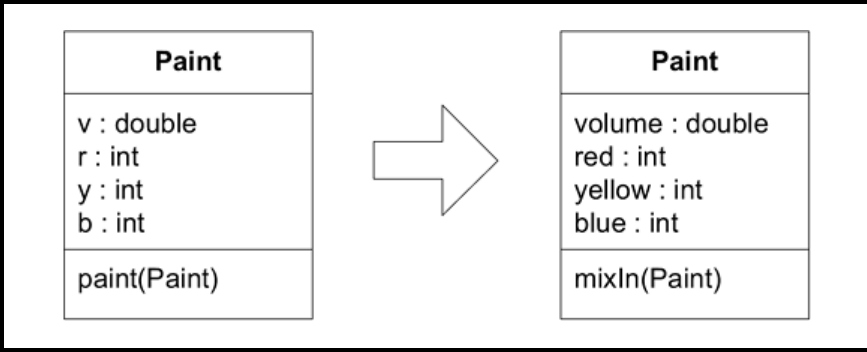
Rule: Don’t abbreviate

- Code should be self documenting.
  - Tools: SonarQube



Public API’s have to be documented i.e. every class, function, interface, exceptions. Mutable objects that can / cannot be modified.

This is possible by choosing the right variable and function names.

Comments are secondary because they tend to lie

Java: Checkstyle, PMD

C#: SytleCop+,FxCop, Simian, Ncover, NDepend for cyclomatic complexity.

C++: Simian or PMD CPD for duplication, coverity for source code analysis

JavaScript: JSHint

```
public List<int[]> getThem() {
    List<int[]> list1 =
        new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

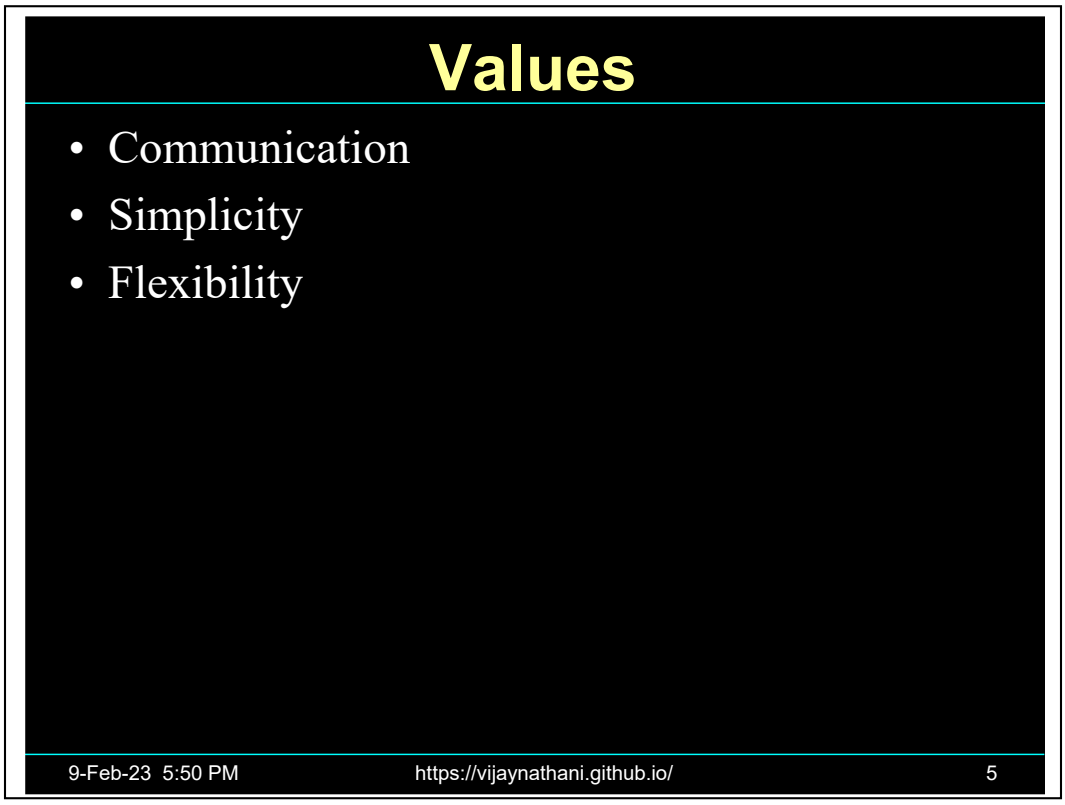
```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells =
        new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;;
    private static final String
        RECORD_ID = "102";
    /* ... */
};
```

Guidelines

- Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser.
  - Avoid words like Manager, Processor, Data, or Info in the name of a class.
  - A class name should not be a verb.
- Methods should have verb or verb phrase names like postPayment, deletePage, or save.



Mostly they complement each other.

Code communicates well, when a reader can understand it, modify it and use it.

Eliminating excess complexity, makes the program easier to understand, modify and use.

Flexibility means that the program can be changed.

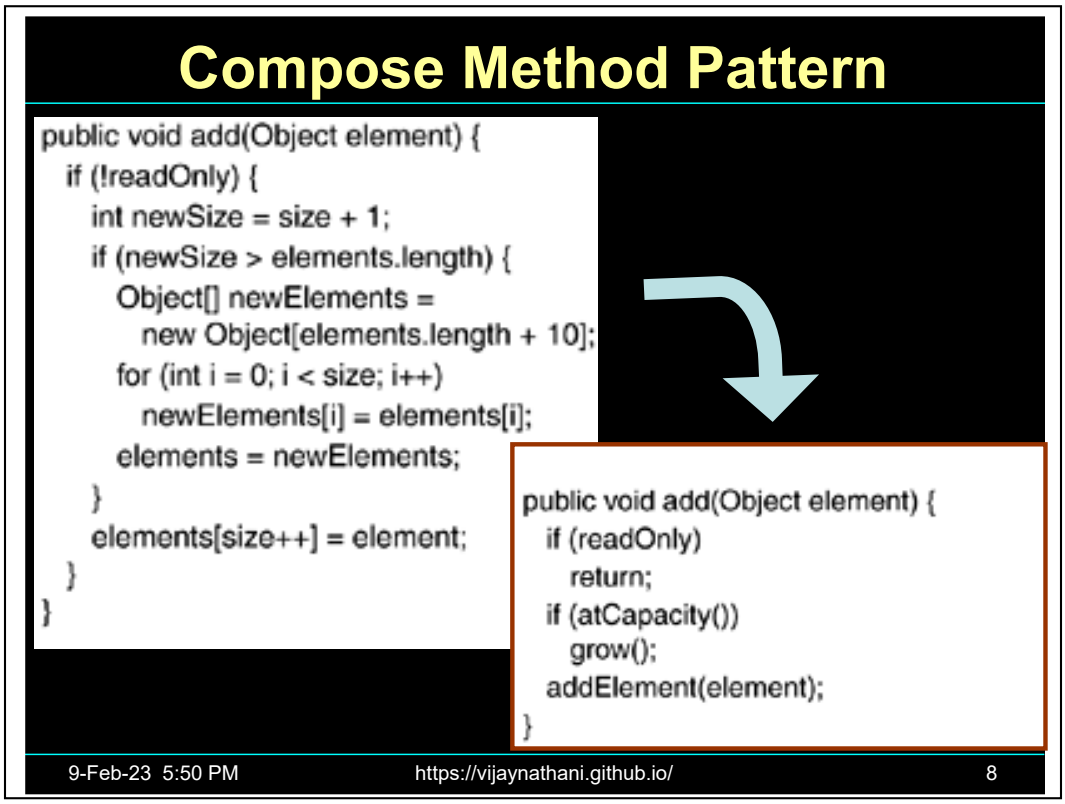
Code should readable

- Any fool can write code that a computer can understand. Good programmers write code that humans can understand. – Martin Fowler
  - ```
    Calendar c=Calendar.getInstance();
    c.set(2005,Calendar.NOVEMBER, 20);
    Date t = c.getTime(); OR
    ```
  - ```
    Date t = november(20, 2005) ;
    public Date november (
         int day, int year)  { … }
    ```

Compare

```
void process() {
    input();
    count++;
    output();
}
```

```
void process() {
    input();
    tally();
    output();
}
private void tally(){
    count++;
}
```



Benefits and Liabilities

+Efficiently communicates what a method does and how it does what it does.

+Simplifies a method by breaking it up into well-named chunks of behavior at the same level of detail.

–Can lead to an overabundance of small methods.

–Can make debugging difficult because logic is spread out across many small methods.

Improve

```
flags |= LOADED_BIT;
```

- Solution: Extract to a message

```
void setLoadedFlag() {
    flags |= LOADED_BIT;
}
```

Improve Code

```
// Check to see if the employee
// is eligible for full benefits
if ((employee.flags & HOURLY_FLAG)
        && (employee.age > 65)) …



if (employee.
         isEligibleForFullBenefits())
```

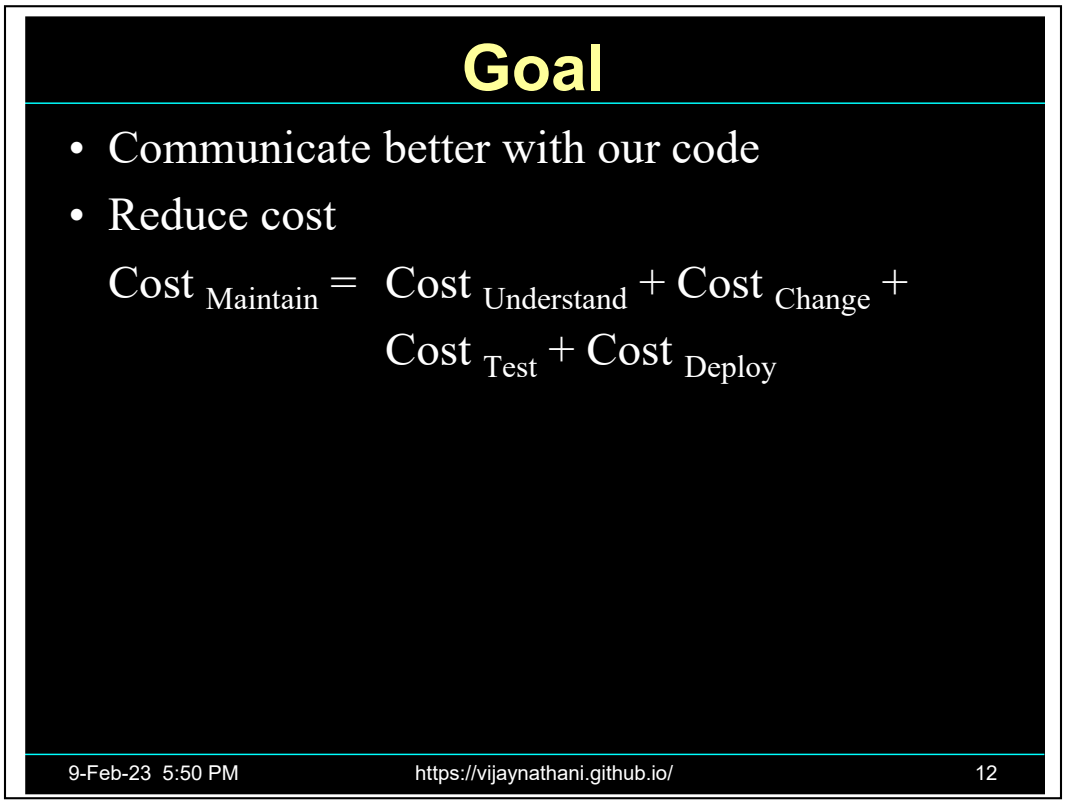
Typical IT budget

This is the problem

New Projects
<20%

Maintenance and Operations
>80%

Most organizations don't scrutinize M&O expenditures
No other major expenditure has so little oversight



Falls between design patterns and Java language manual

Isolate the concurrent portions of the code.

Most programs follow a small set of laws:

•Programs are read more often than they are written

•There is no such thing as “done”. Much more investment will be spent in modifying programs than in developing them initially.

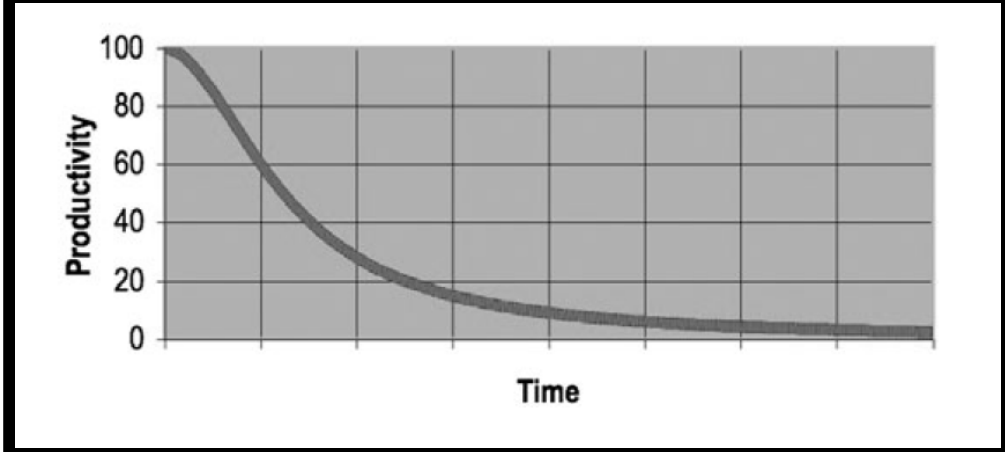
•They are structured using a basic set of state and control flow concepts

•Readers need to understand programs in detail and in concept

Cost to understand code is high. So maintenance is costly. Code will need to change in unanticipated ways.

When code is clear we have fewer defects and smoother development also.

Productivity vs. Time



Improve

```
public static void endMe() {
    if (status == DONE) {
        doSomething();
        …
        return
    } else {
        <other code>
    }
}
```

Avoid “else”

```
public static void endMe() {
    if (status == DONE) {
        doSomething();
        …
        return;
    }
    <other code>
}
```

Improve

```
public Node head() { …
    if (isAdvancing())
        return first;
    else
        return last;
}

  public static Node head() { …
      return isAdvancing()?first:last;
  }
```

Improve

- Function signature

```
void render(boolean isSuite)
```

- Remove boolean variables from functions. Have two functions.

```
void renderForSuite()
void renderForSingleTest()
```

Avoid Long parameter lists

- Long parameter list means more chances of an error.
  - CreateWindow in Win32 has 11 parameters.
- How to solve it?

Break the method or

Create Helper class to hold parameters

Improve

```
class Board {
    ...
    String board() {
        StringBuffer buf = new StringBuffer();
        for(int i = 0; i < 10; i++) {
            for(int j = 0; j < 10; j++)
                buf.append(data[i][j]);
            buf.append("\n" );
        }
        return buf.toString();
    }
}
```

Only one level of indentation per method

```
Class Board {
    ...
    String board() {
        StringBuffer buf = new StringBuffer();
        collectRows(buf);
        return buf.toString();
    }
    void collectRows(StringBuffer buf) {
        for(int I = 0; I < 10; i++)
            collectRow(buf, i);
    }
    void collectRow(StringBuffer buf, int row) {
        for(int I = 0; I < 10; i++)
            buf.append(data[row][i]);
        buf.append("\n" );
    }
}
```

Self documenting code

Q20 – inch; Q06 – NO_GROUPING; Q07 – addHoliday; Q21 – full name in English;

Q22 – complexPassword; Q23 – TokenStream; Q25 - orderItems

Good Comments?

```
String text = "'''bold text'''";
ParentWidget parent = new BoldWidget(
          new MockWidgetRoot(), "'''bold text'''");
AtomicBoolean failFlag = new AtomicBoolean();
failFlag.set(false);
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread = new
         WidgetBuilderThread(widgetBuilder, text,
              parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
}
assertEquals(false, failFlag.get());
```


Comment to

- explain WHY we are doing something?
- Also for external documentation. Javadocs public API
- To give warnings: e.g. Don't run unless you want to kill this program.
- Todo comments