Programmer's Dozen

Thirteen Recommendations for Reviewing, Refactoring and Regaining Control of Code

Kevlin Henney

kevlin@curbralan.com

programmer *a person who writes computer programs.*
dozen *a group or set of twelve.*

The New Oxford Dictionary of English

Asymmetric bounds are most convenient to program in a language like C in which arrays start from zero: the exclusive upper bound of such an array is equal to the number of elements! Thus when we define a C array with [12] elements, 0 is inclusive lower bound and [12] the exclusive upper bound for the subscripts of the array.

Andrew Koenig

A baker's dozen contains thirteen items as opposed to the familiar twelve. This dates from the time when bakers were subject to heavy fines if they served under-weight bread. To avoid this danger bakers provided a surplus number of loaves, the thirteenth loaf in the dozen being called the vantage loaf.

Lock, Stock & Barrel

Code Meaning	*0*	Prefer Code to Comments
	1	Follow Consistent Form
	2	Employ the Contract Metaphor
Code Dependencies	*3*	Express Independent Ideas Independently
	4	Encapsulate
	5	Parameterize from Above
	6	Restrict Mutability of State
	7	Favour Symmetry over Asymmetry
Code Execution	*8*	Sharpen Fuzzy Logic
	9	Go with the Flow
	10	Let Code Decide
Code Consolidation	*11*	Omit Needless Code
	12	Unify Duplicate Code

Recommendation 0

Prefer Code to Comments

1. *If a program is incorrect, it matters little what the documentation says.*
2. *If documentation does not agree with the code, it is not worth much.*
3. *Consequently, code must largely document itself. If it cannot, rewrite the code rather than increase the supplementary documentation. Good code needs fewer comments than bad code does.*
4. *Comments should provide additional information that is not readily obtainable from the code itself. They should never parrot the code.*
5. *Mnemonic variable names and labels, and a layout that emphasizes logical structure, help make a program self-documenting.*

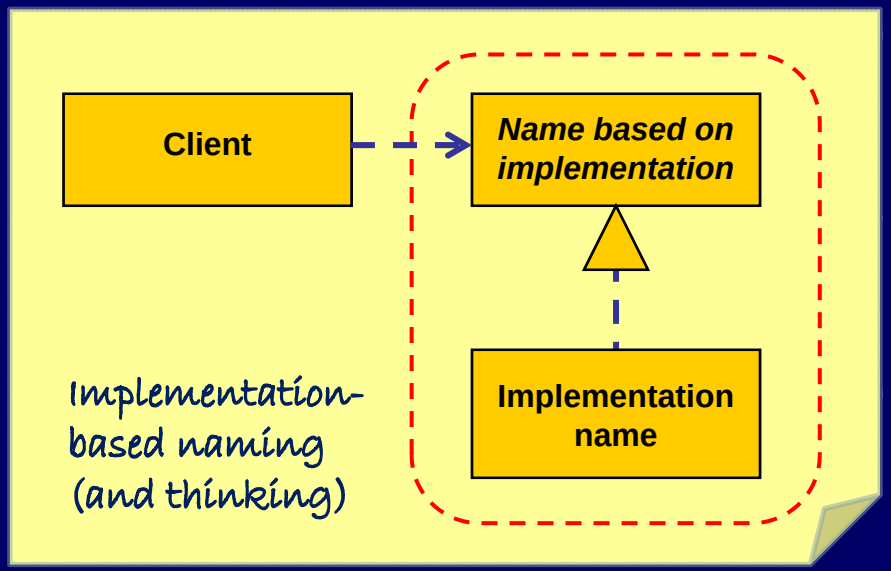
Kernighan and Plauger

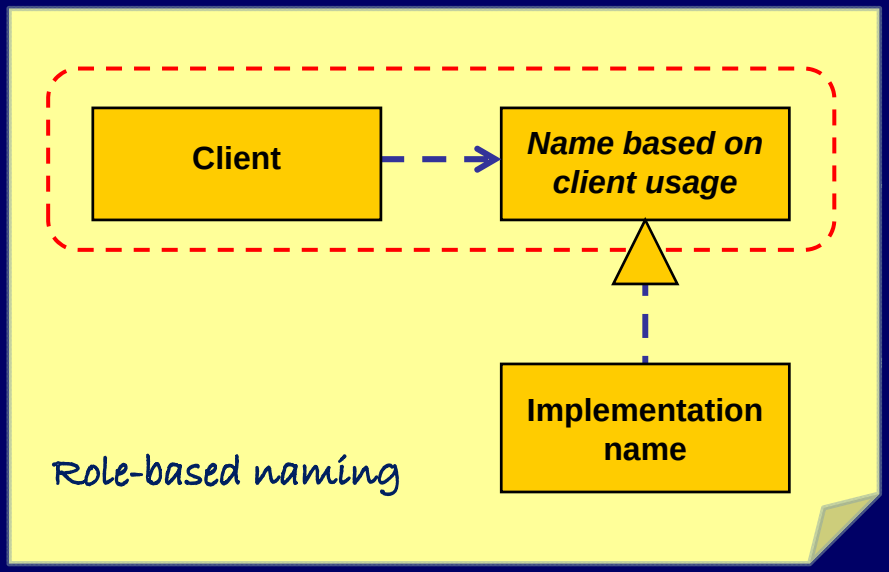
Recommendation 1

Follow Consistent Form

This principle, that of parallel construction, requires that expressions similar in context and function be outwardly similar. The likeness of form enables the reader to recognize more readily the likeness of content and function.

Strunk and White


Client
Name based on implementation
Implementation name
Implementation-based naming (and thinking)


Client
Name based on client usage
Implementation name
Role-based naming

Recommendation 2

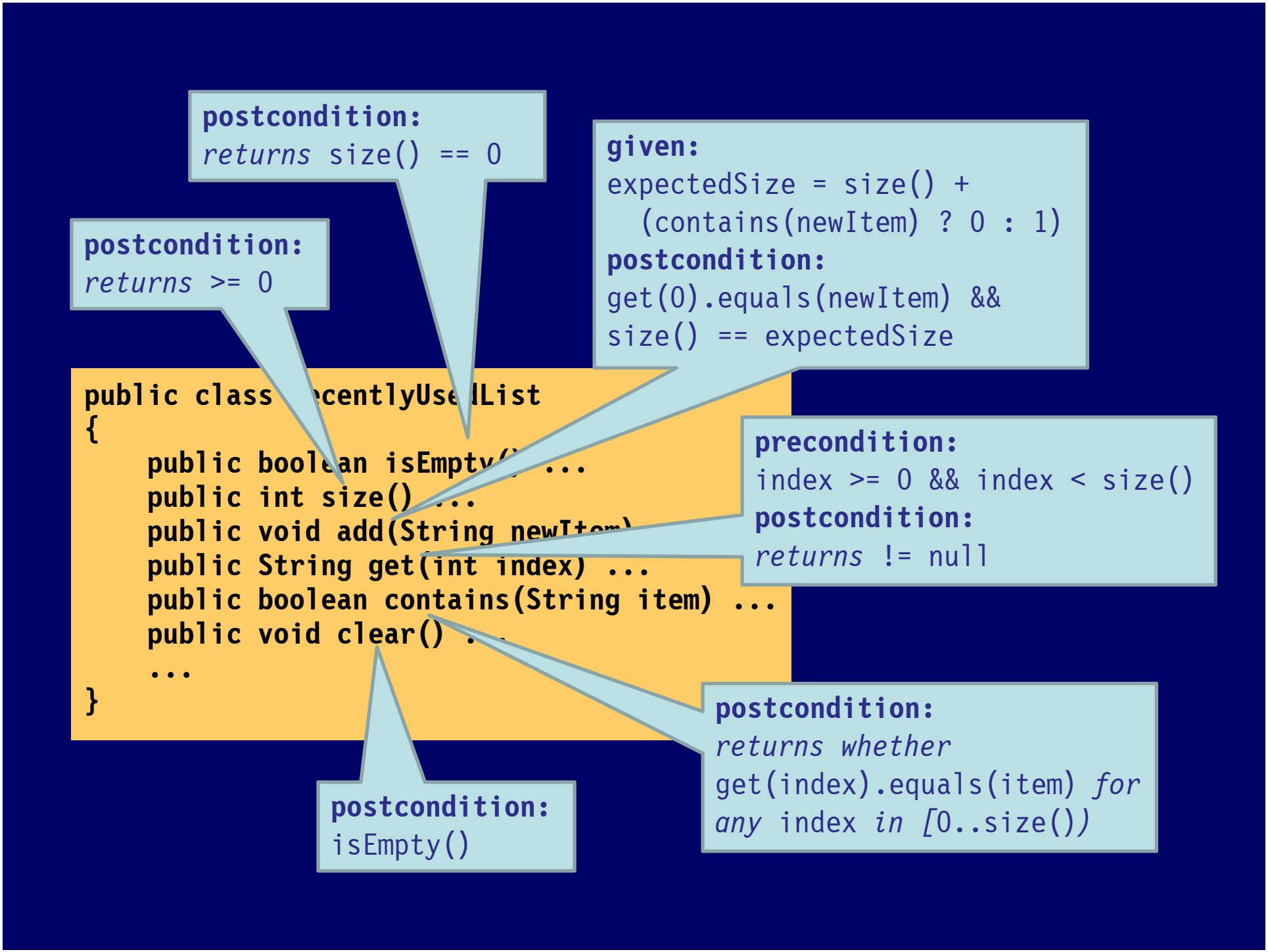
Employ the Contract Metaphor

contract *a written or spoken agreement, especially one concerning employment, sales, or tenancy, that is intended to be enforceable by law.*

metaphor *a figure of speech in which a word or phrase is applied to an object or action to which it is not literally applicable.*

- *a thing regarded as representative or symbolic of something else, especially something abstract.*

The New Oxford Dictionary of English


postcondition:
returns size() == 0
postcondition:
returns >= 0
given:
expectedSize = size() +
(contains(newItem) ? 0 : 1)
postcondition:
get(0).equals(newItem) &&
size() == expectedSize
public class ecentlyUsedList
{
public boolean isEmpty() ...
public int size() ...
public void add(String newItem)
public String get(int index) ...
public boolean contains(String item) ...
public void clear() ...
...
}
precondition:
index >= 0 && index < size()
postcondition:
returns != null
postcondition:
isEmpty()
postcondition:
returns whether
get(index).equals(item) for
any index in [0..size())

```
@Test
public void constructor()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
}
@Test
public void add()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    assertEquals(1, list.size());
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals(3, list.size());
    list.add("Aardvark");
    assertEquals(3, list.size());
}
@Test
public void get()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
    list.add("Aardvark");
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
    assertEquals("Zebra", list.get(2));
    bool thrown;
    try
    {
        list.get(3);
        thrown = false;
    }
    catch(IndexOutOfBoundsException)
    {
        thrown = true;
    }
    assertTrue(thrown);
}
```

Constructor

Add

Get

```
@Test
public void initialListIsEmpty()
{
    RecentlyUsedList list = new RecentlyUsedList();

    assertEquals(0, list.size());
}
@Test
public void additionOfSingleItemToEmptyListIsRetained()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");

    assertEquals(1, list.size());
    assertEquals("Aardvark", list.get(0));
}
@Test
public void additionOfDistinctItemsIsRetainedInStackOrder()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Zebra");
    list.add("Mongoose");

    assertEquals(3, list.size());
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
}
@Test
public void duplicateItemsAreMovedToFrontButNotAdded()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Mongoose");
    list.add("Aardvark");

    assertEquals(2, list.size());
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
}
@Test(expected=IndexOutOfBoundsException.class)
public void outOfRangeIndexThrowsException()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Mongoose");
    list.add("Aardvark");
    list.get(3);
}
```

Initial list is empty

Addition of single item to empty list is retained

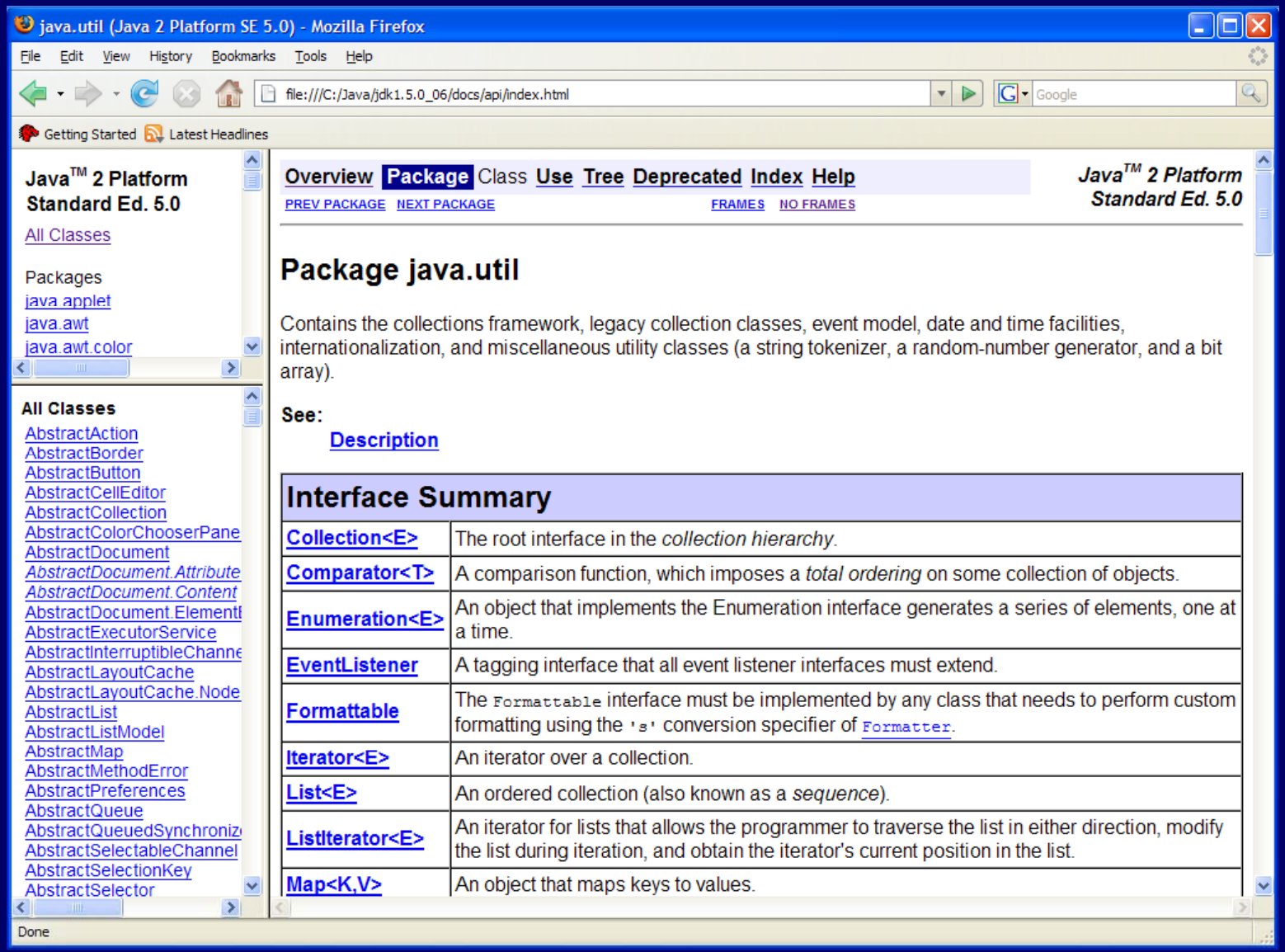
Addition of distinct items is retained in stack order

Duplicate items are moved to front but not added

Out of range index throws exception

Recommendation 3

Express Independent Ideas Independently


java.util (Java 2 Platform SE 5.0) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Java/jdk1.5.0_06/docs/api/index.html
Getting Started Latest Headlines
Java™ 2 Platform Standard Ed. 5.0
All Classes
Packages
java.applet
java.awt
java.awt.color
All Classes
AbstractAction
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPane
AbstractDocument
AbstractDocument.Attribute
AbstractDocument.Content
AbstractDocument.Element
AbstractExecutorService
AbstractInterruptibleChanne
AbstractLayoutCache
AbstractLayoutCache.Node
AbstractList
AbstractListModel
AbstractMap
AbstractMethodError
AbstractPreferences
AbstractQueue
AbstractQueuedSynchroniz
AbstractSelectableChannel
AbstractSelectionKey
AbstractSelector
Overview Package Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES
Java™ 2 Platform Standard Ed. 5.0
Package java.util
Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
See:
Description
Interface Summary
Collection<E> The root interface in the collection hierarchy.
Comparator<T> A comparison function, which imposes a total ordering on some collection of objects.
Enumeration<E> An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener A tagging interface that all event listener interfaces must extend.
Formattable The Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter.
Iterator<E> An iterator over a collection.
List<E> An ordered collection (also known as a sequence).
ListIterator<E> An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V> An object that maps keys to values.
Done

Pure Interface Layer

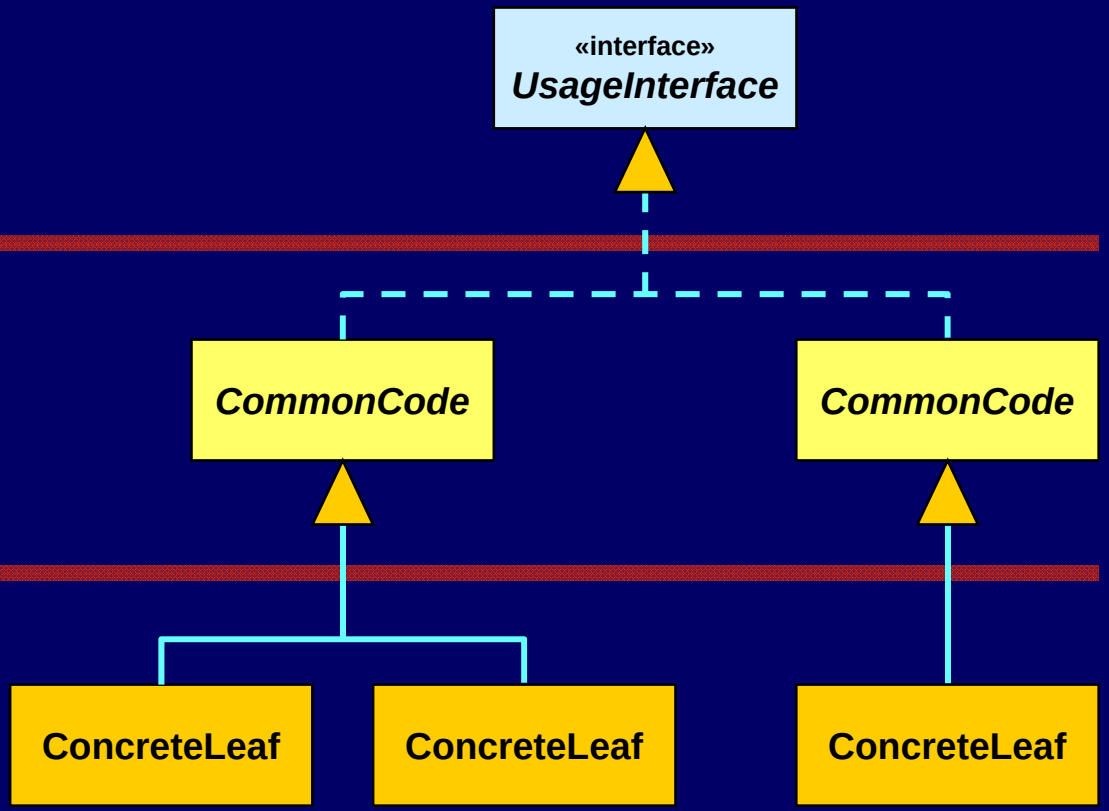
Interfaces may extend interfaces, but there is no implementation defined in this layer.

Common Code Layer

Only abstract classes are defined in this layer, possibly with inheritance, factoring out any common implementation.

Concrete Class Layer

Only concrete classes are defined, and they do not inherit from one another.



Recommendation 4

Encapsulate


Powder
PUSH

encapsulate *enclose (something) in or as if in a capsule.*

- *express the essential feature of (someone or something) succinctly.*
- *enclose (a message or signal) in a set of codes which allow use by or transfer through different computer systems or networks.*
- *provide an interface for (a piece of software or hardware) to allow or simplify access for the user.*

The New Oxford Dictionary of English

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(0, newItem);
    }
    public ArrayList<String> getList()
    {
        return list;
    }
    public void setList(ArrayList<String> newList)
    {
        list = newList;
    }
    private ArrayList<String> list;
}
```

```
RecentlyUsedList list = new RecentlyUsedList();
list.setList(new ArrayList<String>());
list.add("Hello, World!");
assert list.getList().size() == 1;
list.getList().clear();
assert list.getList().isEmpty();
```

Don't ever invite a vampire into your house, you silly boy. It renders you powerless.

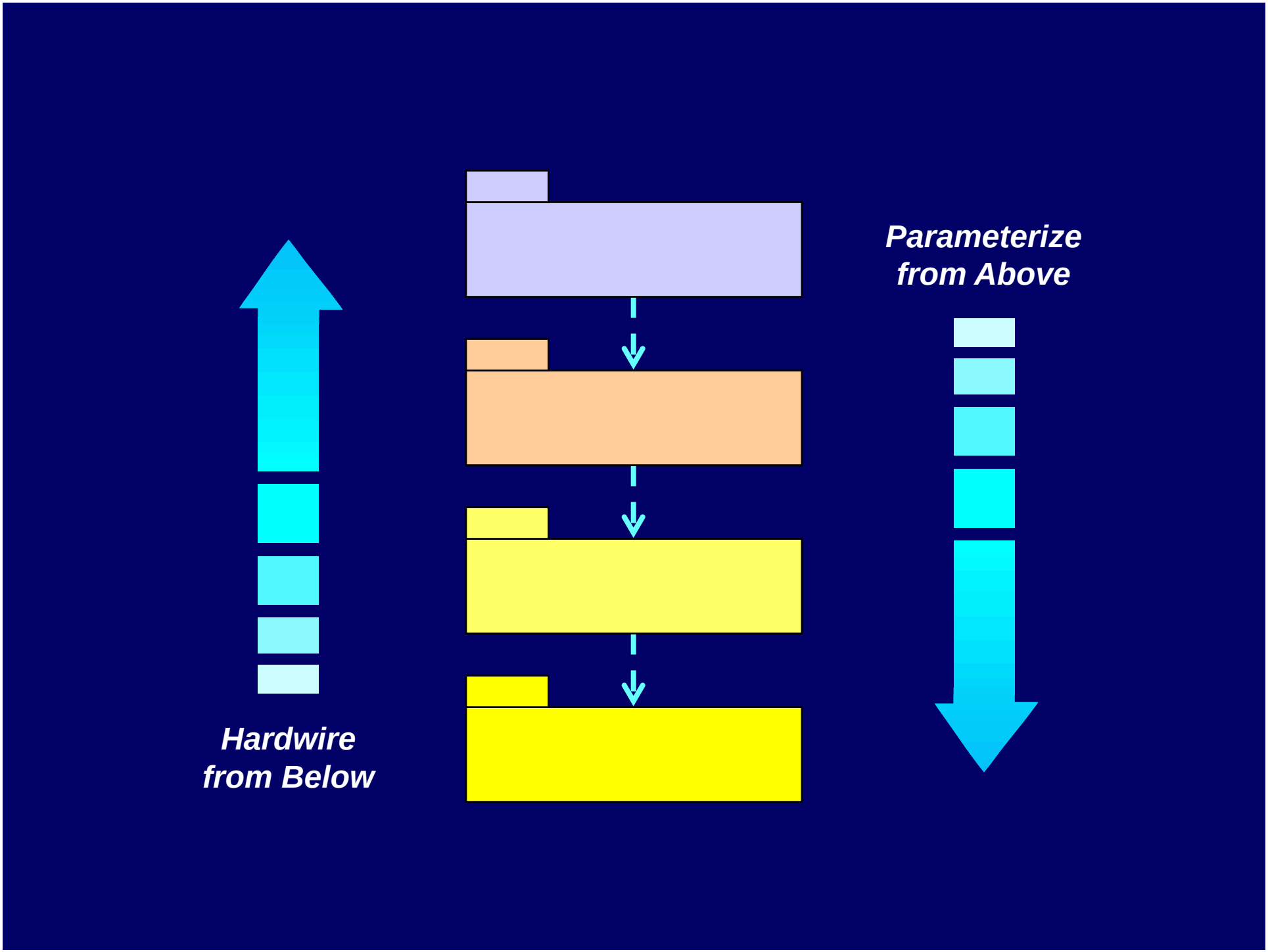

STRICTLY
PRIVATE

```
public class RecentlyUsedList
{
    public boolean isEmpty()
    {
        return list.isEmpty();
    }
    public int size()
    {
        return list.size();
    }
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(0, newItem);
    }
    public void clear()
    {
        list.clear();
    }
    private List<String> list = new ArrayList<String>();
}
```

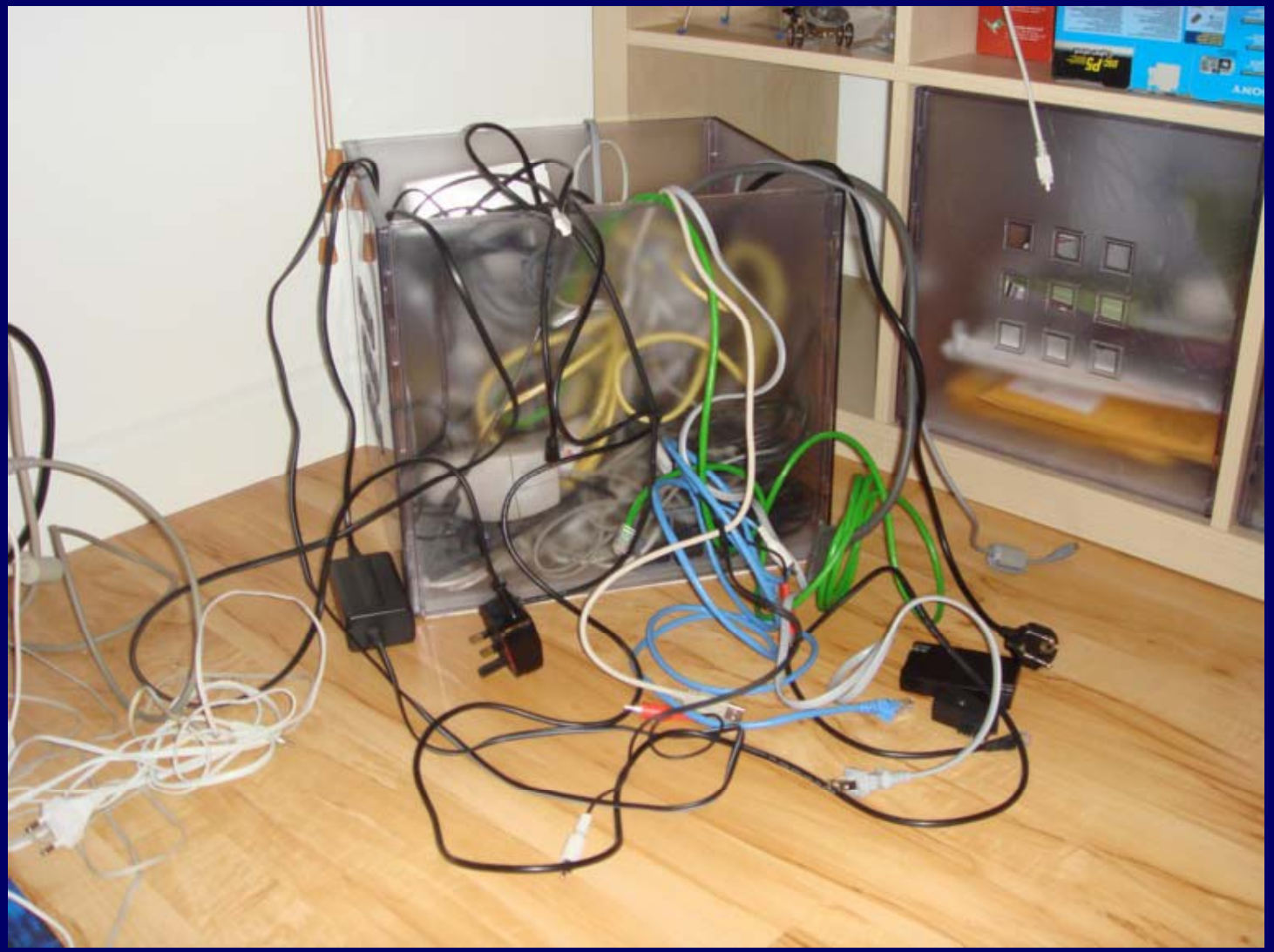
```
RecentlyUsedList list = new RecentlyUsedList();
list.add("Hello, World!");
assert list.size() == 1;
list.clear();
assert list.isEmpty();
```

Recommendation 5

Parameterize from Above


Parameterize
from Above
Hardwire
from Below


THE SINGLETON
Single Malt Scotch Whisky
of Dufftown
12
PERFECTLY BALANCED, NATURALLY RICH & SMOOTH



Recommendation 6

Restrict Mutability of State

```
public class Date
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getDayInMonth() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setDayInMonth(int newDayInMonth) ...
    ...
}
```

```
public class Date
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
}
```

```
public class Date
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int year, month, dayInMonth;
}
```

```
public class Date
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int daysSinceEpoch;
}
```

When it is not necessary to change, it is necessary not to change.

Lucius Cary

```
public class Date
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    ...
}
```

```
public class Date
{
    ...
    public int year() ...
    public int month() ...
    public int weekInYear() ...
    public int dayInYear() ...
    public int dayInMonth() ...
    public int dayInWeek() ...
    ...
}
```

Recommendation 7

Favour Symmetry over Asymmetry

symmetry *the quality of being made up of exactly similar parts facing each other or around an axis.*

- *correct or pleasing proportion of the parts of a thing.*
- *similarity of exact correspondence between different things.*

The New Oxford Dictionary of English

When in doubt, make it symmetrical.

Christopher Alexander

La symétrie, c'est l'ennui, et l'ennui est le fond même du deuil. Le désespoir baîlle.

Victor Hugo

Recommendation 8

Sharpen Fuzzy Logic

```
if(enabled != false)
    enabled = false;
else
    enabled = true;
```

```
enabled = !enabled;
```

```
if(value > limit)
    markOverLimit(true);
else
    markOverLimit(false);
```

```
markOverLimit(value > limit);
```

```
if(loaded() == true)
    if(valid() == true)
        return true;
    else
        return false;
else
    return false;
```

```
return loaded() && valid();
```

Recommendation 9

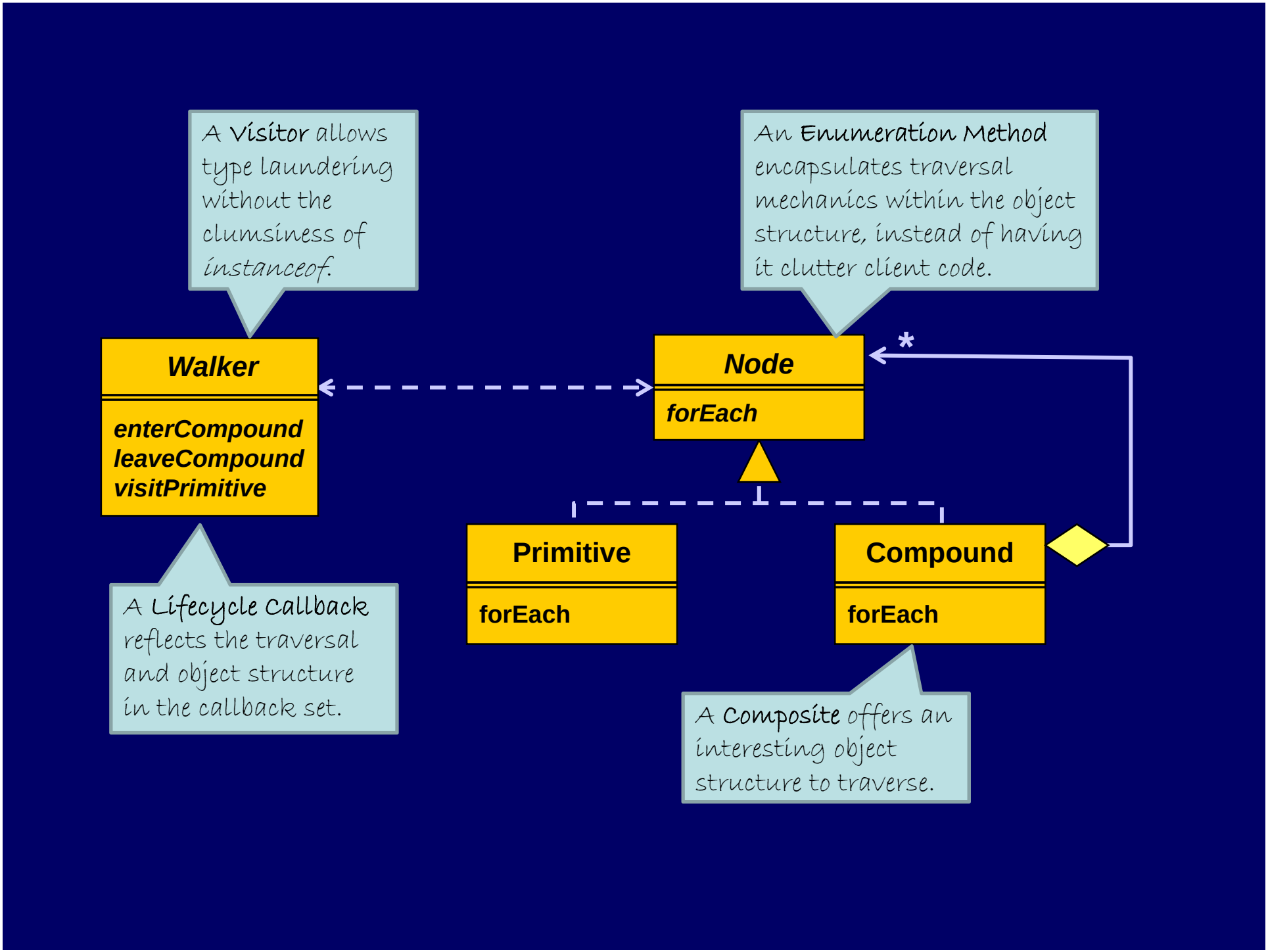
Go with the Flow

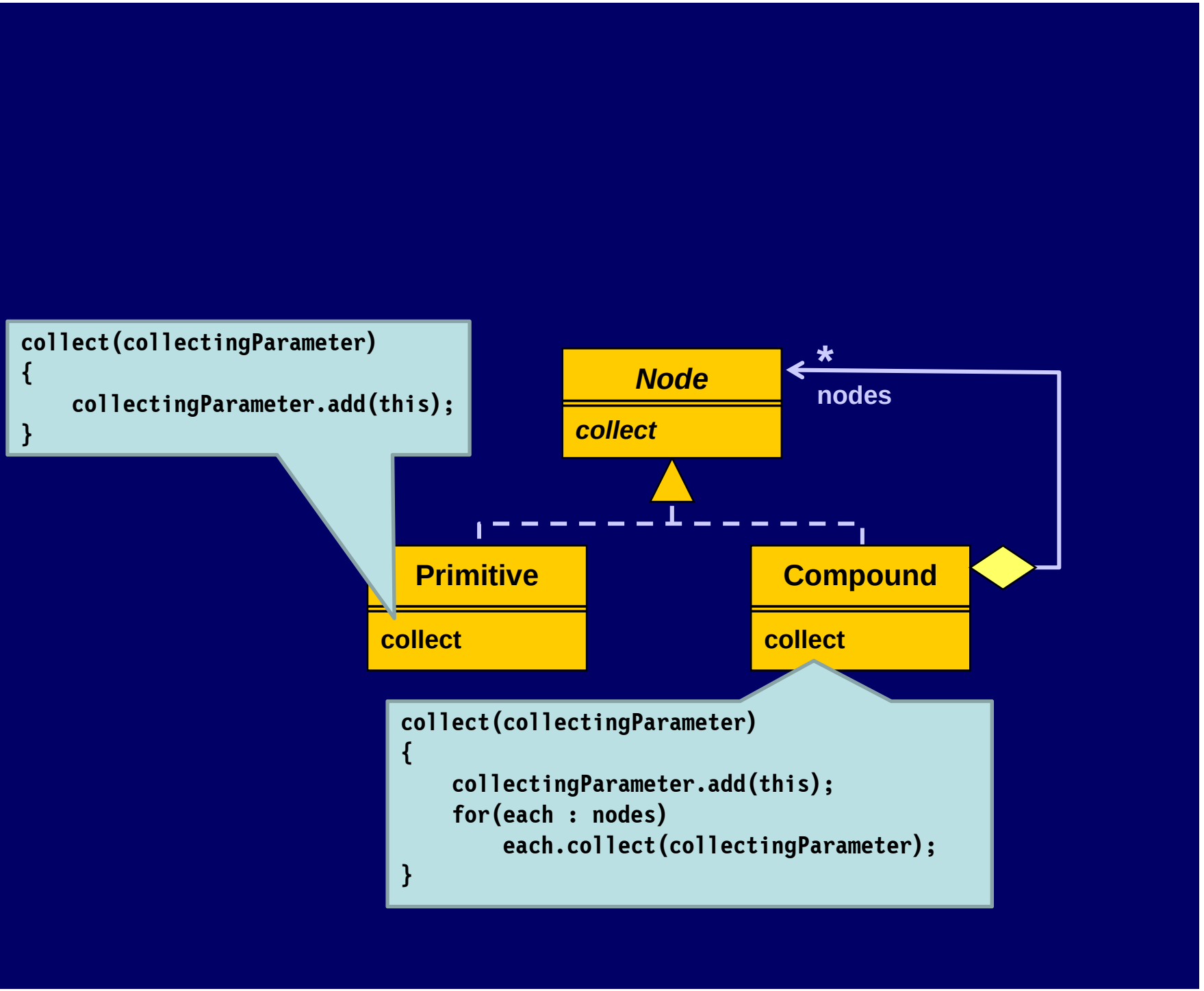
When the visible appearance of code and the control flow correspond, it greatly aids comprehension and correctness. Using `goto` subverts this. With `goto`, you must read every line or you don't know what is going on. A `goto` completely invalidates the high-level structure of the code. returning from the middle of a procedure is similarly suspect. Don't use either of these constructs. If you feel a burning need to do this, consult your architect.

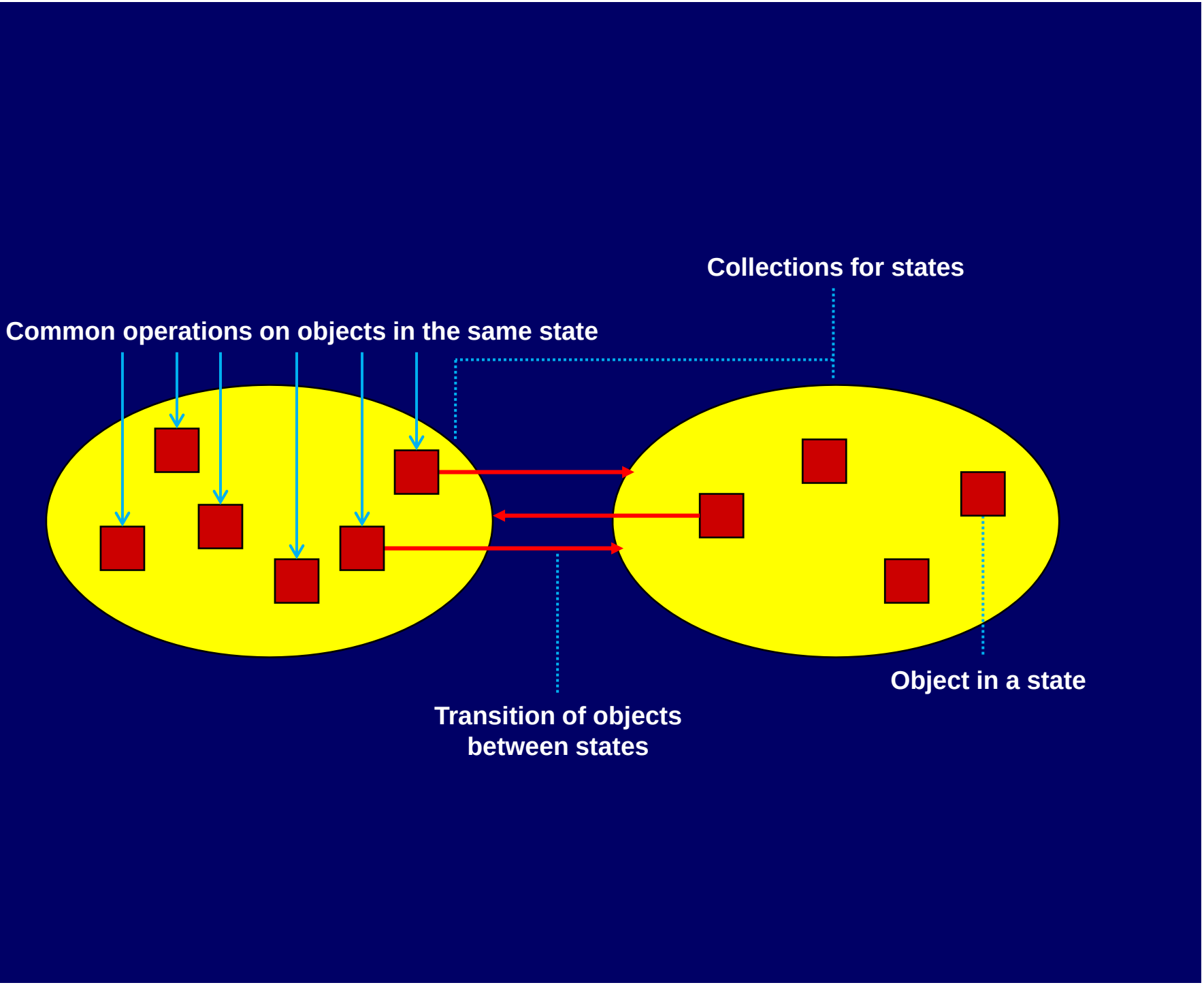
Taligent's Guide to Designing Programs

Recommendation 10

Let Code Decide


A Visitor allows type laundering without the clumsiness of instanceof.
An Enumeration Method encapsulates traversal mechanics within the object structure, instead of having it clutter client code.
Walker
enterCompound
leaveCompound
visitPrimitive
Node
forEach
*
Primitive
forEach
Compound
forEach
A Lifecycle Callback reflects the traversal and object structure in the callback set.
A Composite offers an interesting object structure to traverse.


collect(collectingParameter)
{
collectingParameter.add(this);
}
Node
collect
*
nodes
Primitive
collect
Compound
collect
collect(collectingParameter)
{
collectingParameter.add(this);
for(each : nodes)
each.collect(collectingParameter);
}


Collections for states
Common operations on objects in the same state
Object in a state
Transition of objects
between states

Recommendation 11

Omit Needless Code

```
public class RecentlyUsedList
{
    public RecentlyUsedList()
    {
        list = new ArrayList<String>();
    }
    public void add(String newItem)
    {
        if(list.contains(newItem))
        {
            int position;
            position = list.indexOf(newItem);
            string existingItem;
            existingItem = list.get(position);
            list.remove(position);
            list.add(0, existingItem);
        }
        else
        {
            list.add(0, newItem);
        }
    }
    public int size()
    {
        int size;
        size = list.size();
        return size;
    }
    public String get(int index)
    {
        int position;
        position = 0;
        for(String value : list)
        {
            if(position == index)
            {
                return value;
            }
            ++position;
        }
        throw new IndexOutOfBoundsException();
    }
    private List<String> list;
}
```

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(newItem);
    }
    public int size()
    {
        return list.size();
    }
    public String get(int index)
    {
        return list.get(size() - index – 1);
    }
    private List<String> list = new ArrayList<String>();
}
```

```
public class Date
{
    public Date(Year year, Month month, int dayInMonth) ...
    public Date(int dayInMonth, Month month, Year year) ...
    public Date(Month month, int dayInMonth, Year year) ...
    ...
}
```

```
public class Date
{
    public Date(Year year, Month month, int dayInMonth) ...
    ...
}
```

Recommendation 12

Unify Duplicate Code

Ask not for whom the photocopier jams; it jams for thee.

The only thing to do with good advice is to pass it on. It is never any use to oneself.

Oscar Wilde