Principles of Software Construction: Design Case Study of the Collections API

Christian Kaestner    Bogdan Vasilecu

Many slides stolen with permission from: Josh Bloch (thanks!)
Midsemester Feedback
public class ActionListenerPanel extends JPanel implements ActionListener {

    public ActionListenerPanel() { setup(); }
    private void setup() {
        button1 = new JButton("a");
        button1.addActionListener(this);
        button2 = new JButton("b");
        button2.addActionListener(this);
        add(button1); add(button2);
    }
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==button1)
            display.setText(BUTTON_1);
        else if(e.getSource()==button1) ... 
    }
}
Learning goals for today

• Understand the design aspects of collection libraries
• Recognize the design goals and design patterns used ("learn from experts")
• Be able to use common collection classes, including helpers in the Collections class.
Designing a data structure library

• Different data types: lists, sets, maps, stacks, queues, ...

• Different representations
  – Array-based lists vs. linked lists
  – Hash-based sets vs. tree-based sets
  – ...

• Many alternative designs
  – Mutable vs. immutable
  – Sorted vs. unsorted
  – Accepts null or not
  – Accepts duplicates or not
  – Concurrency/thread-safe or not
  – ...

We take you back now to the late ‘90s

• It was a simpler time
  – Java had only Vector, Hashtable & Enumeration
  – But it needed more; platform was growing!
• The barbarians were pounding the gates
  – JGL was a transliteration of STL to Java
  – It had 130 (!) classes and interfaces
  – The JGL designers wanted badly to put it in the JDK
• It fell to me to design something better 😊
Here’s the first collections talk ever

• Debuted at JavaOne 1998
• No one knew what a collections framework was
  – Or why they needed one
• Talk aimed to
  – Explain the concept
  – Sell Java programmers on this framework
  – Teach them to use it
The Java™ Platform Collections Framework

Joshua Bloch
Sr. Staff Engineer, Collections Architect
Sun Microsystems, Inc.
What is a Collection?

- Object that groups elements
- Main Uses
  - Data storage and retrieval
  - Data transmission
- Familiar Examples
  - java.util.Vector
  - java.util.Hashtable
  - array
What is a Collections Framework?

• Unified Architecture
  – Interfaces - implementation-independence
  – Implementations - reusable data structures
  – Algorithms - reusable functionality

• Best-known examples
  – C++ Standard Template Library (STL)
  – Smalltalk collections
Benefits

- Reduces programming effort
- Increases program speed and quality
- Interoperability among unrelated APIs
- Reduces effort to learn new APIs
- Reduces effort to design new APIs
- Fosters software reuse
Design Goals

• Small and simple
• Reasonably powerful
• Easily extensible
• Compatible with preexisting collections
• Must feel familiar
Architecture Overview

- Core Collection Interfaces
- General-Purpose Implementations
- Wrapper Implementations
- Abstract Implementations
- Algorithms
Core Collection Interfaces
Collection Interface

public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

    Object[] toArray();
    T[] toArray(T a[]);

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> c); // Optional
    boolean removeAll(Collection<?> c); // Optional
    boolean retainAll(Collection<?> c); // Optional
    void clear(); // Optional
}

Iterator Interface

• Replacement for Enumeration interface
  – Adds remove method
  – Improves method names

public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();    // Optional
}
Collection Example

Reusable algorithm to eliminate nulls

```java
public static boolean removeNulls(Collection<?> c) {
    for (Iterator<?> i = c.iterator(); i.hasNext(); ) {
        if (i.next() == null) {
            i.remove();
        }
    }
}
```
Set Interface

• Adds no methods to Collection!
• Adds stipulation: no duplicate elements
• Mandates equals and hashCode calculation

public interface Set<E> extends Collection<E> {
}

Set Idioms

Set<Type> s1, s2;

boolean isSubset = s1.containsAll(s2);

Set<Type> union = new HashSet<>(s1);
union = union.addAll(s2);

Set<Type> intersection = new HashSet<>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<>(s1);
difference.removeAll(s2);

Collection<Type> c;
Collection<Type> noDups = new HashSet<>(c);
List Interface

A sequence of objects

public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);    // Optional
    void add(int index, E element); // Optional
    Object remove(int index);       // Optional
    boolean addAll(int index, Collection<? extends E> c);
        // Optional
    int indexOf(Object o);
    int lastIndexOf(Object o);

    List<E> subList(int from, int to);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
}
List Example

Reusable algorithms to swap and randomize

```java
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}

private static Random r = new Random();

public static void shuffle(List<?> a) {
    for (int i = a.size(); i > 1; i--)
        swap(a, i - 1, r.nextInt(i));
}
```
List Idioms

List<
\[
\text{Type}\] a, b;
\]

// Concatenate two lists
a.addAll(b);

// Range-remove
a.subList(from, to).clear();

// Range-extract
List<
\[
\text{Type}\] partView = a.subList(from, to);
List<
\[
\text{Type}\] part = new ArrayList<>(partView);
partView.clear();
Map Interface

A key-value mapping

public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Object get(Object key);
    Object put(K key, V value);   // Optional
    Object remove(Object key);    // Optional
    void putAll(Map<? Extends K, ? Extends V> t); // Opt.
    void clear();                 // Optional
}

// Collection Views
public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K,V>> entrySet();
Map Idioms

// Iterate over all keys in Map m
Map<Key, Val> m;
for (iterator<Key> i = m.keySet().iterator(); i.hasNext(); )
    System.out.println(i.next());

// As of Java 5 (2004)
for (Key k : m.keySet())
    System.out.println(i.next());

// "Map algebra"
Map<Key, Val> a, b;
boolean isSubMap = a.entrySet().containsAll(b.entrySet());
Set<Key> commonKeys =
    new HashSet<>(a.keySet()).retainAll(b.keySet()); [sic!]
// Remove keys from a that have mappings in b
a.keySet().removeAll(b.keySet());
General Purpose Implementations

Consistent Naming and Behavior
Choosing an Implementation

- **Set**
  - `HashSet` -- O(1) access, no order guarantee
  - `TreeSet` -- O(log n) access, sorted

- **Map**
  - `HashMap` -- (See `HashSet`)
  - `TreeMap` -- (See `TreeSet`)

- **List**
  - `ArrayList` -- O(1) random access, O(n) insert/remove
  - `LinkedList` -- O(n) random access, O(1) insert/remove;
    - Use for queues and deques *(no longer a good idea!)*
Implementation Behavior

Unlike Vector and Hashtable...

• Fail-fast iterator
• Null elements, keys, values permitted
• **Not** thread-safe
Synchronization Wrappers

A new approach to thread safety

- Anonymous implementations, one per core interface
- Static factories take collection of appropriate type
- Thread-safety assured if all access through wrapper
- Must manually synchronize iteration

- It was new then; it’s old now!
  - Synch wrappers are largely obsolete
  - Made obsolete by concurrent collections
Synchronization Wrapper Example

Set<String> s = Collections.synchronizedSet(new HashSet<>());
...
s.add("wombat");  // Thread-safe
...
synchronized(s) {
    Iterator<String> i = s.iterator();  // In synch block!
    while (i.hasNext())
        System.out.println(i.next());
}

// In Java 5 (post-2004)
synchronized(s) {
    for (String t : s)
        System.out.println(i.next());
}
Unmodifiable Wrappers

• Analogous to synchronization wrappers
  – Anonymous implementations
  – Static factory methods
  – One for each core interface

• Provide read-only access
Convenience Implementations

- **Arrays.asList(E[] a)**
  - Allows array to be "viewed" as List
  - Bridge to Collection-based APIs

- **EMPTY_SET, EMPTY_LIST, EMPTY_MAP**
  - immutable constants

- **singleton(E o)**
  - immutable set with specified object

- **nCopies(E o)**
  - immutable list with n copies of object
Custom Implementation Ideas

- Persistent
- Highly concurrent
- High-performance, special-purpose
- Space-efficient representations
- Fancy data structures
- Convenience classes
Custom Implementation Example

*It’s easy with our abstract implementations*

// List adapter for primitive int array
public static List intArrayList(int[] a) {
    return new AbstractList() {
        public Integer get(int i) {
            return new Integer(a[i]);
        }

        public int size() { return a.length; }

        public Object set(int i, Integer e) {
            int oldVal = a[i];
            a[i] = e.intValue();
            return new Integer(oldVal);
        }
    };
}
Reusable Algorithms

static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<? extends T> list);
static void shuffle(List<? extends T> list);
Algorithm Example 1

*Sorting lists of comparable elements*

List<String> strings;  // Elements type: String

...  

Collections.sort(strings);  // *Alphabetical order*


LinkedList<Date> dates;  // Elements type: Date

...  

Collections.sort(dates);  // *Chronological order*

// Comparable interface (Infrastructure)
public interface Comparable<E extends Comparable<E>> {
    int compareTo(Object o);
}
Comparator Interface

*Infrastructure*

• Specifies order among objects
  – Overrides natural order on comparables
  – Provides order on non-comparables

```java
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```
Algorithm Example 2

*Sorting with a comparator*

List<String> strings; // Element type: String

Collections.sort(strings, Collections.ReverseOrder());

// Case-independent alphabetical order
static Comparator<String> cia = new Comparator<>(){
    public int compare(String c1, String c2) {
        return c1.toLowerCase().compareTo(c2.toLowerCase());
    }
};

Collections.sort(strings, cia);
Compatibility

*Old and new collections interoperate freely*

- **Upward Compatibility**
  - Vector<E> implements List<E>
  - Hashtable<K,V> implements Map<K,V>
  - Arrays.asList(myArray)

- **Backward Compatibility**
  - myCollection.toArray()
  - new Vector<>(myCollection)
  - new Hashtable<>(myMap)
API Design Guidelines

• Avoid ad hoc collections
  – Input parameter type:
    • Any collection **interface** (Collection, Map best)
    • Array may sometimes be preferable
  – Output value type:
    • Any collection **interface** or **class**
    • Array

• Provide adapters for your legacy collections
Sermon

• Programmers:
  – Use new implementations and algorithms
  – Write reusable algorithms
  – Implement custom collections

• API Designers:
  – Take collection interface objects as input
  – Furnish collections as output
For More Information

Takeaways

• Collections haven’t changed that much since ‘98
• API has grown, but essential character unchanged
  – With arguable exception of Java 8 streams (2014)
FINDING DESIGN PATTERNS IN COLLECTIONS
One problem: Java arrays are not Collections

• To convert a Collection to an array
  – Use the toArray() method
    List<String> arguments = new LinkedList<String>();
    ...  // puts something into the list
    String[] arr = (String[]) arguments.toArray();
    String[] brr = arguments.toArray(new String[0]);

• To view an array as a Collection
  – Use the java.util.Arrays.asList() method
    String[] arr = {"foo", "bar", "baz", "qux"};
    List<String> arguments = Arrays.asList(arr);
The Iterator Pattern

- Single Iterator interface
- Iterate over Lists, Sets, Maps, Map values, ...
Reminder: The Iterator design pattern

• Provide a strategy to uniformly access all elements of a container in a sequence
  – Independent of the container implementation
  – Ordering is unspecified, but every element visited once

• Design for change, information hiding
  – Hides internal implementation of container behind uniform explicit interface

• Design for reuse, division of labor
  – Hides complex data structure behind simple interface
  – Facilitates communication between parts of the program
Factory Method Pattern for Creating Iterators

public interface Collection<E> {
    boolean add(E e);
    boolean addAll(Collection<E> c);
    boolean remove(E e);
    boolean removeAll(Collection<E> c);
    boolean retainAll(Collection<E> c);
    boolean contains(E e);
    boolean containsAll(Collection<E> c);
    void clear();
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    E[] toArray(E[] a);
    ...
}
The Factory Method design pattern

```
AnOperation() {
    product = FactoryMethod();
}
```
The Factory Method design pattern

- **Applicability**
  - A class can’t anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates

- **Consequences**
  - Provides hooks for subclasses to customize creation behavior
  - Connects parallel class hierarchies
The abstract `java.util.AbstractList<T>`

```java
abstract T get(int i); // Template Method pattern
abstract int size(); // Template Method pattern
boolean set(int i, E e); // set add remove are
boolean add(E e); // pseudo-abstract,
boolean remove(E e); // Template Methods pattern
boolean addAll(Collection<E> c);
boolean removeAll(Collection<E> c);
boolean retainAll(Collection<E> c);
boolean contains(E e);
boolean containsAll(Collection<E> c);
void clear();
boolean isEmpty();
Iterator<E> iterator();
Object[] toArray();
E[] toArray(E[] a);
...
```
Sorting Collections

- Comparable Interface
- Comparator Interface
Marker Interfaces

• UnmodifiableCollection
• RandomAccess
• Set (used to be)

• Serializable

• Marker interfaces add invariants, no code
The java.util.Collections class

Immutable collections:

```java
static List<T> unmodifiableList(List<T> lst);
static Set<T> unmodifiableSet(Set<T> set);
static Map<K,V> unmodifiableMap(Map<K,V> map);
```

Similar for synchronization:

```java
static List<T> synchronizedList(List<T> lst);
static Set<T> synchronizedSet(Set<T> set);
static Map<K,V> synchronizedMap(Map<K,V> map);
```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E>
    implements Collection<E>, Serializable {

    final Collection<E> c;

    UnmodifiableCollection(Collection<> c) {this.c = c; }

    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) { throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) { return c.containsAll(coll); }
    public boolean addAll(Collection<? extends E> coll) { throw new UnsupportedOperationException();}
    public boolean removeAll(Collection<?> coll) { throw new UnsupportedOperationException();}
    public boolean retainAll(Collection<?> coll) { throw new UnsupportedOperationException();}
    public void clear() { throw new UnsupportedOperationException();}
}

What design pattern is this?

UnmodifiableCollection decorates Collection by removing functionality.
Design Patterns in Collections

- Design Goals: Collections as reusable and extensible data structures
  - design for reuse
  - design for change
- Iterators to abstract over internal structure
- Decorator to attach behavior at runtime
- Template methods and factory methods to support behavior customization in subclasses
- Adapters to bridge between implementations
- Strategy pattern for sorting
DISCUSSION: DESIGNING AN API (JOSH BLOCH)
A wonderful source of use cases

“Good artists copy, great artists steal.” – Pablo Picasso
You must maintain an *issues list*

- Centralizes all open and closed design issues
- List pros and cons for each possible decision
- Essential for efficient progress
- Forms the basis of a design rationale
The first draft of API was not so nice

- Map was called Table
- No HashMap, only Hashtable
- No algorithms (Collections, Arrays)
- Contained some unbelievable garbage
Automatic alias detection
A horrible idea that died on the vine

/**
 * This interface must be implemented by Collections and Tables that are
 * `views` on some backing collection. (It is necessary to
 * implement this interface only if the backing collection is not
 * `encapsulated` by this Collection or Table; that is, if the
 * backing collection might conceivably be be accessed in some way other
 * than through this Collection or Table.) This allows users
 * to detect potential `aliasing` between collections.
 * <p>
 * If a user attempts to modify one collection
 * object while iterating over another, and they are in fact views on
 * the same backing object, the iteration may behave erratically.
 * However, these problems can be prevented by recognizing the
 * situation, and "defensively copying" the Collection over which
 * iteration is to take place, prior to the iteration.
 * */

public interface Alias {

    /**
     * Returns the `identityHashCode` of the object "ultimately backing" this
     * collection, or zero if the backing object is undefined or unknown.
     * The purpose of this method is to allow the programmer to determine
     * when the possibility of `aliasing` exists between two collections
     * (in other words, modifying one collection could affect the other). This
     * is critical if the programmer wants to iterate over one collection and
     * modify another; if the two collections are aliases, the effects of
     * the iteration are undefined, and it could loop forever. To avoid
     * this behavior, the careful programmer must "defensively copy" the
     * collection prior to iterating over it whenever the possibility of
     * aliasing exists.
     * <p>
     * If this collection is a view on an Object that does not implement
     * `Alias`, this method must return the `backingObjectId` of the backing
     * Object. For example, a List backed by a user-provided array would
     * return the `IdentityHashCode` of the array.
     * 
     * If this collection is a `view` on another Object that implements
     * `Alias`, this method must return the `backingObjectId` of the backing
     * Object. (To avoid the cost of recursive calls to this method, the
     * `backingObjectId` may be cached at creation time).
     * <p>
     * For all collections backed by a particular "external data source" (a
     * SQL database, for example), this method must return the same value.
     * The `IdentityHashCode` of a "proxy" Object created just for this
     * purpose will do nicely, as will a pseudo-random integer permanently
     * associated with the external data source.
     * <p>
     * For any collection backed by multiple Objects (a "concatenation
     * view" of two Lists, for instance), this method must return zero.
     * Similarly, for any `view` collection for which it cannot be
     * determined what Object backs the collection, this method must return
     * zero. It is always safe for a collection to return zero as its
     * `backingObjectId`, but doing so when it is not necessary will lead to
     * inefficiency.
     * <p>
     * The possibility of aliasing between two collections exists iff
     * any of the following conditions are true:
     * <ol>
     * <li>The two collections are the same Object.
     * <li>Either collection implements `Alias` and has a
     *     `backingObjectId` that is the `identityHashCode` of
     *     the other collection.
     * <li>Either collection implements `Alias` and has a
     *     `backingObjectId` of zero.
     * <li>Both collections implement `Alias` and they have equal
     *     `backingObjectId`'s.</li>
     * </ol>
     * <p>
     * @see `java.lang.System#identityHashCode`
     * @since JDK1.2
     */
    int backingObjectId();
}
I received a *lot* of feedback

- Initially from a small circle of colleagues
  - Some *very* good advice
  - Some not so good

- Then from the public at large: beta releases
  - Hundreds of messages
  - Many API flaws were fixed in this stage
  - I put up with a lot of flaming
Review from a **very** senior engineer

<table>
<thead>
<tr>
<th>API</th>
<th>vote</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array</strong></td>
<td>yes</td>
<td>But remove binarySearch* and toList</td>
</tr>
<tr>
<td><strong>BasicCollection</strong></td>
<td>no</td>
<td>I don't expect lots of collection classes</td>
</tr>
<tr>
<td><strong>BasicList</strong></td>
<td>no</td>
<td>see List below</td>
</tr>
<tr>
<td><strong>Collection</strong></td>
<td>yes</td>
<td>But cut toArray</td>
</tr>
<tr>
<td><strong>Comparator</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>DoublyLinkedList</strong></td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td><strong>HashSet</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>LinkedList</strong></td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>no</td>
<td>I'd like to say yes, but it's just way bigger than I was expecting</td>
</tr>
<tr>
<td><strong>RemovalEnumeration</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td>yes</td>
<td>BUT IT NEEDS A DIFFERENT NAME</td>
</tr>
<tr>
<td><strong>TreeSet</strong></td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

I'm generally not keen on the toArray methods because they add complexity

Similarly, I don't think that the table Entry subclass or the various views mechanisms carry their weight.
# III. Evolution of Java collections

<table>
<thead>
<tr>
<th>Release, Year</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 1.0, 1996</td>
<td>Java Released: Vector, Hashtable, Enumeration</td>
</tr>
<tr>
<td>JDK 1.1, 1996</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.2, 1998</td>
<td>Collections framework added</td>
</tr>
<tr>
<td>J2SE 1.3, 2000</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.4, 2002</td>
<td>LinkedHashMap{Map,Set}, IdentityHashSet, 6 new algorithms</td>
</tr>
<tr>
<td>J2SE 5.0, 2004</td>
<td>Generics, for-each, enums: generified everything, Iterable Queue, Enum{Set,Map}, concurrent collections</td>
</tr>
<tr>
<td>Java 6, 2006</td>
<td>Deque, Navigable{Set,Map}, newSetFromMap, asLifoQueue</td>
</tr>
<tr>
<td>Java 7, 2011</td>
<td>No API changes. Improved sorts &amp; defensive hashing</td>
</tr>
<tr>
<td>Java 8, 2014</td>
<td>Lambdas (+ streams and internal iterators)</td>
</tr>
</tbody>
</table>
IV. Example – How to find anagrams

• Alphabetize the characters in each word
  – cat → act, dog → dgo, mouse → emosu
  – Resulting string is called *alphagram*

• Anagrams share the same alphagram!
  – stop → opst, post → opst, tops → opst, opts → opst

• So go through word list making “multimap” from alphagram to word!
public static void main(String[] args) throws IOException {
    // Read words from file and put into a simulated multimap
    Map<String, List<String>> groups = new HashMap<>();
    try (Scanner s = new Scanner(new File(args[0]))) {
        while (s.hasNext()) {
            String word = s.next();
            String alpha = alphabetize(word);
            List<String> group = groups.get(alpha);
            if (group == null)
                groups.put(alpha, group = new ArrayList<>());
            group.add(word);
        }
    }
}
// Print all anagram groups above size threshold
int minGroupSize = Integer.parseInt(args[1]);
for (List<String> group : groups.values())
    if (group.size() >= minGroupSize)
        System.out.println(group.size() + ": " + group);

// Returns the alphagram for a string
private static String alphabetize(String s) {
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
Demo – Anagrams
Two slides in Java vs. a chapter in STL

Java’s verbosity is somewhat exaggerated
V. Critique

Some things I wish I’d done differently

• Algorithms should return collection, not void or boolean
  – Turns ugly multiliners into nice one-liners
  
  ```java
  private static String alphabetize(String s) {
      return new String(Arrays.sort(s.toCharArray()));
  }
  ```

• Collection should have get(), remove()
  – Queue and Deque eventually did this

• Sorted{Set,Map} should have proper navigation
  – Navigable{Set,Map} are warts
Conclusion

• It takes a lot of work to make something that appears obvious
  – Coherent, unified vision
  – Willingness to listen to others
  – Flexibility to accept change
  – Tenacity to resist change
  – Good documentation!

• It’s worth the effort!
  – A solid foundation can last two+ decades