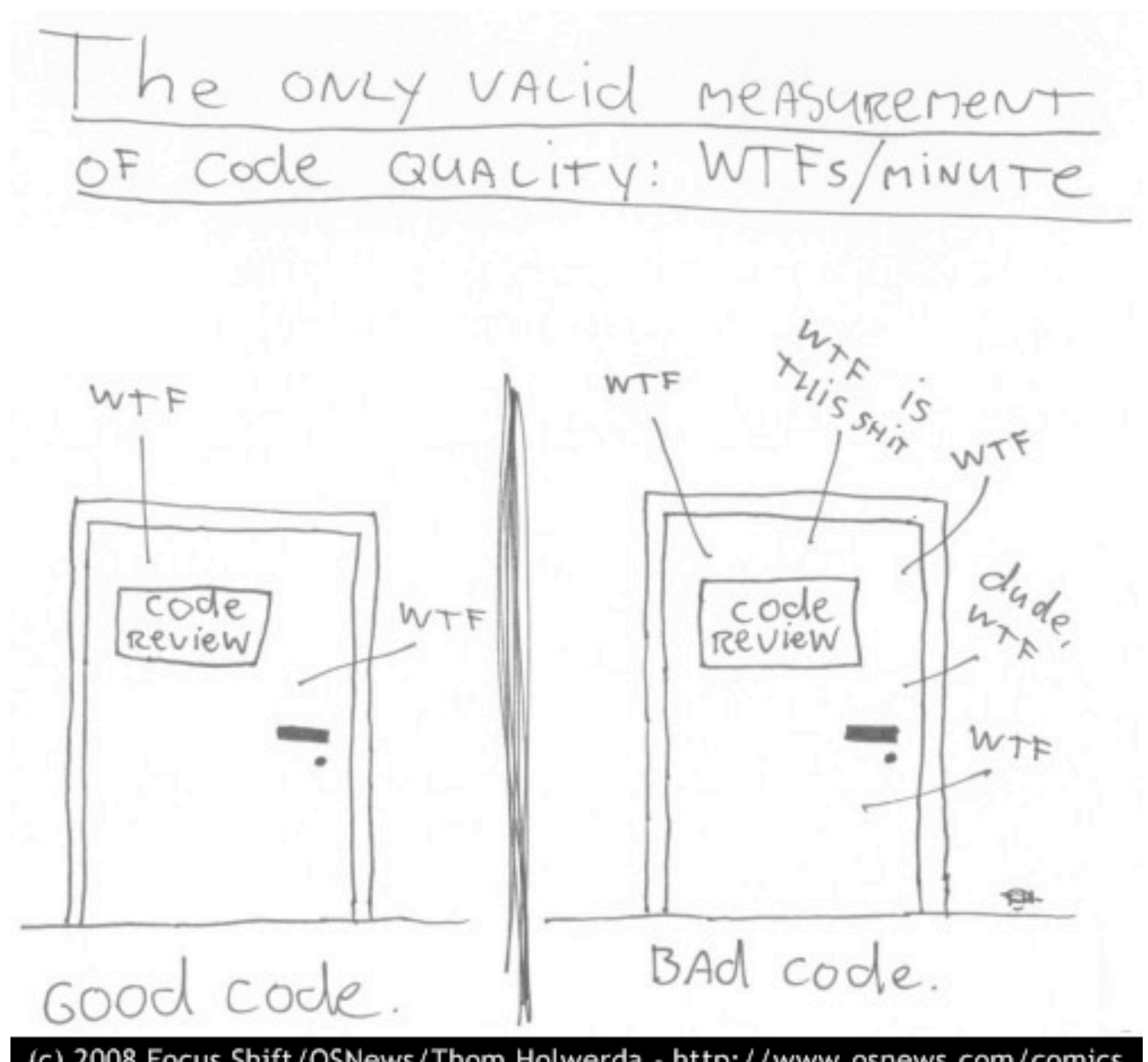
CLEAN CODE

Jeremie Dequidt

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> *-Martin Fowler*

WRITING GOOD / QUALITY CODE

- Matches technical specifications
- Bug-free
- Easy to read / easy to understand / easy to use
- Efficient



> Software entropy: An evolving system increases its complexity unless work is done to reduce it.
>
> *-Meir Lehman*

> Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with refactoring. The danger occurs when the debt is not repaid. Every minute spent on code that is not quite right for the programming task of the moment counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored implementation, object-oriented or otherwise.

-Ward Cunningham

GOOD PRACTICES

- Naming / Comments / Layout
- Principles:
  - KISS
  - DRY
  - YAGNI
  - SOLID
  - Demeter's Law

NAMING

- Variable / functions / class names should display intention
- Character cost for naming is now 0
- Should be auto-descriptive

```
public List<int[]> getList ()
{
    List<int[]> l2 = new ArrayList<int[]>();
    for (int[] x : this.l1)
        if (x[0] == 4) l2.add(x);
    return l2;
}
```

NAMING

- Variable / functions / class names should display intention
- Character cost for naming is now 0
- Should be auto-descriptive

```
public List<int[]> getDeadCells ()
{
    List<int[]> deadCells = new ArrayList<int[]>();
    for (int[] cell : this.allTheCells)
        if (cell[STATUS_OFFSET] == DEAD) deadCells.add(cell);
    return deadCells;
}
```

NAMING

- Variable / functions / class names should display intention
- Character cost for naming is now 0
- Should be auto-descriptive
- Names should be pronounceable (no abbr)
- Functions should be verb: getXXXX(), setXXXX(), validateXXXX()…
- Booleans should answer true/false: isXXXX(), areXXXX()
- Name should be meaningful and easy to look-up
- Don’t use Magic Numbers: if (x == 4)

Comments are always failure

-"Uncle Bob" Robert C. Martin

COMMENTS

- They:
  - Lie
  - Ages badly
  - Are not *refactorable*
  - Illustrates the failing at:
    - Choosing a good name
    - Splitting code into single intention functions
    - Abstraction creation

EXAMPLE: COMPUTING 1/√X

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, can be removed

    return y;
}
```

EXAMPLE: COMPUTING 1/√X… EXPLANATIONS

Algorithm [edit]

The algorithm computes $\frac{1}{\sqrt{x}}$ by performing the following steps:

1. Alias the argument x to an integer as a way to compute an approximation of the binary logarithm $\log_2(x)$
2. Use this approximation to compute an approximation of $\log_2\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2}\log_2(x)$
3. Alias back to a float, as a way to compute an approximation of the base-2 exponential
4. Refine the approximation using a single iteration of Newton's method.

Floating-point representation [edit]

Main article: Single-precision floating-point format

Since this algorithm relies heavily on the bit-level representation of single-precision floating-point numbers, a short overview of this representation is provided here. In order to encode a non-zero real number x as a single precision float, the first step is to write $''x''$ as a normalized binary number:[17]

$$\begin{aligned} x &= \pm 1.b_1b_2b_3\ldots \times 2^{e_x} \\ &= \pm 2^{e_x}(1+m_x) \end{aligned}$$

where the exponent e_x is an integer, $m_x \in [0,1)$, and $1.b_1b_2b_3\ldots$ is the binary representation of the "significand" $(1+m_x)$. Since the single bit before the point in the significand is always 1, it need not be stored. From this form, three unsigned integers are computed:[18]

- S_x, the "sign bit", is 0 if x is positive and 1 negative or zero (1 bit)
- $E_x = e_x + B$ is the "biased exponent", where $B = 127$ is the "exponent bias"[note 3] (8 bits)
- $M_x = m_x \times L$, where $L = 2^{23}$[note 4] (23 bits)

These fields are then packed, left to right, into a 32-bit container.[19]

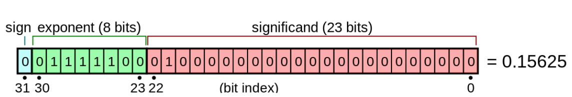
As an example, consider again the number $x = 0.15625 = 0.00101_2$. Normalizing x yields:

$$x = +2^{-3}(1+0.25)$$

and thus, the three unsigned integer fields are:

- $S = 0$
- $E = -3 + 127 = 124 = 0111\ 1100_2$
- $M = 0.25 \times 2^{23} = 2\ 097\ 152 = 0010\ 0000\ 0000\ 0000\ 0000\ 0000_2$

these fields are packed as shown in the figure below:



Aliasing to an integer as an approximate logarithm [edit]

If $\frac{1}{\sqrt{x}}$ were to be calculated without a computer or a calculator, a table of logarithms would be useful, together with the identity $\log_b\left(\frac{1}{\sqrt{x}}\right) = \log_b\left(x^{-\frac{1}{2}}\right) = -\frac{1}{2}\log_b(x)$, which is valid for every base b. The fast inverse square root is based on this identity, and on the fact that aliasing a float32 to an integer gives a rough approximation of its logarithm. Here is how:

If x is a positive normal number:

$$x = 2^{e_x}(1+m_x)$$

then

$$\log_2(x) = e_x + \log_2(1+m_x)$$

and since $m_x \in [0,1)$, the logarithm on the right hand side can be approximated by[20]

$$\log_2(1+m_x) \approx m_x + \sigma$$

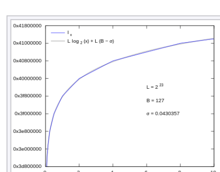
where σ is a free parameter used to tune the approximation. For example, $\sigma = 0$ yields exact results at both ends of the interval, while $\sigma = \frac{1}{2} - \frac{1+\ln(\ln(2))}{2\ln(2)} \approx 0.0430357$ yields the optimal approximation (the best in the sense of the uniform norm of the error). However, this value is not used by the algorithm as it does not take subsequent steps into account.

Thus there is the approximation

$$\log_2(x) \approx e_x + m_x + \sigma.$$

Interpreting the floating-point bit-pattern of x as an integer I_x yields[note 5]

$$\begin{aligned} I_x &= E_xL + M_x \\ &= L(e_x + B + m_x) \\ &= L(e_x + m_x + \sigma + B - \sigma) \\ &\approx L\log_2(x) + L(B-\sigma). \end{aligned}$$



The integer aliased to a floating point number (in blue), compared to a scaled and shifted logarithm (in gray).

It then appears that I_x is a scaled and shifted piecewise-linear approximation of $\log_2(x)$, as illustrated in the figure on the right. In other words, $\log_2(x)$ is approximated by

It then appears that I_x is a scaled and shifted piecewise-linear approximation of $\log_2(x)$, as illustrated in the figure on the right. In other words, $\log_2(x)$ is approximated by

The integer aliased to a floating point number (in blue), compared to a scaled and shifted logarithm (in gray).

$$\log_2(x) \approx \frac{I_x}{L} - (B-\sigma).$$

First approximation of the result [edit]

The calculation of $y = \frac{1}{\sqrt{x}}$ is based on the identity

$$\log_2(y) = -\tfrac{1}{2}\log_2(x)$$

Using the approximation of the logarithm above, applied to both x and y, the above equation gives:

$$\frac{I_y}{L} - (B-\sigma) \approx -\frac{1}{2}\left(\frac{I_x}{L} - (B-\sigma)\right)$$

Thus, an approximation of I_y is:

$$I_y \approx \tfrac{3}{2}L(B-\sigma) - \tfrac{1}{2}I_x$$

which is written in the code as

```
i  = 0x5f3759df - ( i >> 1 );
```

The first term above is the magic number

$$\tfrac{3}{2}L(B-\sigma) = \texttt{0x5F3759DF}$$

from which it can be inferred that $\sigma \approx 0.0450466$. The second term, $\frac{1}{2}I_x$, is calculated by shifting the bits of I_x one position to the right.[21]

Newton's method [edit]

Main article: Newton's method

With y as the inverse square root, $f(y) = \frac{1}{y^2} - x = 0$. The approximation yielded by the earlier steps can be refined by using a root-finding method, a method that finds the zero of a function. The algorithm uses Newton's method: if there is an approximation, y_n for y, then a better approximation y_{n+1} can be calculated by taking $y_n - \frac{f(y_n)}{f'(y_n)}$, where $f'(y_n)$ is the derivative of $f(y)$ at y_n.[22] For the above $f(y)$,

$$y_{n+1} = \frac{y_n\left(3 - xy_n^2\right)}{2}$$

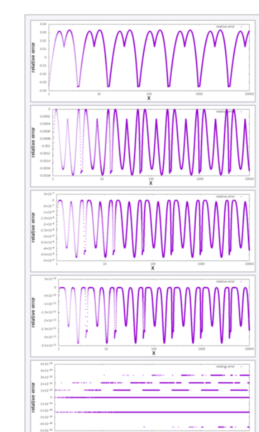
where $f(y) = \frac{1}{y^2} - x$ and $f'(y) = -\frac{2}{y^3}$.

Treating y as a floating-point number, `y = y*(threehalfs - x/2*y*y);` is equivalent to

$$y_{n+1} = y_n\left(\frac{3}{2} - \frac{x}{2}y_n^2\right) = \frac{y_n\left(3 - xy_n^2\right)}{2}.$$

By repeating this step, using the output of the function (y_{n+1}) as the input of the next iteration, the algorithm causes y to converge to the inverse square root.[23] For the purposes of the *Quake III* engine, only one iteration was used. A second iteration remained in the code but was commented out.[15]

Accuracy [edit]



Relative error between direct calculation and fast inverse square root carrying out 0, 1, 2, 3, and 4 iterations of Newton's root-finding method. Note that double precision is adopted and the smallest representab

As noted above, the approximation is surprisingly accurate. The single graph on the right plots the error of the function (that is, the error of the approximation after it has been improved by running one iteration of Newton's method), for inputs starting at 0.01, where the standard library gives 10.0 as a result, while InvSqrt() gives 9.982522, making the difference 0.0017478, or 0.175% of the true value, 10. The absolute error only drops from then on, while the relative error stays within the same bounds across all orders of magnitude.

CODE LAYOUT

- Files should be small ~200 lines (max: 500)
- Lines should have reasonable size [80; 120] characters
- Code should be correctly indented and spaced
- Code should be read from the beginning of the file to the end

```
#include <stdio.h> #define THIS printf(
#define IS "%s\n"
#define OBFUSCATION ,v);
double h[2]; int main(_, v) char *v; int _; { int a = 0; char f[32]; h[2%2] =
21914441197069634153456391018824026170709523170177760997320759459436800394073072125018704290409006721463
38833938303659439237740635160500855813030357492372682887858054616489605441589829740433065995076650229 1
52079883597110973562880.000000; h[4%3] = 1867980801.569119; switch (_) { case 0: THIS IS OBFUSCATION
break; default: main(0,(char *)h); break; } }
```

If you can't explain it simply, you don't understand it well enough.

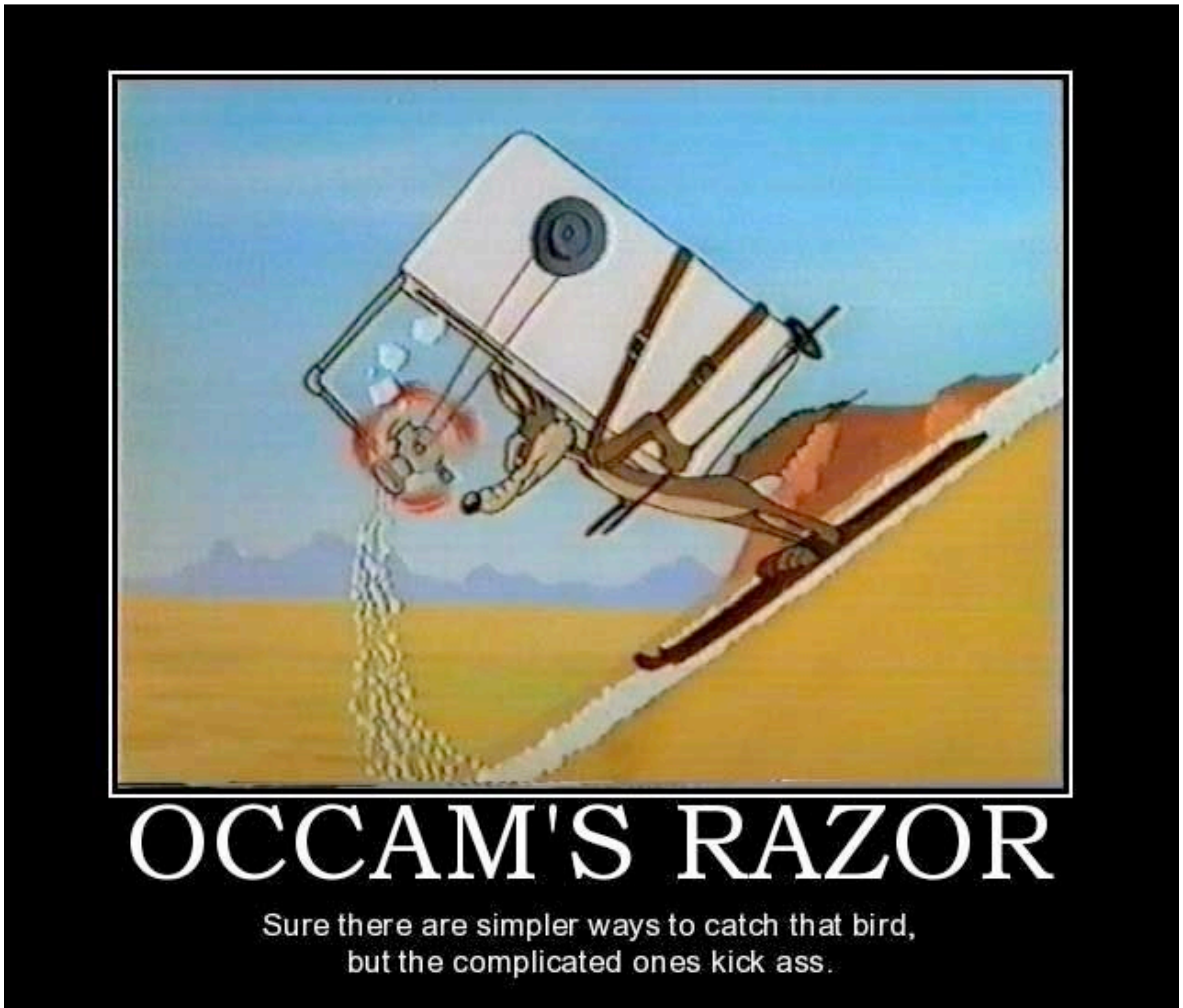
-Albert Einstein

KISS

- Keep it Simple, Stupid
- Use simplest logical way
- Avoid relying on many abstractions
- Will be easier to read later

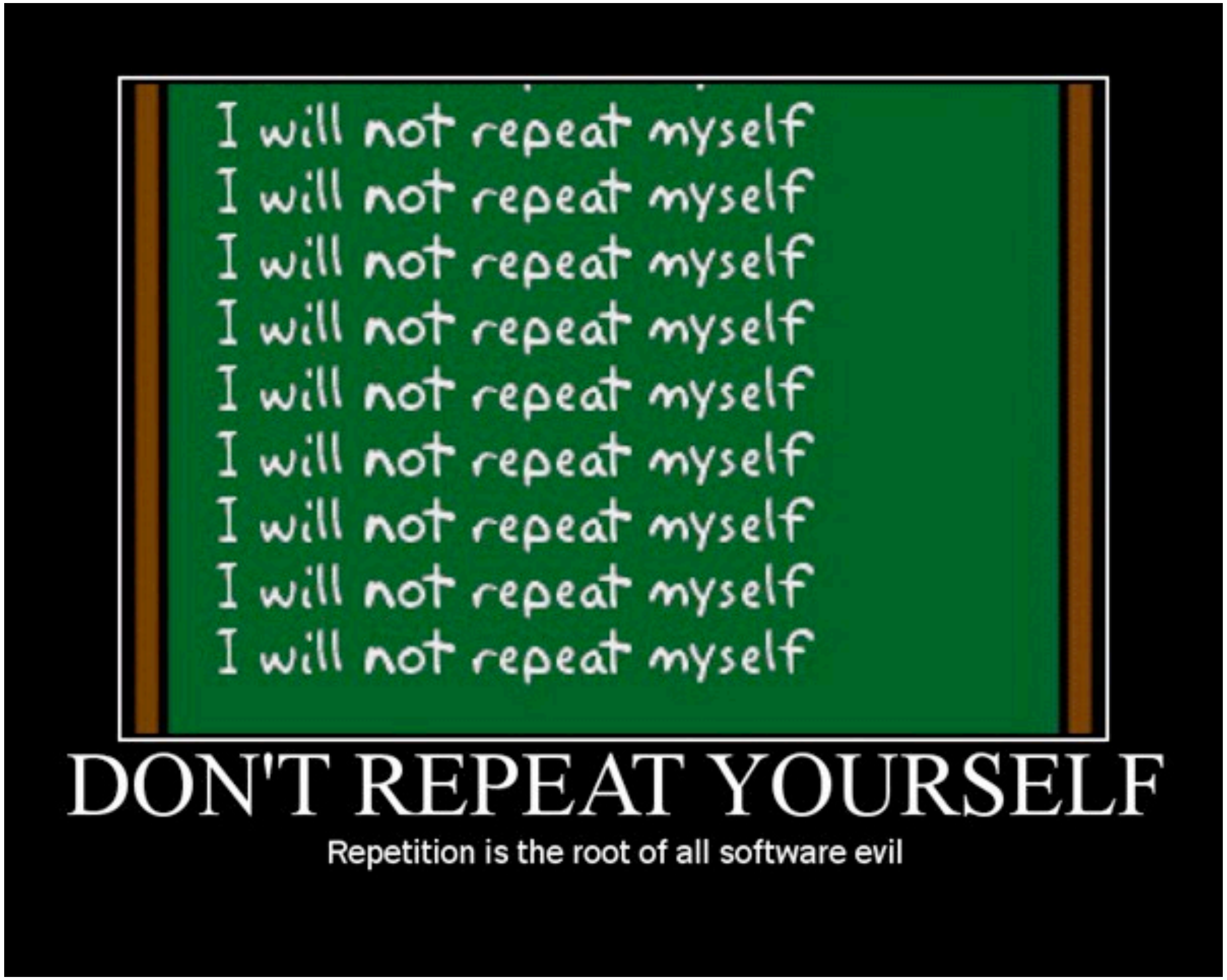
KISS

- Keep it Simple, Stupid



DRY / DIE

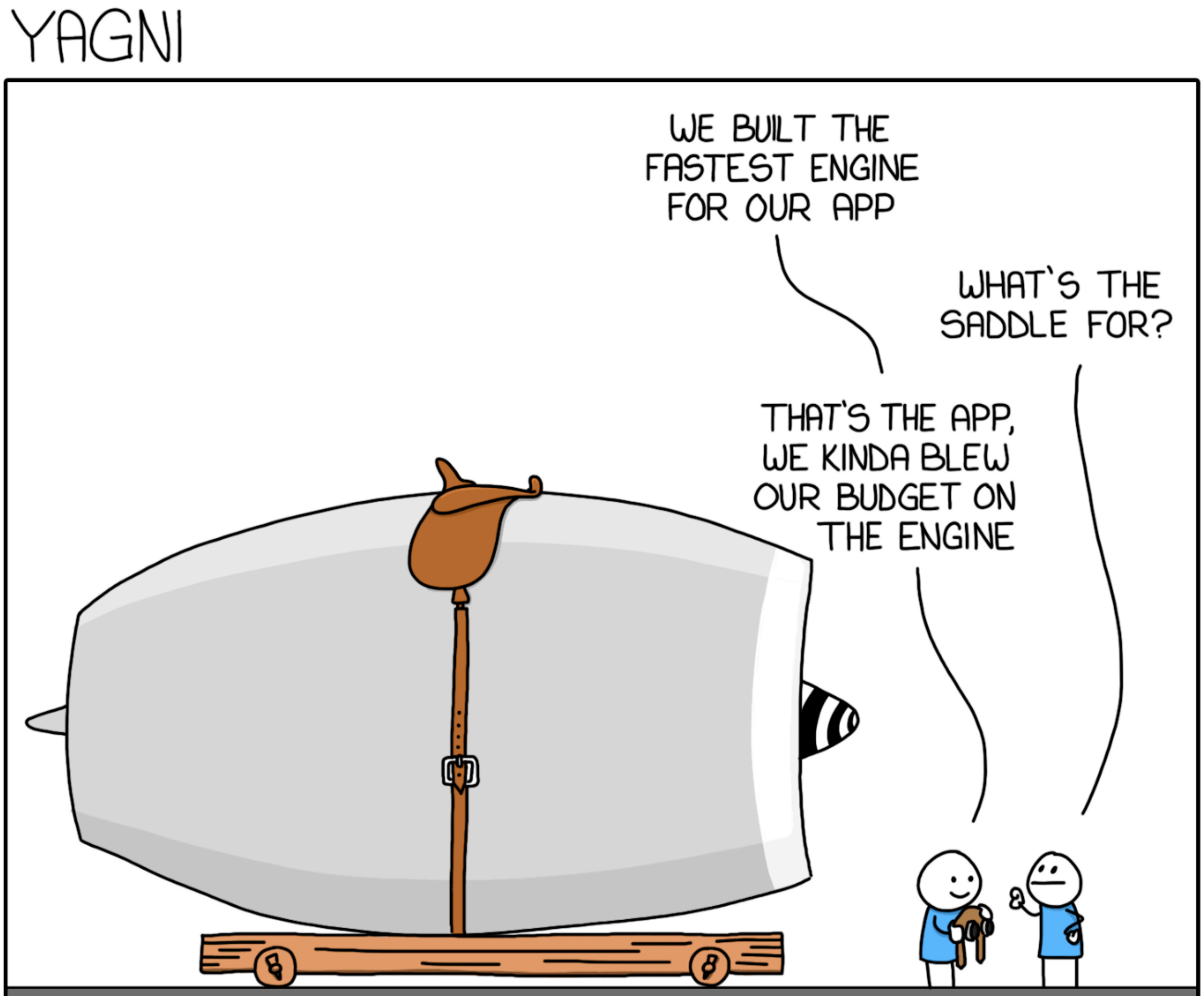
- Don't Repeat Yourself / Duplication Is Evil
- Avoid code duplication (hard to refactor)
  - Factorize
  - Limit responsibilities of entities



YAGNI

- You Ain't Gonna Need It
- Prefer refactoring over new features (according to the Agile Manifesto)
- Time dedicated to this feature will not be used for tests or refactoring

YAGNI


YAGNI
WE BUILT THE FASTEST ENGINE FOR OUR APP
WHAT'S THE SADDLE FOR?
THAT'S THE APP, WE KINDA BLEW OUR BUDGET ON THE ENGINE

SOLID

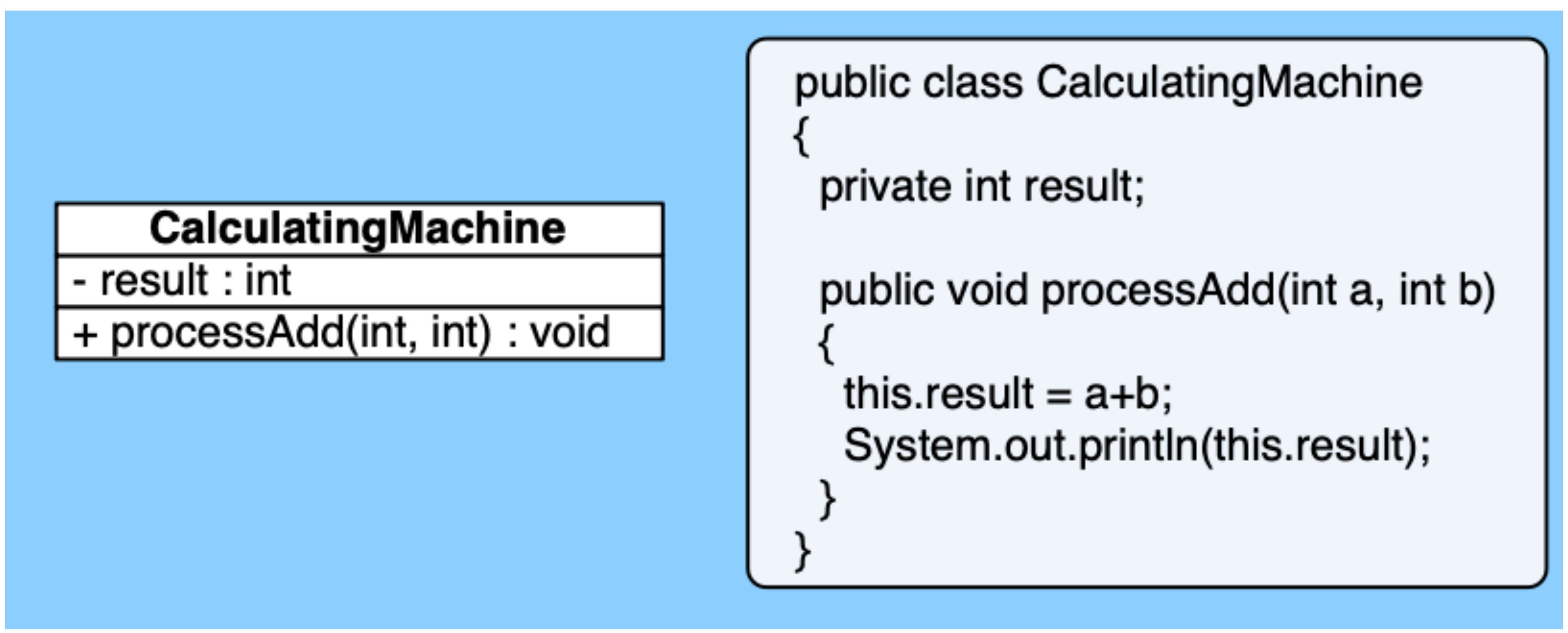
- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

SINGLE-RESPONSIBILITY PRINCIPLE

- Every object, class, and method needs to have a single responsibility. If your objects/class/methods are doing too much, you will end up with the well-known spaghetti code.

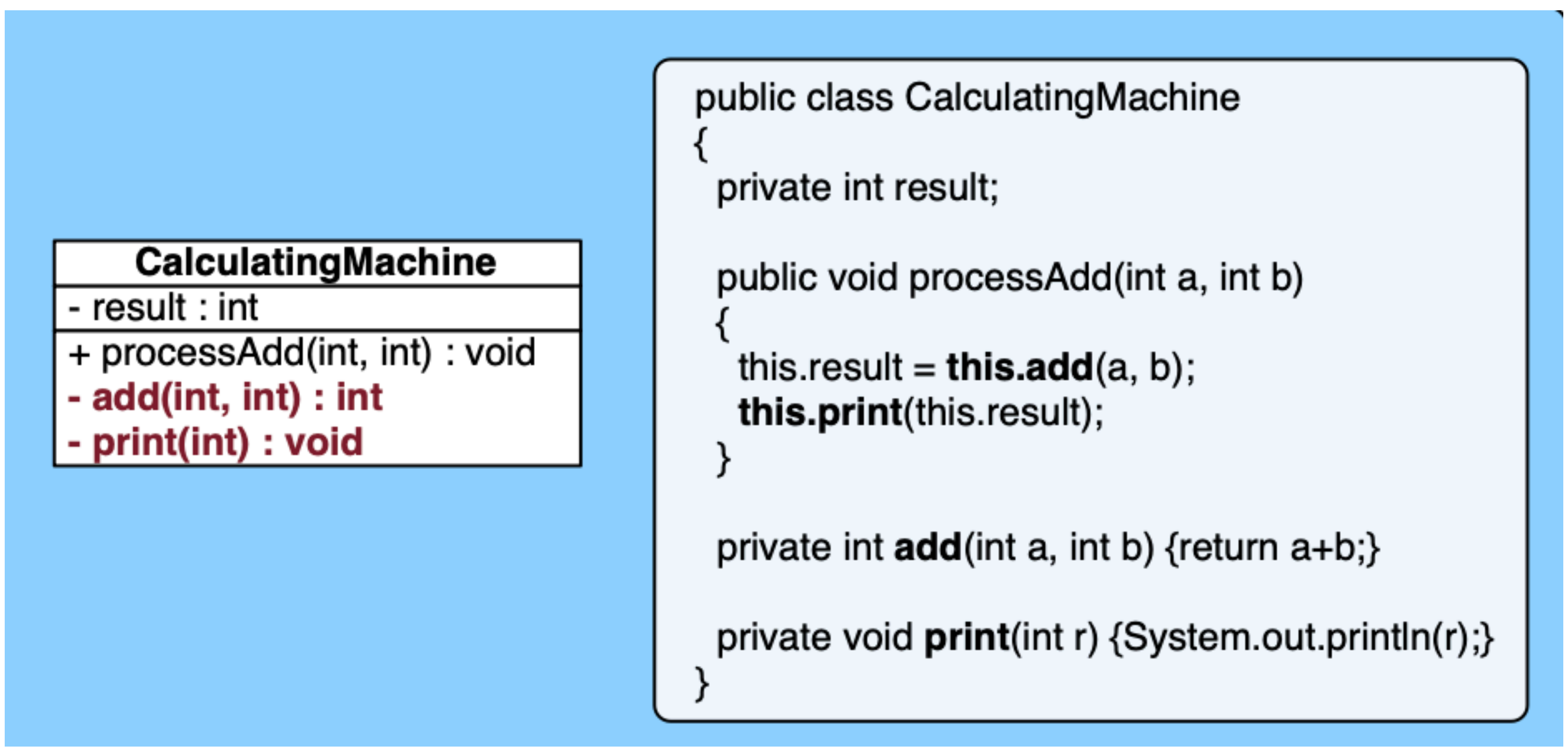
SINGLE-RESPONSIBILITY PRINCIPLE

- Every object, class, and method needs to have a single responsibility. If your objects/class/methods are doing too much, you will end up with the well-known spaghetti code.



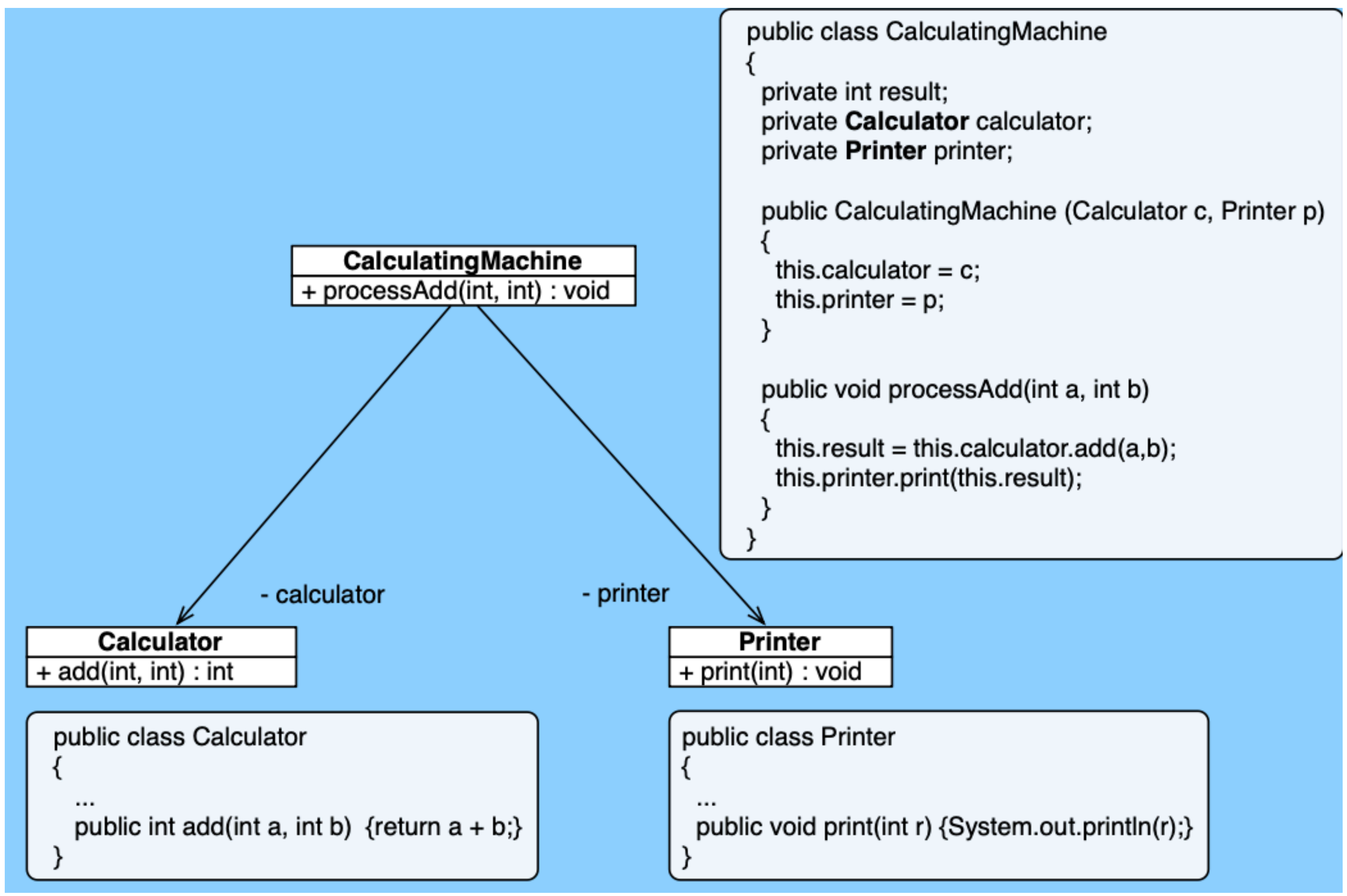
SINGLE-RESPONSIBILITY PRINCIPLE

- Every object, class, and method needs to have a single responsibility. If your objects/class/methods are doing too much, you will end up with the well-known spaghetti code.



SINGLE-RESPONSIBILITY PRINCIPLE

- Every object, class, and method needs to have a single responsibility. If your objects/class/methods are doing too much, you will end up with the well-known spaghetti code.

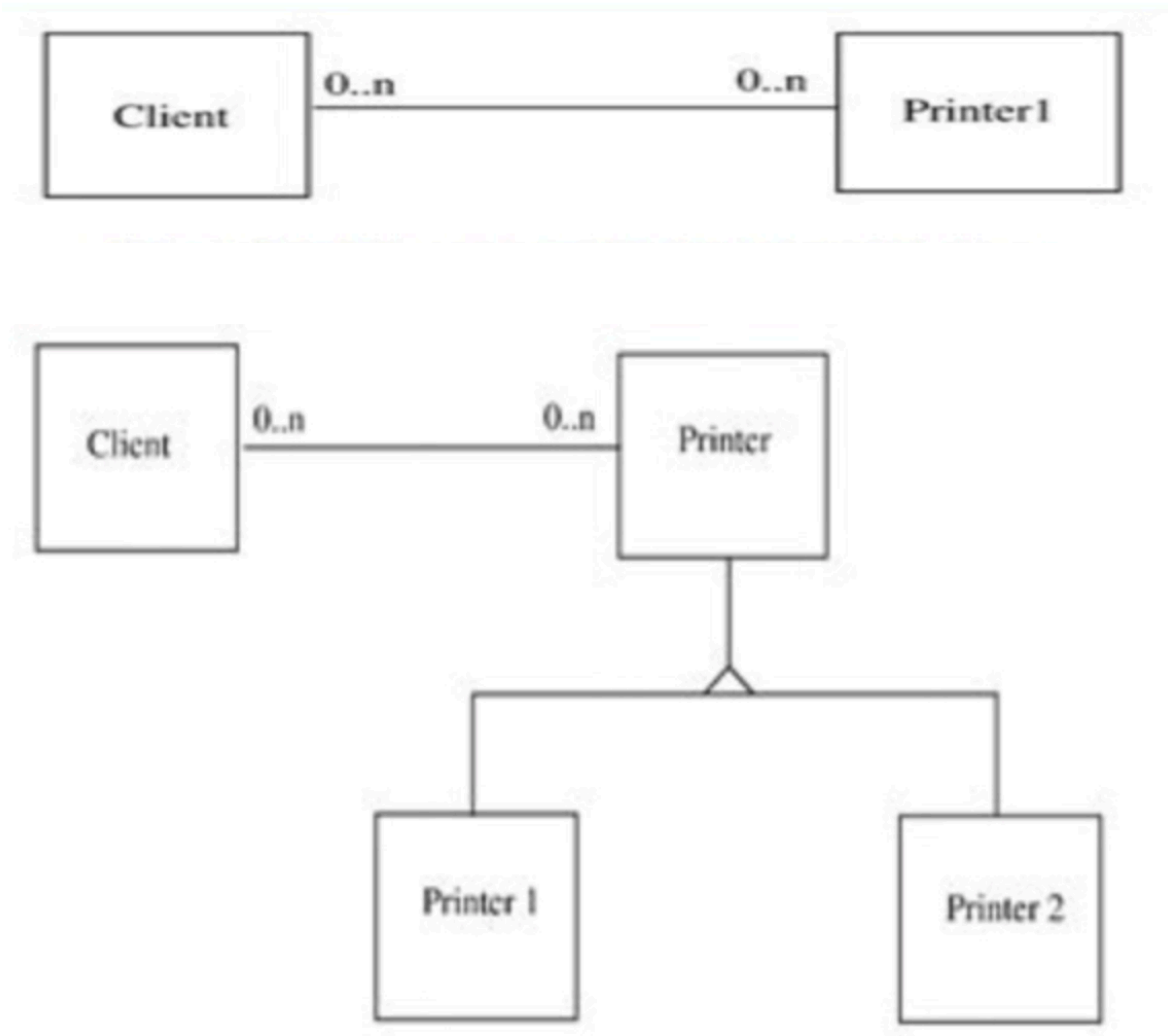


OPEN-CLOSED PRINCIPLE

- Software entities should be open for extension but closed for modification

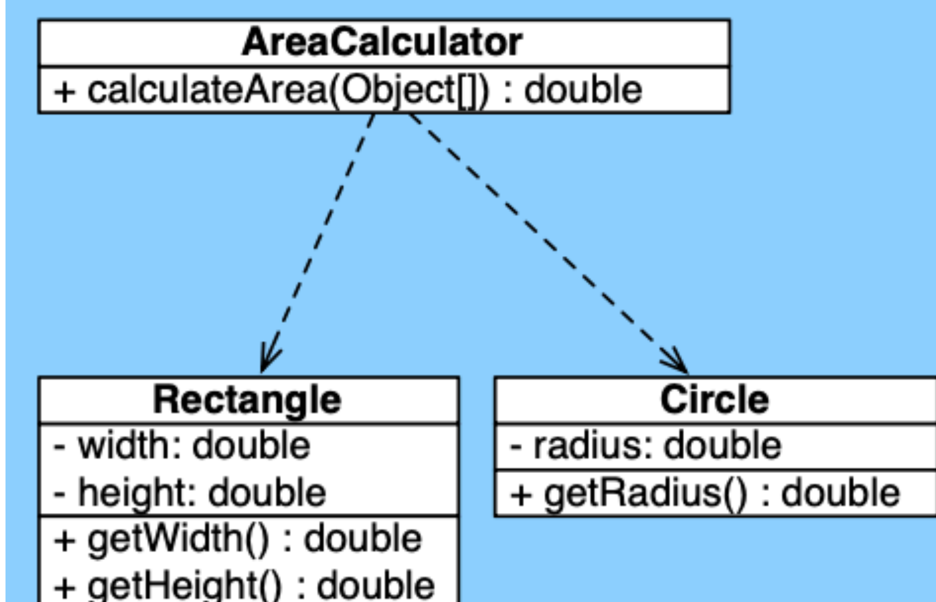
OPEN-CLOSED PRINCIPLE

- Software entities should be open for extension but closed for modification



OPEN-CLOSED PRINCIPLE

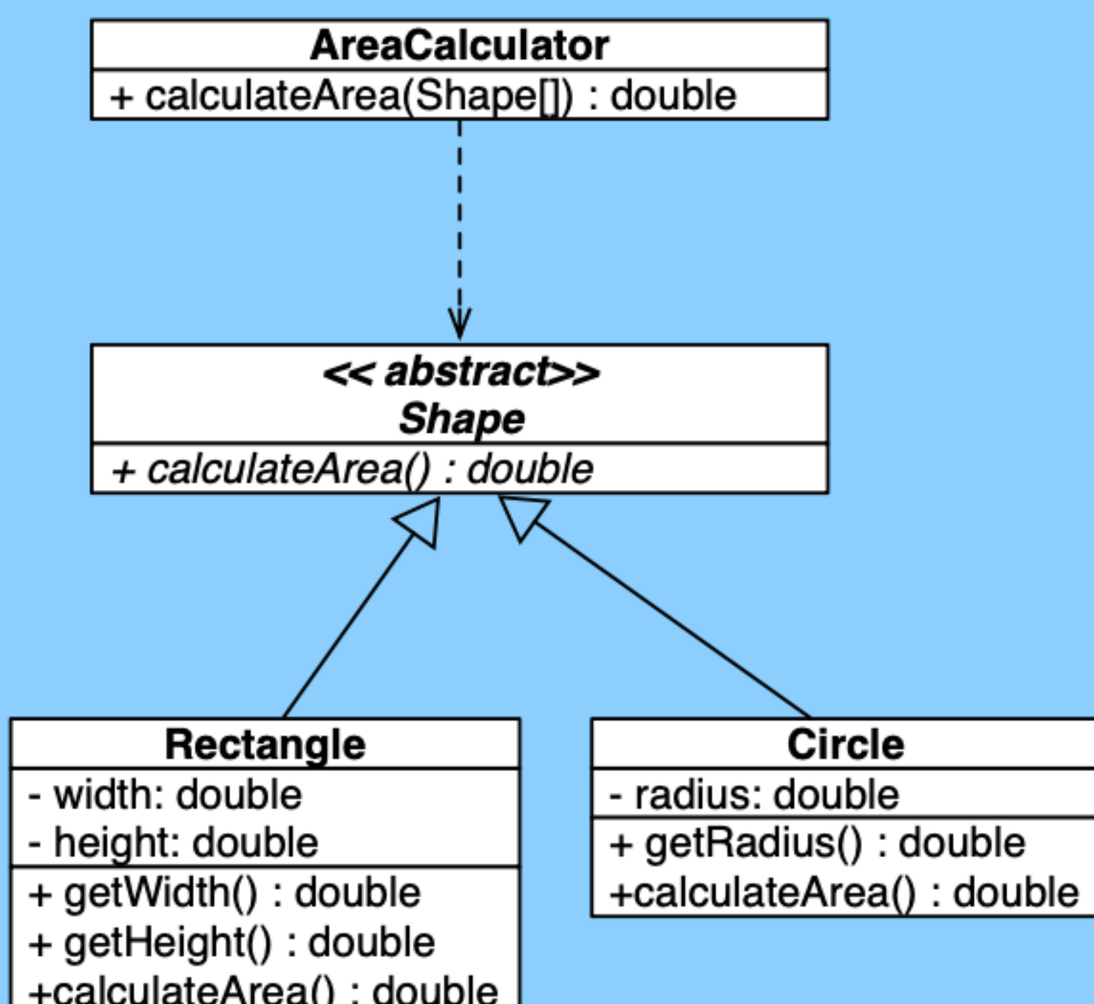
➤ Software entities should be open for extension but closed for modification




```
public class AreaCalculator
{
  public double calculateArea(Object[] shapes)
  {
    double result = 0;

    for (Object shape : shapes)
    {
      if (shape instanceof Rectangle)
      {
        Rectangle rect = (Rectangle) shape;
        result += (rect.getWidth()*rect.getHeight());
      }

      if (shape instanceof Circle)
      {
       Circle circ = (Circle) shape;

        result += Math.PI*(circ.getRadius()*circ.getRadius());
      }
    }
    return result;
  }
}
```

OPEN-CLOSED PRINCIPLE

➤ Software entities should be open for extension but closed for modification

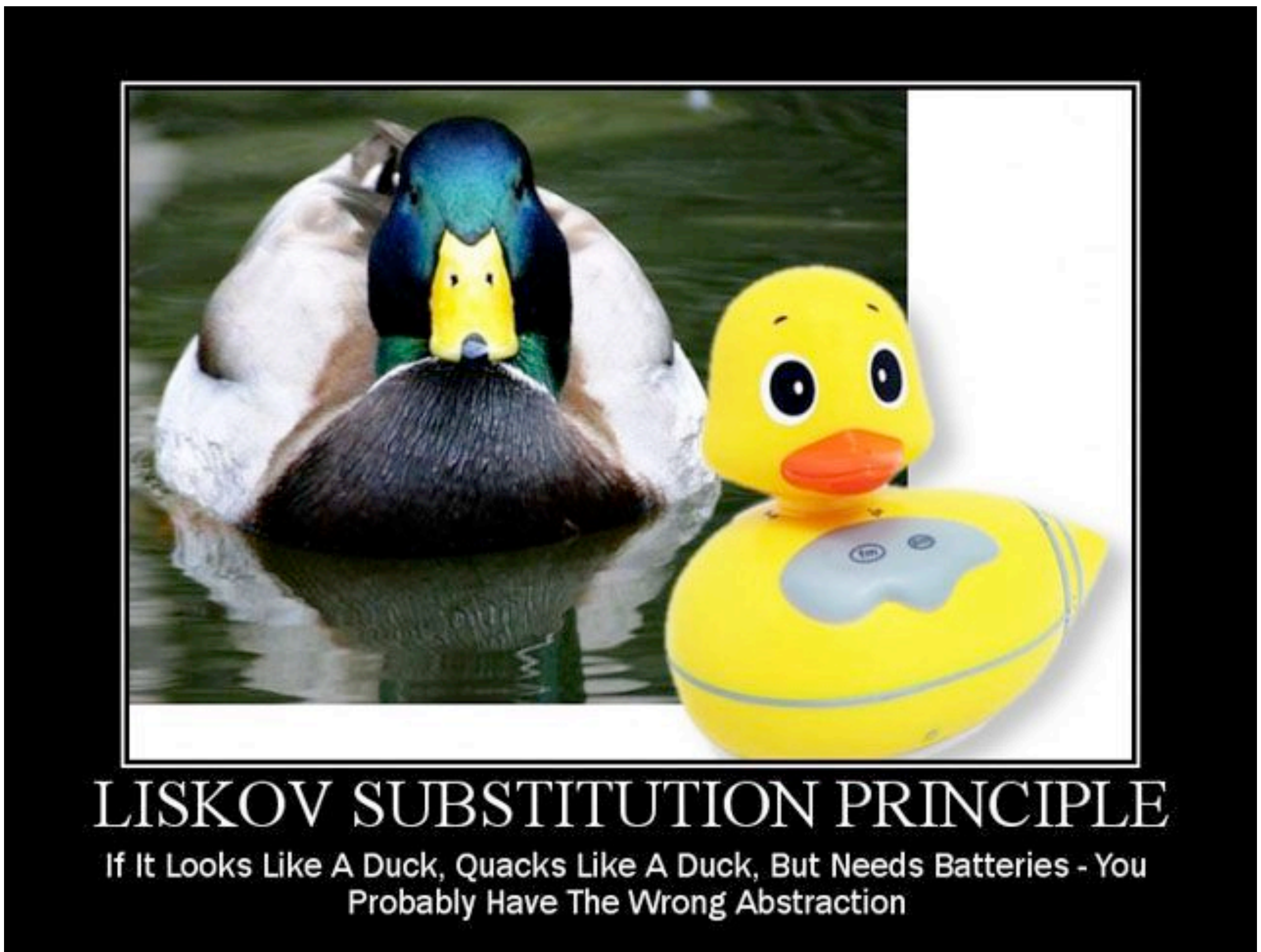



```
public class AreaCalculator
{
  public double calculateArea(Shape[] shapes)
  {
    double result = 0;

    for (Shape shape : shapes)
    {
      result += shape.calculateArea();
    }

    return result;
}
```

```
public class Circle extends Shape
{
  ...
  public double calculateArea()
  {
    return Math.PI*this.radius*this.radius;
  }
}
```

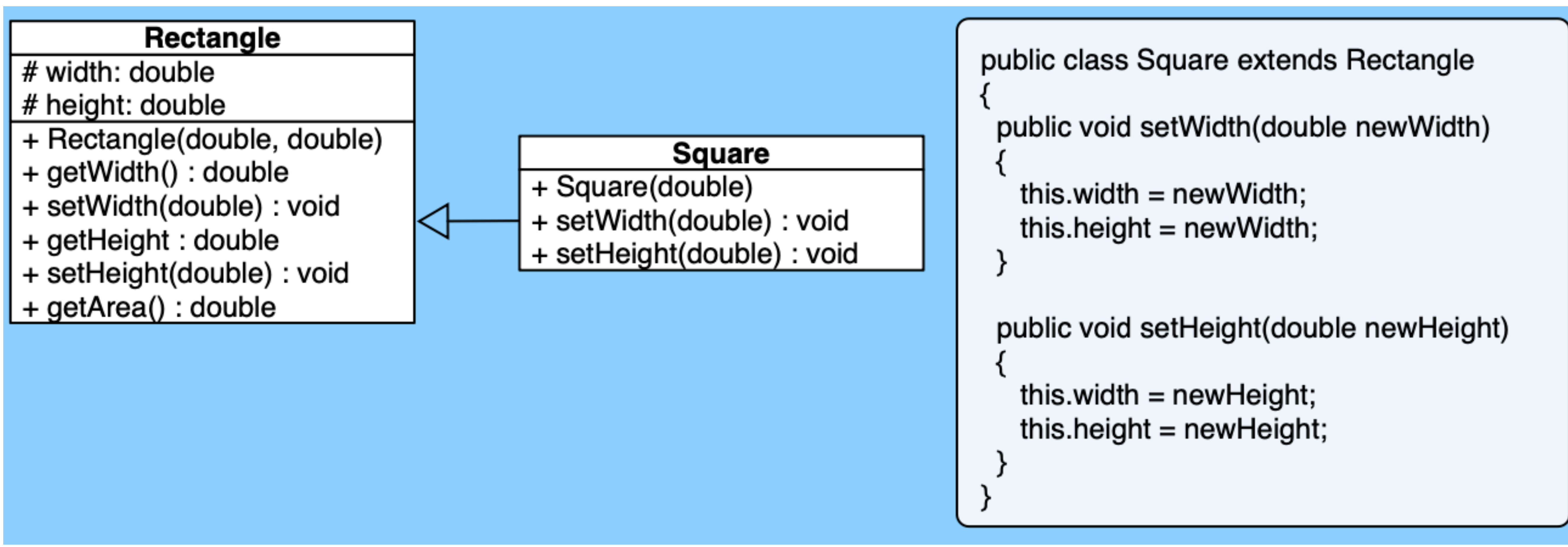
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- This principle says that objects of a superclass must be replaceable with objects of their subclasses, and the application should still work as expected.



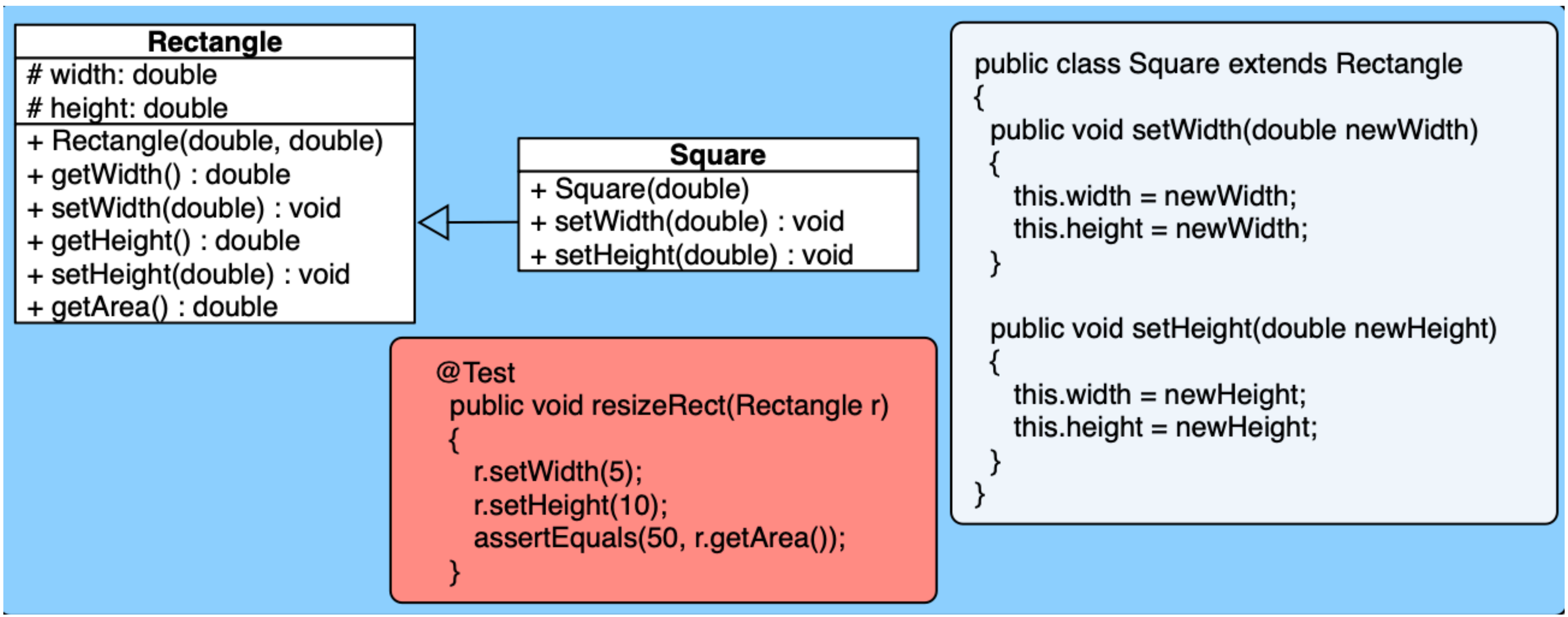
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- This principle says that objects of a superclass must be replaceable with objects of their subclasses, and the application should still work as expected.



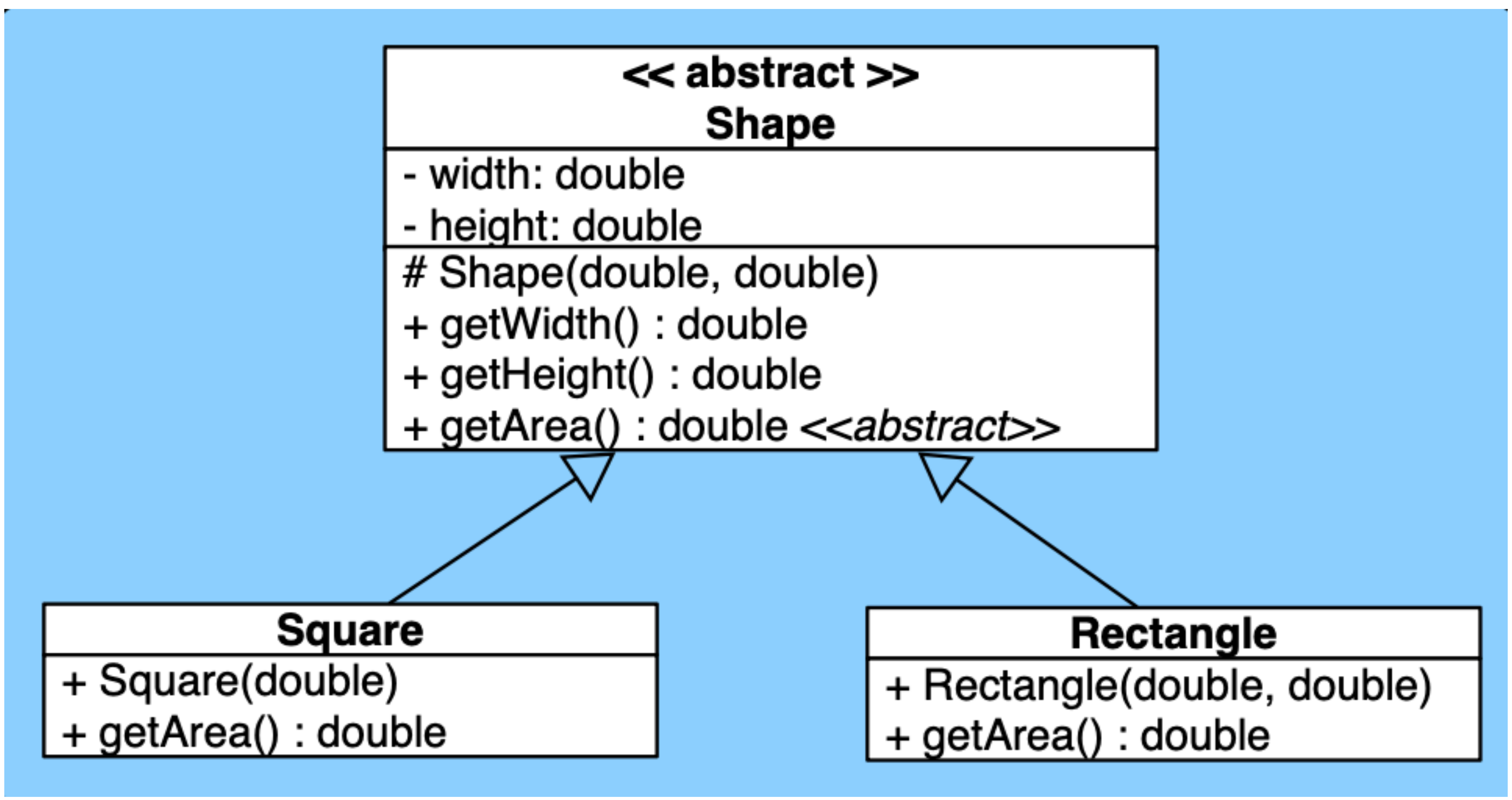
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- This principle says that objects of a superclass must be replaceable with objects of their subclasses, and the application should still work as expected.



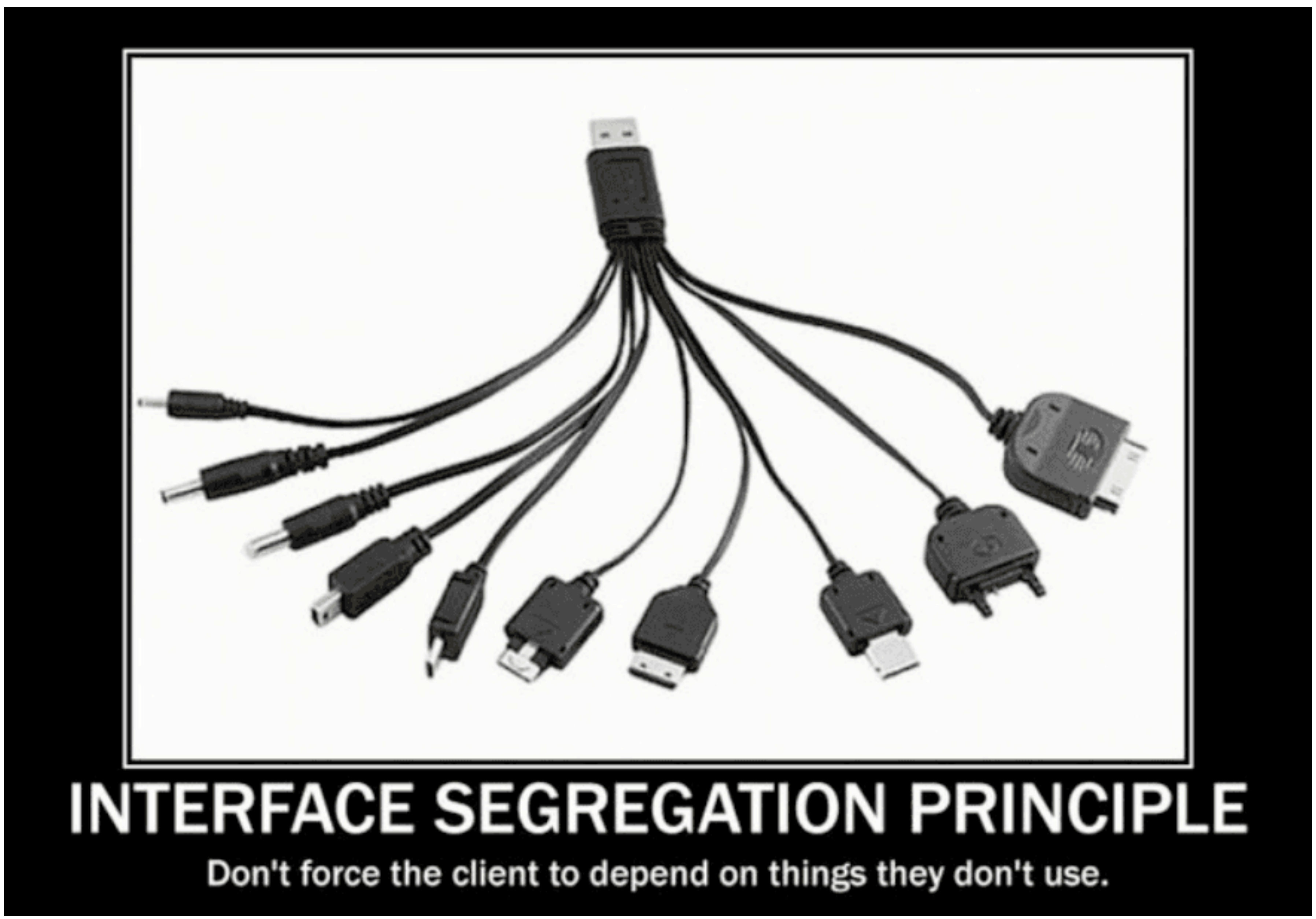
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- This principle says that objects of a superclass must be replaceable with objects of their subclasses, and the application should still work as expected.

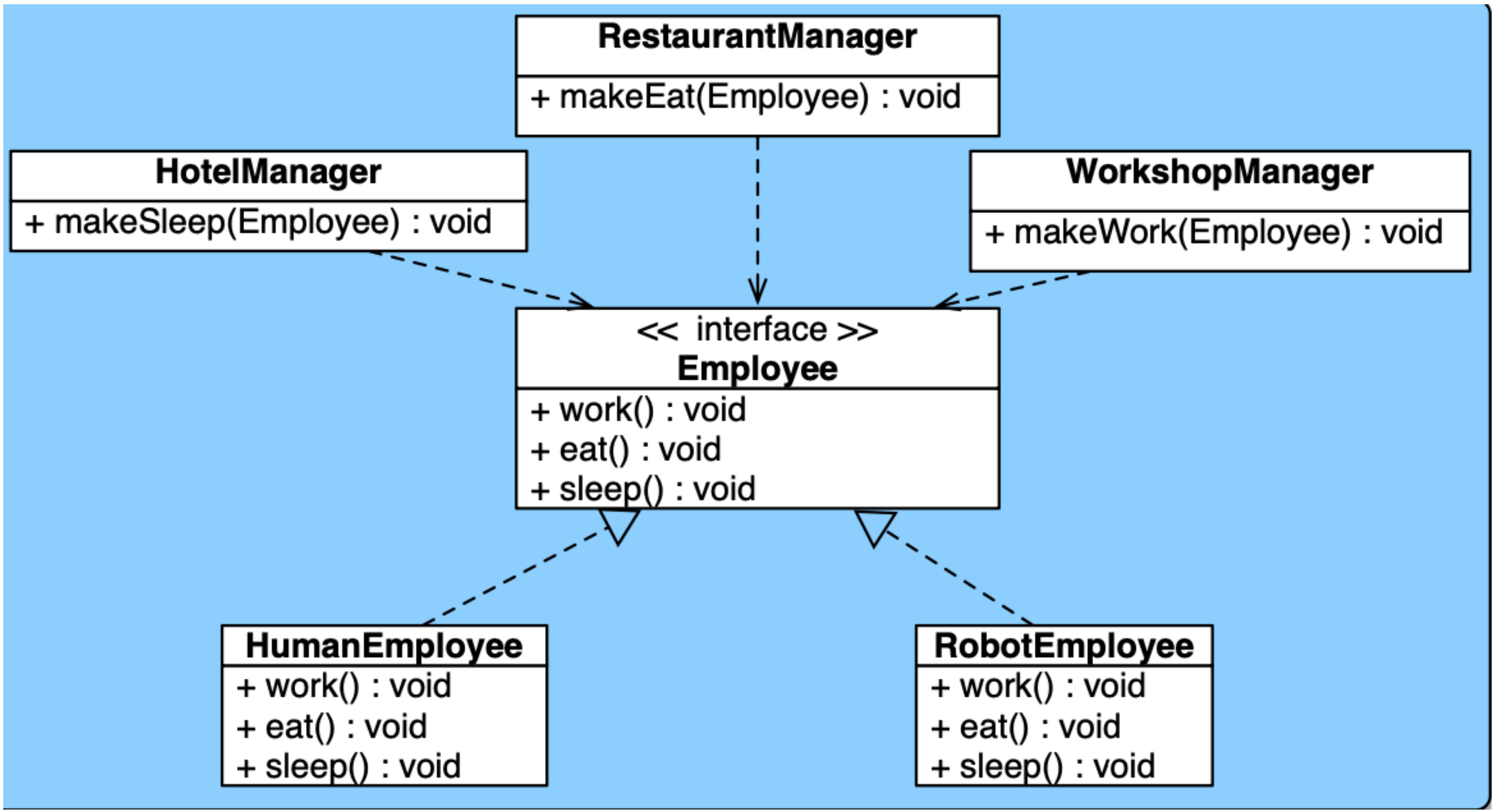


INTERFACE SEGREGATION PRINCIPLE

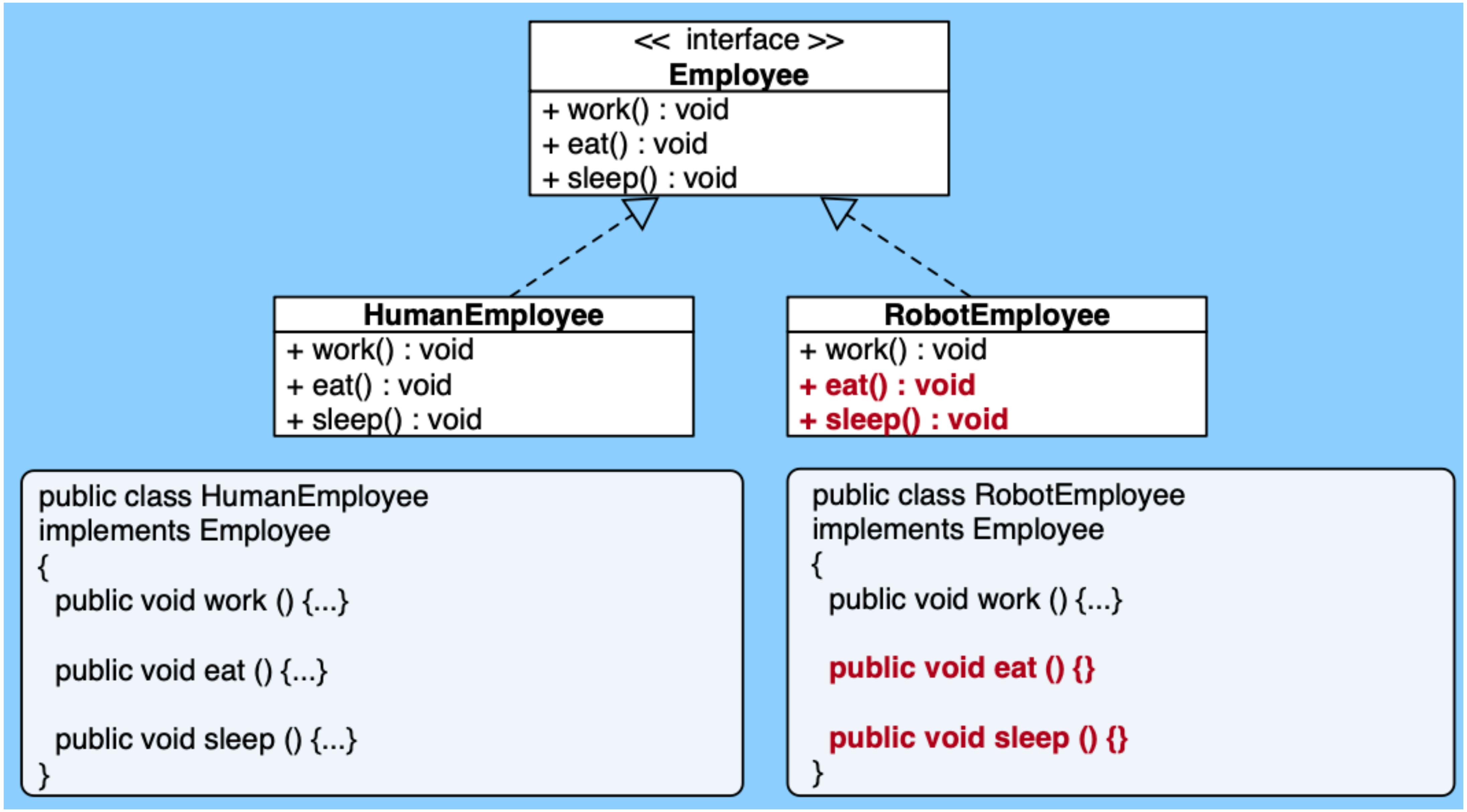
- The software should be split into multiple independents parts. Side-effects should be reduced as much as possible to ensure independence



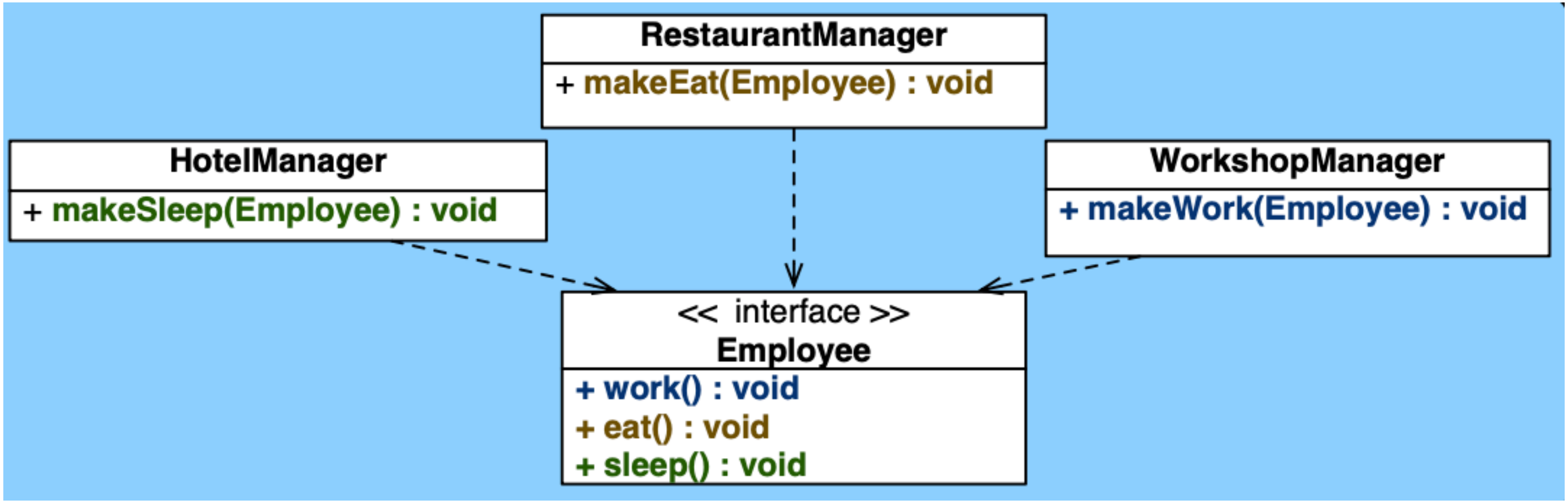
INTERFACE SEGREGATION PRINCIPLE



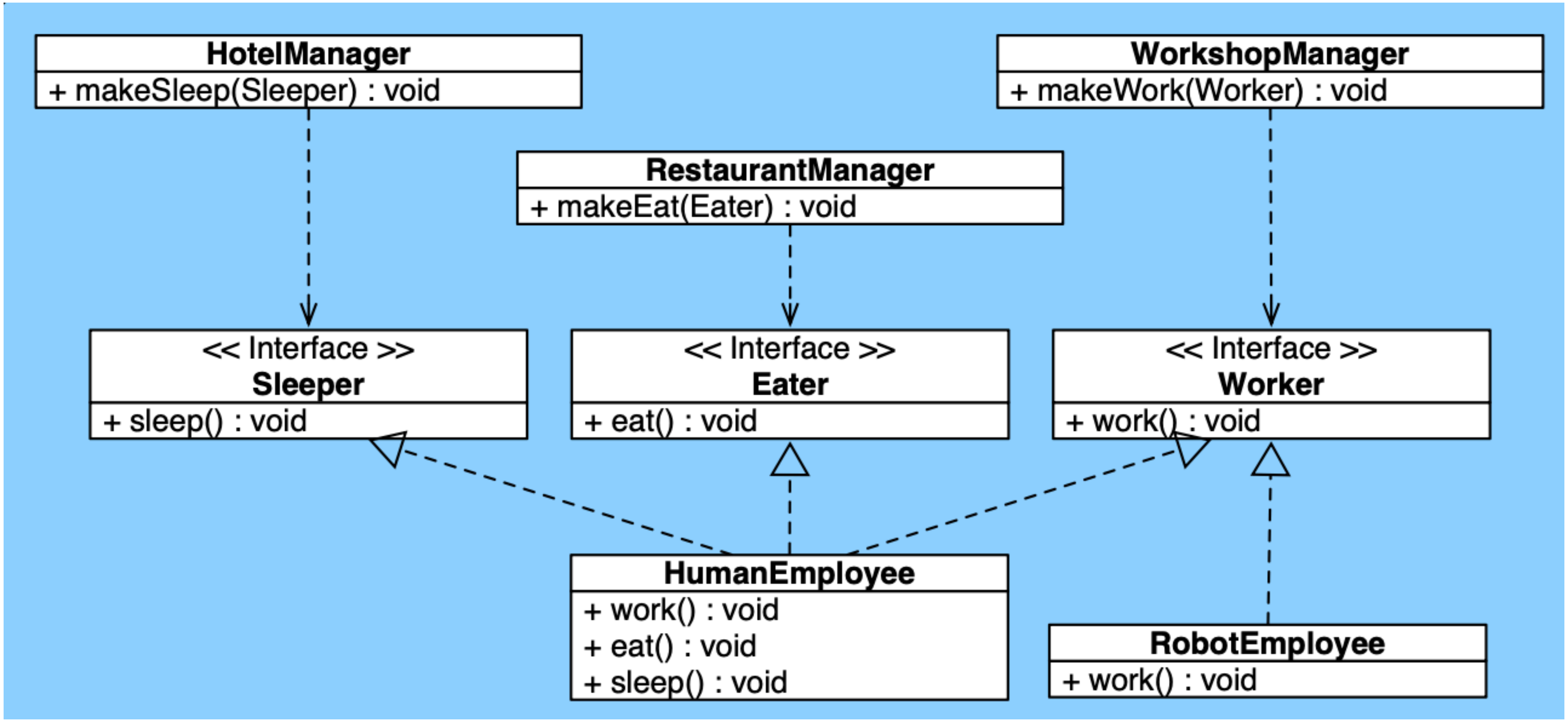
INTERFACE SEGREGATION PRINCIPLE



INTERFACE SEGREGATION PRINCIPLE

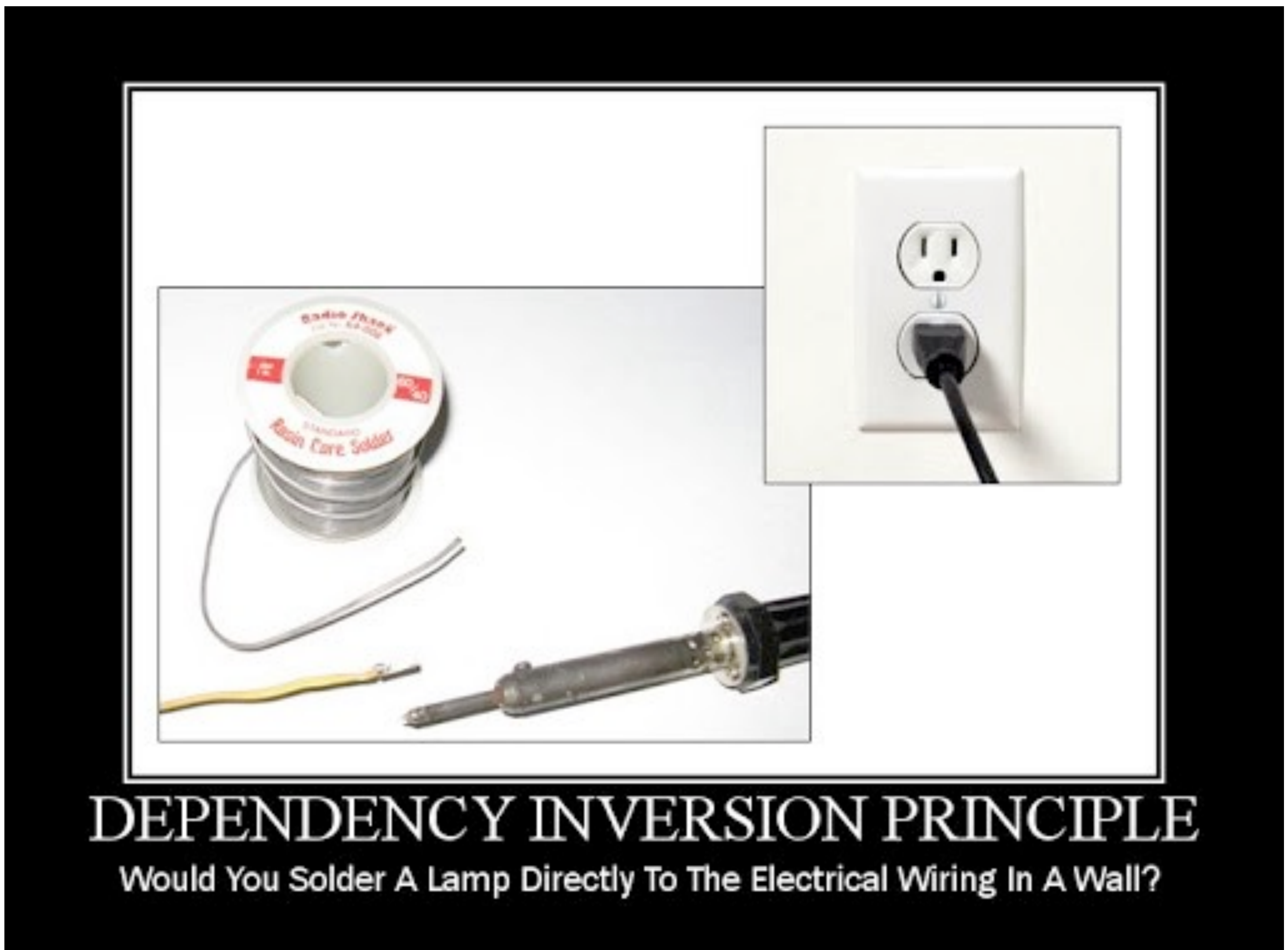


INTERFACE SEGREGATION PRINCIPLE



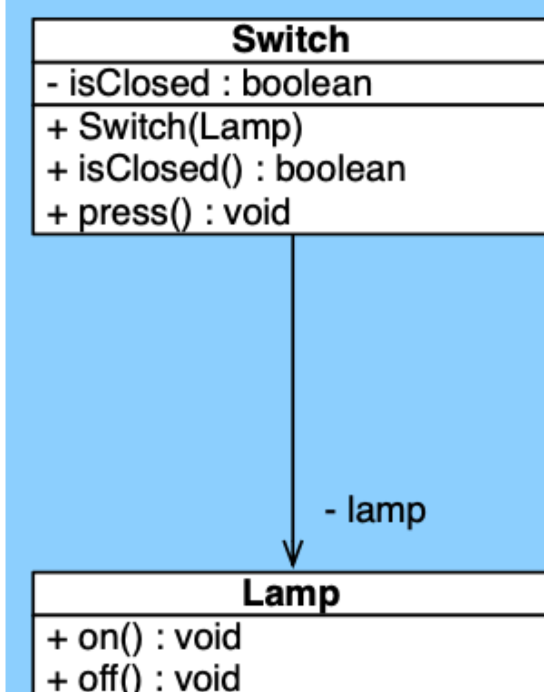
DEPENDENCY INVERSION PRINCIPLE

- We should rely on abstractions, not on concrete implementations. The software should have low coupling and high cohesion.



DEPENDENCY INVERSION PRINCIPLE

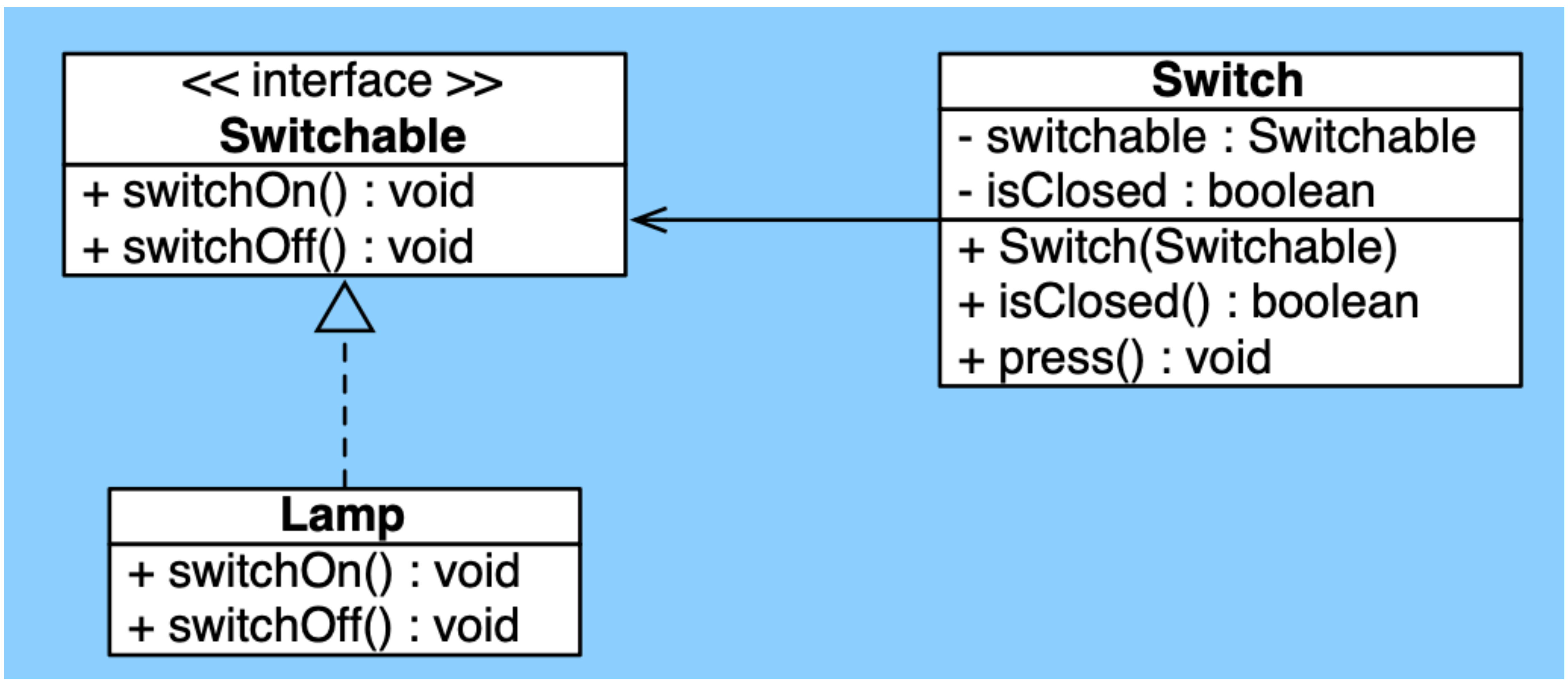
- We should rely on abstractions, not on concrete implementations. The software should have low coupling and high cohesion.




```
public class Switch
{
  private Lamp lamp;
  private boolean isClosed;

  public Switch(Lamp theLamp)
  {
    this.lamp = theLamp;
    this.isClosed = false;
  }

  public boolean isClosed() {return this.isClosed;}

  public void press()
  {
    if (this.isClosed)
    {
      this.isClosed = false;
      this.lamp.off();
    }
    else
    {
      this.isClosed = true;
      this.lamp.on();
    }
}
```

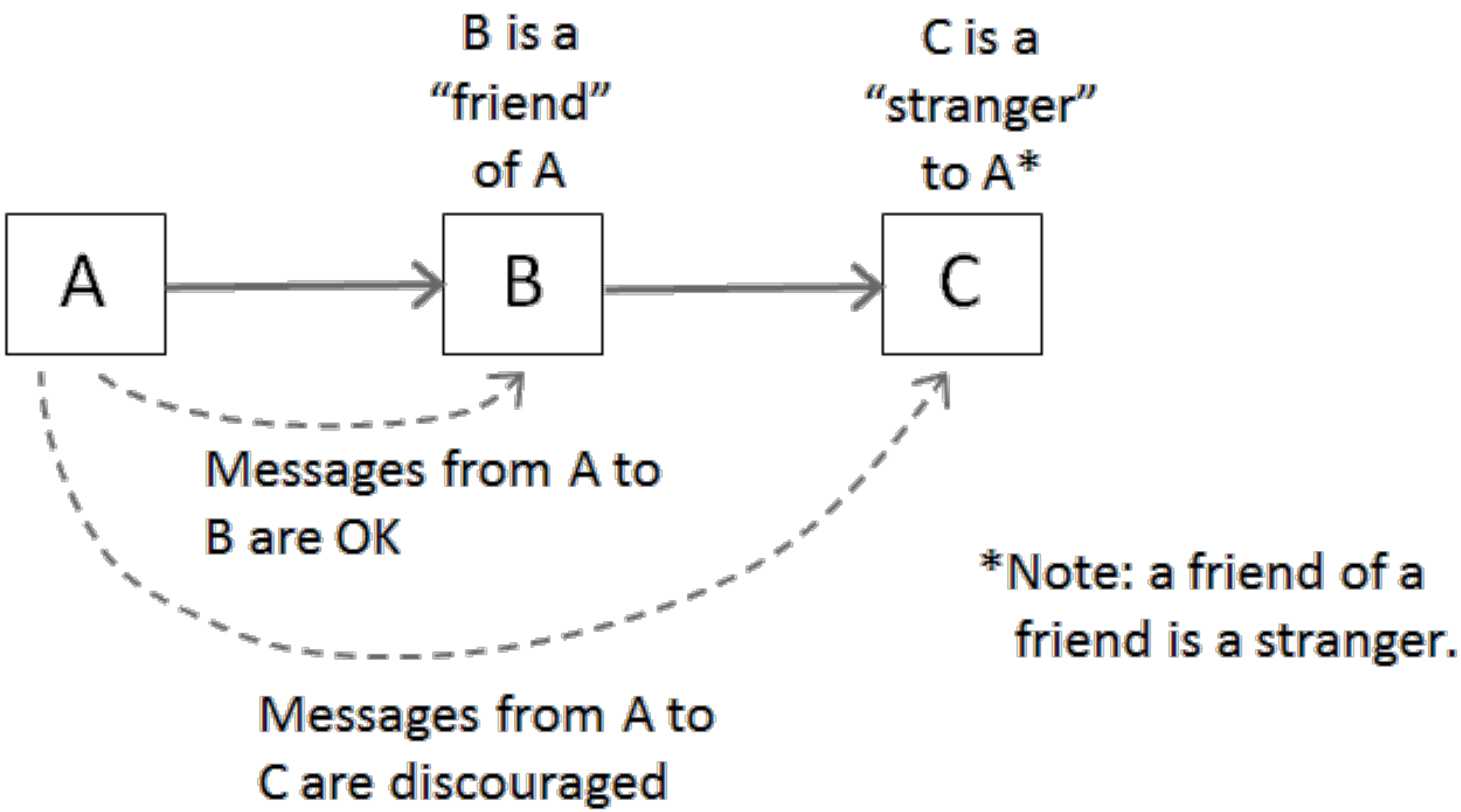
DEPENDENCY INVERSION PRINCIPLE

- We should rely on abstractions, not on concrete implementations. The software should have low coupling and high cohesion.



DEMETER'S LAW

- a module should not have the knowledge on the inner details of the objects it manipulates.
- Don't talk to strangers



DEMETER'S LAW

- a module should not have the knowledge on the inner details of the objects it manipulates.
- Don't talk to strangers

```
var data = new A().GetObjectB().GetObjectC().GetData();
```

OTHER PRINCIPLES

- Prevent side-effects in functions / methods
- Use early returns principle:

```java
public int confusingFonction(String name, int value, AuthenticationInfo permissions) {
    int retval = SUCCESS;
    if (globalCondition) {
        if (name != null && !name.equals("")) {
            if (value != 0) {
                if (permissions.allow(name)) {
                    // Action if allowed
                } else {
                    retval = DENY;
                }
            } else {
                retval = BAD_VALUE;
            }
        } else {
            retval = INVALID_NAME;
        }
    } else {
        retval = BAD_COND;
    }
    return retval;
}
```

OTHER PRINCIPLES

- Prevent side-effects in functions / methods
- Use early returns principle:

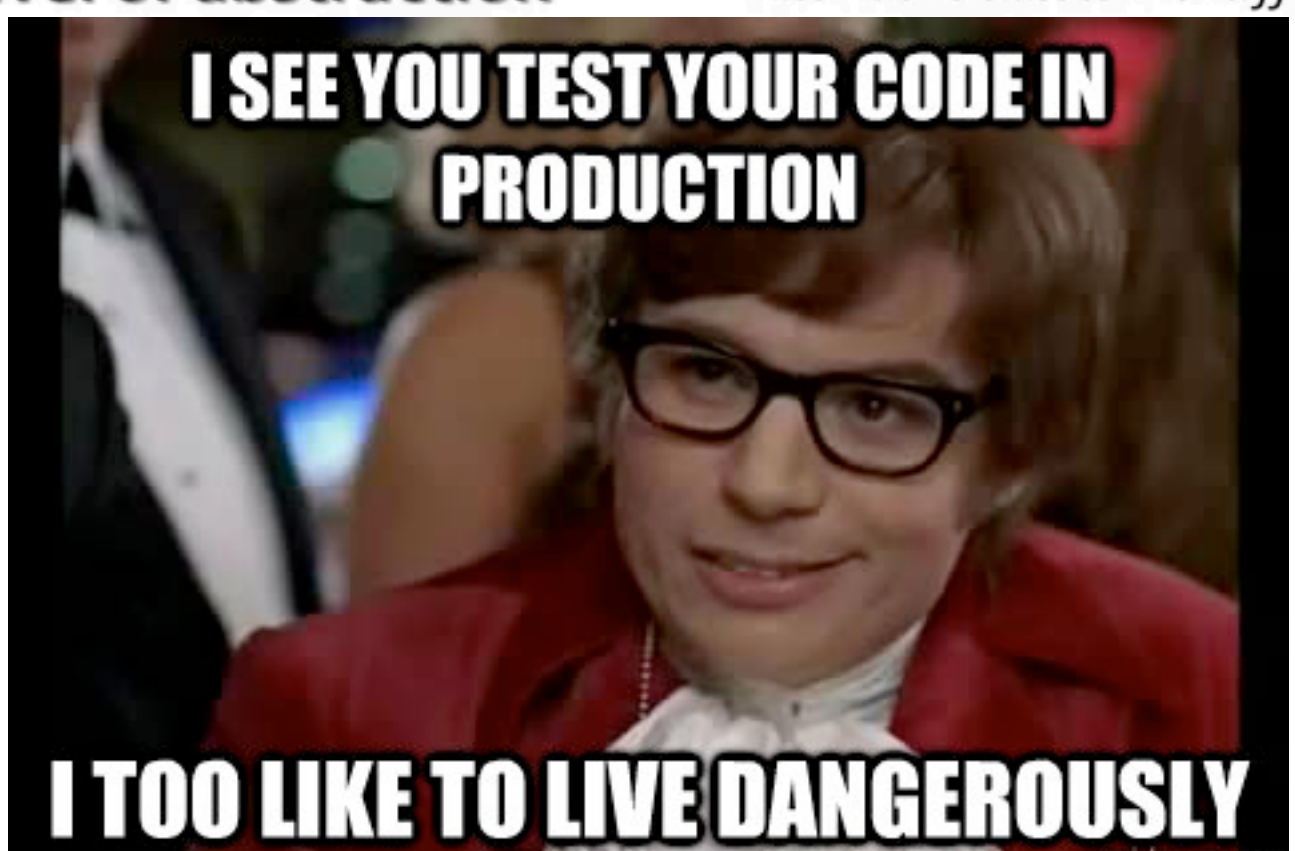
```java
public int lessConfusingFonction(String name, int value, AuthenticationInfo perms) {
    if (!globalCondition) {
        return BAD_COND;
    }
    if (name == null || name.equals("")) {
        return BAD_NAME;
    }
    if (value == 0) {
        return BAD_VALUE;
    }
    if (!perms.allow(name)) {
        return DENY;
    }
    // Action if allowed
    return SUCCESS;
}
```

OTHER PRINCIPLES

- Prevent side-effects in functions / methods
- Use early returns principle
- Limit the numbers of parameters (2 / 3)

CODE SMELLS / CODDING HORS

Large class
Data class
Duplicated code
Refused bequest
Uncommunicative name
Lazy class
Type embedded in name
Message chain
Conditional complexity
Inappropriate intimacy
Data clumps
Dead code
Speculative generality
Comments
Shotgun Surgery
Primitive obsession
Inconsistent names
Divergent change
Feature envy
Middle man
Temporary field
Long parameter list
Long method
Wrong level of abstraction
Alternative classes with different interfaces



TOOLS

CODE INDENTATION

- Clang-format

```
clang-format -I fichier.c
```

CODE INDENTATION

- Clang-format

```
void selectionR(unsigned char m[HAUTEUR][LARGEUR],int x,int y,int x1,int y1, unsigned char
 reception[HAUTEUR][LARGEUR]){



  int j,i;



  for(i=x;i< x1;i++)

    for(j=y;j< y1;++j){

      reception[i][j]=m[i][j];

       m[i][j]=0;

    }
}
```

CODE INDENTATION

➤ Clang-format

```
void selectionR (unsigned char m[HAUTEUR][LARGEUR],
                 int           x,
                 int           y,
                 int           x1,
                 int           y1,
                 unsigned char reception[HAUTEUR][LARGEUR])
{
    int j, i;
    for (i = x; i < x1; i++)
        for (j = y; j < y1; ++j)
        {
            reception[i][j] = m[i][j];
            m[i][j]         = 0;
        }
}
```

DUPLICATES SEARCH

➤ pmd

```
$ pmd cpd --minimum-tokens 70 --language c --files projet.c
    Found a 4 line (112 tokens) duplication in the following files:
    Starting at line 43 of projet.c
    Starting at line 79 of projet.c

 void selection_ellipse(unsigned char image[HAUTEUR][LARGEUR],int C_x, int C_y, int a, int b, unsigned
char selection[2*b][2*a]){
    for (int y = 0; y <2*b; y++) {  //balaie un rectangle de hauteur 2b et de largeur 2a (rectangle
contenant l'ellipse)
        for (int x = 0; x < 2*a; x++) {
            if   ( (  (pow((x-a),2))/(a*a) + (pow((y-b),2))/(b*b))  <= 1 ) //vérifie que la portion
d'image est bien dans l'ellipse
```

USE STATIC CHECKERS

- clang-tidy, rats, cppcheck, oclint…

```
$ gcc negative.c
$ clang-tidy --quiet -checks='*' negative.c --
    negative.c:12:3: warning: multiple declarations in a single statement reduces readability
[readability-isolate-declaration]
    negative.c:12:7: warning: variable 'j' is not initialized [cppcoreguidelines-init-variables]
    negative.c:12:9: warning: variable 'i' is not initialized [cppcoreguidelines-init-variables]
    negative.c:14:20: warning: statement should be inside braces [google-readability-braces-around-
statements,hicpp-braces-around-statements,readability-braces-around-statements]
    negative.c:25:3: warning: multiple declarations in a single statement reduces readability
[readability-isolate-declaration]
    negative.c:25:7: warning: variable 'i' is not initialized [cppcoreguidelines-init-variables]
    negative.c:25:9: warning: variable 'j' is not initialized [cppcoreguidelines-init-variables]
    negative.c:28:9: warning: narrowing conversion from 'float' to 'int' [bugprone-narrowing-
conversions,cppcoreguidelines-narrowing-conversions]
    negative.c:28:14: warning: narrowing conversion from 'int' to 'float' [bugprone-narrowing-
conversions,cppcoreguidelines-narrowing-conversions]
    negative.c:44:14: warning: 255 is a magic number; consider replacing it with a named constant
[cppcoreguidelines-avoid-magic-numbers,readability-magic-numbers]
    [...]
```

USE MEMORY CHECKERS / DATA VALIDATOR

- valgrind, clang
- yamllint, csvlint

BUILD AUTOMATION PIPELINE — GITLAB

```
image: debian

before_script:
    - apt-get update -y
    - apt-get upgrade -y
    - apt-get install -y clang-format clang-tidy clang-tools clang make check cppcheck libcppunit-
subunit-dev lcov llvm valgrind

stages:
    - codestyling
    - check
    - build
    - test
    - coverage
    - clean

job:codestyling:
    stage: codestyling
    script: ./scripts/run-clang-format.py -r src includes tests

job:check:tidy:
    stage: check
    script: clang-tidy src/*.c -- -Iincludes
    when: always

job:check:cppcheck:
    stage: check
    script: cppcheck --enable=warning,style,portability src/*.c
    when: always
```

BUILD AUTOMATION PIPELINE — GITLAB

```
    script: cppcheck --enable=warning,style,portability src/*.c
    when: always

job:build:
    stage: build
    script:
        - mkdir build
        - make main
    when: always
    artifacts:
        paths:
            - build/

job:test:
    stage: test
    script:
        - make tests
        - build/tri_tests
    when: on_success
    artifacts:
        paths:
            - build/

job:memcheck:
    stage: test
    script: valgrind build/tri_comp
    when: on_success

job:coverage:
    stage: coverage
    script:
```

THE END !