# 3 OO Class Design Principles

# 3.1 Dependency Management

- The parts of a project depend on each other
  - components, programs, groups of classes, libraries
- Dependencies limit
  - flexibility
  - ease of maintainance
  - reuse of components or parts
- Dependency management tries to control dependencies

# 3.1 Dependency Management and Software

- Software systems are the most complex artificial systems
- There will be a lot of dependencies
- Software development was and is always concerned with dependencies
- OOAD gives us tools to manage dependencies
  - trace dependencies e.g. in UML models
  - use OO language to manipulate dependencies

# 3.1 Problems with Software

- Rigid
- Fragile
- Not Reuseable
- High Viscosity
- Useless Complexity
- Repetition
- Opacity

These statements apply to an average physicist/programmer who develops and/or maintains some software system.

Software gurus will always find some solution in their code.

Do you want to rely on the guru?

What if that person retires, finds a well-paid job or gets moved to another project?

# 3.1 Rigid Software

- Difficulties with changes
  - Unforeseen sideeffects occur frequently
  - Hard to estimate time to complete modifications
- "Roach Motel"
  - Always in need of more effort
- Management reluctant to allow changes
  - Official rigidity "don't touch a working system"
  - Users forced to develop workarounds

# 3.1 Fragile Software

- Small changes have large side effects
  - New bugs appear regularily
  - In the limit of P(bug | change) = 1 system is impossible to maintain
- It looks like control has been lost
  - Users become critical
  - Program looses credibility
  - Developers loose credibility

# 3.1 Not Reuseable

- You have a problem and find some piece of code which might solve it
  - but it brings in a lot of other stuff
  - it needs changes here and there
- Eventually you have two choices
  - Take over maintainance of the branched code
  - Roll your own
- You would like to include headers and link a library maintained by somebody else

# 3.1 High Viscosity

- Viscosity of the design
  - Hard to make changes properly, i.e. without breaking the design → make hacks instead
- Viscosity of the environment
  - Slow and inefficient development environment
  - Large incentive to keep changes localised even if they break designs
  - Design changes are very difficult

# 3.1 Useless Complexity

- Design/code contains useless elements
- Often for anticipated changes or extension
  - May pay off
  - Meanwhile makes design/code harder to understand
- Or leftovers of previous design changes?
  - Time for a clean-up
- Tradeoff between complexity now and anticipated changes later

# 3.1 Repetition

- Added functionality using cut-and-paste
  - Then slight modifications for local purpose
- Find same structure repeatedly
  - More code
  - Harder to debug and modify
- There is an abstraction somewhere
  - Refactor into function/method
  - Create class(es) to do the job

# 3.1 Opacity

- Design/code difficult to understand
  - We have all suffered ...
  - What is clear now may seem strange later
- Ok when its your code
  - You suffer in silence
- Not acceptable in collaboration
  - Need to code clearly, may need to rearrange
  - Code reviews?

# 3.1 Dependencies Managed

- Code is less rigid
- Code is less fragile
- Reuse is possible
- Viscosity is low

# 3.1 Less Rigid Code

- Modules can be interchanged
- Changes are confined to a few modules
- Cost of changes can be estimated
- Changes can be planned and scheduled
- Management is possible

# 3.1 Less Fragile Code

- Confined changes: $P(\text{bug} \mid \text{change}) \ll 1$
- New bugs will most likely appear where the changes was made, i.e. localised
  - Easier to fix (hopefully)
- Risk of changes can be estimated
- Credibility of code and developers conserved

# 3.1 Reuseable Code

- A module can be used in a different context without changes
  - Just use headers and link a library
- No need to compile and/or link lots of unrelated stuff

# 3.1 Low Viscosity

- Design is easy to modify
  - No quick hacks needed
  - Proper design improvements will actually happen
- Large scale changes affecting many modules are possible
  - Reasonable compile and link times for the whole system
  - May depend on adequate hardware as well

# 3.1 Compile and Link Times

- Compile and link times are unproductive
- In a project with N modules compile and link time can grow like $N^2$ (assuming every module is tested) when dependencies are not controlled
- Loss of productivity
- Long turnaround times → slow development
- Dependency management essential in large projects

# 3.1 Code Changes

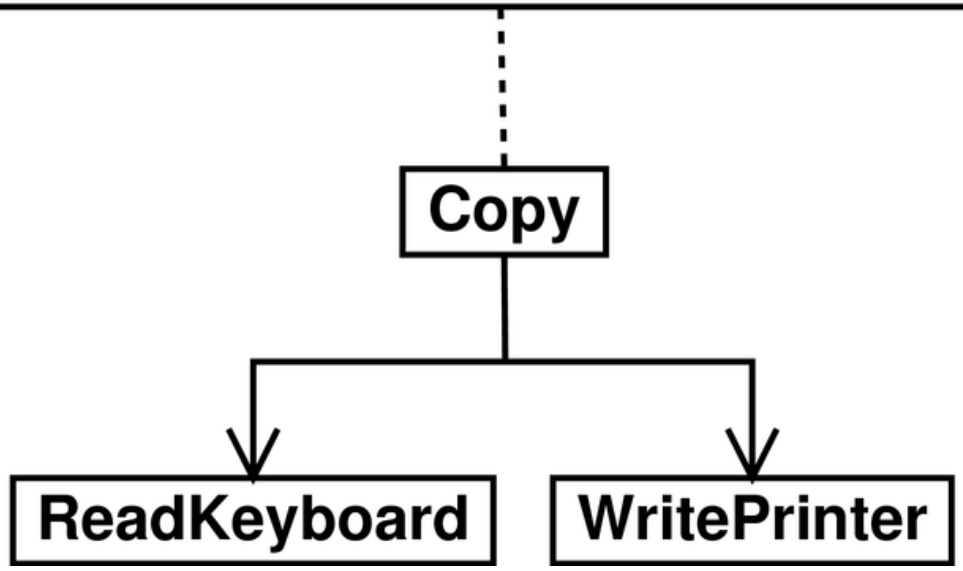
- Modules/packages and makefiles
  - Verify that makefiles are reliable
- Changes to libraries (reuseable code)
  - All affected users must relink (and retest)
- Shared libraries
  - Need to distribute (and restart programs)
  - Validation by users still needed
  - Need recompile after interface changes

# 3.2 The Copy Routine

- Code rots
- There are many reasons for code rot
- We'll make a case study (R. Martin)
- A routine which reads the keyboard and writes to a printer

# 3.2 Copy Version 1

```
void Copy(void) {
  char ch;
  while( (ch= ReadKeyboard()) != EOF ) {
    WritePrinter( ch );
  }
}
```



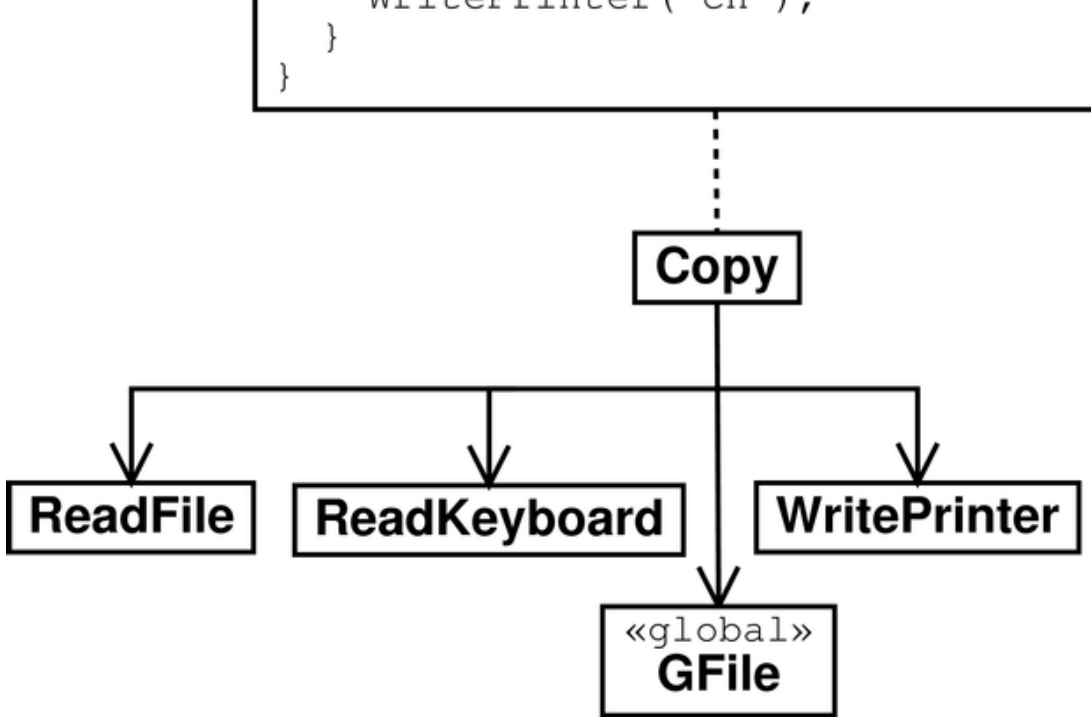

A simple solution
to a simple problem

ReadKeyboard and
WritePrinter are probably
reuseable

# 3.2 Copy Version 2

```
bool GFile;

void Copy(void) {
  char ch= 0;
  while( ch != EOF ) {
    if( GFile ) { ch= ReadFile(); }
    else { ch= ReadKeyboard(); }
    WritePrinter( ch );
  }
}
```




Many users want to read files too ...
But they don't want to change their code ... can't put a flag in the call

Ok, so we use a global flag

Its backwards compatible
To read files you have to set the flag first

# 3.2 Copy Version 3

Oh dear, we introduced a bug in version 2 (printing EOF isn't nice)

Version 3 fixes this bug

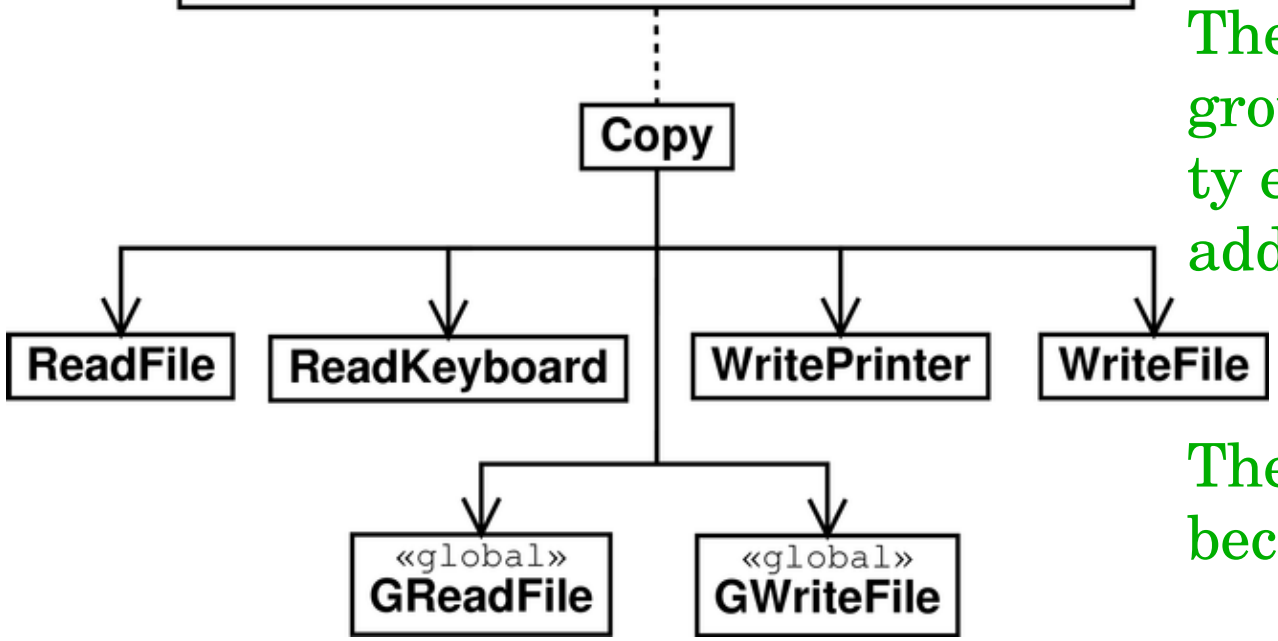
```
bool GFile;

void Copy(void) {
  char ch;
  while( 1 ) {
    if( GFile ) { ch= ReadFile(); }
    else { ch= ReadKeyboard(); }
    if( ch == EOF ) break;
    WritePrinter( ch );
  }
}
```

# 3.2 Copy Version 4

```
bool GReadFile;
bool GWriteFile;

void Copy(void) {
  char ch;
  while( 1 ) {
    if( GReadFile ) { ch= ReadFile(); }
    else { ch= ReadKeyboard(); }
    if( ch == EOF ) break;
    if( GWriteFile ) { WriteFile( ch ); }
    else { WritePrinter( ch ); }
  }
}
```




Users want to write to files, of course they want it backwards compatible

We know how to do that!

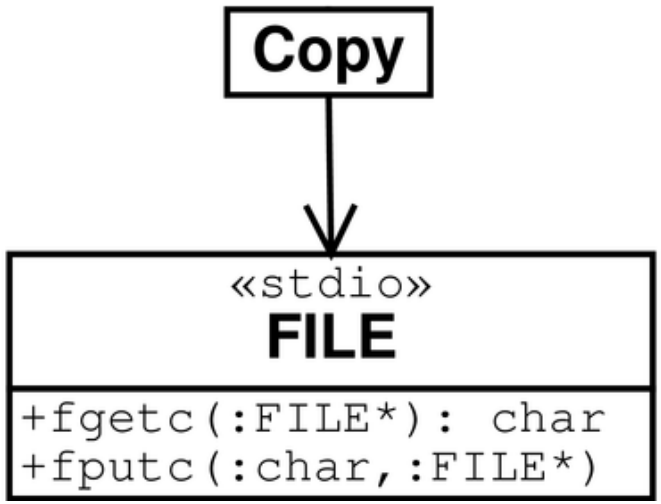
The Copy routine seems to grow in size and complexity every time a feature is added

The protocol to use it becomes more complicated

# 3.2 Copy done properly in C

```
#include <stdio.h>

void Copy( FILE* in, FILE* out ) {
  char ch;
  while( (ch= fgetc( in )) != EOF ) {
    fputc( ch, out );
  }
}
```



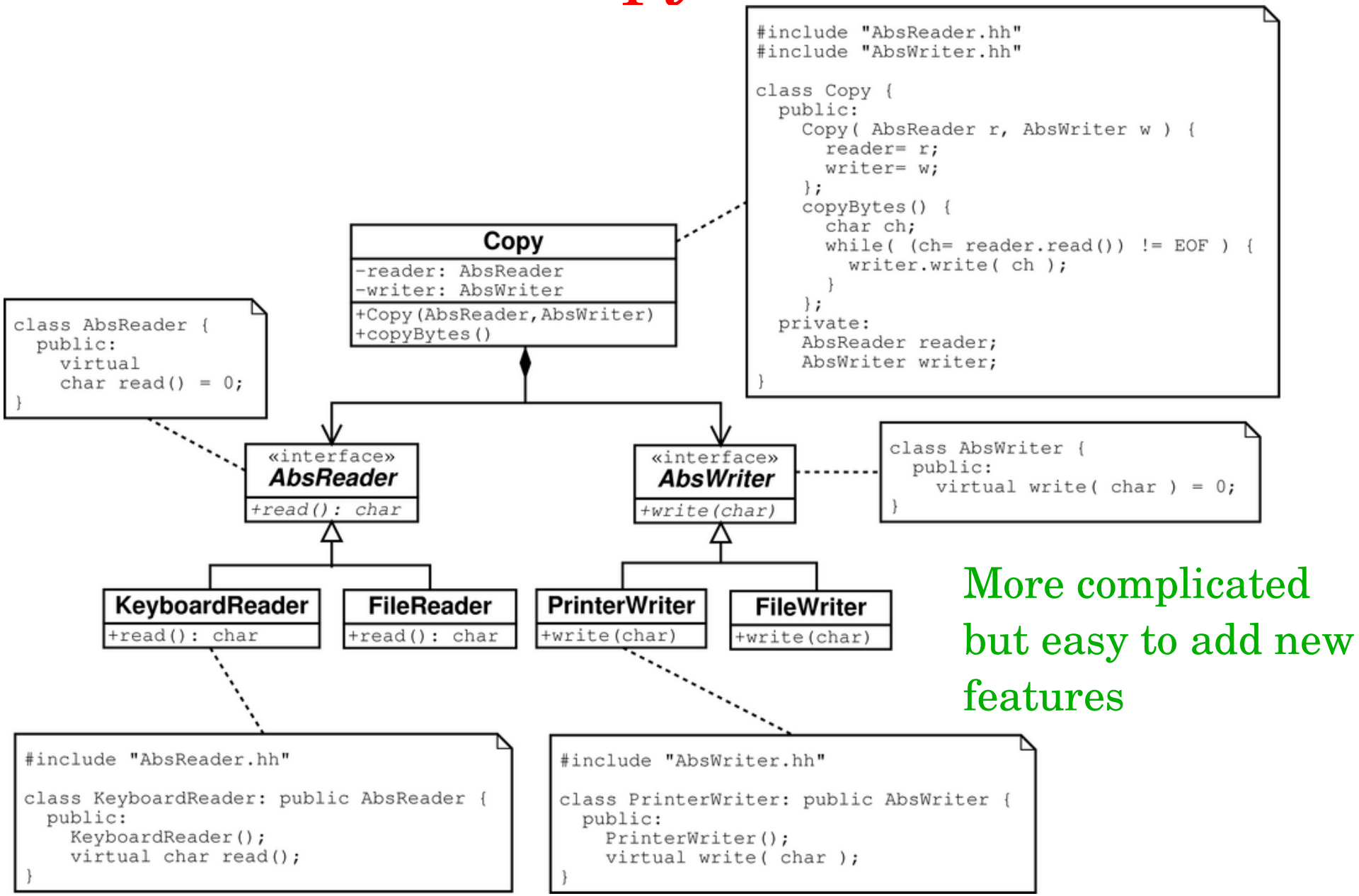

Finally a good C programmer comes to the rescue!

But this is C?!

FILE, fgetc and fputc behave like an interface class

FILE represents a generic byte stream manipulated by fgetc, fputc etc.

# 3.2 Copy in C++




More complicated but easy to add new features

# 3.2 Copy Routine Summary

- Lack of sensible design leads to code rot
  - Useless complexity, repetition, opacity
- Software systems are dynamic
  - New requirements, new hardware
- A good design makes the system flexible and allows easy extensions
  - Abstractions and interfaces
- An OO design may be more complex but it builds in the ability to make changes

# 3.2 Dependency Management Summary

- Controlling dependencies has several advantages for software system
  - Not rigid, not fragile, reuseable, low viscosity
- Also affects development environment
  - Lower compile and link times, less testing
  - More productive work
- Plan for changes and maintainance