
- The idea is to choose another time to create some or all of the objects (perhaps in a partially initialized stage) and store those objects until they are needed.
- **In one word:** Create objects early, when there is spare time in the application, and hold those objects until required.

Late (Lazy) Initialization

- *Lazy initialization* means that you do not initialize objects until the first time they are used.
- Typically, this comes about when you are unsure of what initial value an instance variable might have but want to provide a default.
- Rather than initialize explicitly in the constructor (or class static initializer), it is left until access time for the variable to be initialized, using a test for `null` to determine if it has been initialized.

```
public getSomething( ){  
    if (something == null)  
        something = defaultSomething( );  
    return something;  
}
```



4 Code Tuning for Logics, Loops, Data Type, Expressions, and Functions





(1) Logic-related code tuning



Stop Testing When You Know the Answer



```
if ( x>5 && x<10 ) ...
```

- Some languages provide a form of expression evaluation known as “short-circuit evaluation,” which means that the compiler generates code that automatically stops testing as soon as it knows the answer.
- Short-circuit evaluation is part of Java’s “conditional” operators.

```
if (x>5) {  
    if (x <10 ) { ...}  
}
```

Stop Testing When You Know the Answer

- **E.g. for search loop:**

```
boolean found = false;
for (int i = 0; i < iCount; i++ ) {
    if (input[i] < 0 ) {
        found = True;
    }
}
```

- **To stop scanning as soon as you find a negative value in the array:**
 - Add a `break` statement after the `found = True` line.
 - Change the `for` loop to a `while` loop and check for `found` as well as for incrementing the loop counter past `iCount`.

Substitute Table Lookups for Complicated Logics

- In some circumstances, a table lookup may be quicker than traversing a complicated chain of logic.
- The point of a complicated chain is usually to categorize something and then to take an action based on its category. (See **Table-Driven Methods**)

```
if ( ( a && !c ) || ( a && b && c ) ) {
    category = 1;
}
else if ( ( b && !a ) || ( a && c && !b ) ) {
    category = 2;
}
else if ( c && !a && !b ) {
    category = 3;
}
else {
    category = 0;
}
```

```
// define categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
    // !b!c  !bc  b!c  bc
    0,   3,   2,   2,   // !a
    1,   2,   1,   1    //  a
};
...

category = categoryTable[ a ][ b ][ c ];
```

Use Lazy Evaluation

- **Lazy evaluation:** If a program uses lazy evaluation, it avoids doing any work until the work is needed.
- Lazy evaluation is similar to just-in-time strategies that do the work closest to when it's needed.
- E.g., a program contains a table of 5000 values, generates the whole table at startup time, and then uses it as the program executes. If the program uses only a small percentage of the entries in the table, it might make more sense to compute them as they're needed rather than all at once. Once an entry is computed, it can still be stored for future reference ("cached").



(2) Loops-related code tuning



Unswitching-将判断外提

- Switching refers to making a decision inside a loop every time it's executed.
- If the decision doesn't change while the loop is executing, you can unswitch the loop by making the decision outside the loop.
- Usually this requires turning the loop inside out, putting loops inside the conditional rather than putting the conditional inside the loop.

```
for ( i = 0; i < count; i++ ) {  
    if ( sumType == SUMTYPE_NET ) {  
        netSum = netSum + amount[ i ];  
    }  
    else {  
        grossSum = grossSum + amount[ i ];  
    }  
}
```



```
if ( sumType == SUMTYPE_NET ) {  
    for ( i = 0; i < count; i++ ) {  
        netSum = netSum + amount[ i ];  
    }  
}  
else {  
    for ( i = 0; i < count; i++ ) {  
        grossSum = grossSum + amount[ i ];  
    }  
}
```

Jamming-合并

- **Jamming, or “fusion,” is the result of combining two loops that operate on the same set of elements. The gain lies in cutting the loop overhead from two loops to one.**

```
For i = 0 to employeeCount - 1
    employeeName( i ) = ""
Next
...
For i = 0 to employeeCount - 1
    employeeEarnings( i ) = 0
Next
```

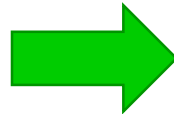


```
For i = 0 to employeeCount - 1
    employeeName( i ) = ""
    employeeEarnings( i ) = 0
Next
```

Unrolling-展开

- The goal of loop unrolling is to reduce the amount of loop housekeeping.

```
i = 0;
while ( i < count ) {
    a[ i ] = i;
    i = i + 1;
}
```



```
i = 0;
while ( i < count - 1 ) {
    a[ i ] = i;
    a[ i + 1 ] = i + 1;
    i = i + 2;
}
```

```
if ( i == count ) {
    a[ count - 1 ] = count - 1;
}
```

Deal with the case that might fall through the cracks if the loop went by twos instead of by ones.

Minimizing the Work Inside Loops

- **To minimize the work done inside a loop.**
 - If you can evaluate a statement or part of a statement outside a loop so that only the result is used inside the loop, do so. It's good programming practice, and, in some cases, it improves readability.

C++ Example of a Complicated Pointer Expression Inside a Loop

```
for ( i = 0; i < rateCount; i++ ) {  
    netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;  
}
```

C++ Example of Simplifying a Complicated Pointer Expression

```
quantityDiscount = rates->discounts->factors->net;  
for ( i = 0; i < rateCount; i++ ) {  
    netRate[ i ] = baseRate[ i ] * quantityDiscount;  
}
```

Sentinel Values – 哨兵值

- The classic example of a compound test that can be improved by use of a sentinel is the search loop that checks both whether it has found the value it is seeking and whether it has run out of values.

C# Example of Compound Tests in a Search Loop

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
    if ( item[ i ] == testValue ) {
        found = TRUE;
    }
    else {
        i++;
    }
}

if ( found ) {
    ...
}
```

the loop really has three tests
for each iteration.

C# Example of Using a Sentinel Value to Speed Up a Loop

```
// set sentinel value, preserving the original value
initialValue = item[ count ];
item[ count ] = testValue;

i = 0;
while ( item[ i ] != testValue ) {
    i++;
}

// restore the value displaced by the sentinel
item[ count ] = initialValue;

// check if value was found
if ( i < count ) {
    ...
}
```

the loop really has one test
for each iteration.

Putting the Busiest Loop on the Inside

- When you have nested loops, think about which loop you want on the outside and which you want on the inside.
- Following is an example of a nested loop that can be improved.

Java Example of a Nested Loop That Can Be Improved

```
for ( column = 0; column < 100; column++ ) {  
    for ( row = 0; row < 5; row++ ) {  
        sum = sum + table[ row ][ column ];  
    }  
}
```

- The key to improving the loop is that the outer loop executes much more often than the inner loop. The loop has a total of 600 iterations.

Putting the Busiest Loop on the Inside

- By merely switching the inner and outer loops, you can change the total number of iterations to 5 for the outer loop and $5 * 100 = 500$ for the inner loop, for a total of 505 iterations. You'd expect to save about $(600 - 505) / 600 = 16$ percent.

Java Example of a Nested Loop That Can Be Improved

```
for ( column = 0; column < 100; column++ ) {  
    for ( row = 0; row < 5; row++ ) {  
        sum = sum + table[ row ][ column ];  
    }  
}
```

Strength Reduct—削減強度

- Reducing strength means replacing an expensive operation such as multiplication with a cheaper operation such as addition.
- Sometimes you'll have an expression inside a loop that depends on multiplying the loop index by a factor. Addition is usually faster than multiplication, and if you can compute the same number by adding the amount on each iteration of the loop rather than by multiplying, the code will run faster.

Visual Basic Example of Multiplying a Loop Index

```
For i = 0 to saleCount - 1
    commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

- This code is straightforward but expensive. You can rewrite the loop so that you accumulate multiples rather than computing them each time. This reduces the strength of the operations from multiplication to addition.

Strength Reduct—削減強度

- This code is straightforward but expensive. You can rewrite the loop so that you accumulate multiples rather than computing them each time. This reduces the strength of the operations from **multiplication to addition**

Visual Basic Example of Multiplying a Loop Index

```
For i = 0 to saleCount - 1
    commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

Visual Basic Example of Adding Rather Than Multiplying

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
    commission( i ) = cumulativeCommission
    cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```



(3) Data type related code tuning



Use Integers Rather Than Floating-Point Numbers

- Integer addition and multiplication tend to be faster than floating point. Changing a loop index from a floating point to an integer, for example, can save time.

Visual Basic Example of a Loop That Uses a Time-Consuming Floating-Point Loop Index

```
Dim i As Single  
For i = 0 to 99  
    x( i ) = 0  
Next
```



```
Dim i As Integer  
For i = 0 to 99  
    x( i ) = 0  
Next
```

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	2.80	0.801	71%	3.5:1
PHP	5.01	4.65	7%	1:1
Visual Basic	6.84	0.280	96%	25:1

Use the Fewest Array Dimensions Possible

- **Conventional wisdom maintains that multiple dimensions on arrays are expensive. If you can structure your data so that it's in a one-dimensional array rather than a two-dimensional or three-dimensional array, you might be able to save some time.**

Java Example of a Standard, Two-Dimensional Array Initialization

```
for ( row = 0; row < numRows; row++ ) {  
    for ( column = 0; column < numColumns; column++ ) {  
        matrix[ row ][ column ] = 0;  
    }  
}
```



Java Example of a One-Dimensional Representation of an Array

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {  
    matrix[ entry ] = 0;  
}
```

Minimize Array References

- In addition to minimizing accesses to doubly or triply dimensioned arrays, it's often advantageous to minimize array accesses, period.
- A loop that repeatedly uses one element of an array is a good candidate for optimization.

C++ Example of Unnecessarily Referencing an Array Inside a Loop

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {  
    for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {  
        rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];  
    }  
}
```

- The reference to `discount[discountType]` doesn't change when `discountLevel` changes in the inner loop. Consequently, you can move it out of the inner loop so that you'll have only one array access per execution of the outer loop rather than one for each

Minimize Array References

C++ Example of Moving an Array Reference Outside a Loop

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {  
    thisDiscount = discount[ discountType ];  
    for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {  
        rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;  
    }  
}
```

Language	Straight Time	Code-Tuned Time	Time Savings
C++	32.1	34.5	-7%
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

Use Supplementary Indexes

- **Using a supplementary index means adding related data that makes accessing a data type more efficient. You can add the related data to the main data type, or you can store it in a parallel structure.**
- **E.g.,**
 - String-Length Index
 - In C, strings are terminated by a byte that's set to 0.
 - You can apply the idea of indexing for length to any variable-length data type. It's often more efficient to keep track of the length of the structure rather than computing the length each time you need it.
 - Independent, Parallel Index Structure
 - Sometimes it's more efficient to manipulate an index to a data type than it is to manipulate the data type itself.
 - If each data item is large, you can create an auxiliary structure that consists of key values and pointers to the detailed information. If the difference in size between the data-structure item and the auxiliary-structure item is great enough, sometimes you can store the key item in memory even when the data item has to be stored externally.

Use Caching

- **Caching means saving a few values in such a way that you can retrieve the most commonly used values more easily than the less commonly used values.**
 - E.g., If a program randomly reads records from a disk, a routine might use a cache to save the records read most frequently. When the routine receives a request for a record, it checks the cache to see whether it has the record. If it does, the record is returned directly from memory rather than from disk.
 - In addition to caching records on disk, you can apply caching in other areas. In a Microsoft Windows font-proofing program, the performance bottleneck was in retrieving the width of each character as it was displayed. Caching the most recently used character width roughly doubled the display speed. You can cache the results of time-consuming computations too — especially if the parameters to the calculation are simple.
 - You can cache the results of time-consuming computations too — especially if the parameters to the calculation are simple.

Use Caching

- Suppose, for example, that you need to compute the length of the hypotenuse of a right triangle, given the lengths of the other two sides. The straightforward implementation of the routine would

Java Example of a Routine That's Conducive to Caching

```
double Hypotenuse(  
    double sideA,  
  
    double sideB  
) {  
    return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );  
}
```

Use Caching

- If you know that the same values tend to be requested repeatedly, you can cache values this way:

Java Example of Caching to Avoid an Expensive Computation

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;

public double Hypotenuse(
    double sideA,
    double sideB
) {
    // check to see if the triangle is already in the cache
    if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
        return cachedHypotenuse;
    }

    // compute new hypotenuse and cache it
    cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
    cachedSideA = sideA;
    cachedSideB = sideB;

    return cachedHypotenuse;
}
```

Use Caching

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	4.06	1.05	74%	4:1
Java	2.54	1.40	45%	2:1
Python	8.16	4.17	49%	2:1
Visual Basic	24.0	12.9	47%	2:1

- The success of the cache depends on the relative costs of accessing a cached element, creating an uncached element, and saving a new element in the cache.
- Success also depends on how often the cached information is requested. In some cases, success might also depend on caching done by the hardware.
- Generally, the more it costs to generate a new element and the more times the same information is requested, the more valuable a cache is. The cheaper it is to access a cached element and save new elements in the cache, the more valuable a cache is.
- As with other optimization techniques, caching adds complexity and tends to be error prone.



(4) Expression-related code tuning



Exploit Algebraic Identities-代数恒等式

- You can use algebraic identities to replace costly operations with cheaper ones.
- For example, the following expressions are logically equivalent:
 - not a and not B
 - not (a or B)
- the second expression can save a not operation.

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	7.43	0.010	99.9%	750:1
Visual Basic	4.59	0.220	95%	20:1
Python	4.21	0.401	90%	10:1

Use Strength Reduction – 削弱运算强度

- **strength reduction means replacing an expensive operation with a cheaper one.**
- **Here are some possible substitutions:**
 - Replace multiplication with addition.
 - Replace exponentiation with multiplication.
 - Replace trigonometric routines with their trigonometric identities.
 - Replace longlong integers with longs or ints (but watch for performance issues associated with using native-length vs. non-native-length integers)
 - Replace floating-point numbers with fixed-point numbers or integers.
 - Replace double-precision floating points with single-precision numbers.
 - Replace integer multiplication-by-two and division-by-two with shift operations.

Use Strength Reduction – 削弱运算强度

- $Ax^2 + Bx + C$

Visual Basic Example of Evaluating a Polynomial

```
value = coefficient( 0 )  
For power = 1 To order  
    value = value + coefficient( power ) * x^power  
Next
```



Visual Basic Example of a Reduced-Strength Method of Evaluating a Polynomial

```
value = coefficient( 0 )  
powerOfX = x  
For power = 1 to order  
    value = value + coefficient( power ) * powerOfX  
    powerOfX = powerOfX * x  
Next
```

Initialize at Compile Time

- If you're using a named constant or a magic number in a routine call and it's the only argument, that's a clue that you could precompute the number, put it into a constant, and avoid the routine call. The same principle applies to multiplications, divisions, additions, and other op

$$\log(x)_{\text{base}} = \log(x) / \log(\text{base})$$

- Log() is time-consuming routine .

C++ Example of a Log-Base-Two Routine Based on System Routines

```
unsigned int Log2( unsigned int x ) {
    return (unsigned int) ( log( x ) / log( 2 ) );
}
```



```
const double LOG2=0.69314718;
```

```
.....
```

```
unsigned int Log2( unsigned int x ) {
    return (unsigned int) ( log( x ) / LOG2 );
}
```


Use the Correct Type of Constants

- Use named constants and literals that are the same type as the variables they're assigned to. When a constant and its related variable are different types, the compiler has to do a type conversion to assign the constant to the variable.
- A good compiler does the type conversion at compile time so that it doesn't affect run-time performance. But a less advanced compiler or an interpreter generates code for a runtime conversion.

Use the Correct Type of Constants

- **In the first case, the initializations look like this:**
 - $x = 5$
 - $i = 3.14$
 - assuming x is a floating point variable and i is an integer, require type conversions.
- **In the second case, they look like this:**
 - $x = 3.14$
 - $i = 5$
 - don't require type conversions.

Precompute Results

- A common low-level design decision is the choice of whether to compute results on the fly or compute them once, save them, and look them up as needed.
- If the results are used many times, it's often cheaper to compute them once and look them up the rest of the time.
- At the simplest level, you might compute part of an expression outside a loop rather than inside. At a more complicated level, you might compute a lookup table once when program execution begins, using it every time thereafter, or you might store results in a data file or embed them in a program.

Precompute Results

- **Optimizing a program by precomputation can take several forms:**
 - Computing results before the program executes and wiring them into constants that are assigned at compile time
 - Computing results before the program executes and hard-coding them into variables used at run time
 - Computing results before the program executes and putting them into a file that's loaded at run time
 - Computing results once, at program startup, and then referencing them each time they're needed
 - Computing as much as possible before a loop begins, minimizing the work done inside the loop
 - Computing results the first time they're needed and storing them so that you can retrieve them when they're needed again

Eliminate Common Subexpressions

- If you find an expression that's repeated several times, assign it to a variable and refer to the variable rather than recomputing the expression in several places.
- The loan-calculation example has a common subexpression that you could eliminate. Here's the original code:

Java Example of a Common Subexpression

```
payment = loanAmount / (  
    ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /  
    ( interestRate / 12.0 )  
);
```



Java Example of Eliminating a Common Subexpression

```
monthlyInterest = interestRate / 12.0;  
payment = loanAmount / (  
    ( 1.0 - Math.pow( 1.0 + monthlyInterest, -months ) ) /  
    monthlyInterest  
);
```



(5) Function-related code tuning



Routines

- **One of the most powerful tools in code tuning is a good routine decomposition.**
 - Small, well-defined routines save space because they take the place of doing jobs separately in multiple places.
 - They make a program easy to optimize because you can refactor code in one routine and thus improve every routine that calls it.
 - Small routines are relatively easy to rewrite in assembler. Long, tortuous routines are hard enough to understand on their own; in assembler they're impossible.



The end

May 16, 2018