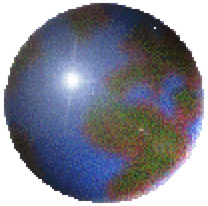


Is it Computer Science, Software Engineering, or Hacking?

Dr. Bruce K. Haddon

Paladin Software International

INCORPORATED

All trade and service marks, and logos, used in this presentation are the property of their respective owners, and are used solely to refer to those owners and their property. Copyrighted material is shown for fair comment. Copyright on this presentation does not include such material, but does include the author’s comments and forms of expression.

The Origins of the Problem

- A parser processing a Unicode input file, with a need to build *first* and *follow* sets (traditionally done with bit sets);
- Unicode has code points in the range 0 ~ 2^{21}
- The Java JDK has a class, **BitSet**: range 0 ~ 2^{32} - 1
- A match made in heaven!   ?

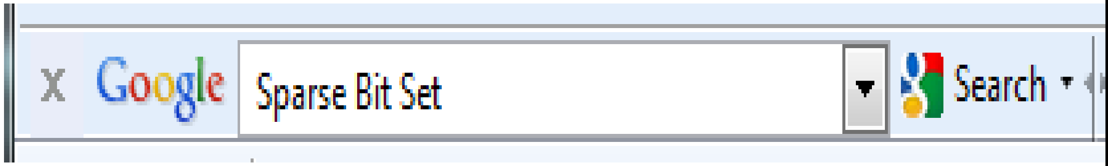
Copyright © 2010 Paladin Software Incorporated, Inc.

Enter Reality!

- To store a single bit using **BitSet** at bit 2^{32}-1 takes 2^{27} 32-bit words (2^{26} 64bit “words”), not counting any Java object overhead.
- The amount of memory available in a 32-bit JVM in a 4 GB Windows system is approximately 2^{26} 32-bit words at best (not quite 2^{27} on a Linux system).
- ***Oops!***

Copyright © 2010 Paladin Software Incorporated, Inc.

No. 1 “top of the search” return!



Sparse Bit Set

Tutorials & Code Camps

Help Using a HashSet for a Sparse Bit Set

by MageLang Institute

A *sparse* bitset is a large collection of boolean values where many of the values are off (or false). For maintaining these sparsely populated sets, the **BitSet** class can be very inefficient. Since the majority of the bits will be off, space will be occupied to store nothing. For working with these sparse bitsets, you can create an alternate representation, backed instead by a hashtable, or HashMap. Only those positions where a value is set are then stored in the mapping.

Copyright © 2010 Paladin Software Incorporated, Inc.

The Instructions

To create a sparse bitset, subclass **BitSet** and override the necessary methods (everything). The skeleton code should help get your started, so you can focus on the set-oriented routines.

The following UML diagram shows you the **BitSet** operations:

Skeleton Code

SparseBitSet.java

Tester.java

Tasks

Copyright © 2010 Paladin Software Incorporated, Inc.

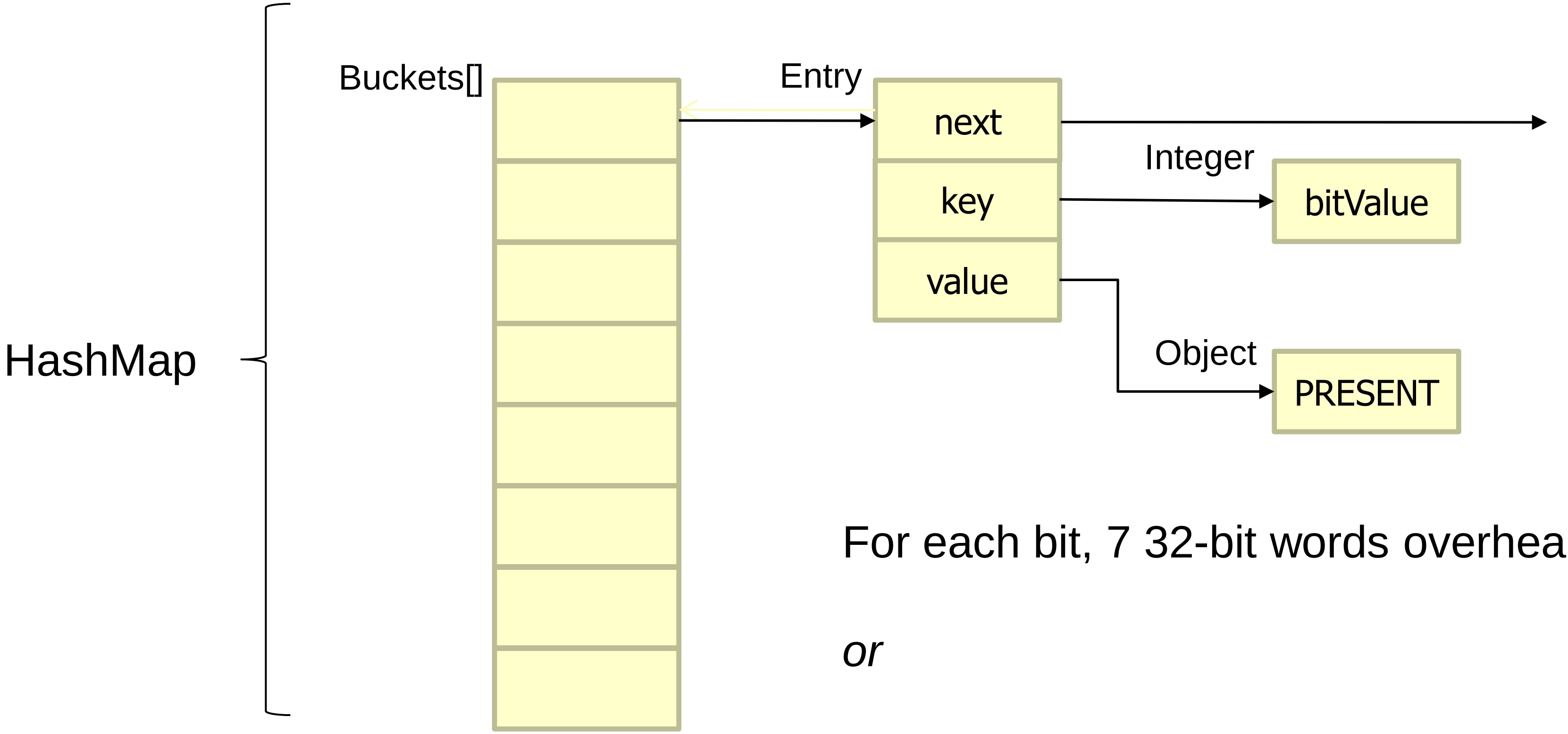
BitSet *(UML)*

+BitSet()
+BitSet(int : nbits)
+and(set : BitSet) : void
+andNot(set : BitSet) : void
+cardinality() : int
+clear() : void
+clear(i : int) : void
+clear(i : int, j: int) : void
+clone() : Object
+equals(obj : Object) : boolean
+flip(i : int) : void
+flip(i : int, j: int) : void
+get(i : int) : boolean
+get(i : int, j : int) : BitSet
+hashCode() : int

+intersects(set : BitSet) : boolean
+isEmpty() : boolean
+length() : int
+nextClearBit(i : int) : int
+nextSetBit(i : int) : int
+or(set : BitSet) : void
+set(i : int) : void
+set(i : int, value : boolean) : void
+set(i : int, j : int) : void
+set(i : int, j : int, value : boolean) : void
+size() : int
+toString() : String
+xor(set : BitSet) : void

Copyright © 2010 Paladin Software Incorporated, Inc.

The HashSet solution

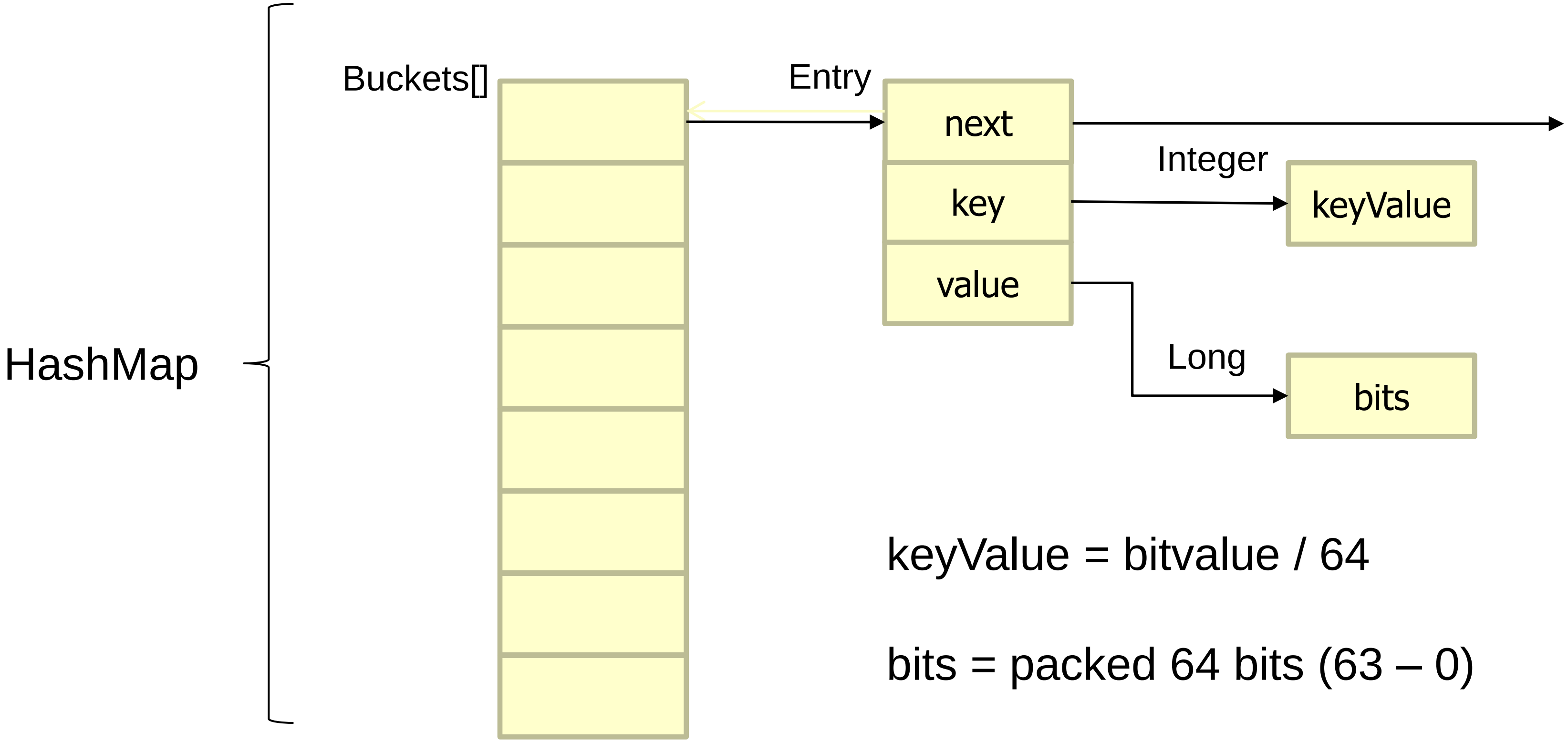


For each bit, 7 32-bit words overhead

or

For 64 bits, ~448 32-bit words overhead

The HashMap solution



HashMap

Buckets[]

Entry

next

key

value

Integer

keyValue

Long

bits

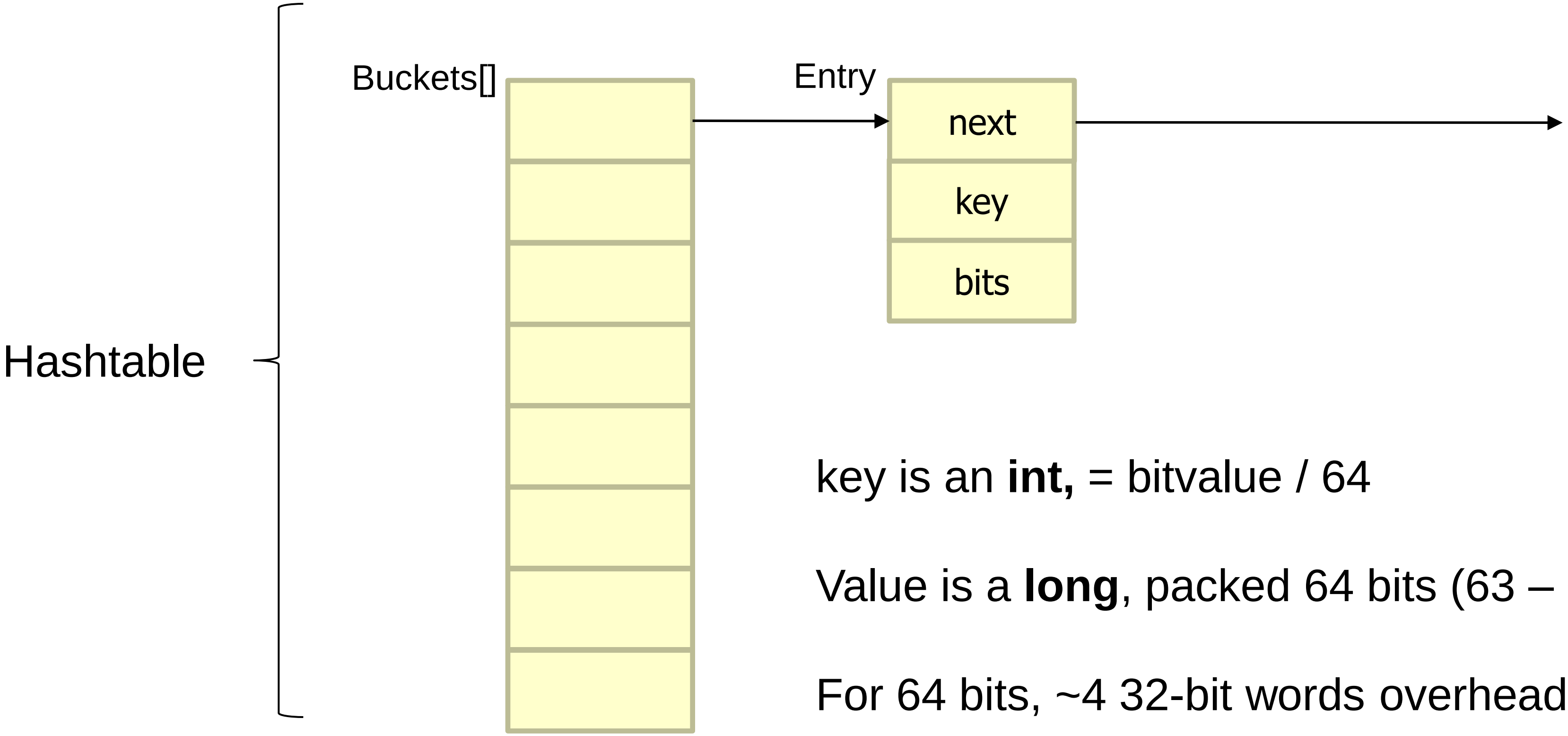
keyValue = bitvalue / 64

bits = packed 64 bits (63 – 0)

For 64 bits, ~8 32-bit words overhead

Copyright © 2010 Paladin Software Incorporated, Inc.

The (homegrown) Hashtable solution



key is an **int,** = bitvalue / 64

Value is a **long**, packed 64 bits (63 – 0)

For 64 bits, ~4 32-bit words overhead

Copyright © 2010 Paladin Software Incorporated, Inc.

The "Bucket by Masking" Problem

- HashMap uses masking by powers of 2 to obtain a bucket number.
- If clumping occurs, it remains when the bucket table is resized.
- Replaced masking by remaindering modulo the given table of primes, for table sizes of 2^n
- Every table resize returned average list length to 1.1.

- Primes <= 2^n (n = 0, 25)

  1, 2, 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381, 32749, 65521, 131071, 262139, 524287, 1048573, 2097143, 4194301, 8388593, 16777213, 33554393

Copyright © 2010 Paladin Software Incorporated, Inc.

The Assessment and Response

- JUnit tests (with timers) were built to test every method (verified against **BitSet**).
- The speed of operation seems good, but this effort started with concern over memory usage.
- Other problems:
  - Sequential access is less than efficient: hash computation is done to find the “next” 64-bits.
  - Bucket lists can be very long, and this destroys performance.
- Perhaps time for some real research into the theory—

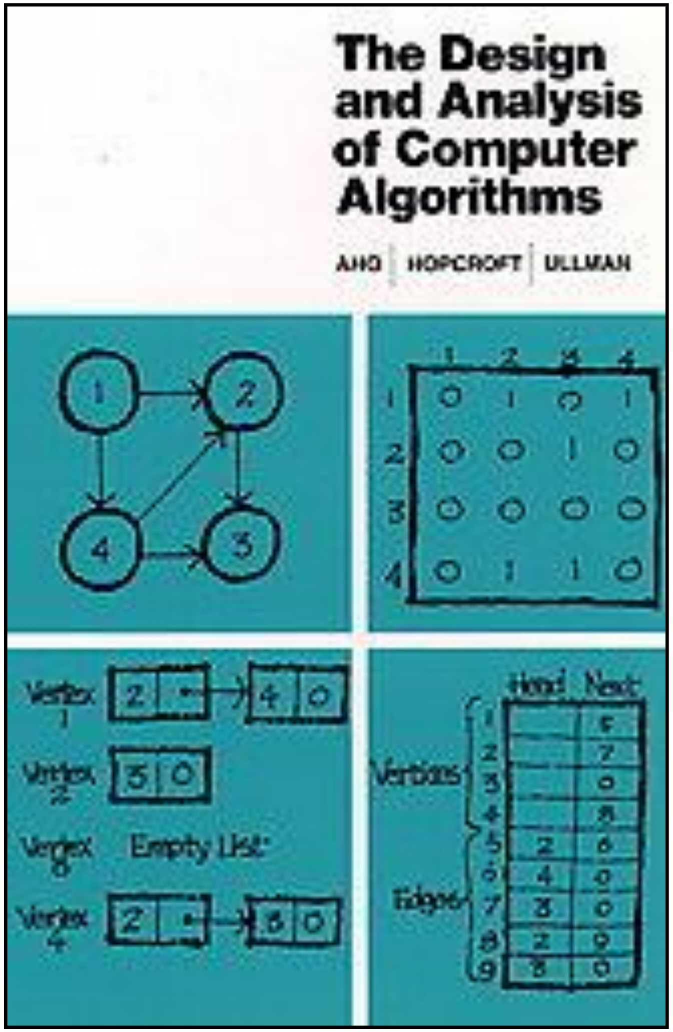
Copyright © 2010 Paladin Software Incorporated, Inc.

The Theory

An alternative to the list is a bit vector representation of sets. Assume the universe of discourse U (of which all sets are subsets) has n members. Linearly order the elements of U. A subset $S \subseteq U$ is represented as a vector $\mathbf{v}_S$, of n bits, where the ith bit in $\mathbf{v}_S$ is 1 if and only if the ith element of U is an element of S. We call $\mathbf{v}_S$ the *characteristic vector* of S.

The bit vector representation has the advantage that one can determine whether the ith element of U is an element of a set in time independent of the size of the set. Furthermore, basic operations on sets such as union and intersection can be carried out by the bit vector operations $\vee$ and $\wedge$.

(page 49)



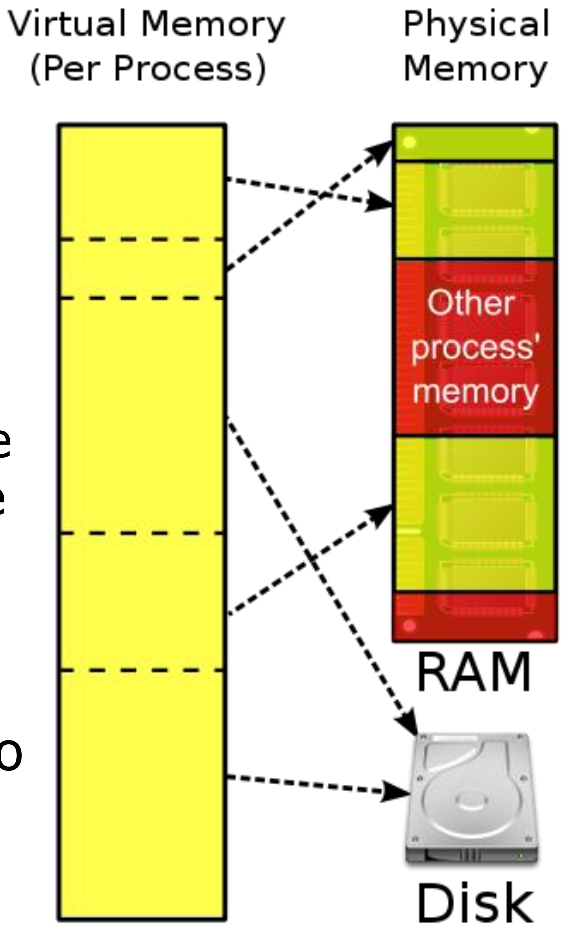
The Design and Analysis of Computer Algorithms

AHO | HOPCROFT | ULLMAN

Copyright © 2010 Paladin Software Incorporated, Inc.

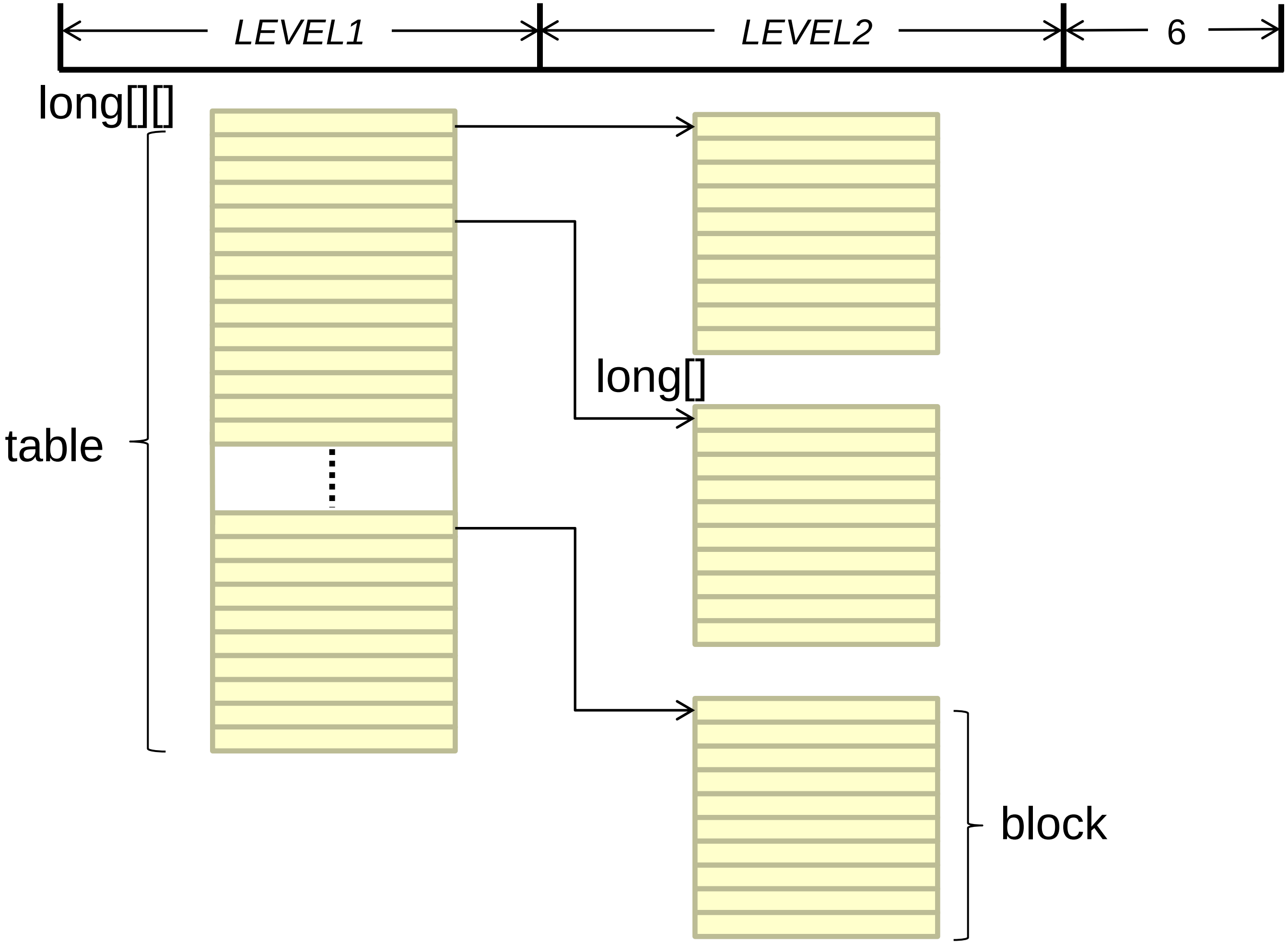
"Virtual Memory"

- After some thought, decided to try (as an experiment) a "virtual memory" approach
  - This thought pattern evolved over a period of not touching the code, and the process of evolution was neither science nor engineering (otherwise, a true "hack").
- In a "virtual memory" scenario, a table of virtual addresses is kept, that point to real memory if that memory is actually needed, and do not point anywhere if that memory is not actually in use.



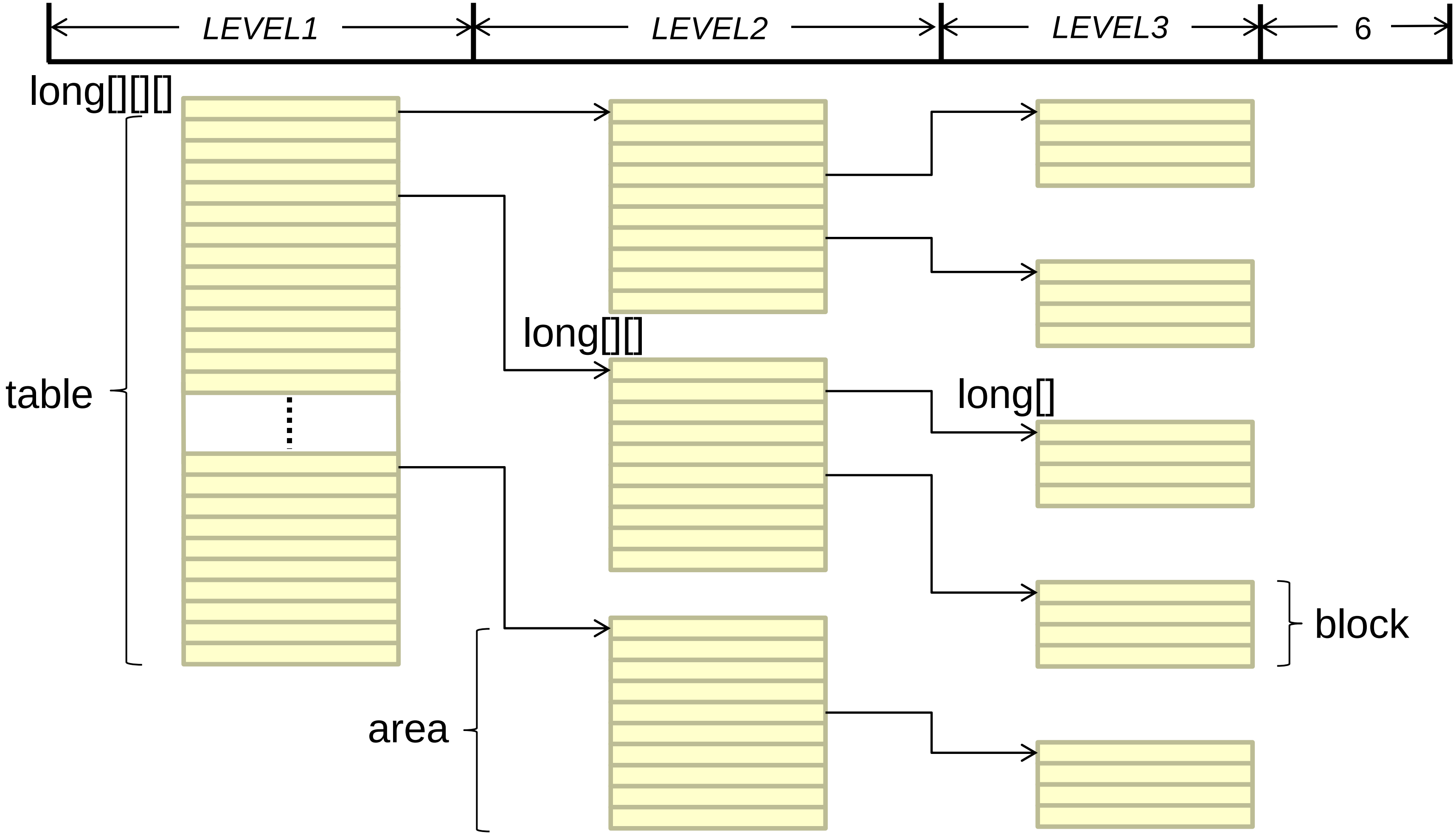
Copyright © 2010 Paladin Software Incorporated, Inc.

The first "virtual memory" structure



Copyright © 2010 Paladin Software Incorporated, Inc.

The final "virtual memory" structure



Copyright © 2010 Paladin Software Incorporated, Inc.

Example: code based on “virtual memory”: flip

```
public void flip(int i)
{
   if( (i + 1) < 1  )throw new IndexOutOfBoundsException("i=" + i);
   final int w = i >> SHIFT3;
   final int w1 = w >> SHIFT1;
   final int w2 = (w >> SHIFT2) & MASK2;

   if( i >= bitsLength ) resize(i);
   long[][] a2;
   if( (a2 = bits[w1]) == null )
         a2 = bits[w1] = new long[LENGTH2][];
   long[] a3;
   if( (a3 = a2[w2]) == null )   a3 = a2[w2] = new long[LENGTH3];
   a3[w & MASK3] ^= 1L << i;  //Flip the designated bit
   cache.hash = 0;      // Invalidate size, etc., values
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

Example: code based on “virtual memory”: clear

```
public void clear(int i)
{
   /* In the interests of speed, no check is made here on whether
      the level3 block goes to all zero. This may be found and
      corrected in some later operation. */
   if( (i + 1) < 1 ) throw new IndexOutOfBoundsException("i=" + i);
   if( i > bitsLength ) return;
   final int w = i >> SHIFT3;
   long[][] a2;
   if( (a2 = bits[w >> SHIFT1]) == null ) return;
   long[] a3;
   if( (a3 = a2[(w >> SHIFT2) & MASK2]) == null ) return;
   a3[w & MASK3] &= ~(1L << i);  // Clear the indicated bit
   cache.hash = 0;   // Invalidate size, etc.,
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

Example: code based on “virtual memory”: and

```
for( int w1 = 0; w1 != length1; ++w1 ) {
   if( (b_level2 = b_level1[w1]) == null ) level1[w1] = null;
   else if( (level2 = level1[w1]) != null ) {
      boolean level2_area_is_all_null = true;  // Pre-assumption
      for( int w2 = 0; w2 != LENGTH2; ++w2 ) {
         if( (b_level3 = b_level2[w2]) == null ) level2[w2] = null;
         else if( (level3 = level2[w2]) != null ) {
            final int index_base = (w1 << SHIFT1) + (w2 << SHIFT2);
            boolean level3_block_is_all_zero = true;
            for( int w3 = 0; w3 != LENGTH3; ++w3 )
               if( (word = (level3[w3] &= b_level3[w3])) != 0 )
                  level3_block_is_all_zero = false;
            if( level3_block_is_all_zero ) level2[w2] = null;
         }
         if( level2[w2] != null ) level2_area_is_all_null = false;
      }
      if( level2_area_is_all_null ) level1[w1] = null;
   }
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

Real Software Engineering Research

- Looking at the way blocks were allocated and discarded, a side project was mounted to investigate keeping of pool of blocks:
    - A stack where discarded blocks were placed (to a given limit). Each discarded block had to be guaranteed empty.
    - From which needed blocks were pop’ed as needed, and if the stack ran out, new blocks were constructed.
- A program was constructed to time this versus simple discarding with garbage collection, and allocation upon need.

Copyright © 2010 Paladin Software Incorporated, Inc.

New Allocation vs. Pooling Blocks

Count of allocations/discards	**Thread time(ms)** (2 sig. digits)		**Elapsed time(ms)** (2 sig. digits)	
	Using new	**“pooled”**	**Using new**	**“pooled”**
100,000	16	0	17	3
1,000,000	110	60	120	60
10,000,000	1000	600	1100	600
100,000,000	10000	6000	11000	6100
1,000,000,000	100000	60000	110000	60000

All timings in this presentation were made on a Lenovo ThinkPad, with an
Intel Core 2 DUO CPU T9600, 2.80 GHz, with 3GB RAM, under Windows XP SP3.

Copyright © 2010 Paladin Software Incorporated, Inc.

Example: code based on “virtual memory”: and

```
for( int w1 = 0; w1 != length1; ++w1 ) {
   if( (b_level2 = b_level1[w1]) == null ) level1[w1] = null;
   else if( (level2 = level1[w1]) != null ) {
      boolean level2_area_is_all_null = true;  // Pre-assumption
      for( int w2 = 0; w2 != LENGTH2; ++w2 ) {
         if( (b_level3 = b_level2[w2]) == null ) level2[w2] = null;
         else if( (level3 = level2[w2]) != null ) {
            final int index_base = (w1 << SHIFT1) + (w2 << SHIFT2);
            boolean level3_block_is_all_zero = true;
            for( int w3 = 0; w3 != LENGTH3; ++w3 )
               if( (word = (level3[w3] &= b_level3[w3])) != 0 )
                  level3_block_is_all_zero = false;
            if( level3_block_is_all_zero ) level2[w2] = null;
         }
         if( level2[w2] != null ) level2_area_is_all_null = false;
      }
      if( level2_area_is_all_null ) level1[w1] = null;
   }
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

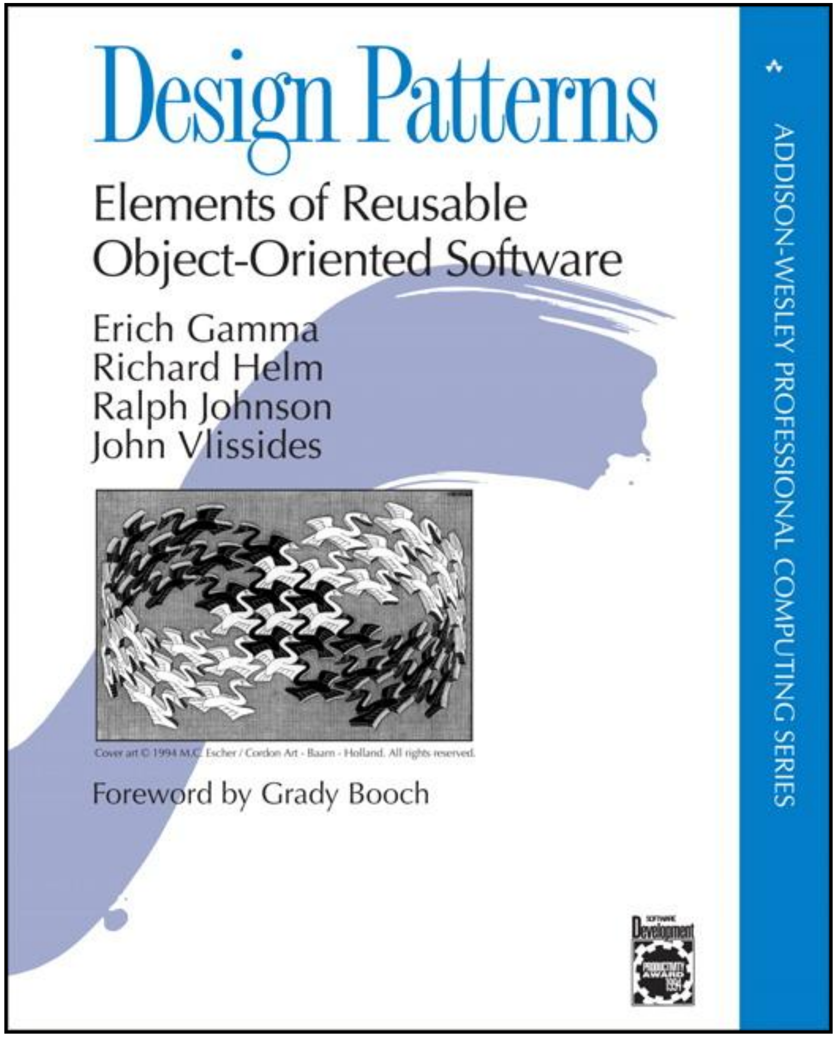
Hand Waving

- The next part is difficult to describe.
- Problem: lots of very similar, repeated code.
- Try a management method for the “whole set” problems, with flags and switches for the various kinds of operations.
- Similarly, for the “scanning set” operations.
- Merge these, by treating “whole set” as scanning from 0 to *bitsLength*.
- Bugs galore in trying to get all the “special” cases to work (everything is fragile).
- Start to suspect that need to walk code over the structure (Visitor Pattern?).

Copyright © 2010 Paladin Software Incorporated, Inc.

Design Patterns

- “Visitor” turned out to be the wrong concept.
- **Strategy** was the needed pattern.
    - One management routine to walk the structure, and maintain it.
    - At the critical points in the walk, invoke the method of a Strategy to perform the appropriate actions.



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Copyright © 2010 Paladin Software Incorporated, Inc.

AbstractStrategy (UML)

#F_OP_F_EQ_F : const int
#F_OP_X_EQ_F : const int
#X_OP_F_EQ_F : const int
#X_OP_F_EQ_X : const int

***#properties*() : int;**
***#start*(b : SparseBitSet) : void**
***#word*(base : int , u3: int, a3 : long[] ,**
b3 : long[] , mask : long) : boolean
***#block*(base : int, u3 : int, v3 : int, a3 : long[] ,**
b3 : long[]) : boolean
#void finish(a2Count : int, a3Count: int)
#isZeroBlock(a3 : long[]) : boolean

- The Strategies
  - AndStrategy
  - AndNotStrategy
  - ClearStrategy
  - CopyStrategy
  - EqualsStrategy
  - FlipStrategy
  - IntersectsStrategy
  - OrStrategy
  - SetStrategy.
  - UpdateStrategy
  - XorStrategy

Copyright © 2010 Paladin Software Incorporated, Inc.

A Strategy

```
protected class AndStrategy extends AbstractStrategy {
protected int properties() {
   return F_OP_F_EQ_F + F_OP_X_EQ_F + X_OP_F_EQ_F;
}
protected void start(SparseBitSet b) {
   if( b == null ) throw new NullPointerException();
   cache.hash = 0;
}
protected boolean word(int base, int u3, long[] a3, long[] b3, long mask) {
   return (a3[u3] &= b3[u3] | ~mask) == 0L;
}
protected boolean block(int base, int u3, int v3, long[] a3, long[] b3) {
   boolean isZero = true;       // Presumption
   for( int w3 = u3; w3 != v3; ++w3 ) isZero &= ((a3[w3] &= b3[w3]) == 0L);
   return isZero;
}
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

And now the code!

```
public void and(SparseBitSet b)
{
   nullify(Math.min(bits.length, b.bits.length));    // Optimisation
   setScanner(0, Math.min(bitsLength, b.bitsLength), b, andStrategy);
}
```

```
public void andNot(int i, int j, SparseBitSet b)
                              throws IndexOutOfBoundsException
{
   setScanner(i, j, b, andNotStrategy);
}
```

```
public void flip(int i, int j) throws IndexOutOfBoundsException
{
   setScanner(i, j, null, flipStrategy);
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.

LEVEL2 and LEVEL3 sizes

level 2

	8	16	32	64	128
128	12.5	8.8	9.9	11.5	15.1
64	7.6	8.0	7.0	8.5	12.5
32	6.1	5.5	6.8	7.6	11.4
16	5.5	4.9	5.5	7.2	10.4

level 3

Copyright © 2010 Paladin Software Incorporated, Inc.

BitSet *(UML)*

+BitSet()
+BitSet(int : nbits)
+and(i : int, j : int, b : SparseBitSet) : void
+andNot(i : int, j : int,
b : SparseBitSet) : void
+or(i : int, j : int, b : SparseBitSet) : void
+statistics() : String
+statistics(String[] values) : String
+toStringCompaction(boolean change) : void
+toStringCompaction(int count) : void
+xor(i : int, j : int, b : SparseBitSet) : void

- Made possible the easy addition of bounded operations (almost impossible to compose using the given operations).
- Added methods to report the internal statistics of the class (to help with memory management).
- Added an option to shorten the *toString* output (for "(1, 2, 3, 4, 8)" give "(1..4, 8)".

Copyright © 2010 Paladin Software Incorporated, Inc.

So How Did It Do?

	BitSet	Sparse (hash)	%	Sparse (VM)	%
Set/flip/clear 1	6.42	5.16	80	4.53	71
Set/flip/clear 2	2.13	4.00	188	2.55	120
Get (bulk)	26.00	8.58	33	1.48	6
Random set/get/clear individual bits	6.50	6.14	94	5.67	87
Performance comparison	24.93	1.97	8	6.76	27

Set/flip/clear 1	Sets bits 0 to 100,000,000 individually, and flips them, then sets them again, and clears them.
Set/flip/clear 2	Bulk sets bits 0 to 1,000,000, and bits 4,000,000 to 10,000,000, bulk flips them, the does it again, and clears them, repeated 300 times.
Get(bulk)	Gets a new set (with bits 0 to 100,000 and bits 9,900,000 to 10,000,000 set) from an existing set 20,000 times
Random *etc.*	Sets random bits in the range 0 to 10,000, 10,000 times, does a get on each bit, and then clears it, the whole sequence done 8,000 times
Performance	A mixture of operations, including construction, setting, getting, clearing, doing various and, or, and xor operations, checking cardinality and hashCode, cloning, and finding intersections, done 100,000 times.

Copyright © 2010 Paladin Software Incorporated, Inc.

And this started with memory concerns

- For 2^{26} 64-bit words the overhead is approximately:
  - 2^{15} 32-bit pointers to areas
  - 2^{15} areas with 32 32-bit pointers plus 1 word object overhead
  - 2^{20} words of overhead across all the blocks.

32768
1081344
1048576
2162688

  - which gives ~0.03 32-bit words overhead per 64 bits.

Copyright © 2010 Paladin Software Incorporated, Inc.

The Experience

- There is no way that the final structure and algorithmic behavior of this component could have been predicted, designed, or otherwise anticipated at the beginning of the project.
- The Computer Science contributed as a source of techniques and their properties: hashing, hash tables, linked lists, bit vectors, object pooling, and design patterns.
- The engineering practices of constant testing, prototyping, performance and resource measurement, benefit trade-offs, tooling, and version tracking both indicated and constrained choices.
- But these disciplines do not capture the entirety of the experience!

Copyright © 2010 Paladin Software Incorporated, Inc.

The Bottom Line

- Developing effective and correct software (applications, components, libraries, etc.) requires understanding the possible—this is the Computer Science.
- Developing efficient and useful software is a trade-off between many issues (more than just space and time, resources, and cost), that constitute the practice of Software Engineering.
- But there remains that activity that lies between inspiration and perspiration, which is not Computer Science, and which is not Software Engineering, for which no name but "hacking" currently exists.
- The challenge, here, and elsewhere, is how to teach this skill (?), and how to bring new programmers to the place where they experience the reward.

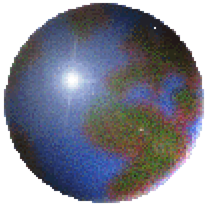
Copyright © 2010 Paladin Software Incorporated, Inc.

Please,

Ask questions!

Thank you!

Copyright © 2010 Paladin Software Incorporated, Inc.



Is it Computer Science, Software Engineering, or Hacking?

Dr. Bruce K. Haddon

`Bruce.Haddon@colorado.edu`

Paladin Software International
INCORPORATED