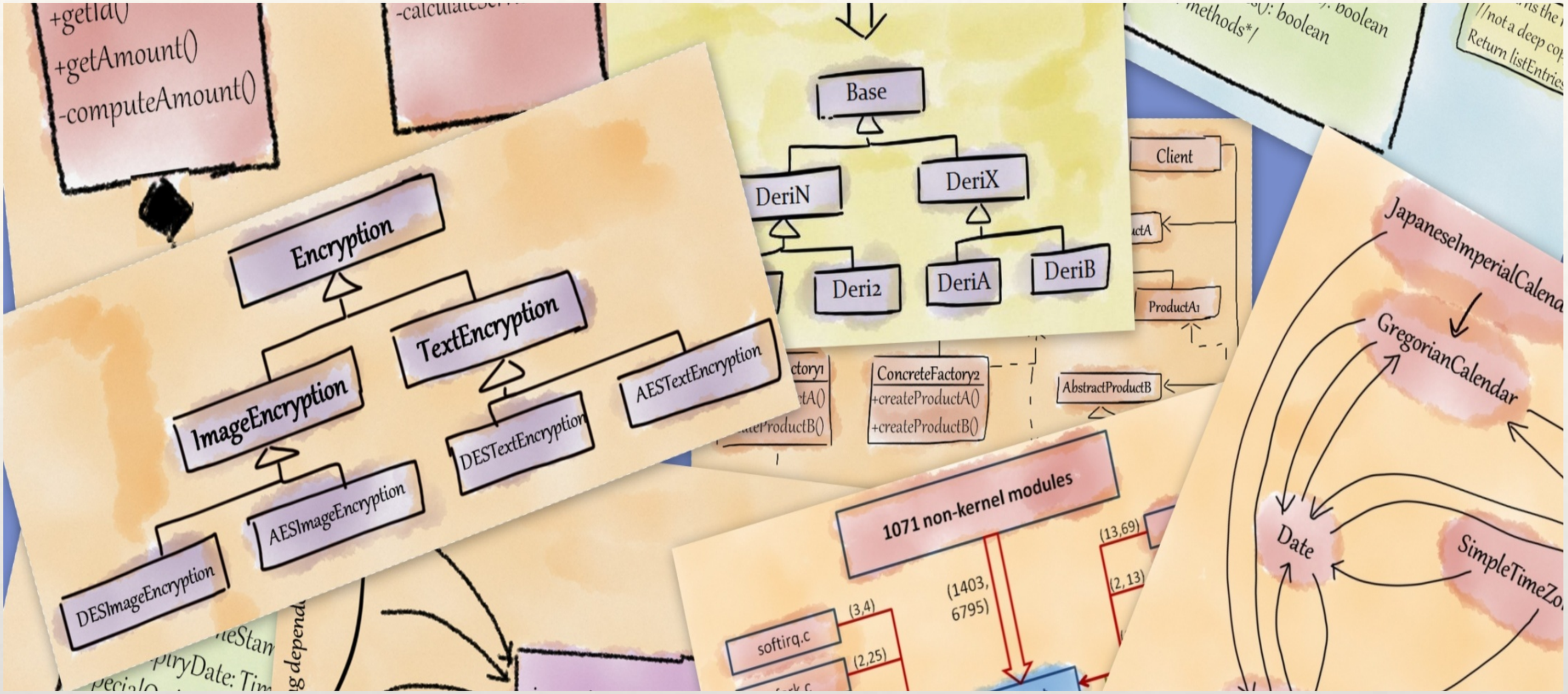


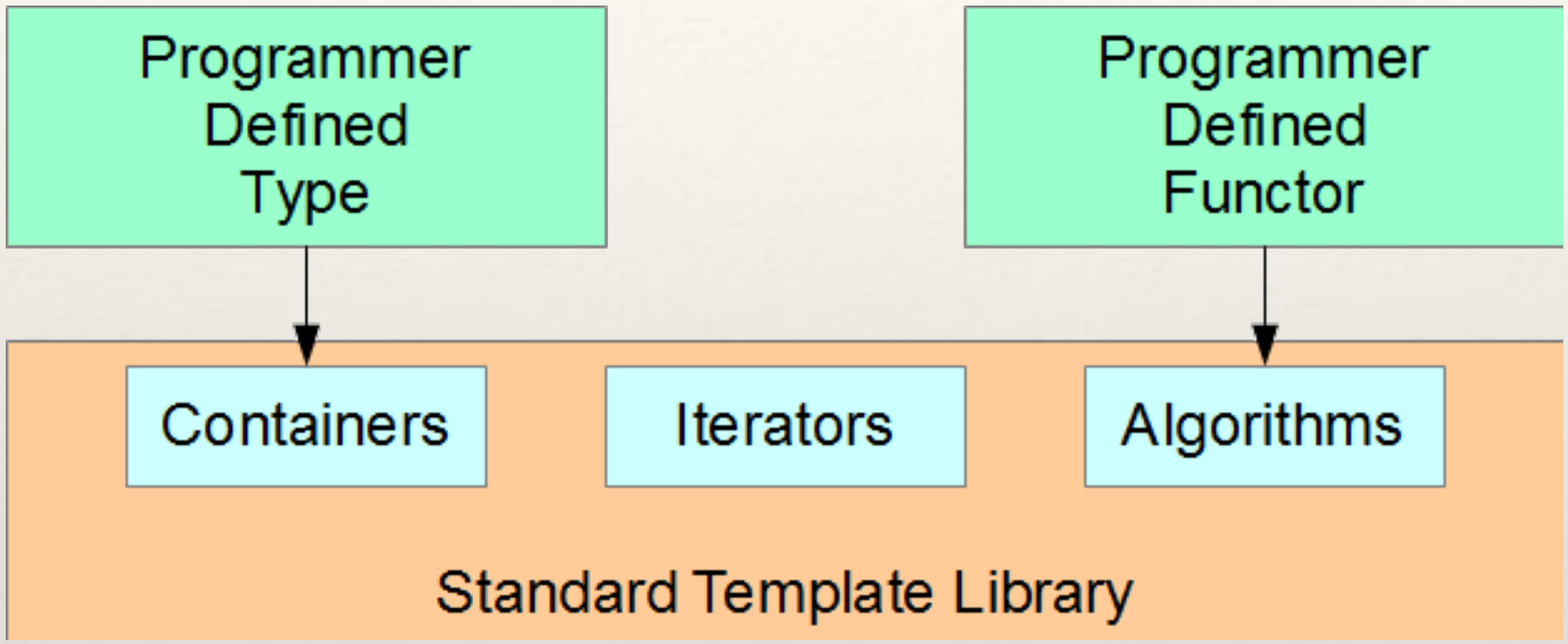
OOP in C++

Ganesh Samarthyam
ganesh@codeops.tech

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

Example of beautiful design



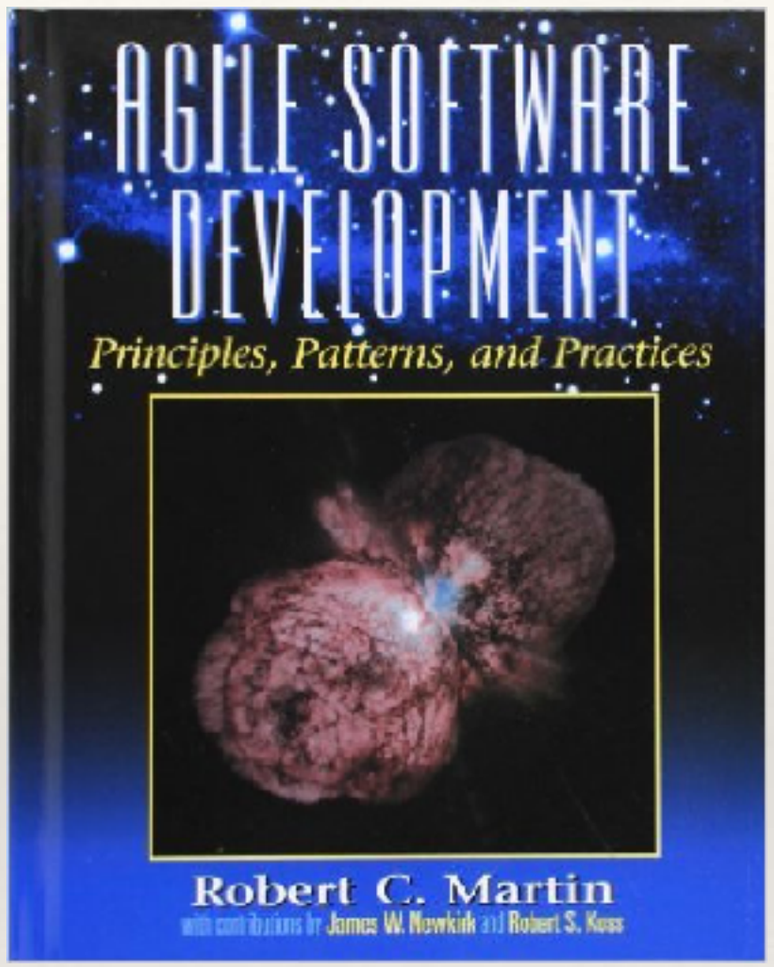

```
int arr[] = {1, 4, 9, 16, 25}; // some values
// find the first occurrence of 9 in the array
int * arr_pos = find(arr, arr + 4, 9);
std::cout<< “array pos = “<< arr_pos - arr << endl;

vector<int> int_vec;
for(int i = 1; i <= 5; i++)
        int_vec.push_back(i*i);
vector<int>::iterator vec_pos = find (int_vec.begin(), int_vec.end(), 9);
std::cout<< “vector pos = “<< (vec_pos - int_vec.begin());
```

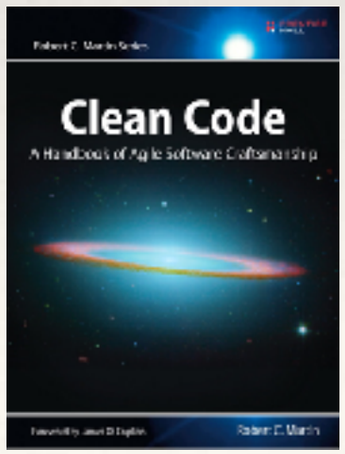
"The critical design tool for software development is a mind well educated in design principles"

- Craig Larman

Robert C. Martin

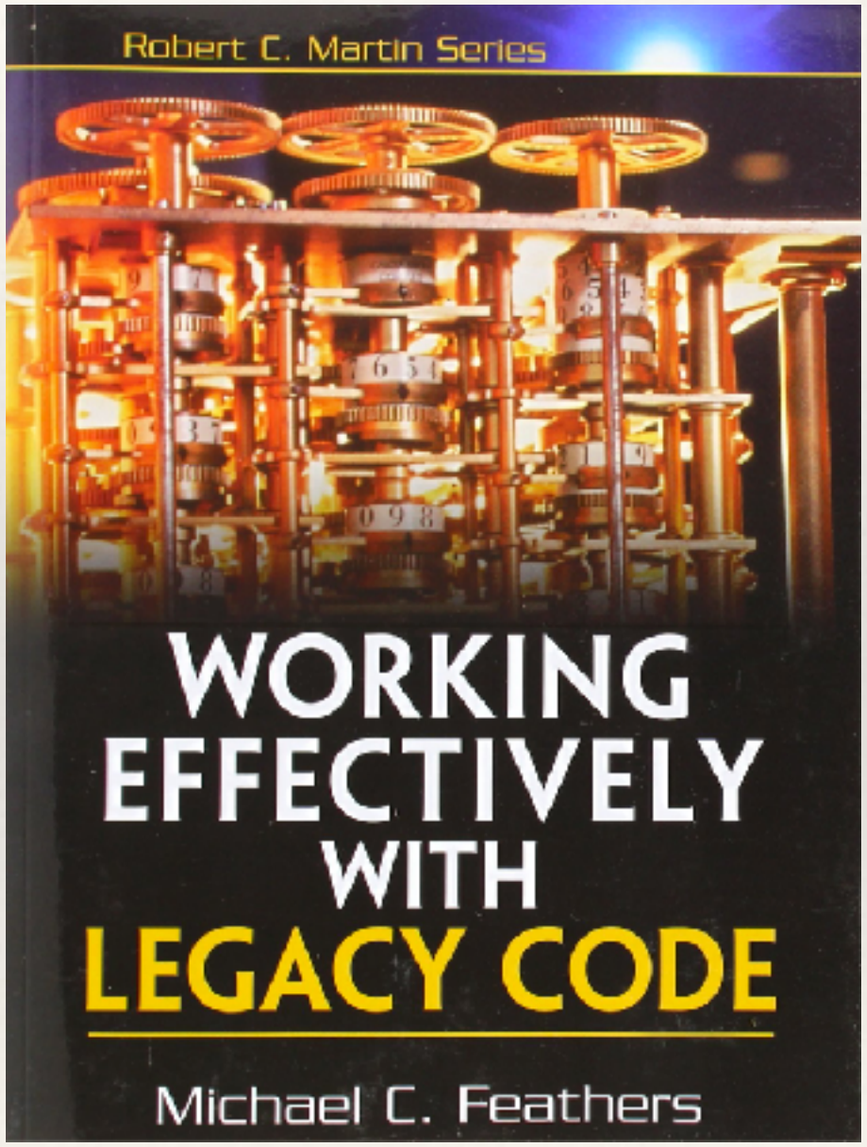
Formulated many principles and described many other important principles





Michael Feathers

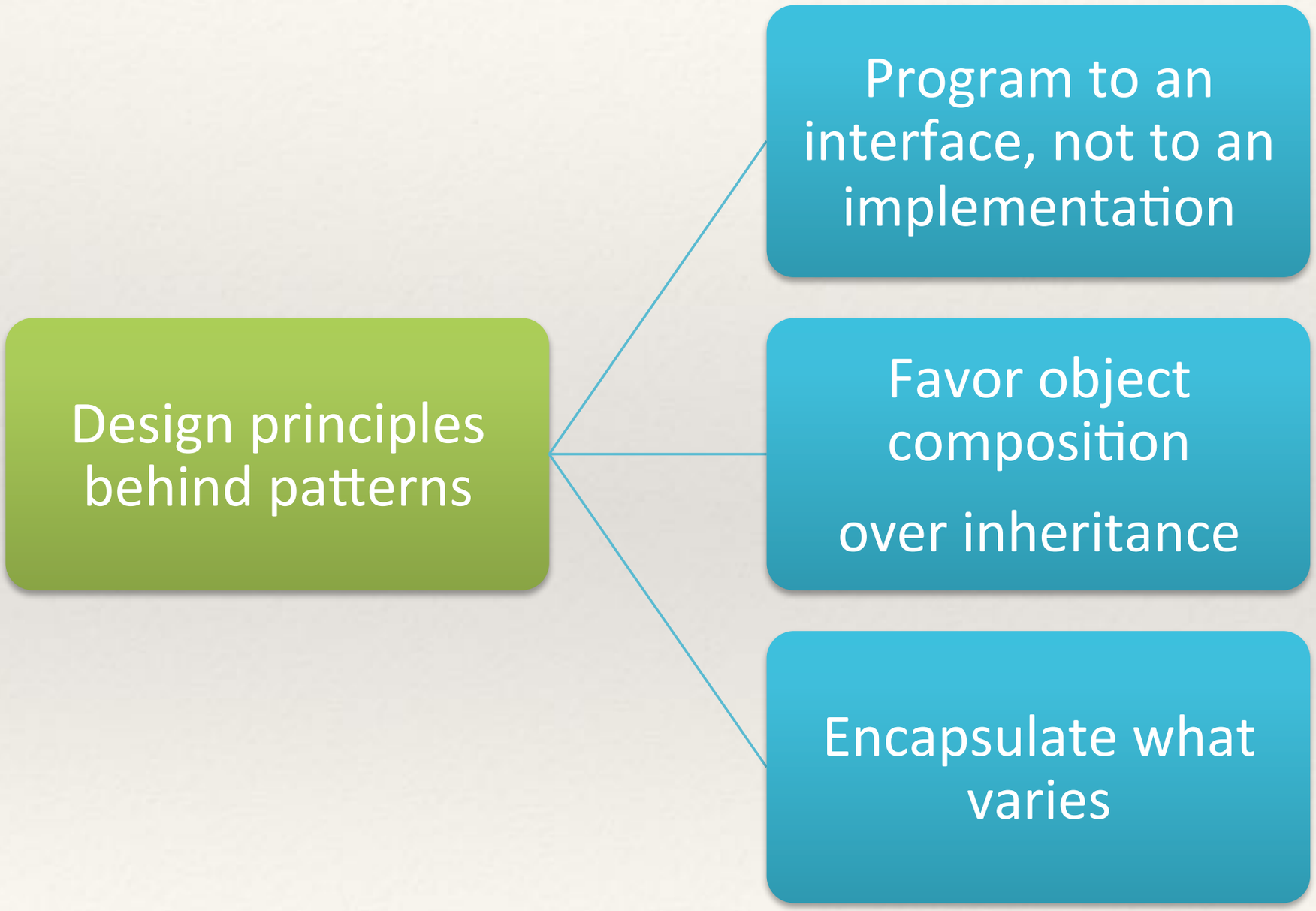
Michael Feathers coined the acronym SOLID in 2000s to remember first five of the numerous principles by Robert C. Martin



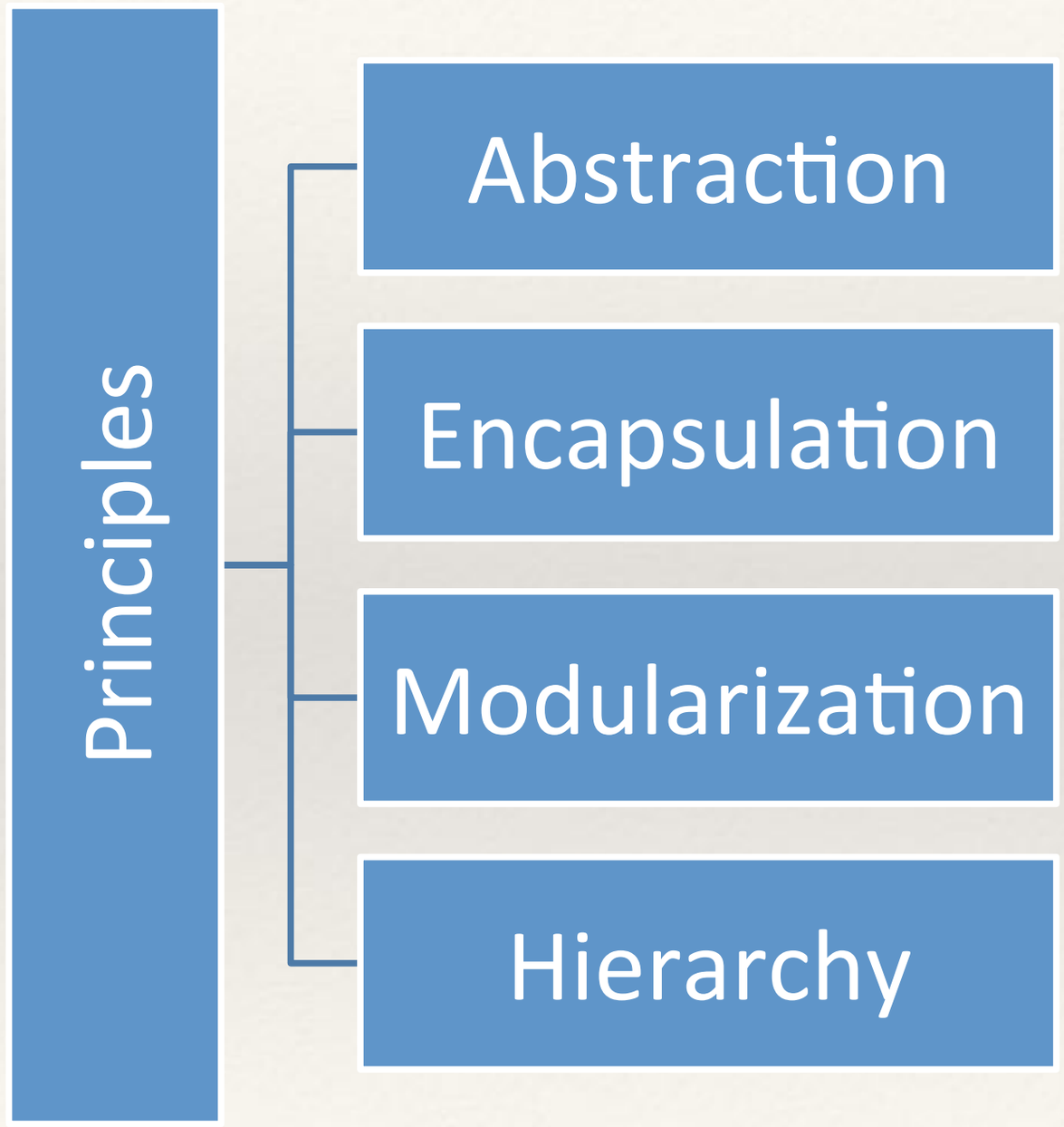
SOLID principles

S	Single Responsibility Principle	Every object should have a single responsibility and that should be encapsulated by the class
O	Open Closed Principle	Software should be open for extension, but closed for modification
L	Liskov's Substitution Principle	Any subclass should always be usable instead of its parent class
I	Interface Segregation Principle	Many client specific interfaces are better than one general purpose interface
D	Dependency Inversion Principle	Abstractions should not depend upon details. Details should depend upon abstractions

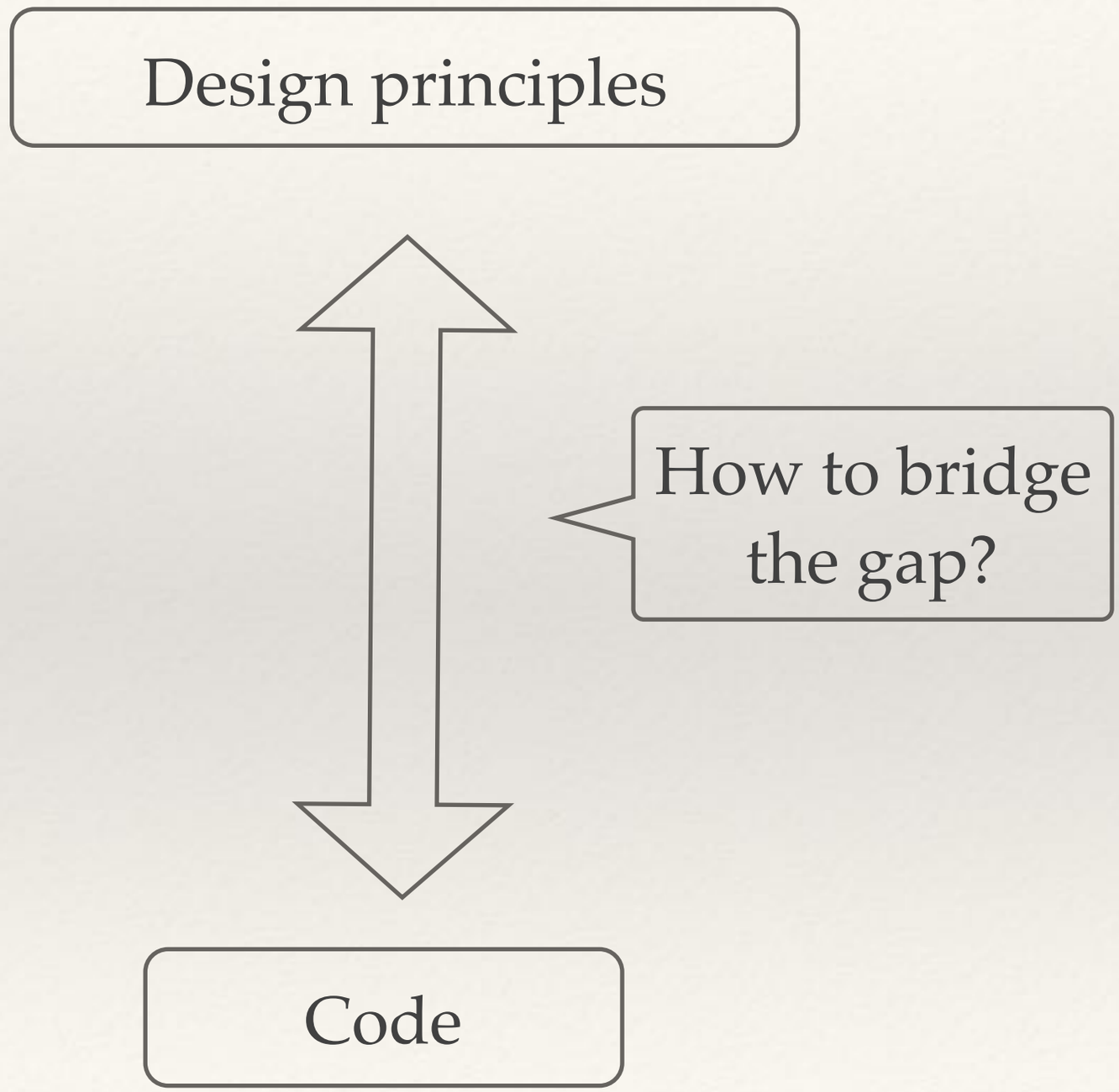
3 principles behind patterns



Booch's *fundamental* principles



How to apply principles in practice?



Why care about refactoring?

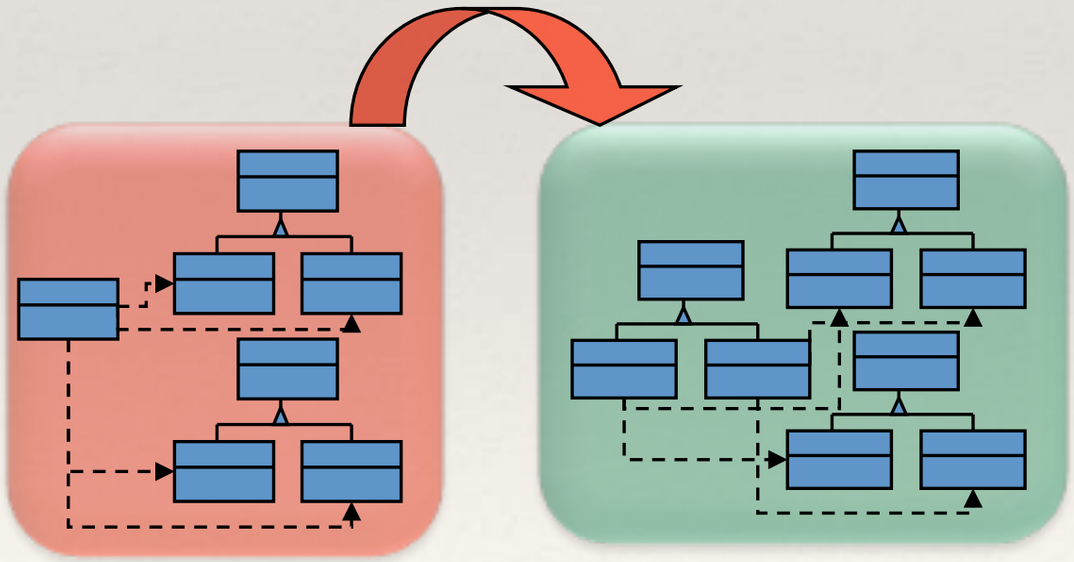
> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it

- Lehman's law of Increasing Complexity

What is refactoring?

Refactoring (noun): a *change* made to the *internal structure* of software to make it *easier to understand and cheaper to modify* without *changing its observable behavior*

Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior



What are smells?

"Smells are certain structures
in the code that suggest
(sometimes they scream for)
the possibility of refactoring."

Granularity of smells

Architectural
Cyclic dependencies between modules Monolithic modules Layering violations (back layer call, skip layer call, vertical layering, etc)
Design
God class Refused bequest Cyclic dependencies between classes
Code (implementation)
Internal duplication (clones within a class) Large method Temporary field

Procedural code

```
#include <string>
#include <iostream>
using namespace std;

enum Animal { rat, pig, frog };

string talk(enum Animal animal) {
  switch(animal) {
  case rat: return "squeak";
  case pig: return "oink";
  case frog: return "ribbit";
  default: return "unrecognized case value";
  }
}

int main() {
  enum Animal animal = pig;
  cout << talk(animal) << endl;
}
```

Object oriented code

```
#include <string>
#include <iostream>
using namespace std;

class Animal {
public:
   virtual string talk() = 0;
};

class Rat : public Animal {
public:
   string talk() override { return "squeak"; }
};

class Pig : public Animal {
public:
   string talk() override { return "oink"; }
};

class Frog : public Animal {
public:
   string talk() override { return "ribbit"; }
};

int main() {
   Pig pig;
   Animal &animal = pig;
   cout << animal.talk() << endl;
}
```

Functional code

```
#include <iostream>
using namespace std;

int main() {
  auto rat = []() { return "squeak"; };
  auto pig = []() { return "oink"; };
  auto frog = []() { return "ribbit"; };
  auto animaltalk = pig;
  cout << animaltalk() << endl;
}
```

Generic code

```
#include <string>
#include <iostream>
using namespace std;

enum Animal {
    rat, pig, frog
};

typedef const char * cstring;

template <enum Animal animal>
struct talk  {
    static constexpr cstring value = "undefined";
};

template <>
struct talk<rat> {
    static constexpr cstring value = "squeak";
};

template <>
struct talk<pig> {
    static constexpr cstring value = "oink";
};

template <>
struct talk<frog> {
    static constexpr cstring value = "ribbit";
};

int main() {
    cout << talk<pig>::value << endl;
}
```

Object Oriented Design

- Essentials

Inheritance vs. composition

- A common base class means common behaviour to its derived types
- Public inheritance means “is-a” relationship
- Private inheritance means “is-implemented-in-terms-of”
- Composition means “has-a” or “is-implemented-in-terms-of”

Virtual or non-virtual methods?

- Use pure virtual when you want to provide an interface to the derived types
- Use virtual functions when you want to provide an interface to its derived types with default behaviour
- Use non-virtual functions when you want to provide the interface to its derived types with fixed behaviour

Templates vs. inheritance

- Does the type of the object affect the behaviour of the functions in the class?
  - If the answer is YES, then use Inheritance to model
  - If the answer is NO, then use Generics/templates to model

Object Oriented Design

- Best Practices

"Object Oriented = Objects + Classes + Inheritance!"

– Peter Wegner (https://en.wikipedia.org/wiki/Peter_Wegner)

Design Practices: Even in C!

Consider the following methods from the C library:

```
void bcopy(const void *src, void *dst, size_t n);
void *memcpy(void *dst, const void *src, size_t n);
```

What is the problem in the interfaces of these methods? Does it follow the principles of good API design?

Design Practices: Even in C!

realloc - “does too many things!”

```
#include <cstdlib>
using namespace std;

// demonstrates how malloc acts as a "complete memory manager!"
int main()
{
   int* ip = (int*) realloc(NULL, sizeof(int) * 100);
   // allocate 100 4-byte ints - equivalent to malloc(sizeof(int) * 100)

   ip = (int*) realloc(ip, sizeof(int) * 200);  // expand the memory area twice

   ip = (int*) realloc(ip, sizeof(int) * 50);   // shrink to half the memory area

   ip = (int*) realloc(ip, 0);              // equivalent to free(ip)
}
```

Design Practices: Even in C!

strtok - “stateful methods!” (not reentrant)

```
#include <stdio.h>
#include <string.h>

int main() {
  char str[] = "Mon Jul 17 17:56:41 IST 2017";
  char *token = strtok(str, ". -,;:");
  do {
    printf("%s \n", token);
  }
  while(token = strtok(0, ". -,;:"));
}
```

Intuitive class design?

```
class MyContainer {
private:
    int m_size;
public:
    void setLength(int size){ m_size = size; }
    void setSize(int size) { m_size = size; }
    int getLength() { return m_size; }
    int getSize() { return m_size; }
    // other members
}
```

Intuitive class design?

```
MyContainer * container = new Container;
container.setSize(0);
container.setLength(20);
cout << “Container's size is ” << container.getSize() <<endl;
cout << “Its length is “ container.getLength() <<endl;

// output:
//  Container's size is 20
// Its length is 20
```

Calling virtual methods from a ctor

```
struct base {
    base() {
        vfun();
    }
    virtual void vfun() {
       cout << “Inside base::vfun\n”;
    }
};

struct deri : base {
    virtual void vfun() {
       cout << “Inside deri::vfun\n”;
    }
};

int main(){
   deri d;
}

 // prints:
 //  Inside base::vfun
```

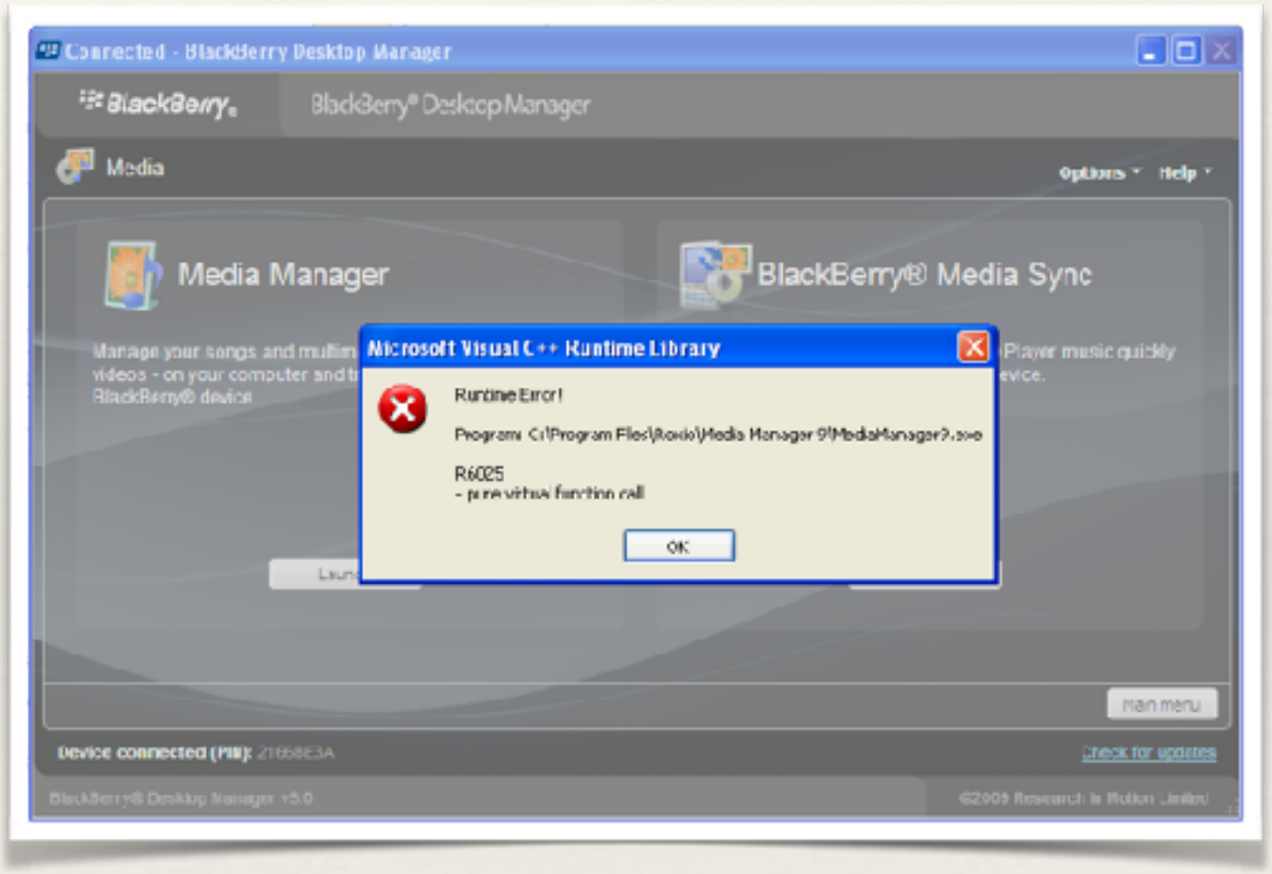
Calling virtual methods from a ctor

```
struct base {
   base() {
        base * bptr = this;
        bptr->bar();
   }
    virtual void bar() =0;
};

struct deri: base {
    void bar(){ }
};

int main() {
     deri d;
}



 // g++ output:
 //   pure virtual method called
 //   ABORT instruction (core dumped)
```

Does it happen in real-world?

- Supporting multimedia software for my Blackberry crashed with this design error!

- Classic OO design problem of "polymorphic call in constructors"
  - Constructors do not support fully as the derived objects are not constructed yet when base class constructor executes



Best Practice

"Avoid calling virtual functions from constructors"

Temporary objects & NRV

```
class String {
public:
     String(const char *str) : cstr(str) {}
     String(const String& arg) {
        std::cout<< "String cctor \n";
        this->cstr = arg.cstr;
     }
private:
     const char * cstr;
}

String function(){
     return String("Hello");
}

int main(){
     String s = function();
}
```

```
// without NRV optimization on, this prints:
//             String cctor
// with NRV optimization on, this prints:
//             String cctor
//             String cctor
```

Intuitive overloading?

```
Colour (int red, int green, int blue);
Colour (float hue, float saturation, float brightness);
```

Intuitive overloading?

```
static Colour* createRGBColour(int red, int blue, int green);
static Colour* createHSBColour(float hue, float saturation, float brightness);
```

Intuitive overloading?

```
void log(double val) {
   // prints the natural logarithm value of val
}

void log(String str) {
   // logs the string str into the log file
}
```

Intuitive overloading?

```
void log(double val) {
   // prints the natural logarithm value of val
}

void log(String str) {
   // logs the string str into the log file
}
```

```
void logarithm(double val);
void logger(String str);
// or
void log(double val);
void logger(String str);
```

Best Practice

"Avoid confusing overloads"

Preventing inheritance

```
// C++ Example
class NoInherit {
private:
     NoInherit() {
        // code for constructor
     }
public:
     static NoInherit* createInstance() {
        return new NoInherit();
     }
};

// Creation of the object
NoInherit* nip = NoInherit::createInstance();
```

Beware of versioning problems

```
class Base {  // provided by a third party tool
public:
     virtual void vfoo(){ // introduced in version 2.0
        cout<< ”vfoo in Base – introduced in a new version”;
     }
     // other members
};

class Derived : public Base {   // some client code
public:
     void vfoo(){    // was available in version 1.0
        cout<< ”vfoo in Deri – now becomes overridden”;
     }
     // other members
};
```

Follow “safe” overriding

```
#include <iostream>
#include <memory>
using namespace std;

class Base {
public:
  virtual void call(int val = 10) {
     cout << "The default value is: " << val << endl;
  }
};

class Derived : public Base {
public:
  virtual void call(int val = 20) {
     cout << "The default value is: " << val << endl;
  }
};

int main() {
  unique_ptr<Base> b = make_unique<Derived>();
  b->call();
}

                    // prints
                    //  The default value is: 10
```

Follow “safe” overriding

```
#include <iostream>
#include <memory>
using namespace std;

class Base {
public:
  void call() {
     cout << "In Base::call" << endl;
  }
};

class Derived : public Base {
public:
  virtual void call() {
     cout << "In Derived::call" << endl;
  }
};

int main() {
  unique_ptr<Base> b = make_unique<Derived>();
  b->call();
}

                    // prints
                    //  In Base::call
```

Follow “safe” overriding

```
#include <iostream>
#include <memory>
using namespace std;

class Base {
public:
  virtual void call() {
     cout << "in Base::call" << endl;
  }
};

class Derived : public Base {
public:
  virtual void call() const {
     cout << "in Derived::call" << endl;
  }
};

int main() {
  unique_ptr<Base> b = make_unique<Derived>();
  b->call();
}

                    // prints
                    //  In Base::call
```

Best Practice

"Use 'override' keyword for safe overriding"

Overloading and overriding

```
#include <iostream>
#include <memory>
using namespace std;

class Base {
public:
        virtual void call(char ch) {
                cout << "In Base::call(char)" << endl;
        }
        virtual void call(int i) {
                cout << "In Base::call(int)" << endl;
        }
};

class Derived : public Base {
public:
        virtual void call(char ch) {
                cout << "In Derived::call(char)" << endl;
        }
};

int main() {
        unique_ptr<Base> b = make_unique<Derived>();
        b->call(10);
}
```

Selectively introduce types

```
// in mymath.h
namespace math {
     class scalar { /* … */ }
     class vector { /* … */ }
     // other members
}
```

```
// in use.cpp
// for using std::vector:
#include <vector>
using namespace std;

// for using the scalar class in your math namespace:
#include “mymath.h”
using namespace math;

int main() {
     vector<scalar *> v;
     // compiler error: vector is ambiguous
     // does vector refer to std::vector or math::vector?
}
```

Selectively expose types to clients

```
namespace {
  int someMem;
  void somefunction();
};
// is a better way to limit names to file scope
//   than the following:
static int someMem;
static void somefunction();
```

Hiding?

```
int x, y;          // global variables x and y

class Point {
public:
     int x, y;         // class members x and y
     Point(int x, int y);  // function arguments x and y
};



// Point constructor for setting values of x and y data members

// C++
Point::Point(int x, int y) {
     x = x;
     y = y;
}
```

Beware of hiding

```
void foo { // outer block
     int x, y;
     {  // inner block
        int x = 10, y = 20;
        // hides the outer x and y
     }
}
```

Long parameter lists

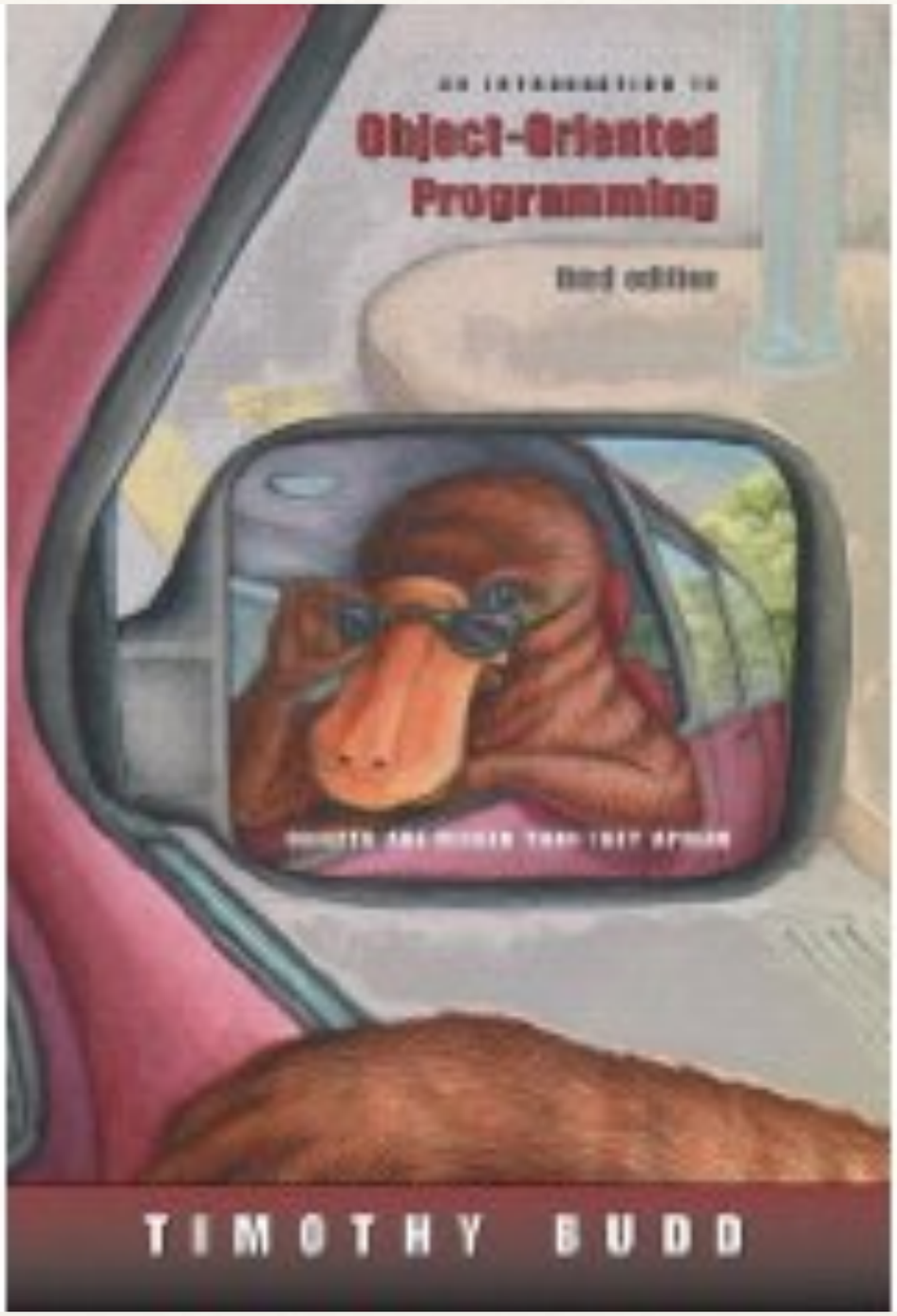
```
HWND CreateWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HANDLE hInstance,
    PVOID lpParam
);
```

Are arrays polymorphic?

```
class base {
   int basemem;
public:
   base() : basemem(10) {}
   int getint() {
      return basemem;
   }
   // other members
};

class derived : public base {
   float derivedmem;
public:
   derived() : base(), derivedmem(20.0f) {}
   // other members
};
```

```
void print(base *bPtr, int size) {
   for(int i = 0; i < size; i++, bPtr++)
      cout<< bPtr->getint() << endl;
}

int main() {
   base b[5];
   // prints five 10's correctly
   print(b, 5);
   derived d[5];
   // does not print five 10's correctly
   print(d, 5);
}
```

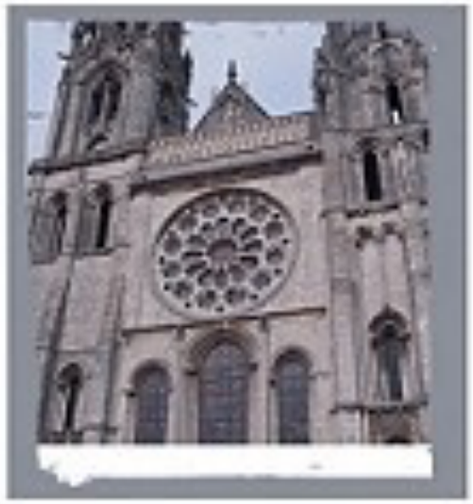
Books to read


Object-Oriented
Programming
TIMOTHY BUDD

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A System of Patterns



Volume 1

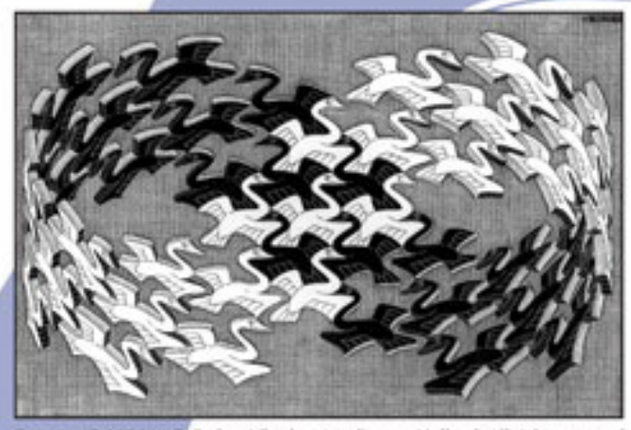
Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

WILEY

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.

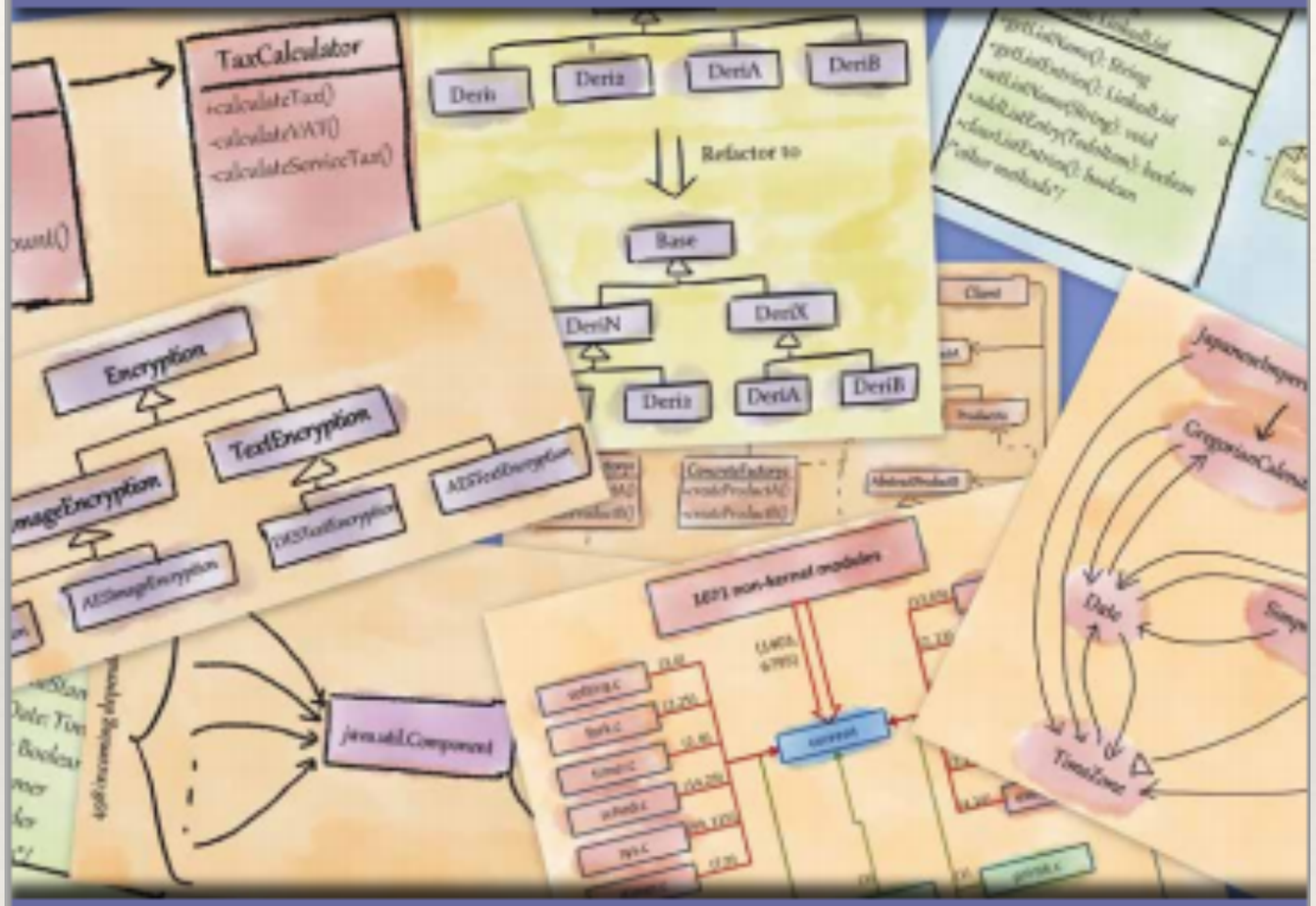
ADDISON-WESLEY OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

SERIES EDITORS

Refactoring for Software Design Smells

Managing Technical Debt



Girish Suryanarayana,
Ganesh Samarthyam, Tushar Sharma

MK
MORGAN KAUFMANN

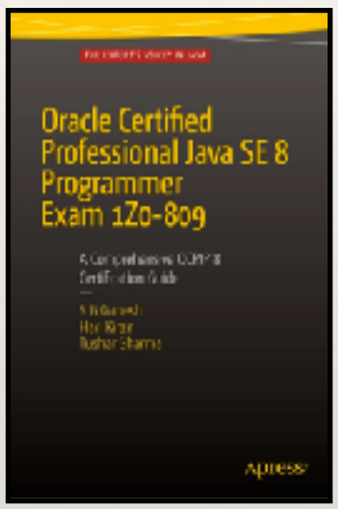
"Applying design principles is the key to creating high-quality software!"

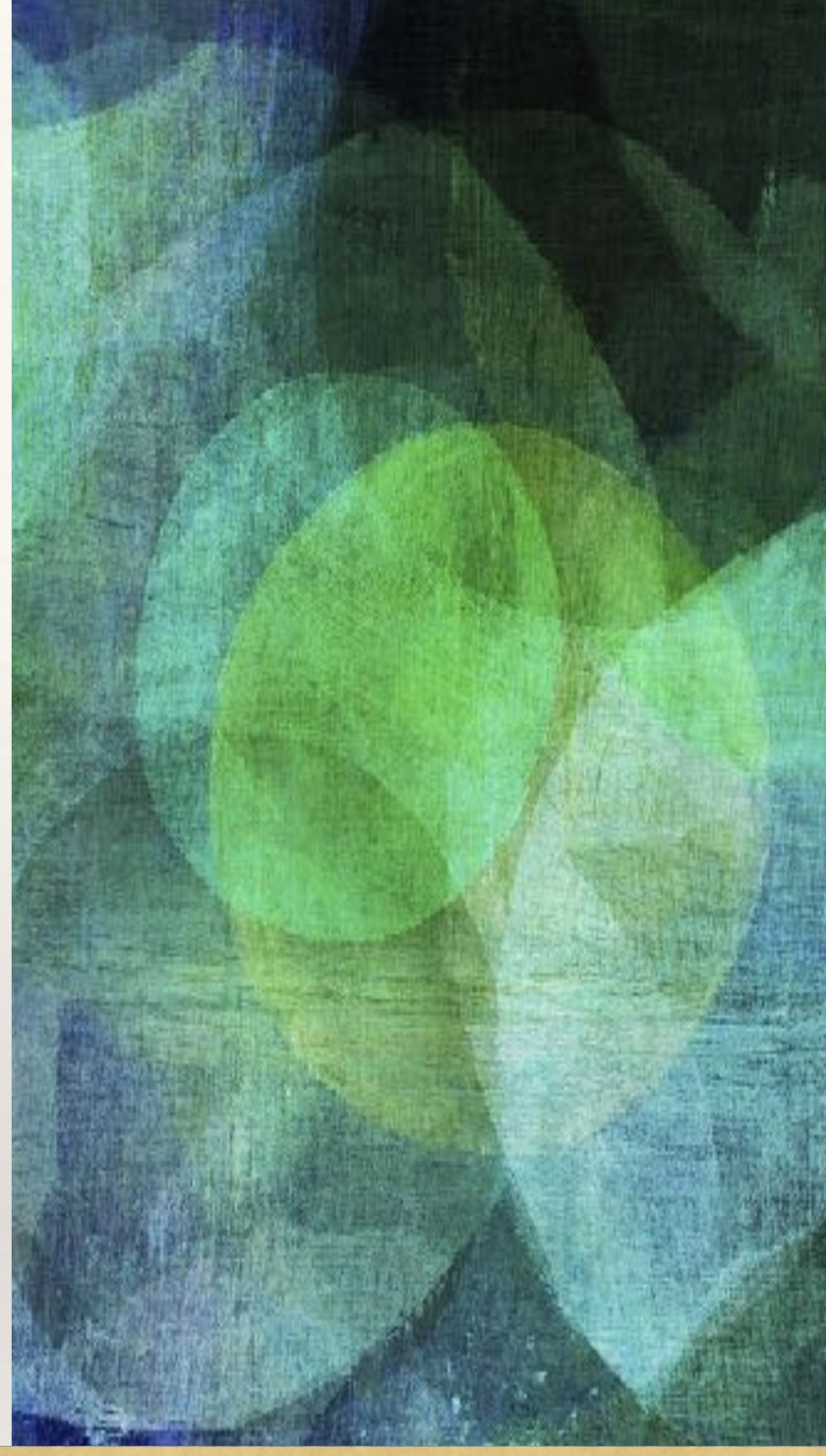
Architectural principles:
Axis, symmetry, rhythm, datum, hierarchy, transformation


MAY
THE
FORCE
BE WITH
YOU

Image/video credits

- http://en.wikipedia.org/wiki/Fear_of_missing_out
- http://lesliejanemoran.blogspot.in/2010_05_01_archive.html
- http://javra.eu/wp-content/uploads/2013/07/angry_laptop2.jpg
- https://www.youtube.com/watch?v=5R8XHrfJkeg
- http://womenworld.org/image/052013/31/113745161.jpg
- http://www.fantom-xp.com/wallpapers/33/I'm_not_sure.jpg
- https://www.flickr.com/photos/31457017@N00/453784086
- https://www.gradtouch.com/uploads/images/question3.jpg
- http://gurujohn.files.wordpress.com/2008/06/bookcover0001.jpg
- http://upload.wikimedia.org/wikipedia/commons/d/d5/Martin_Fowler_-_Swipe_Conference_2012.jpg
- http://www.codeproject.com/KB/architecture/csdespat_2/dpcs_br.gif
- http://upload.wikimedia.org/wikipedia/commons/thumb/2/28/Bertrand_Meyer_IMG_2481.jpg/440px-Bertrand_Meyer_IMG_2481.jpg
- http://takeji-soft.up.n.seesaa.net/takeji-soft/image/GOF-OOPLSA-94-Color-75.jpg?d=a0
- https://developer.apple.com/library/ios/documentation/cocoa/Conceptual/OOP_ObjC/Art/watchcalls_35.gif
- http://www.pluspack.com/files/billeder/Newsletter/25/takeaway_bag.png
- http://cdn1.tnwcdn.com/wp-content/blogs.dir/1/files/2013/03/design.jpg
- http://img01.deviantart.net/d8ab/i/2016/092/c/2/may_the_force_be_with_you___yoda_flag_by_osflag-d9xe904.jpg





ganesh@codeops.tech

www.codeops.tech

+91 98801 64463

@GSamarthyam

slideshare.net/sgganesh

bit.ly/sgganesh

CodeOps