Good Programming Practice

Juli 6, 2017 | Tobias Stockmanns

Why is this necessary?

JÜLICH FORSCHUNGSZENTRUM

When ever

- many people work
- over a long time
- on a complex software project
- with a high demand on reproducibility

it is mandatory to have at least a basic knowledge to write proper code, because

- You want to read and understand what others did (or what you did after a year not looking into the code)
- You want to find bugs easily and fix them just on one place
- You want to add additional features without rewriting the hole code again
- You want to profit from the work of others
- You want that the full project works

panda

LEVEL 1

Do not copy and paste code

JÜLICH FORSCHUNGSZENTRUM

- Most important rule!
- But one of the most broken one

- If you copy code from someone else (or even more often from yourself) you increase the places where you have to fix something if there was a bug in the first place
- If you want to add additional features you have to do it many times

- Better use common methods/functions and common base classes to reuse existing code without copying.

KISS – keep it simple, stupid

- “Keep things as simple as possible bot not simpler” - *A. Einstein*

- Most important for code is that it is understandable
- So prefer simple, short and easily understandable solutions

- “*Any fool can write code that computers can understand, good programmers write code that humans can understand.*”

Coding Conventions

- Every project with more than one developer should have a set of coding conventions to improve the readability of the code

- For PANDA they were defined 2007 and can be found at: http://panda-wiki.gsi.de/cgi-bin/view/Computing/PandaRootCodingRules

Coding Conventions

- PandaRoot follows the ROOT coding conventions
- All PandaRoot classes should start with the suffix `Pnd` !
- Include files in the C++ code will have the extension ".h".
- The implementation files in C++ will have the extension ".cxx“.
- Every include file should contain a mechanism to prevent multiple inclusions. For the file `XxxMyFile.h`, the Panda convention is:

```
#ifndef XXXMYFILE_HH
#define XXXMYFILE_HH
[....]
#endif
```

Coding Conventions

- Use a separate .cxx file, and corresponding .h file, for each C++ class. **The filename should be identical to the class name.**
- Do not create class names (and therefore filenames) that differ only by case.
- The identifier of every globally visible class, function or variable should begin with the package "TLA" (Three Letter Acronym) prefix from the package to which it belongs (e.g. mvd, emc, tpc, etc.) This implies that the implementation (.cxx) and interface (.h) files for C++ classes should also begin with the same prefix.

Coding Conventions

- Avoid overloading functions and operators unless there is a clear improvement in the clarity of the resulting code. For read and write access to data members, use:

```
int  GetMyData( ) const;
void SetMyData( const int value );
```

rather than

```
int MyData( ) const;
void MyData( const int value );
```

In fact, using `SetMyData` is a strict rule. Please use it. `GetMyData` is not a strict rule, but strongly encouraged.

Coding Conventions

- Members of a class should start with an `f` at the beginning:
  - Examples: `fX, fData, …`
- Use the root types insteadt of C++ types:
  - Int_t, Double_t, Bool_t
- Compare the same data types with each other:
  - Unsigned with unsigned
  - Int_t with Int_t and not with Double_t

Coding Conventions

- Don't implicitly compare pointers to nonzero (i.e. do not treat them as having a boolean value). Use

```
if ( 0 != ptr ) ...
```

instead of

```
if ( ptr ) ...
```

If you are doing an assignment in a comparison expression, make the comparison explicit:

```
while ( 0 != (ptr=iterator() ) ) ...
```

instead of

```
while ( ptr=iterator() ) ...
```

Coding Conventions

- format of Comments

When using C++, the preferred form for short comments is:

```
// This is a one-line comment.
```

i.e. use the "//" comment format. If the comment extends over multiple lines, each line must have // at the beginning:

```
// This is a long and boring comment.
// I need to put // at the start of each line.
// Note that the comment starts at the // and
// extends to the end of line. These comments
// can therefore appear on the same line as code,
// following on from the code.
```

Do not use "/* */" comments because they are very error prone

Documentation – self-documenting

- The best code is a self-documenting code
  - Keep it short
    - *No method longer than a page*
    - *No class more complex than necessary*
  - Keep it structured
    - *Not more than one statement ended by a “ ; ” in each line*
    - *Indent your blocks*

```
int Foo(bool isBar)
{
  if (isFoo) {
    bar();
    return 1;
  } else {
    return 0;
  }
}
```

Documentation – self-documenting

- Use speaking variable/method/function/class names
  - *Use English!*
  - *As longer the scope of a variable is as more detailed should be its name*
  - *If the scope is short the name should be short*
  - *Do not use abbreviations*
  - *Use CamelCasing*
  - *Methods with boolean return type should start with an* `Is…`, e.g. `IsFound()`, `IsEmpty()`, … *or in appropriate cases with* `Has…`, `Can…`
  - *Search methods should start with a* `Find…`, e.g. `FindTimeStamp()`
  - *Use enums for type-identification*

Documentation

- Documentation should be in the .h as well as in the .cxx file
- In the .h file the interface is described:
  - What is the method doing?
  - What is the meaning of the parameters (units!)?
  - What is the meaning of the return value?
- In the .cxx file it should be explained how something is done
- Use doxygen for documentation!
  http://www.stack.nl/~dimitri/doxygen/

Use the SVN/GIT

- SVN/GIT allows you to do changes on your code without harming the code of others and your own code
- Therefore the development branch / your fork exists
- A stable version of pandaRoot is moved into your own development branch / fork
- Here you can do your code changes without interfering with the work of others
- Once your work is finished and tested you can merge it back into the trunk / make a pull request
- Documentation can be found at the PandaComputing wiki pages: http://panda-wiki.gsi.de/cgi-bin/view/Computing/PandaRootSvnDev2Trunk

Use an appropriate development suite

- Modern development suits help you in your day-by-day coding work a lot
- They offer code completion, automatic formatting, checking for syntax failures and many more
- There are many free on the market:
  - KDevelop
  - QDevelop
  - eclipse
  - …

- It does not matter which you use but use one
- I am using eclipse including SVN and doxygen.
  How to use svn and cmake within eclipse can be found here

How to write decent classes

LEVEL 2

SOLID - Principle

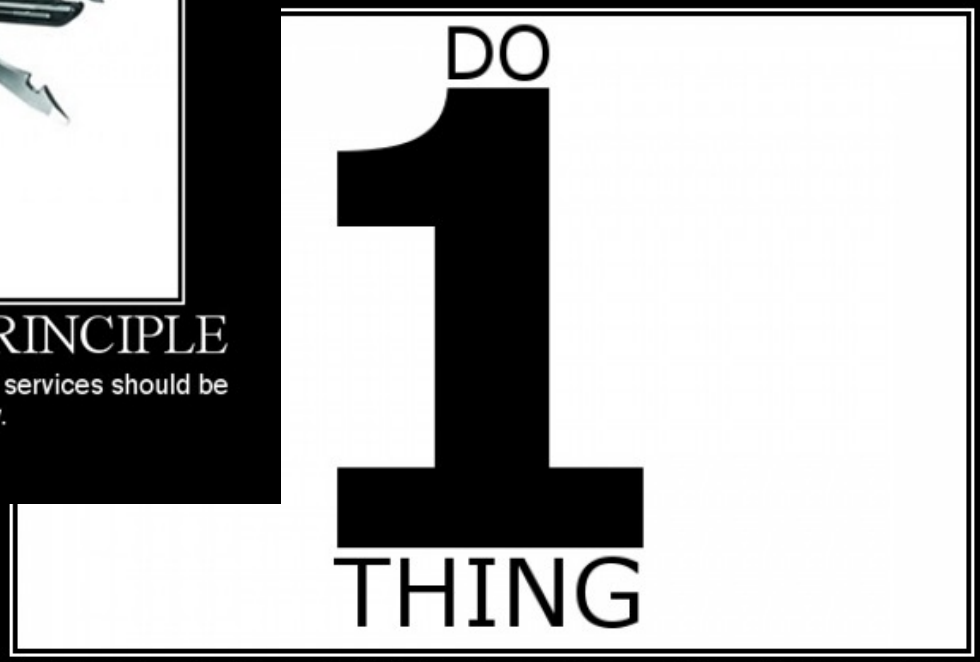
- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

For more information see http://www.clean-code-developer.de/

http://www.codeproject.com/Articles/93369/
How-I-explained-OOD-to-my-wife

SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.



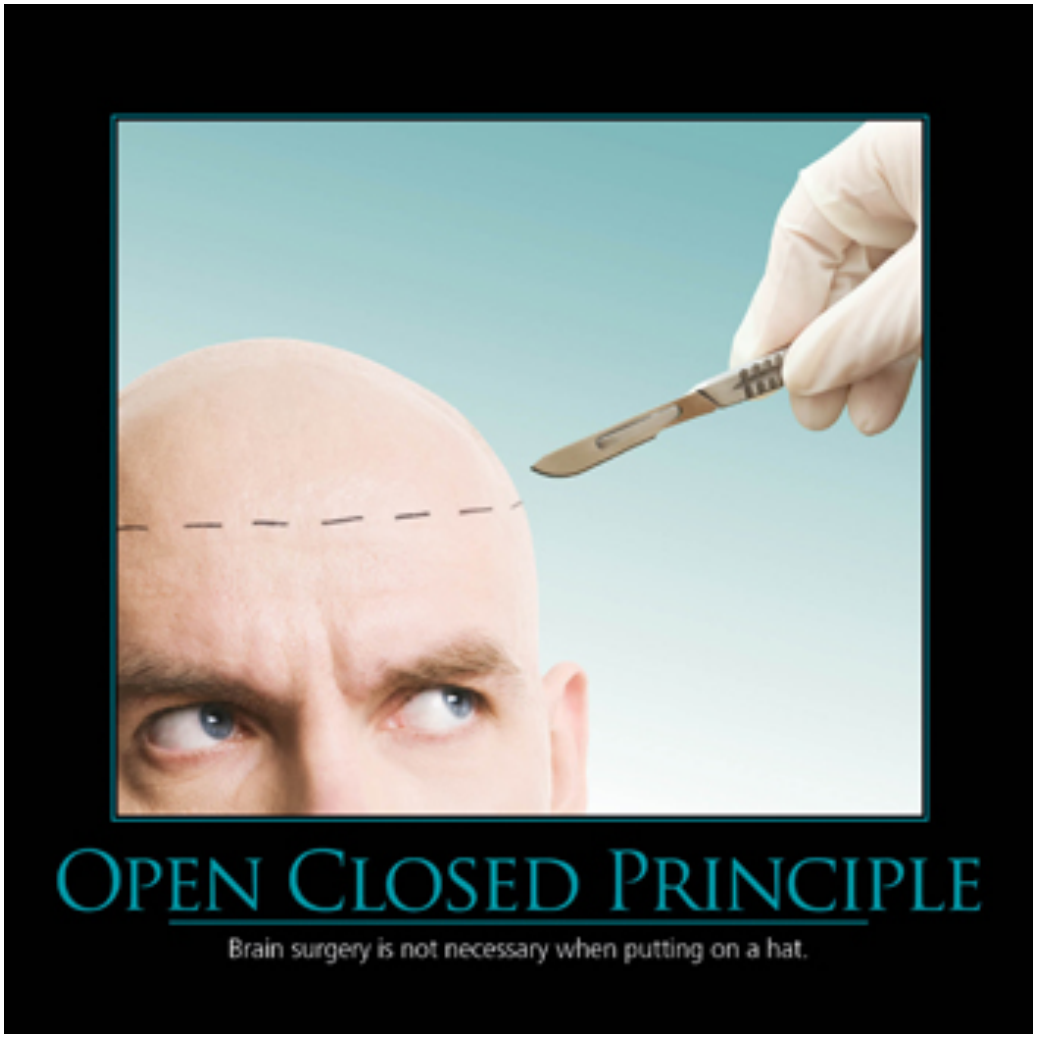
SINGLE RESPONSIBILITY PRINCIPLE (SRP)

A class should have only one reason to change.

Single Responsibility Principle

JÜLICH FORSCHUNGSZENTRUM

- A class should only have one single responsibility
- The responsibility should be entirely encapsulated by the class
- There should be no more than one reason to change a class
  - Reduces the number of dependencies
  - Keeps classes slim
  - Better to understand


OPEN CLOSED PRINCIPLE
Brain surgery is not necessary when putting on a hat.

Open Closed Principle

- A class should be open for extension, but closed for modification
  - Make it easy to add new functionality and different algorithms to your classes
  - Protect the existing ones against any external modifications

Example for bad code

```
enum ShapeType {circle, square};

struct Shape
{
  ShapeType itsType;
};

struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};


struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

//
// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);
```

```
typedef struct Shape *ShapePoint

void DrawAllShapes(ShapePointer
{
  int i;
  for (i=0; i<n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
    case square:
      DrawSquare((struct Square*
    break;

    case circle:
      DrawCircle((struct Circle*
    break;
    }
  }
}
```

How to do it better

```
class Shape
{
  public:
    virtual void Draw() const = 0;
};

class Square : public Shape
{
  public:
    virtual void Draw() const;
};

class Circle : public Shape
{
  public:
    virtual void Draw() const;
};

void DrawAllShapes(Set<Shape*>& list)
{
  for (Iterator<Shape*>i(list); i; i++)
     (*i)->Draw();
}
```

Open / close principle continued

- Make all data variables private
- Access them via Set/Get methods
- Only implement Set if it is really necessary
- Do not use global variables
- Hide as much as you can from the user

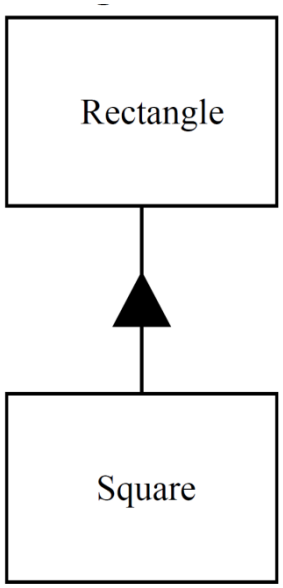

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
  - Protect your code from unwanted behavior
  - Rule to decide if a class can be a subclass of another one

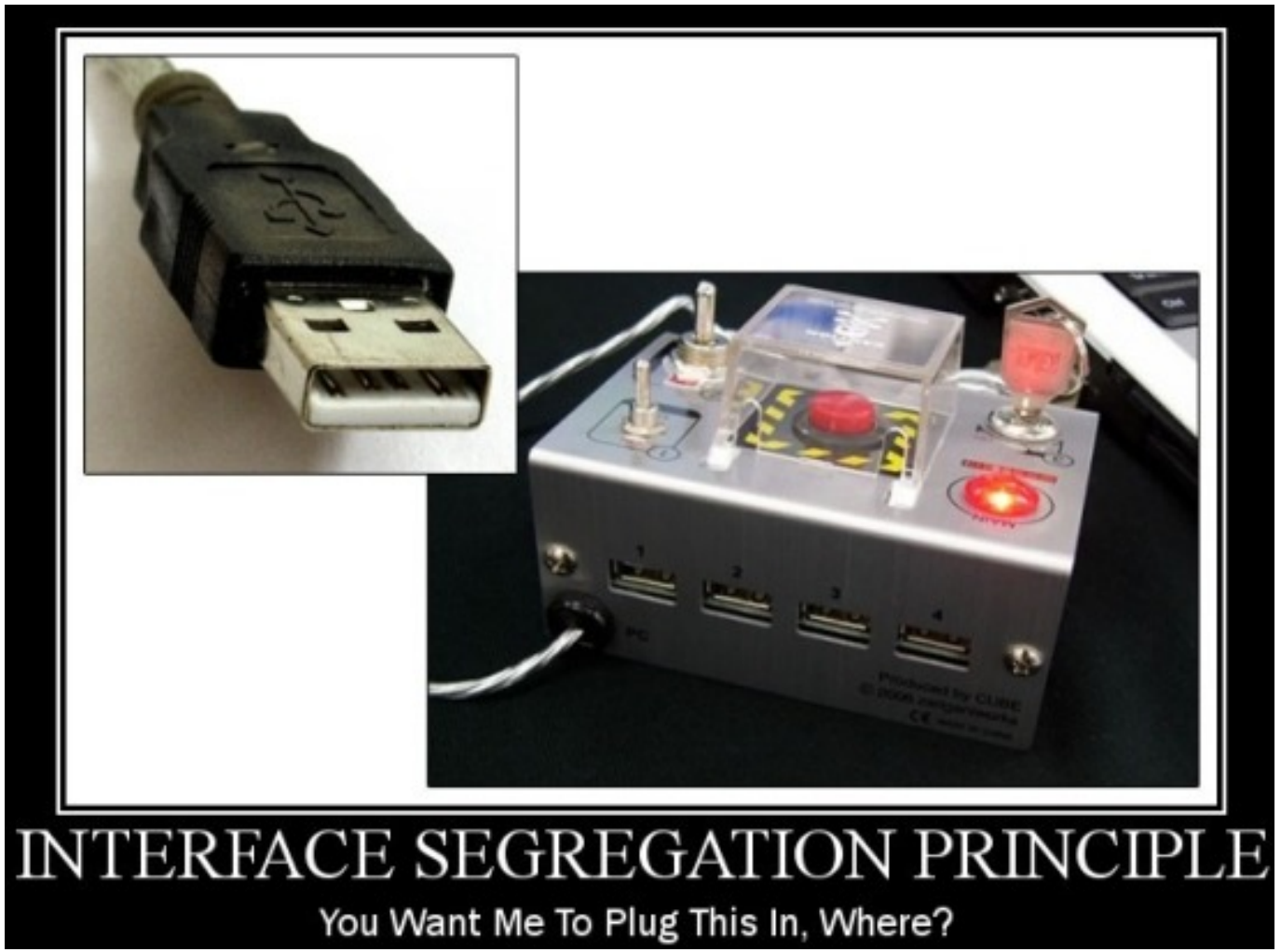
LSP - Example

```
class Rectangle
{
  public:
    virtual void SetWidth(double w)  {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
    double GetHeight() const    {return itsHeight;}
    double GetWidth() const     {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```



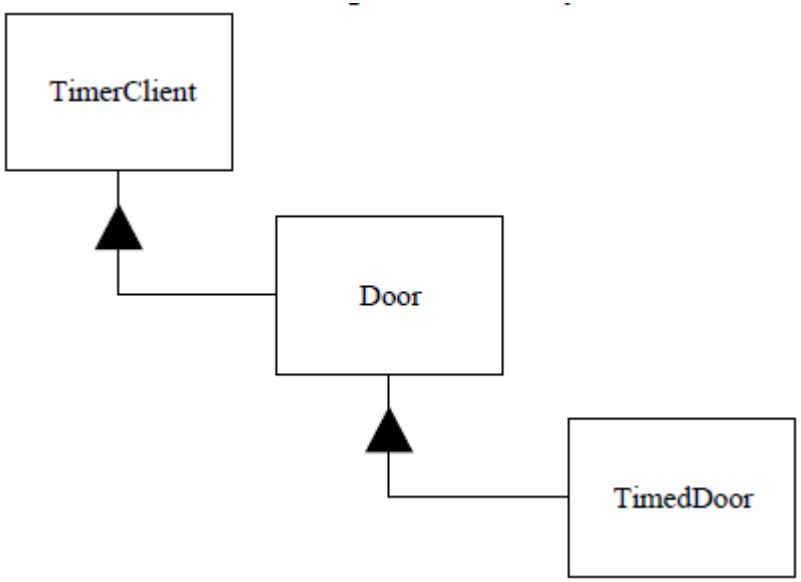
What is the problem?

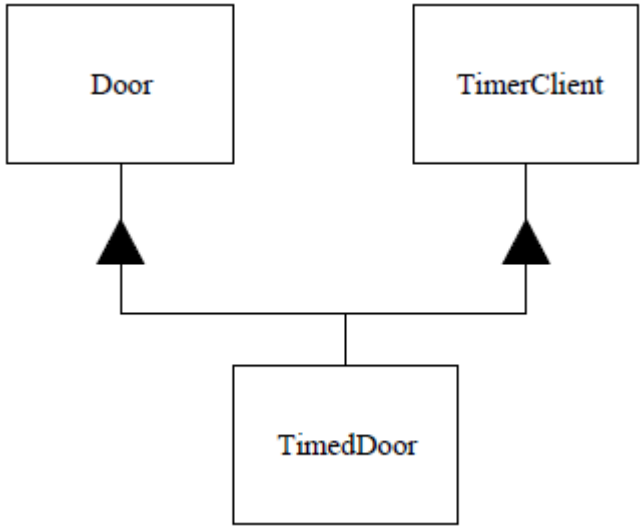
```
void f(Rectangle& r)
{
  r.SetWidth(32); // calls Rectangle::SetWidth
}


void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

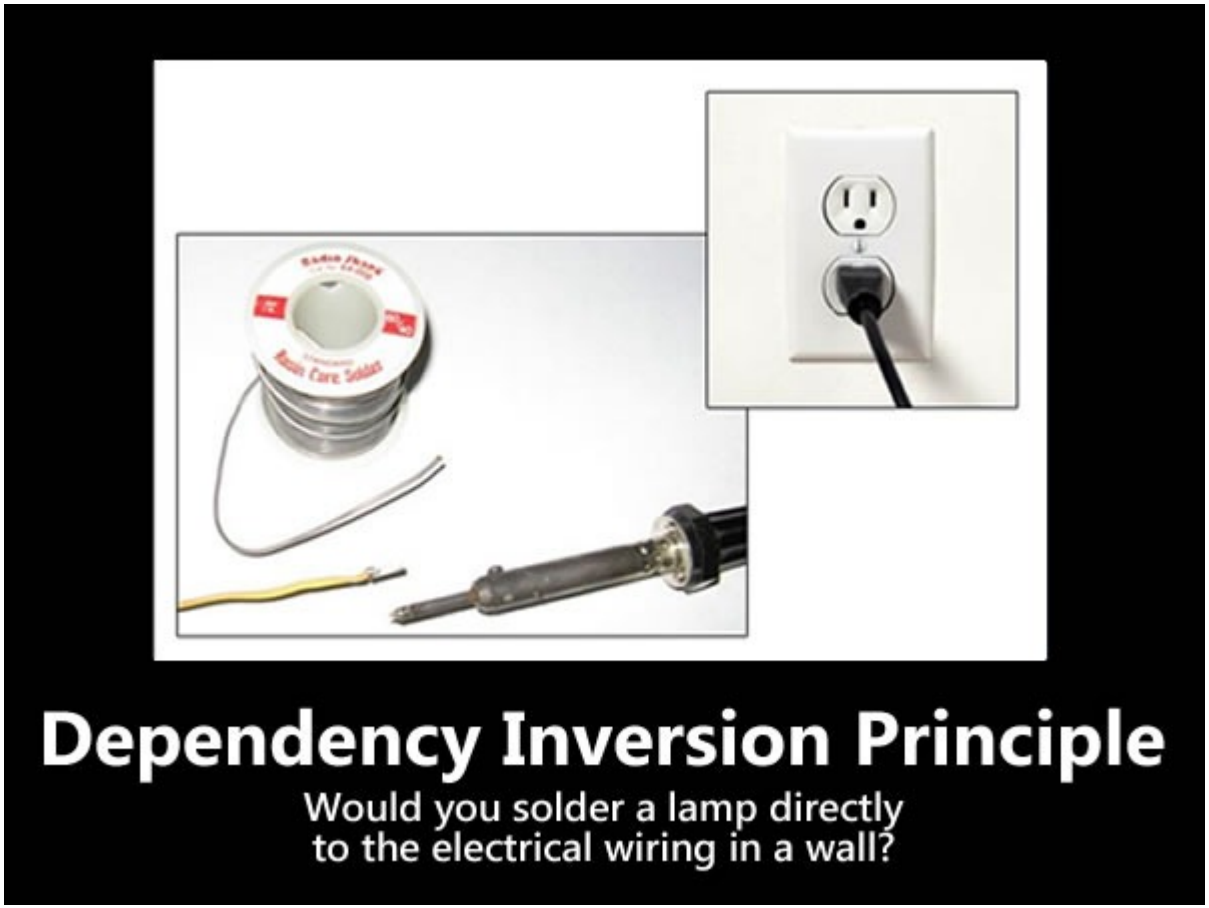

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle

- Many client specific interfaces are better than one general purpose interface
- Clients should not be forced to depend upon interfaces that they do not use





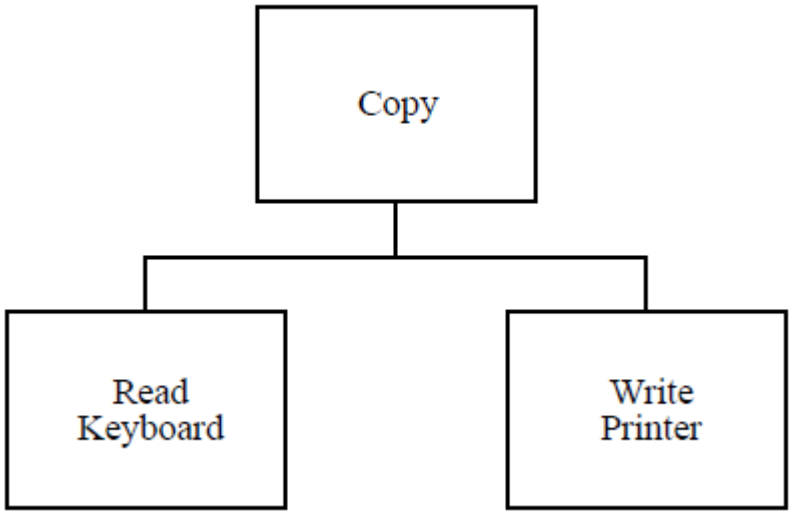

Dependency Inversion Principle
Would you solder a lamp directly
to the electrical wiring in a wall?

Dependency Inversion Principle

- A. High-level modules should not depend on low level modules. Both should depend on abstractions.
- B. Abstractions should not depend upon details. Details should depend upon abstractions

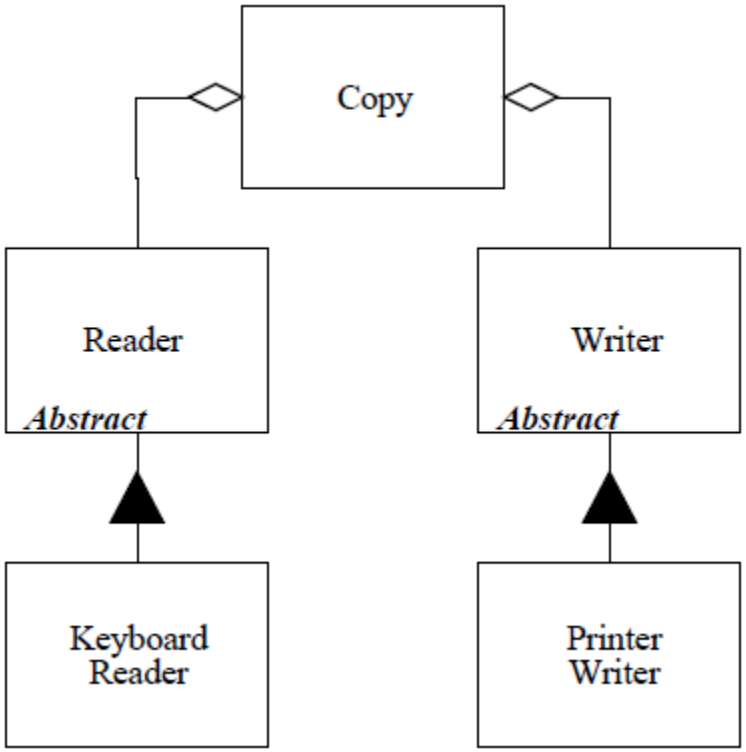
Example for bad code

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```




```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

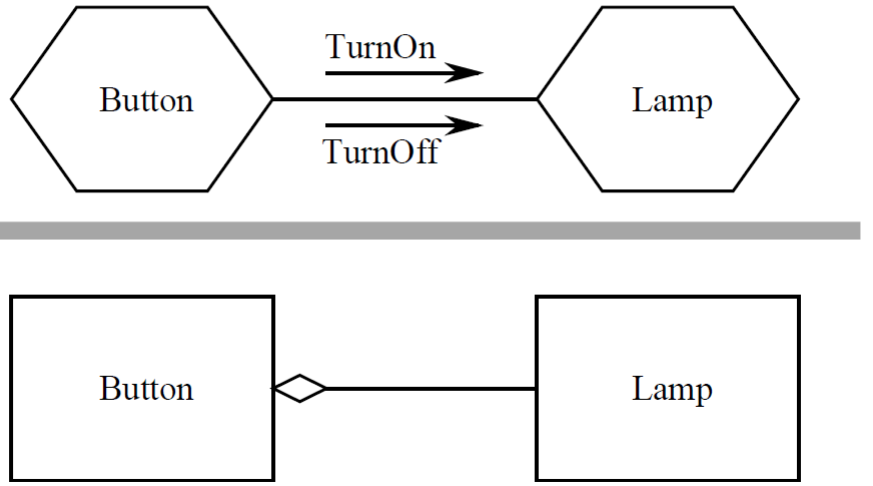
How to do it better




```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

Another example – BAD CODE

Figure 5: Naive Button/Lamp Model

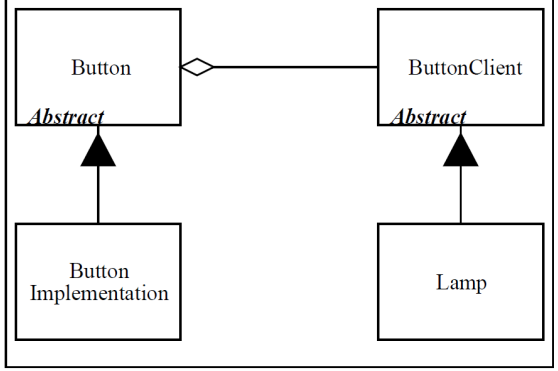


Listing 5: Naive Button/Lamp Code

```
--------------lamp.h----------------
class Lamp
{
  public:
    void TurnOn();
    void TurnOff();
};
-------------button.h---------------
class Lamp;
class Button
{
  public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
  private:
    Lamp* itsLamp;
};
-------------button.cc--------------
#include "button.h"
#include "lamp.h"

void Button::Detect()
{
  bool buttonOn = GetPhysicalState();
  if (buttonOn)
    itsLamp->TurnOn();
  else
    itsLamp->TurnOff();
}
```

Another example – GOOD CODE

Figure 6: Inverted Button Model



Listing 6: Inverted Button Model

```
----------byttonClient.h---------
class ButtonClient
{
  public:
    virtual void TurnOn() = 0;
    virtual void TurnOff() = 0;
};
-----------button.h---------------
class ButtonClient;
class Button
{
  public:
    Button(ButtonClient&);
    void Detect();
    virtual bool GetState() = 0;
  private:
    ButtonClient* itsClient;
};
---------button.cc----------------
#include button.h
#include buttonClient.h

Button::Button(ButtonClient& bc)
: itsClient(&bc) {}

void Button::Detect()
{
  bool buttonOn = GetState();
  if (buttonOn)
    itsClient->TurnOn();
  else
    itsClient->TurnOff();
}
-----------lamp.h-----------------
class Lamp : public ButtonClient
{
  public:
    virtual void TurnOn();
    virtual void TurnOff();
};
---------buttonImp.h-------------
class ButtonImplementation
: public Button
{
  public:
    ButtonImplementaton(
      ButtonClient&);
    virtual bool GetState();
};
```

General statements

- Use std::cout instead of printf
- Do not use C arrays like: double myVal[10]
  The standard library offers a long list of container classes (vector, map, set, list, …) which are more powerful and more safe
- As an alternative there are root classes like TClonesArray
- Do not fear to refactorize (clean up) your code. SVN prevents you from braking something.
- If you use pointers in your classes make sure you clean them up at the end
- If you use pointers in your classes write an appropriate copy constructor and assignment operator