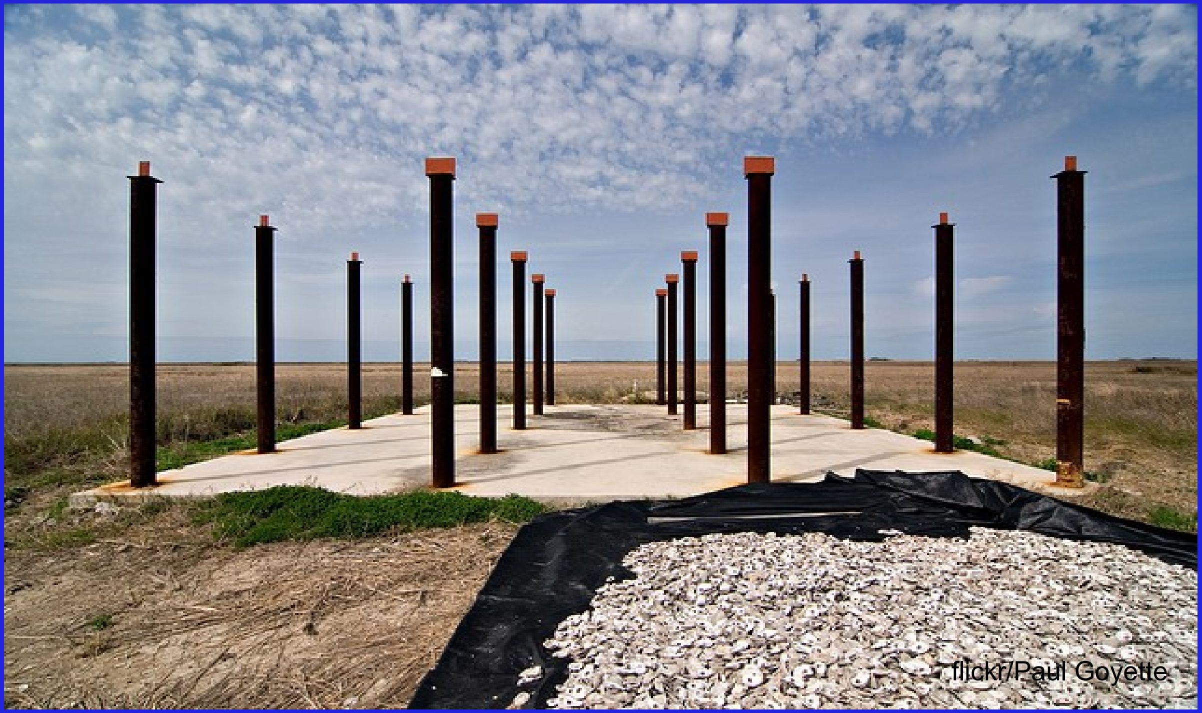
C++ Foundation


flickr/Paul Goyette

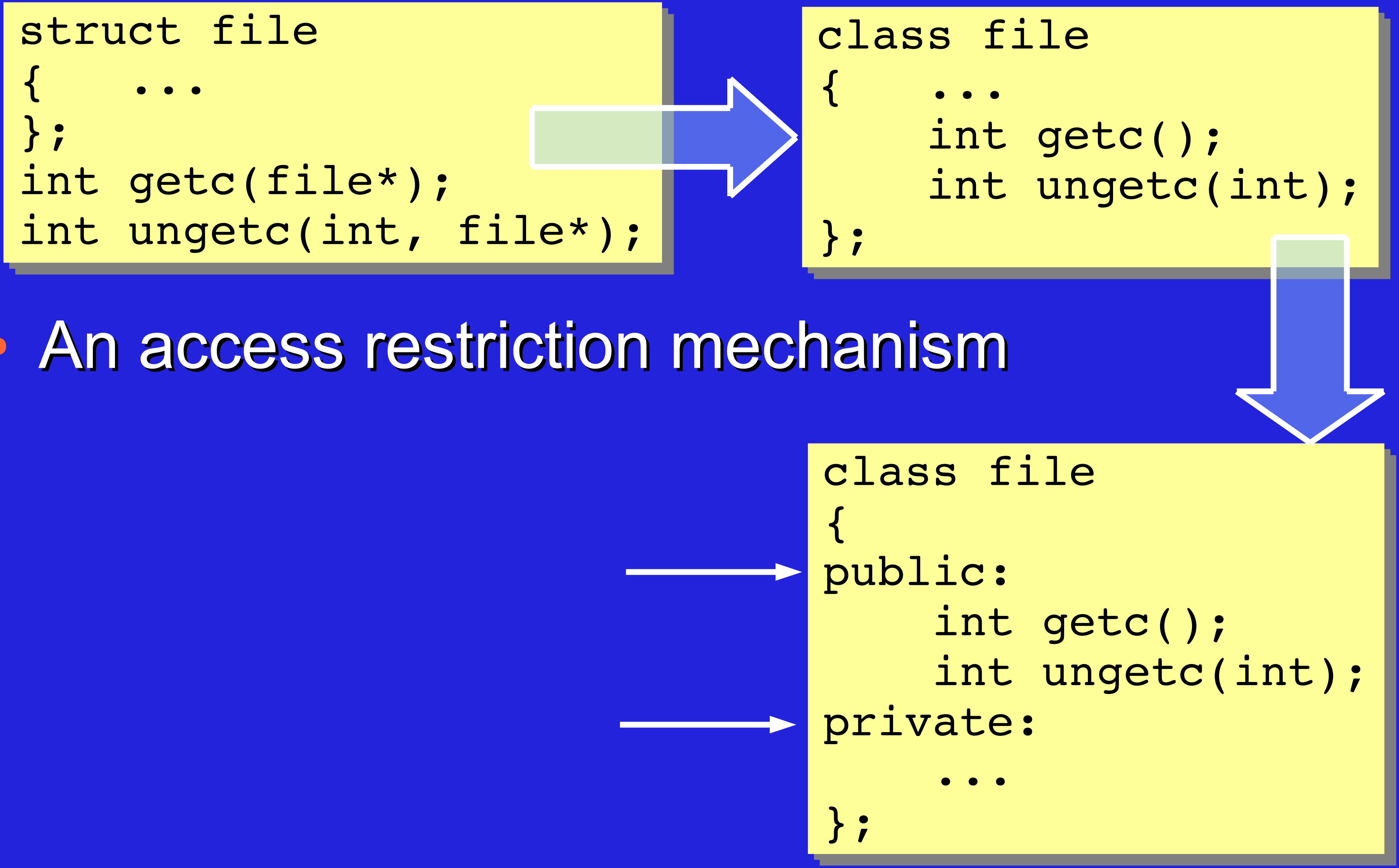
Object Oriented Programming

Object Oriented Programming

- encapsulation
- information hiding
- liskov substitution principle
- parameterize from above
- single repsonsibility principle
- separation of concerns
- dependency
- abstraction
- patterns

En(capsule)ation

- Data and functions can be bundled together



```
struct file
{   ...
};
int getc(file*);
int ungetc(int, file*);
```

```
class file
{   ...
    int getc();
    int ungetc(int);
};
```

- An access restriction mechanism

```
class file
{
public:
    int getc();
    int ungetc(int);
private:
    ...
};
```

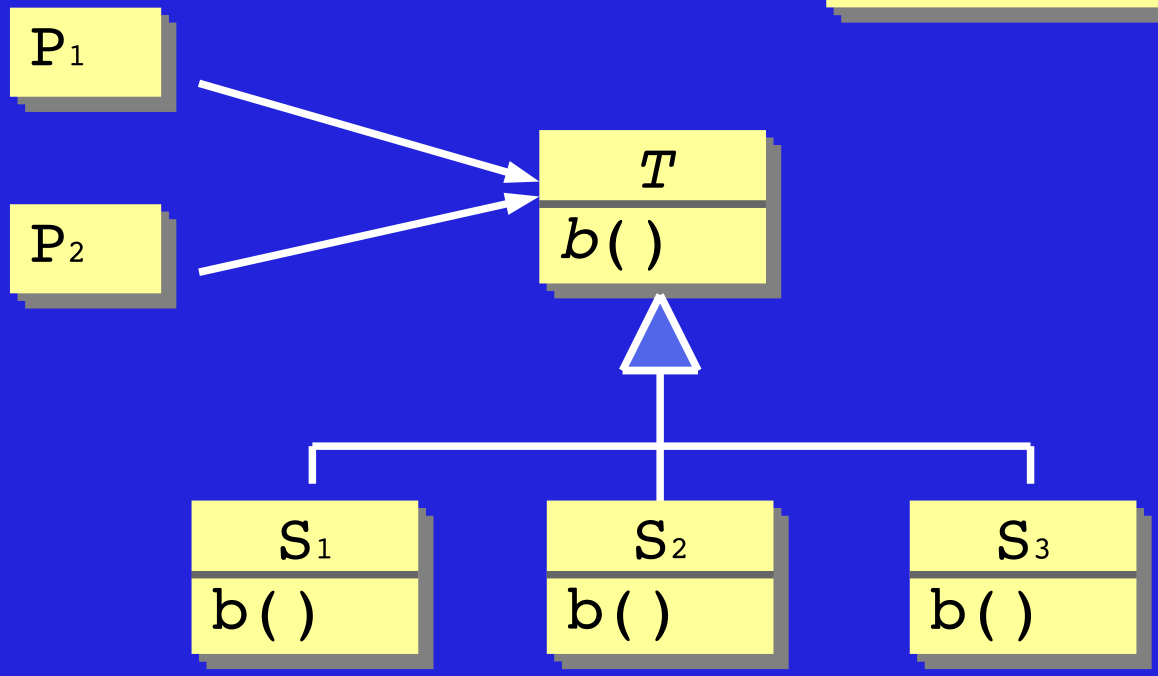
Information Hiding in C++

- We hide information partly so we can change what's hidden and limit the change's impact
  - public - private
    - change requires recompilation
  - header file - source file
    - change of implementation requires relinking
  - opaque types
    - change of representation requires relinking
  - inheritance hierarchies
    - change of type does not require relinking!

Liskov Subtitution Principle

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov



P_1

P_2

T

b( )

S_1

b( )

S_2

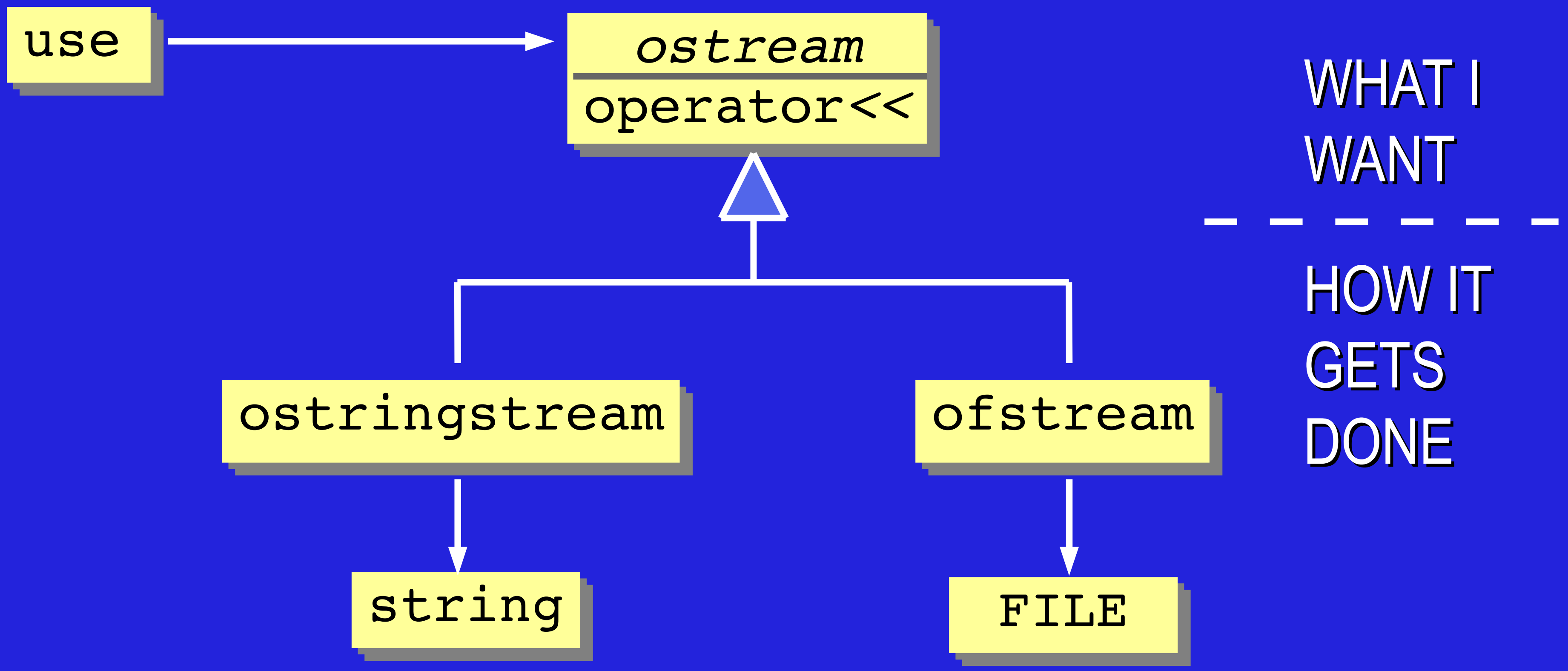
b( )

S_3

b( )

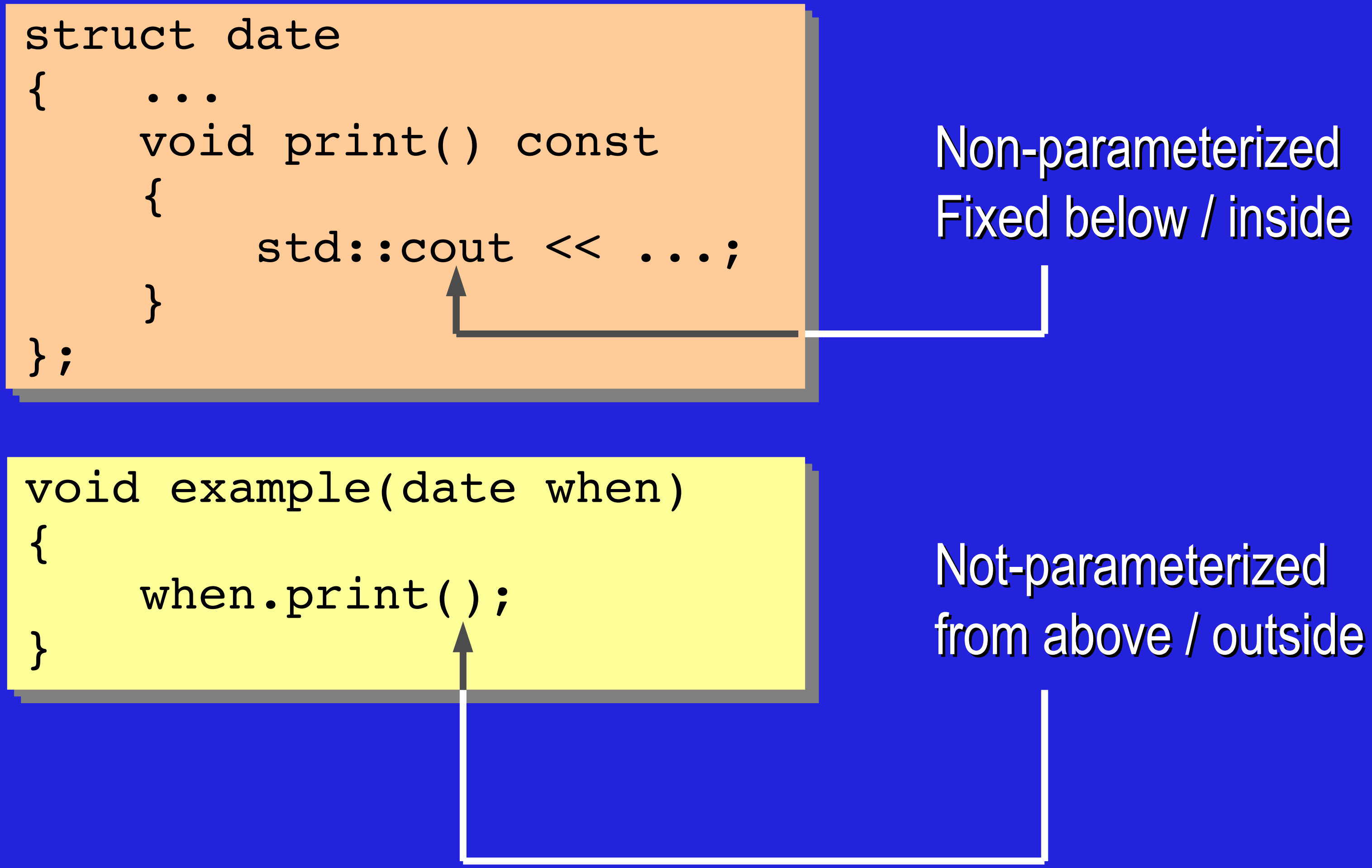
Liskov Subtitution Principle

- For example...



Hard-wired From Below :-(

- Complicates testing, increases dependencies



```
struct date
{    ...
     void print() const
     {
          std::cout << ...;
     }
};
```

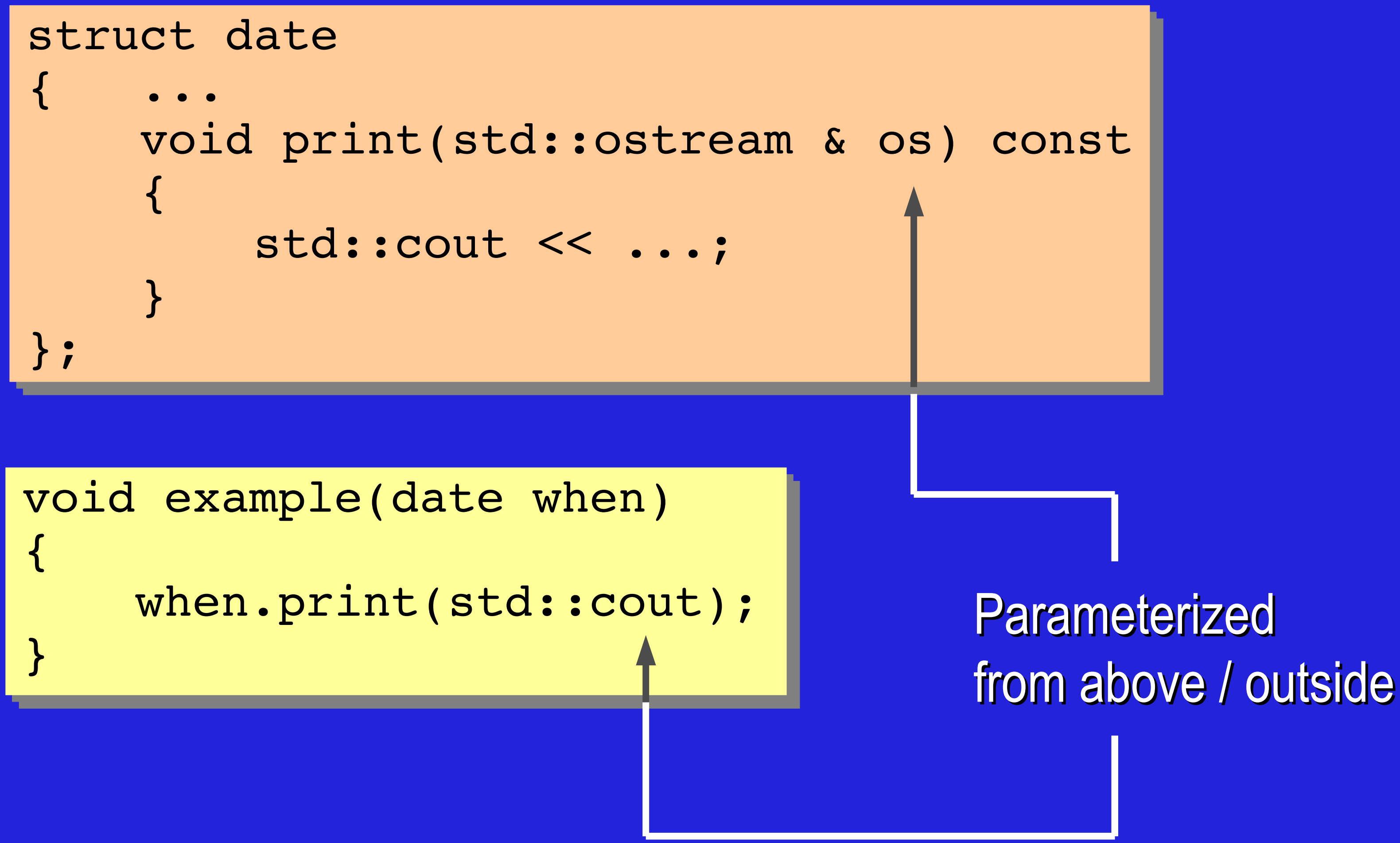
Non-parameterized
Fixed below / inside

```
void example(date when)
{
     when.print();
}
```

Not-parameterized
from above / outside

Parameterize From Above :-)

- Aim to make parameterization an explicit and visible part of the public api of a class/method



```
struct date
{   ...
    void print(std::ostream & os) const
    {
        std::cout << ...;
    }
};
```

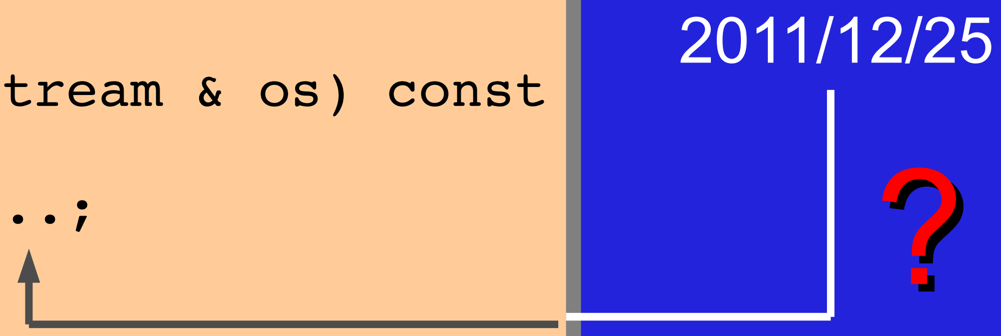
```
void example(date when)
{
    when.print(std::cout);
}
```

Parameterized
from above / outside

Single Responsibility Principle

- A class should be responsible for one thing and one thing only

```
struct date
{   ...
    void print(std::ostream & os) const
    {
        std::cout << ...;
    }
};
```



2011/12/25

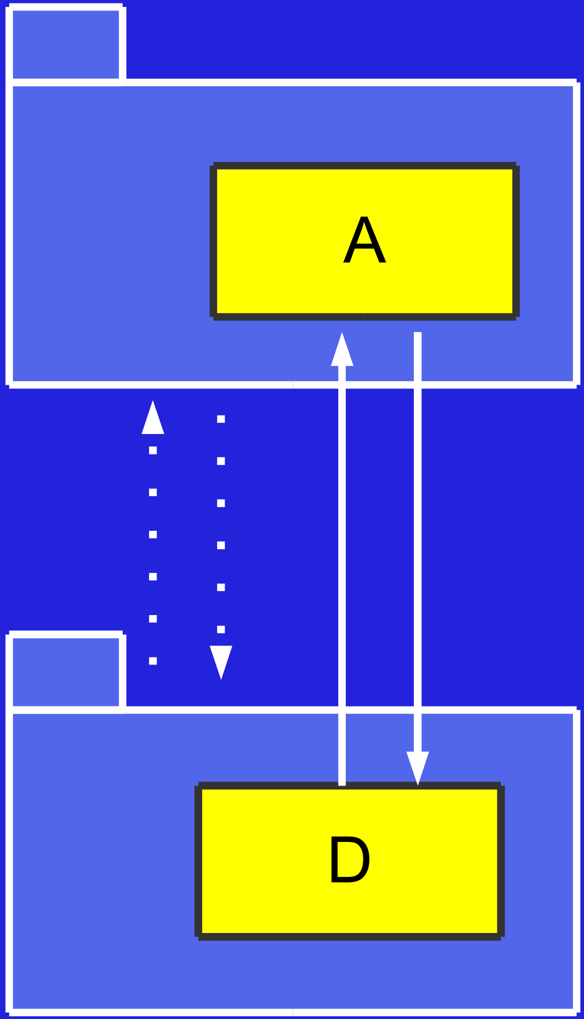
?

```
struct date
{   ...
    int year() const;
    int month() const;
    int day() cont;
};
```

Separation of Concerns

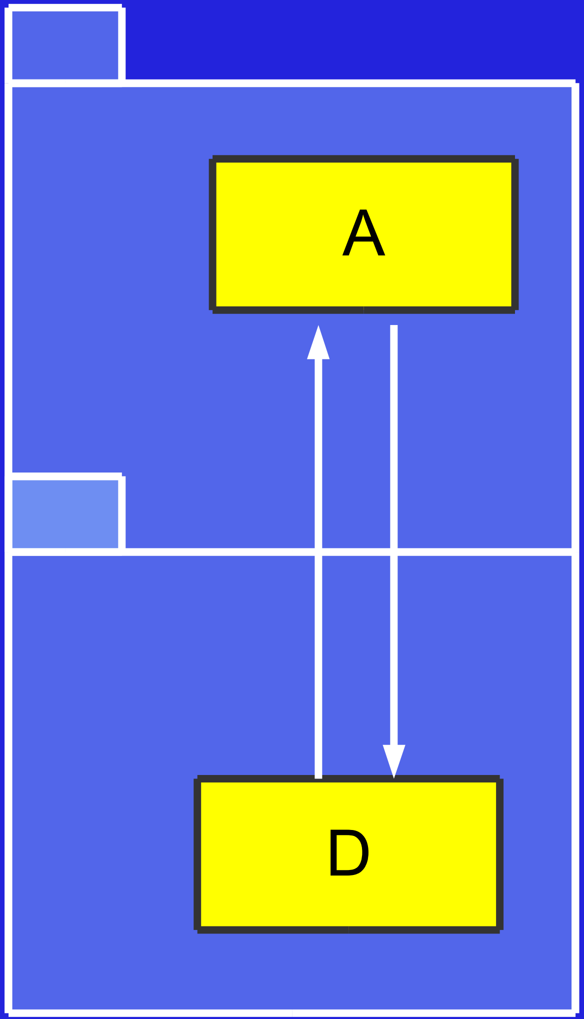
- Codebases tend to have a lot of effort focused on its functionalilty and not so much effort focused on its structure :-(
- Separate but dependent :-)

Dependency



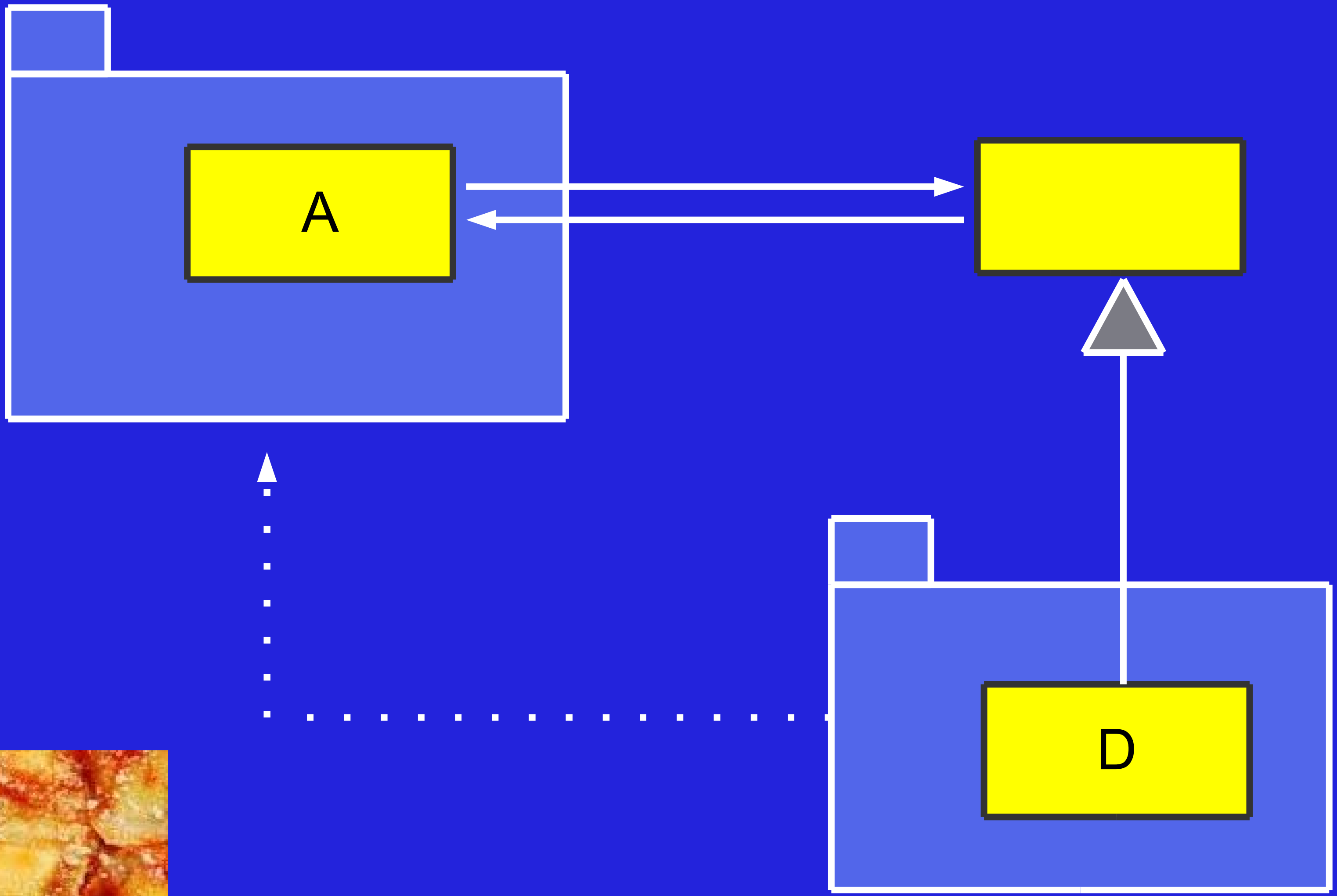
Claimed

Dependency

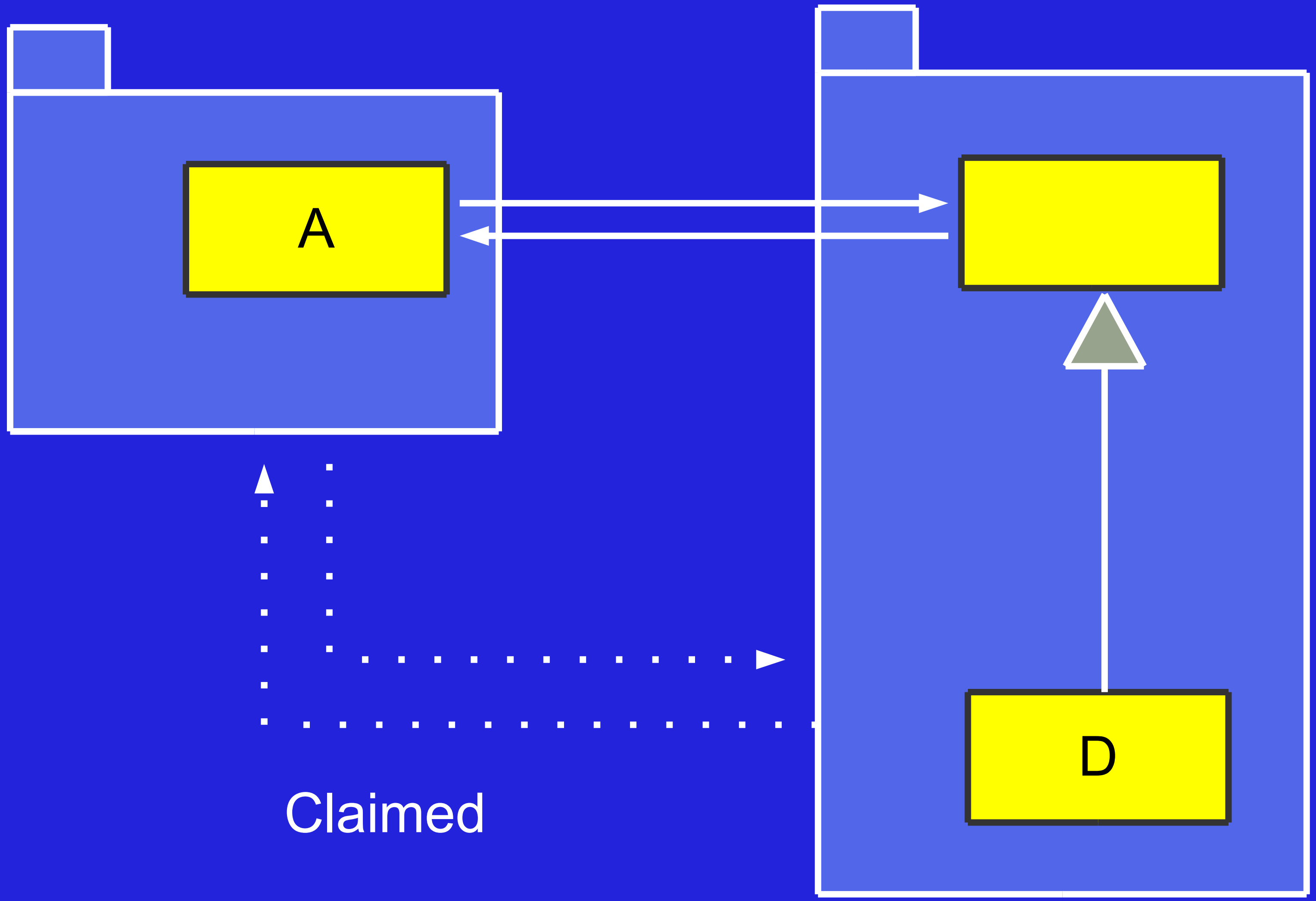


Actual

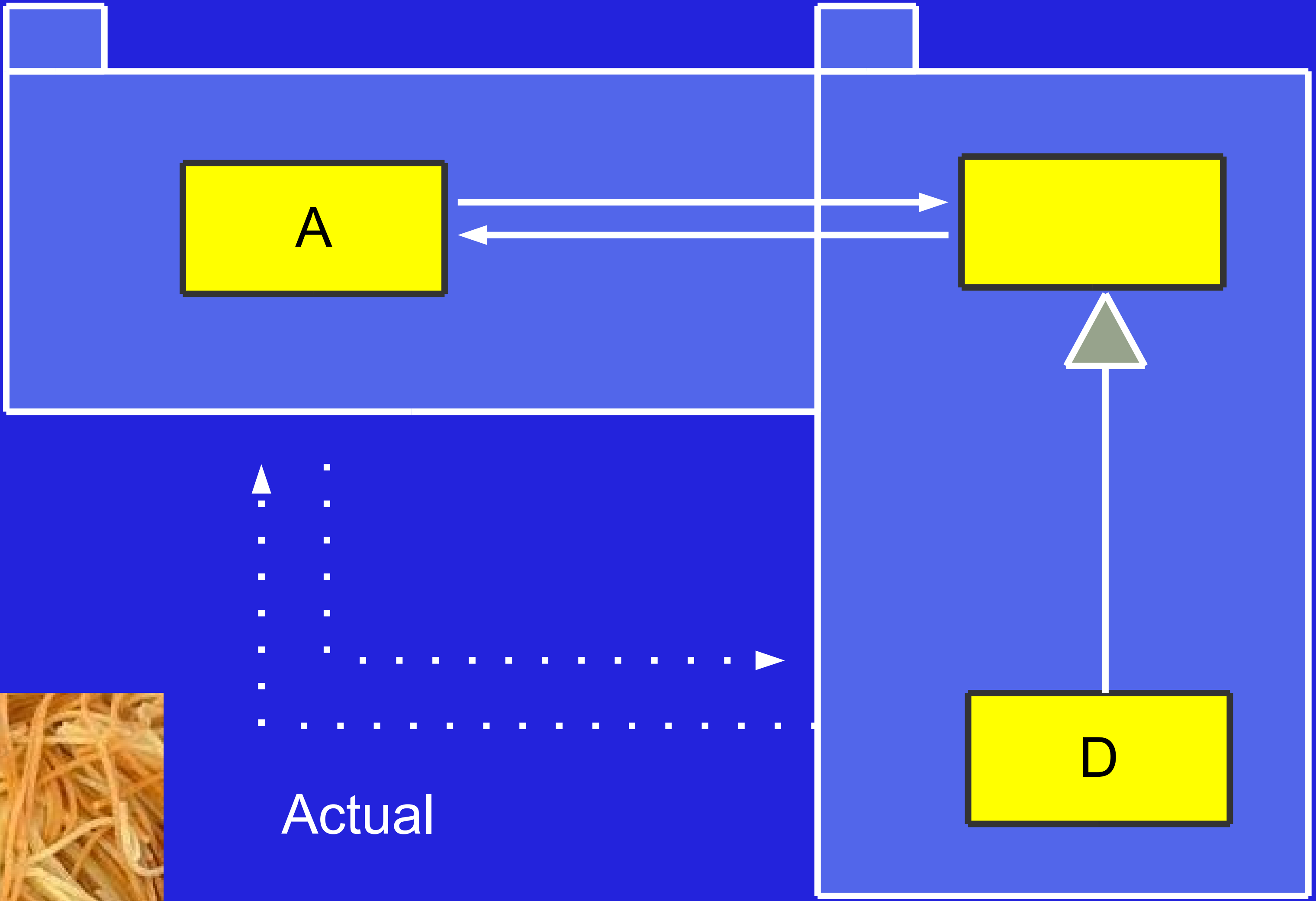
Dependency



Dependency

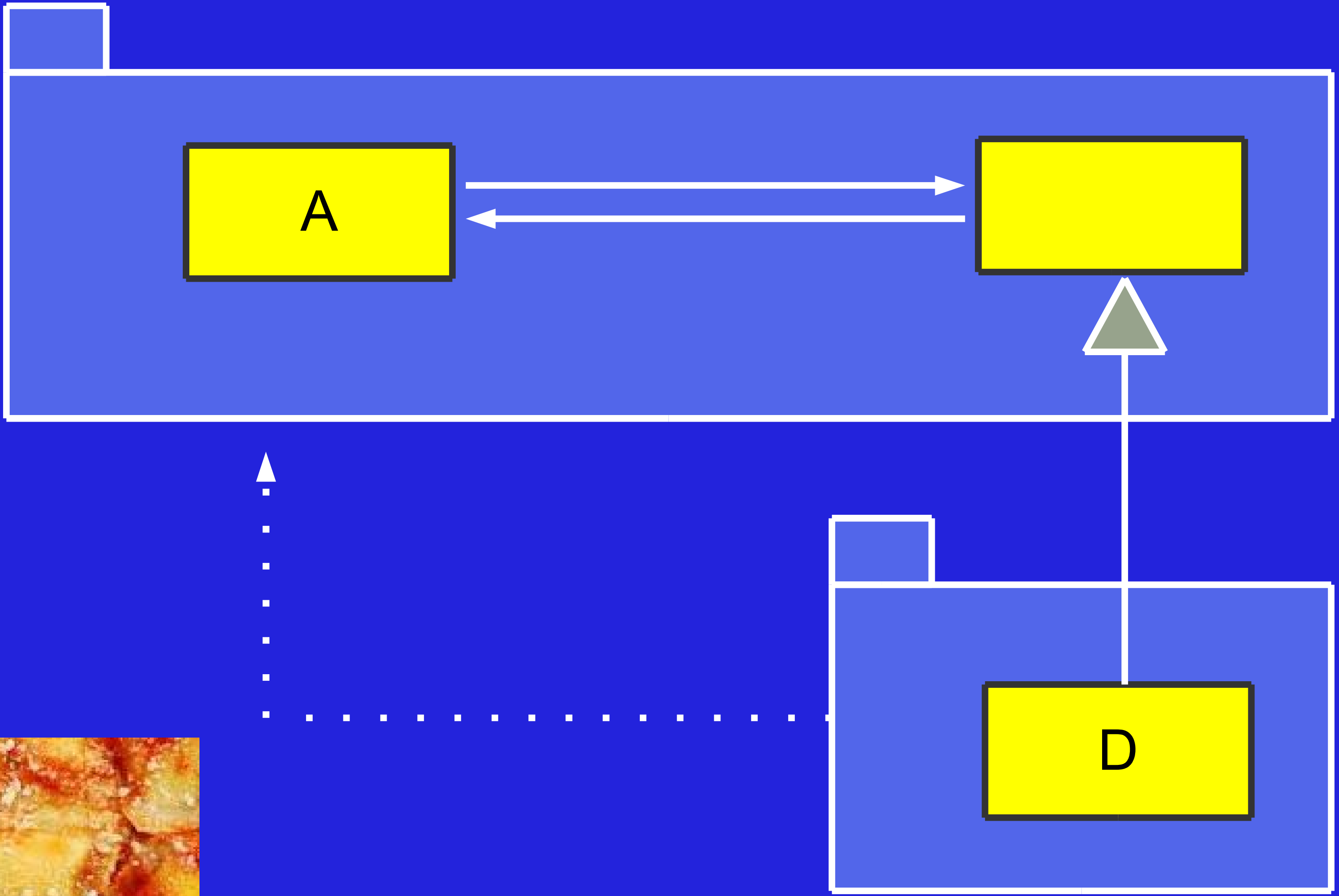


Dependency

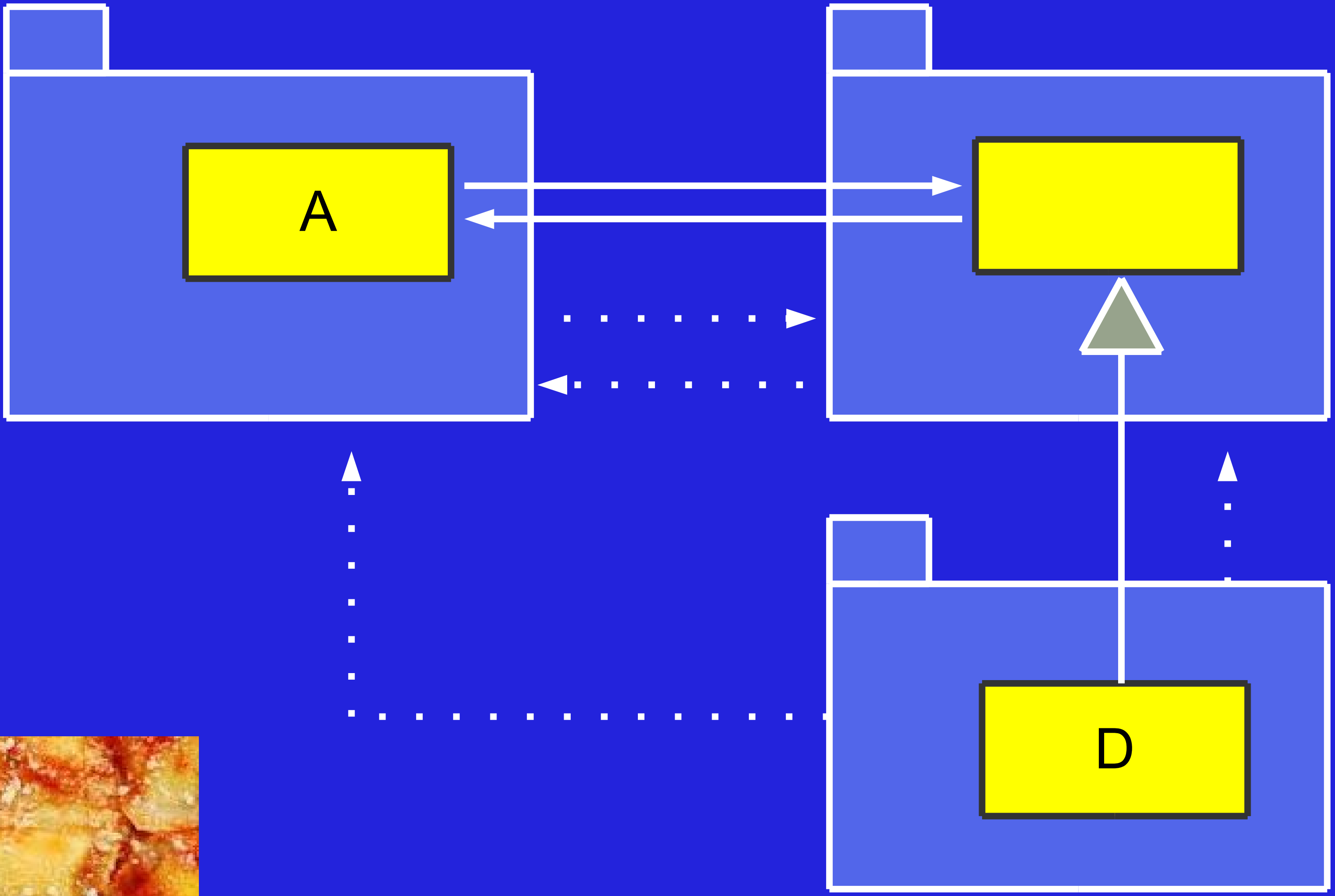


Actual

Dependency



Dependency

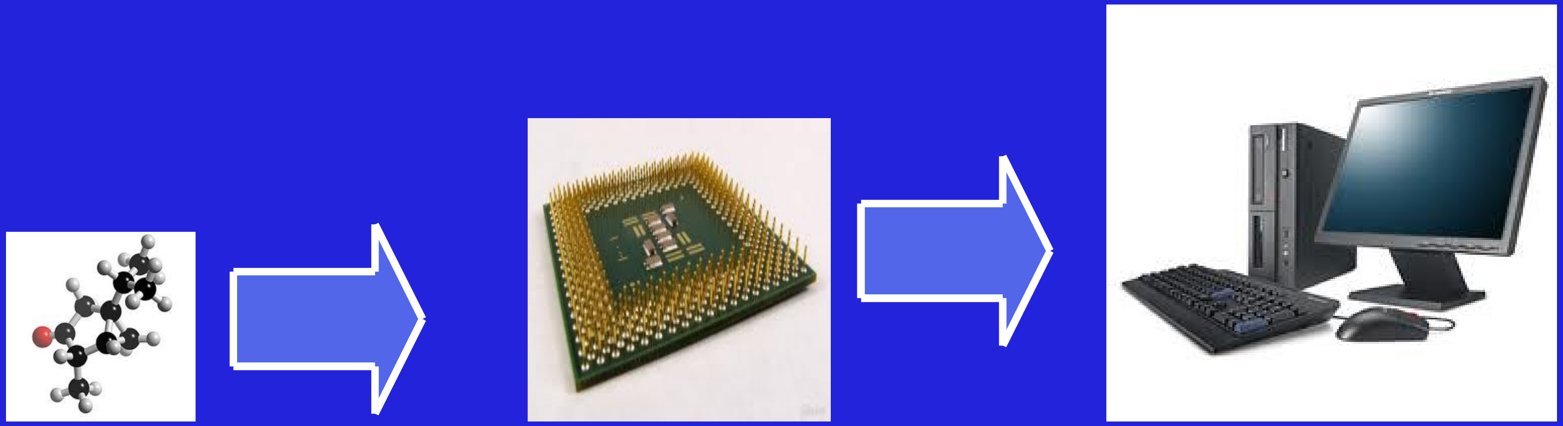


Abstraction

- We also hide information when creating crisp new semantic levels

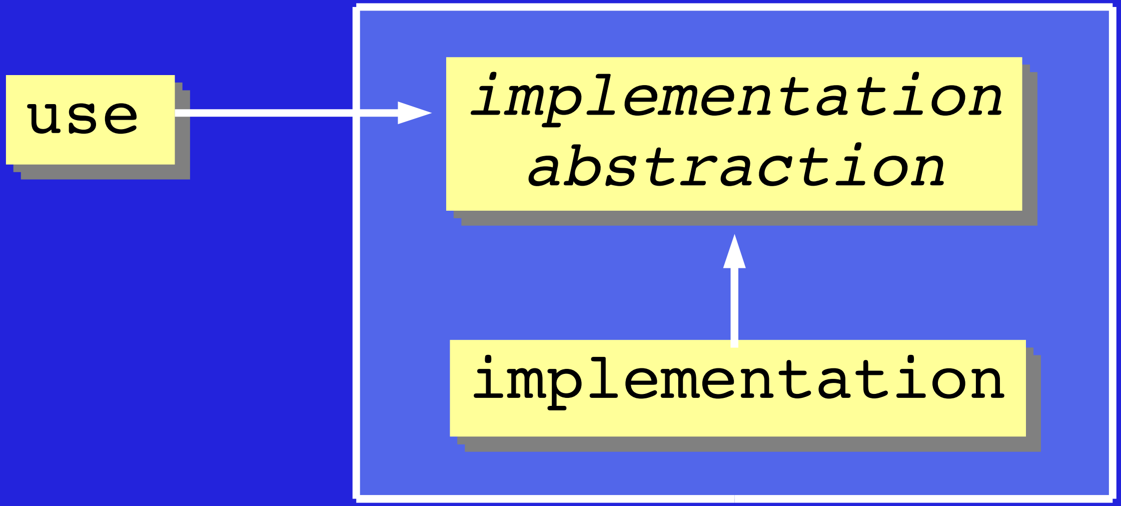
> Being abstract is something profoundly different from being vague... The purpose of an abstraction is not to be vague, but to create a *new semantic level* in which one can be absolutely precise.

Edsger Dijkstra

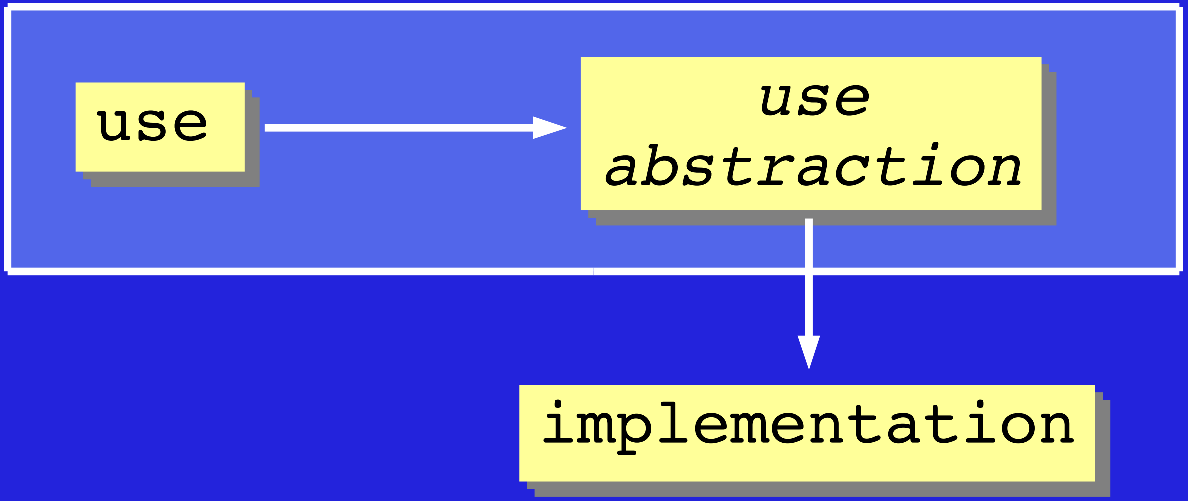


Abstraction

- How different are the semantic levels?



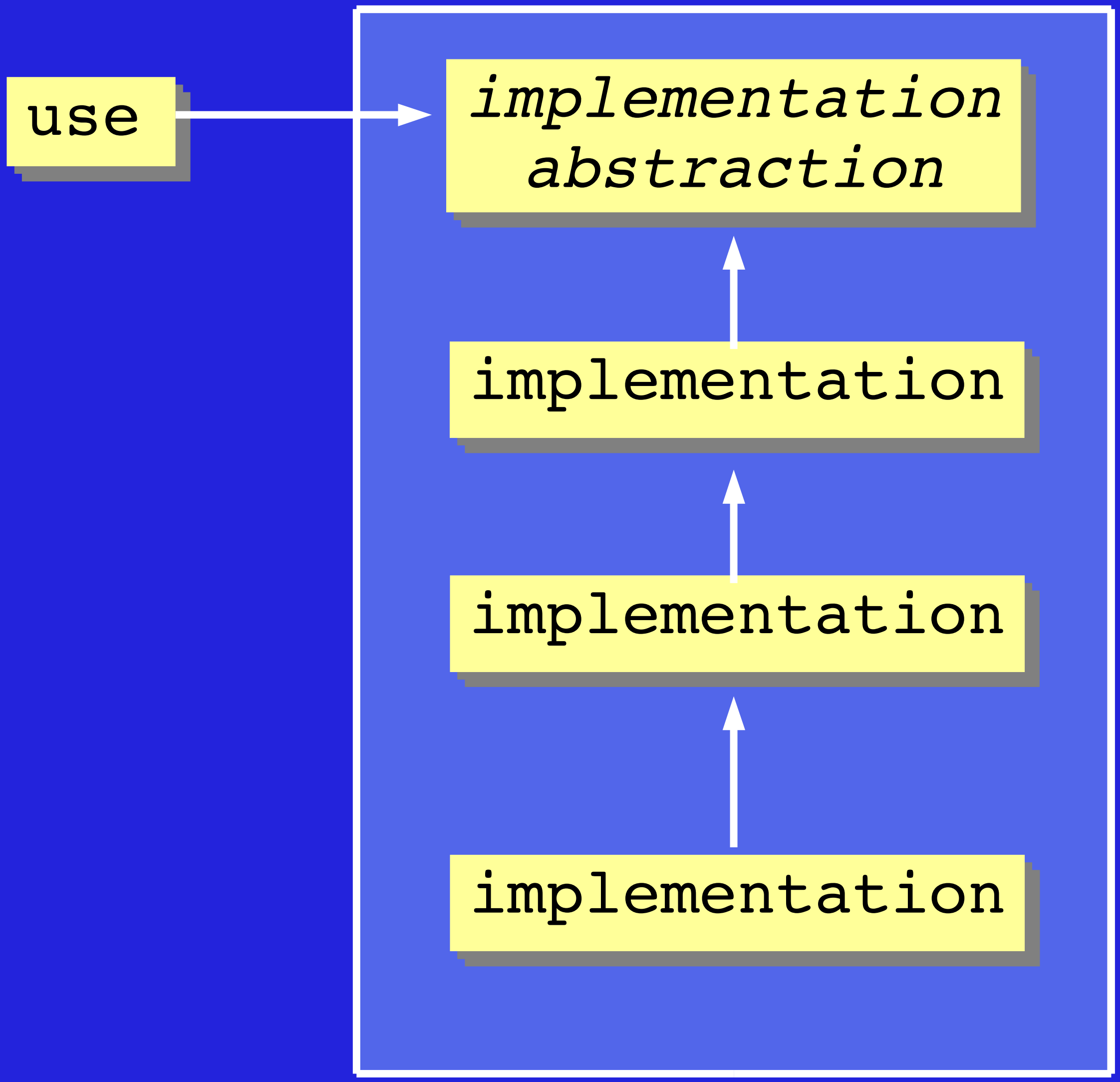
Some abstractions are weak and abstract away very little



Some abstractions are strong and abstract away a lot more

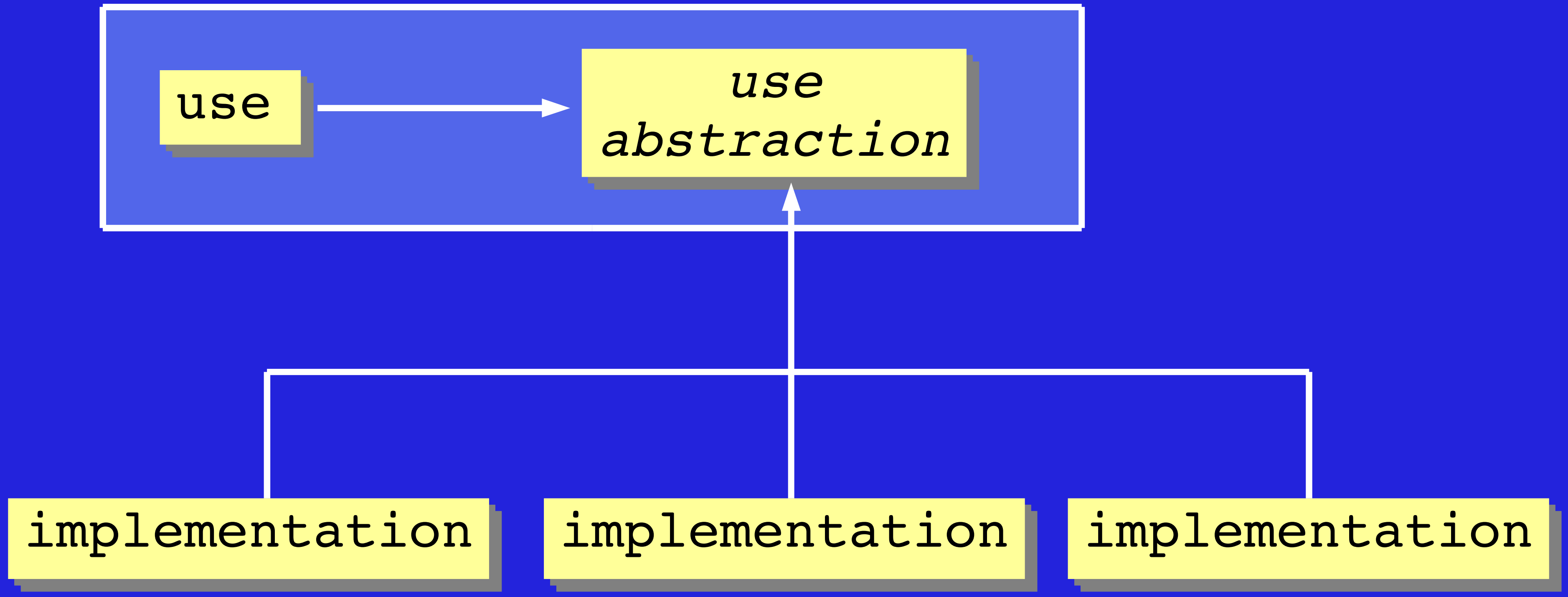
Abstraction?

- How much focus is on the implementation?



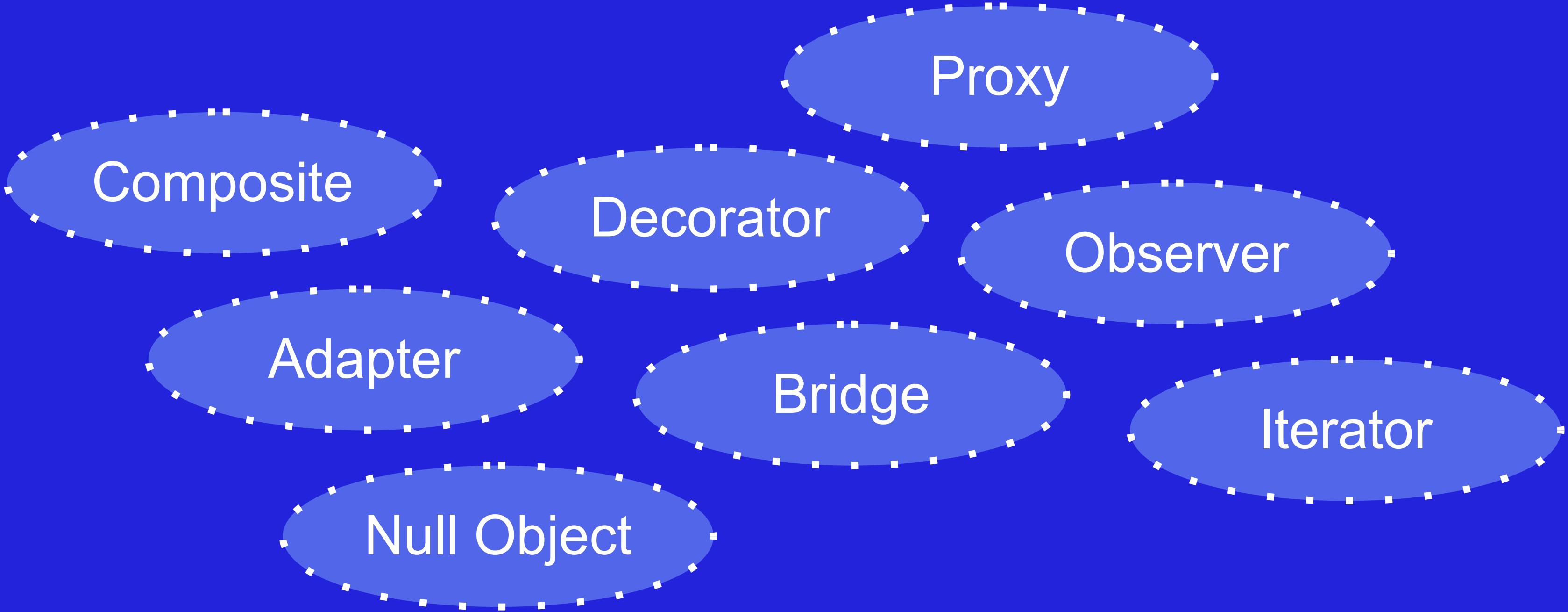
Abstraction

- How much focus is on the structure?



Patterns

- A class is not a useful unit of design!
- Patterns help you raise the level of abstraction
- Patterns document the role each class plays in a cluster of collaboration
- There are many named patterns



Proxy

Composite

Decorator

Observer

Adapter

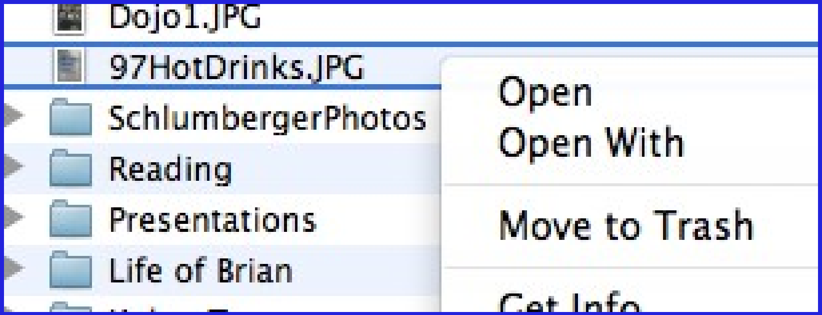
Bridge

Iterator

Null Object

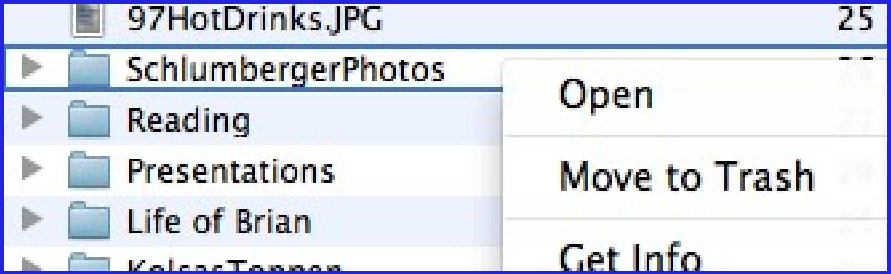
Deleting Files & Folders on a Mac

- Right click on the file or folder
- Click Move To Trash



file

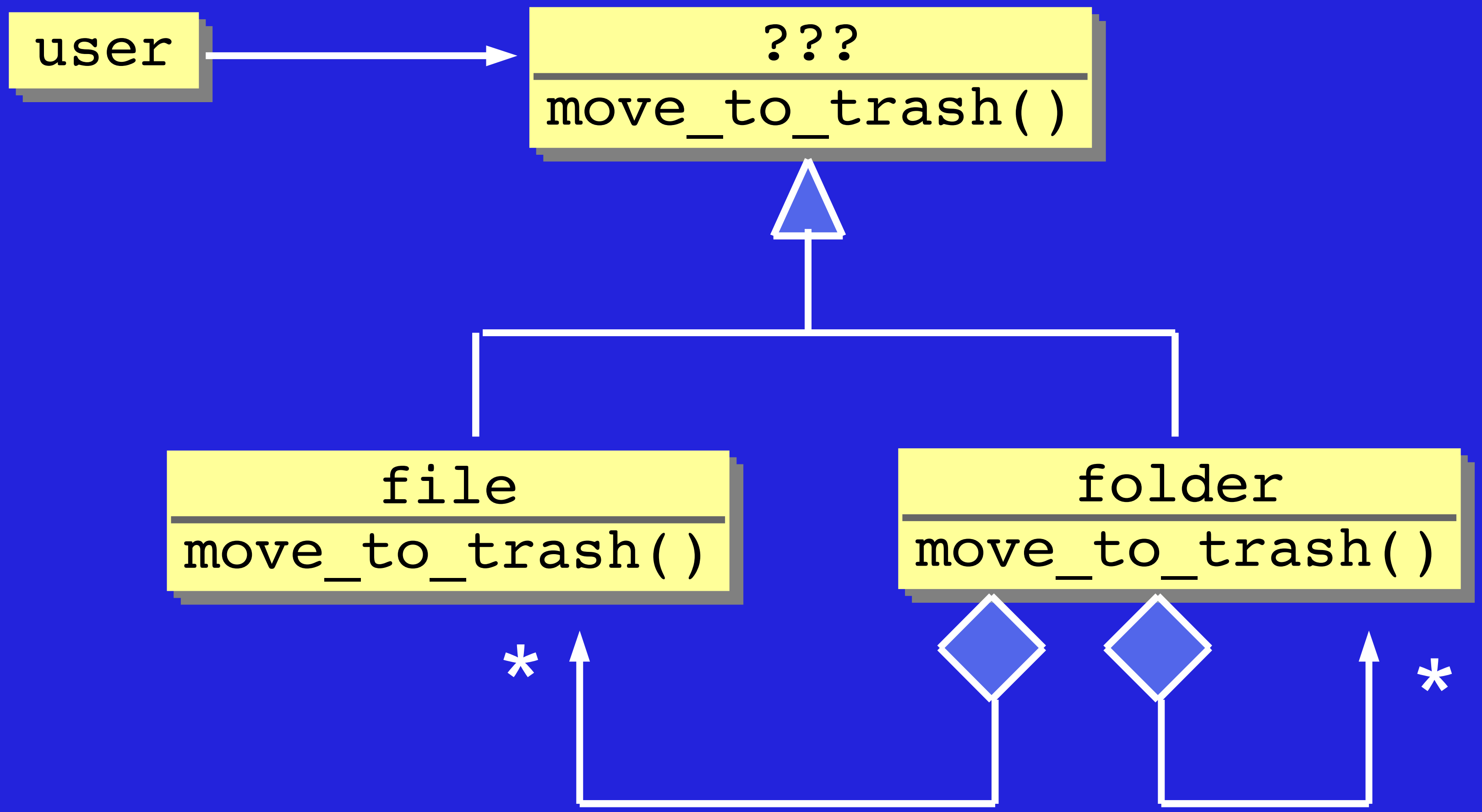
move_to_trash()



folder

move_to_trash()

Files and Folders



user

???

move_to_trash()

file

move_to_trash()

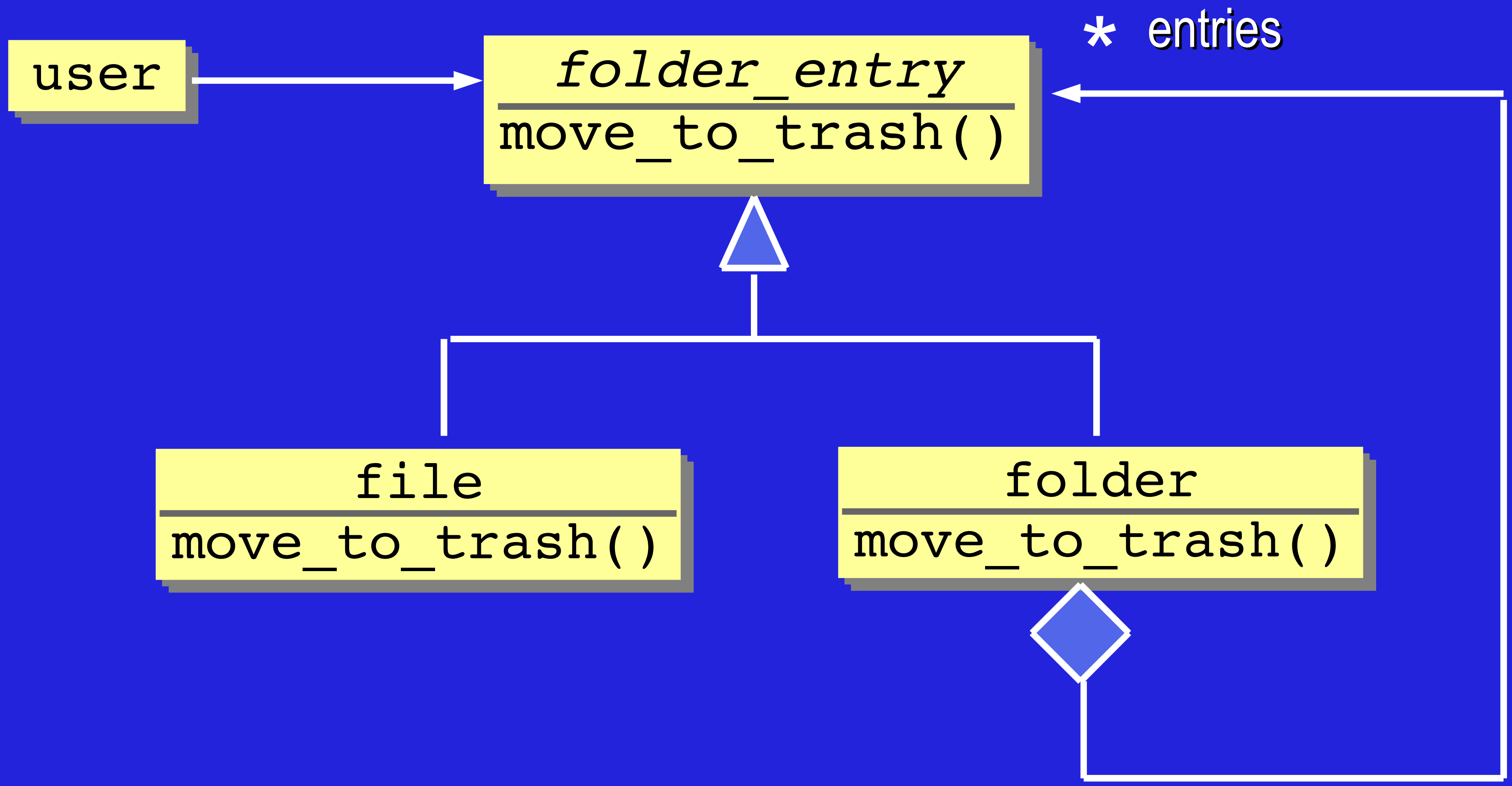
folder

move_to_trash()

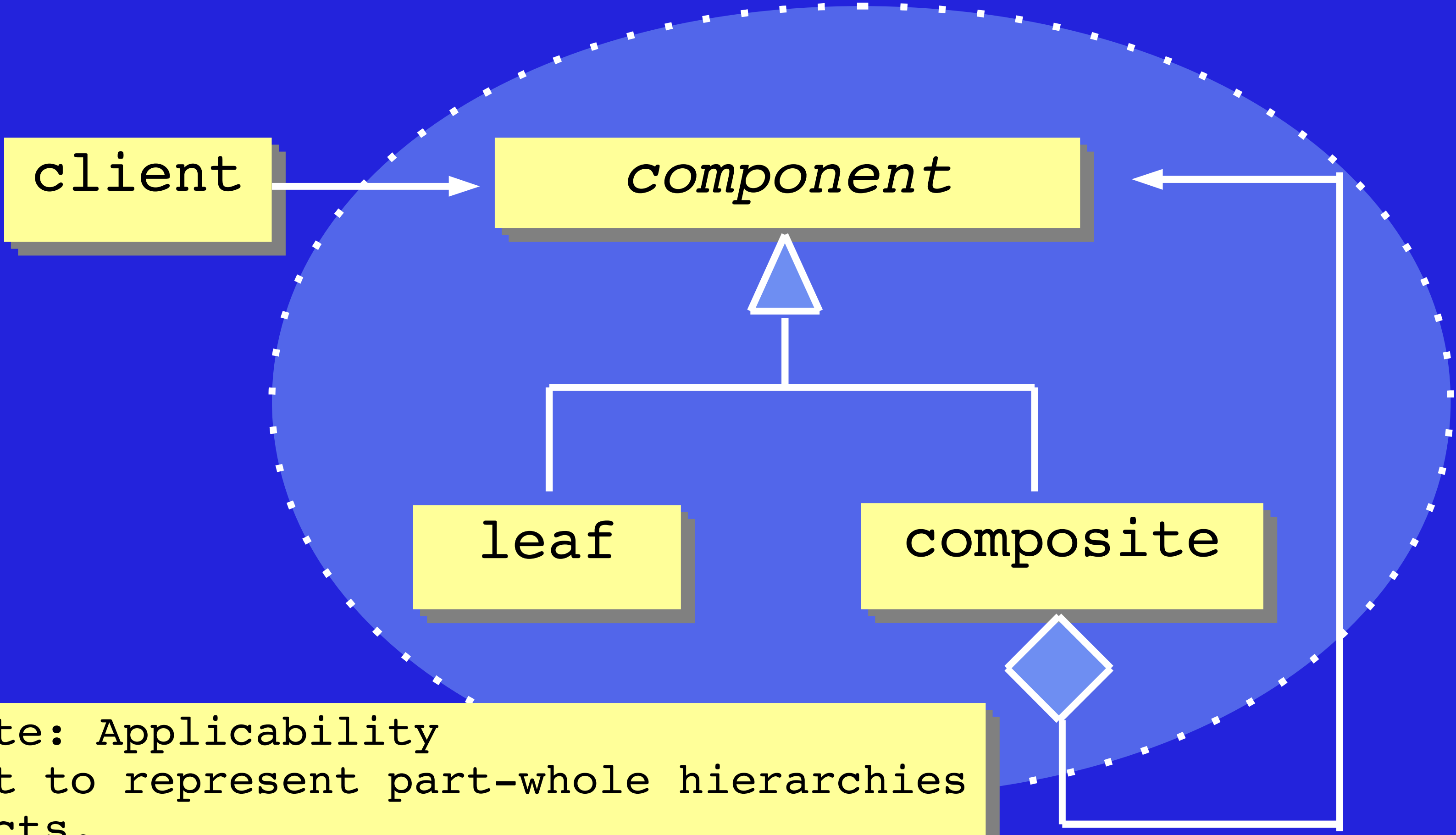
*

*

Files and Folders and Folder Entries



The Composite Pattern

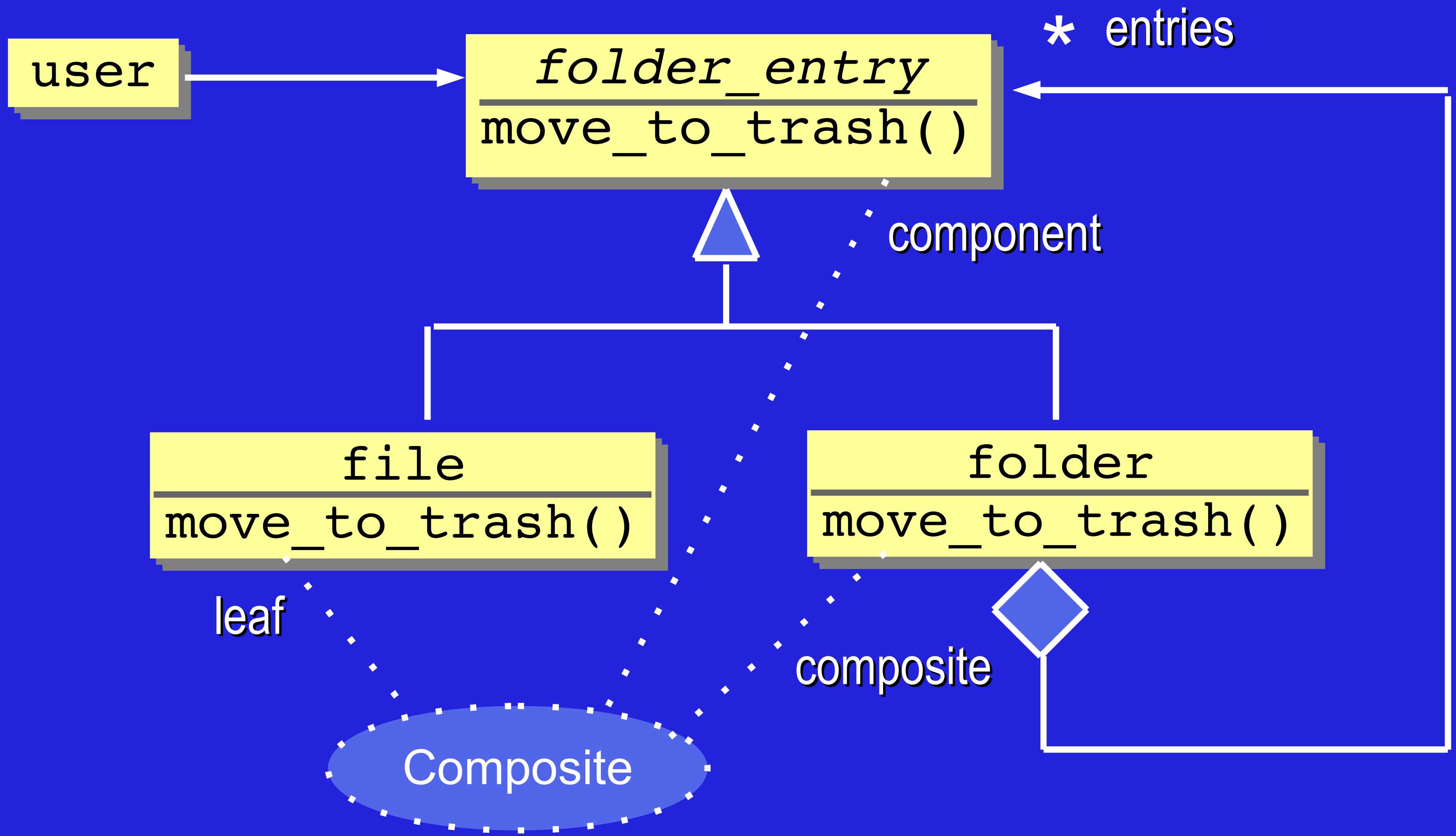


Composite: Applicability

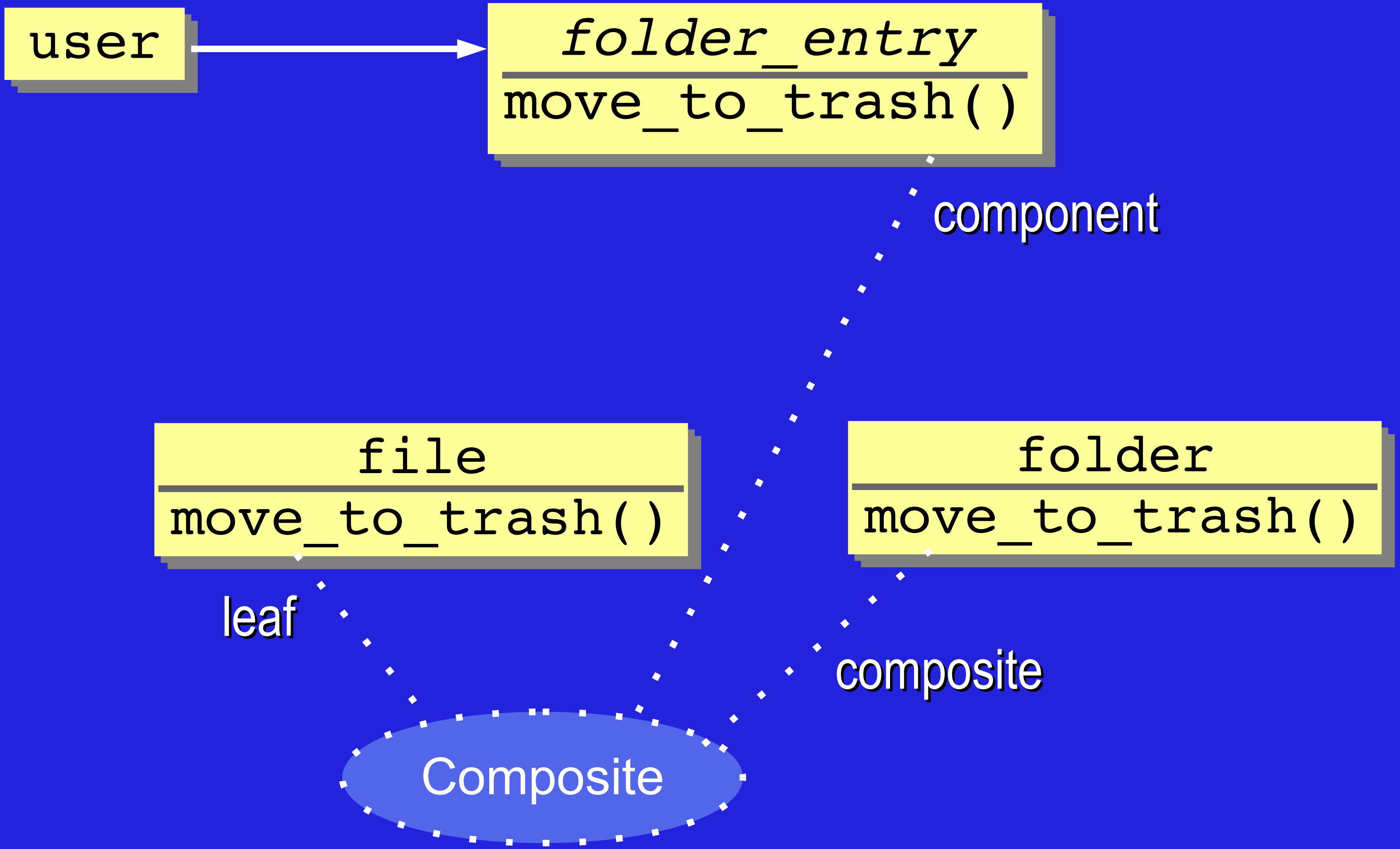
You want to represent part-whole hierarchies of objects.

You want clients to be able to ignore the differences between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Files and Folders



Patterns


user
folder_entry
move_to_trash()
component
file
move_to_trash()
leaf
folder
move_to_trash()
composite
Composite