Overview of last course

- Chapter 6 Class diagram
  - Operations
  - Associations, association generalization
  - Aggregation
  - Composite objects
  - Association classes

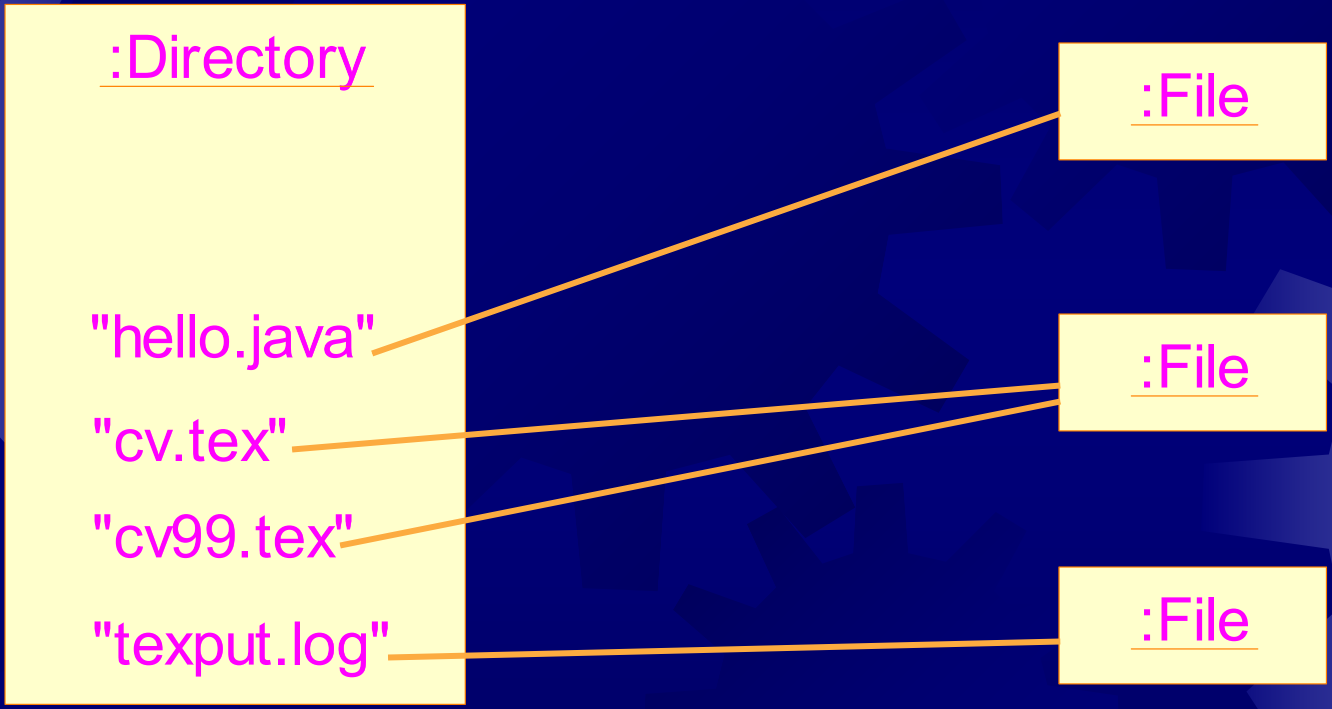
Outline of this course

- Class Diagram
  - Qualified associations
  - Multiple inheritance in U
  - Mixin class
  - Discriminato
- Implementation of class diagram
  - Uni-directional association
  - Bi-directional association
  - Implementing qualifiers
  - Implementation of association classes

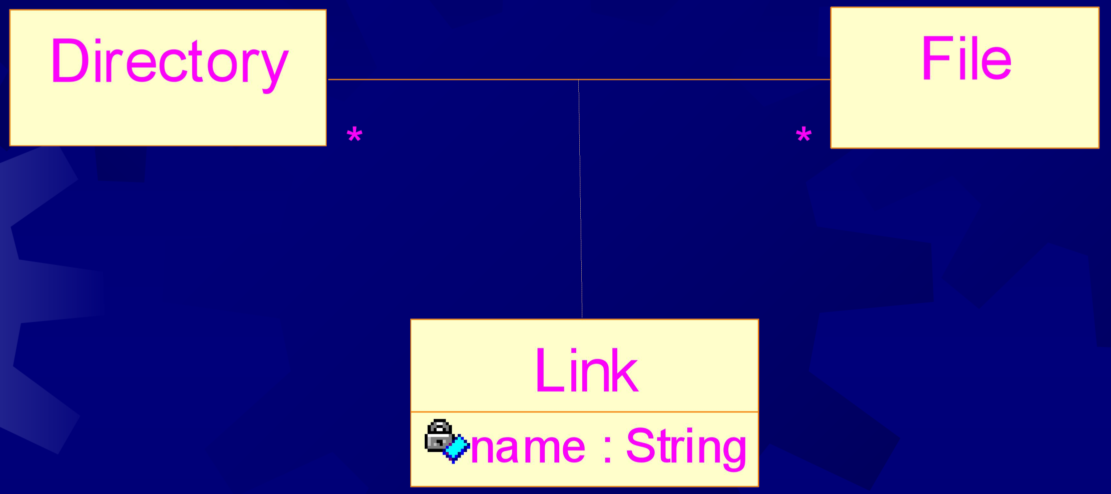
Qualified associations

- An example: Unix file system
  a) Each file has a unique internal identifier
  b) Each file can appear multiple times in different directories, even including in the same directory.
  c) All names within a directory must be different from each other.

Informal illustration of qualified association



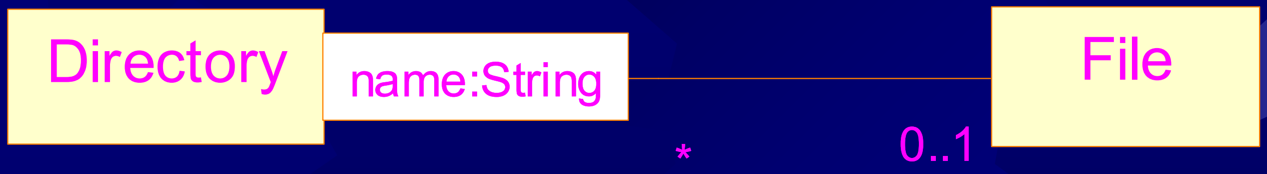
Association class can not describe the relationship



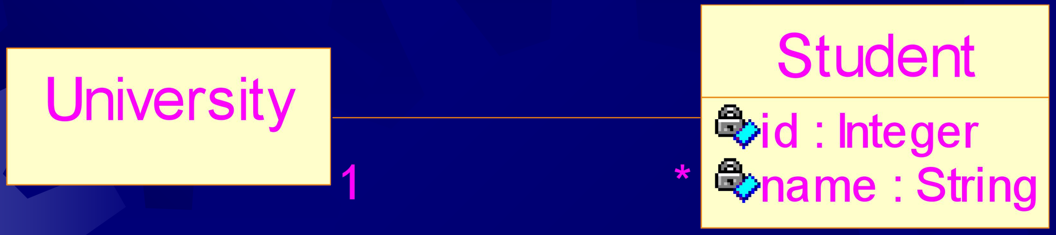
Reasons: It does not allow: a directory may have different names linked to the same file

Qualified associations

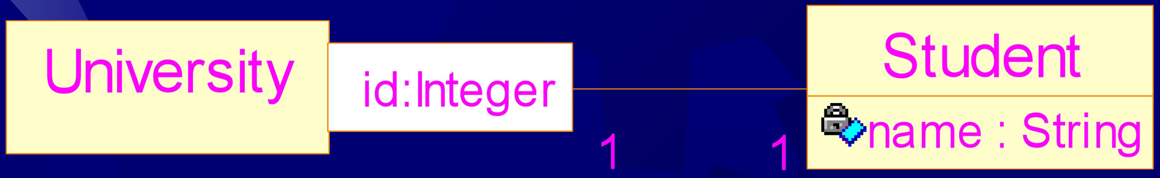
- Definition: an association class which has properties which enable it to act as a key
- Mapping from qualifier values to instances of the class at the other end of the association



Qualifiers and identifiers



Use of an attribute as an identifier

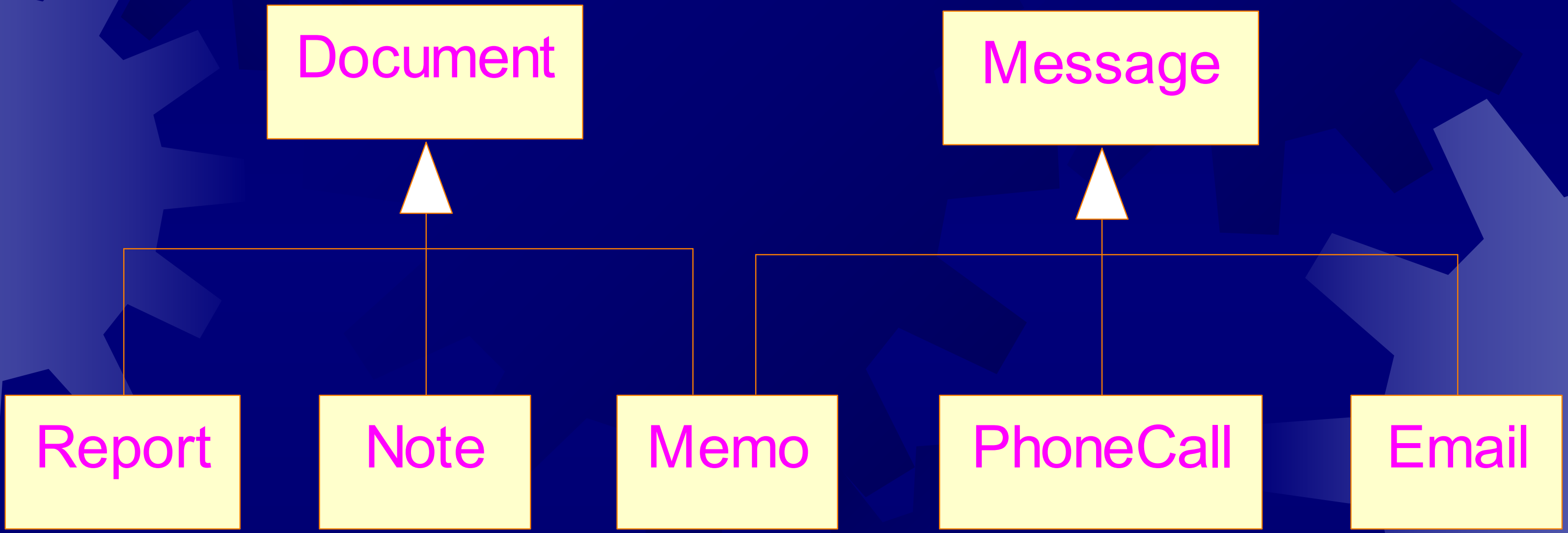


Better: Use of a qualifier

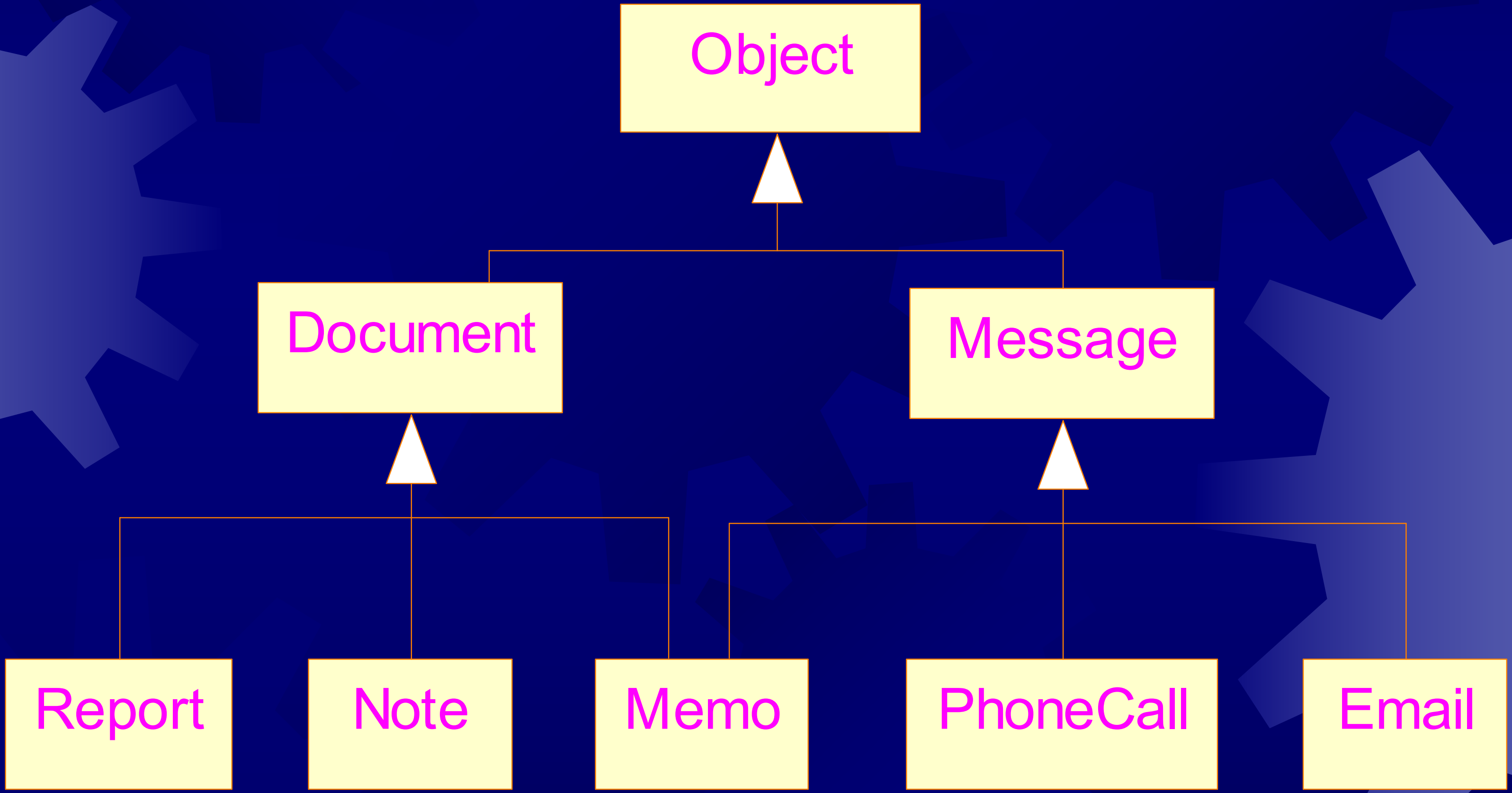
Multiple inheritance in UML

- UML allows multiple inheritance
- Attributes and operations of the common ancestor (if any) are inherited only once

An example of multiple inheritance



Multiple inheritance with a common ancestor

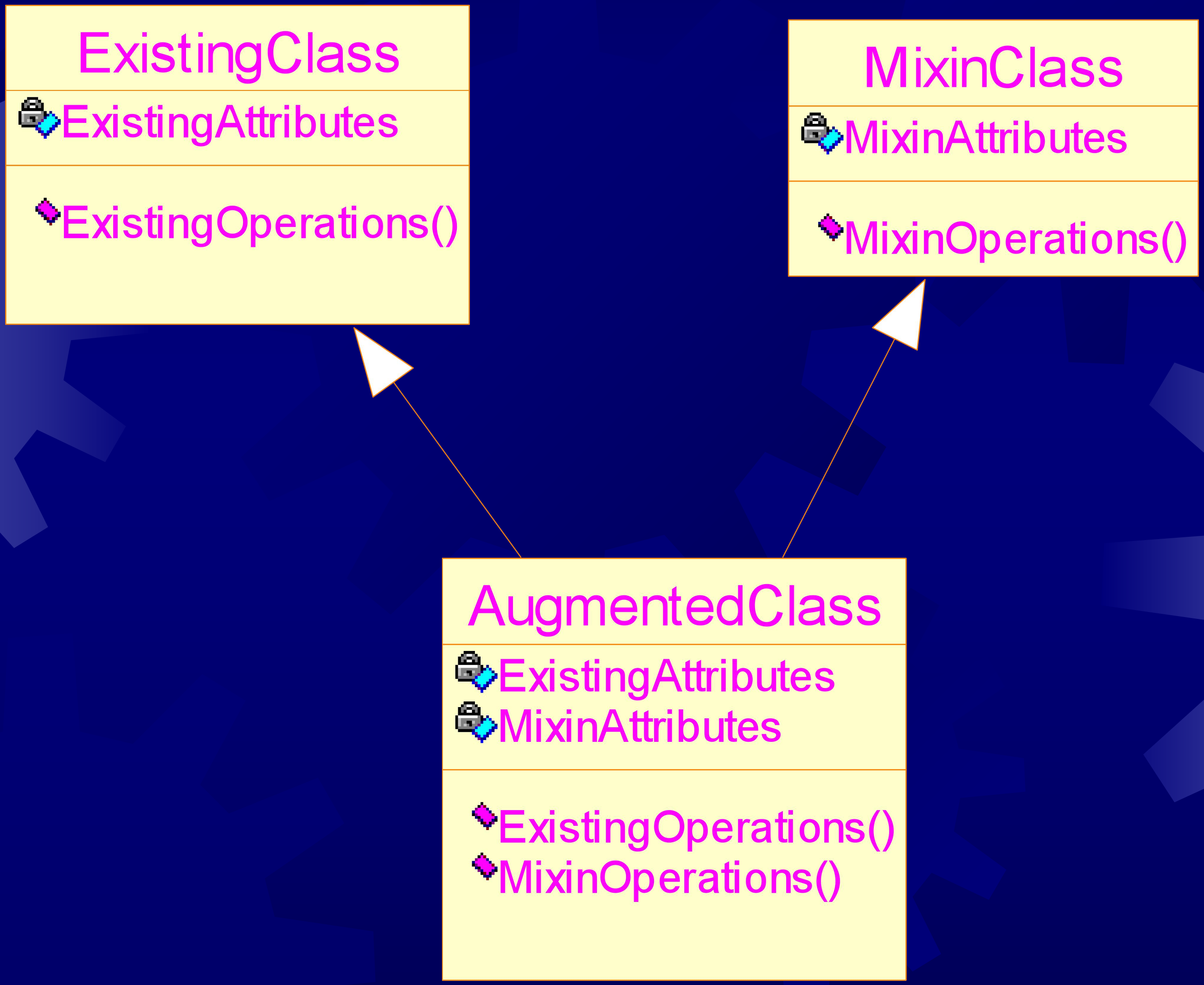


Mixin class

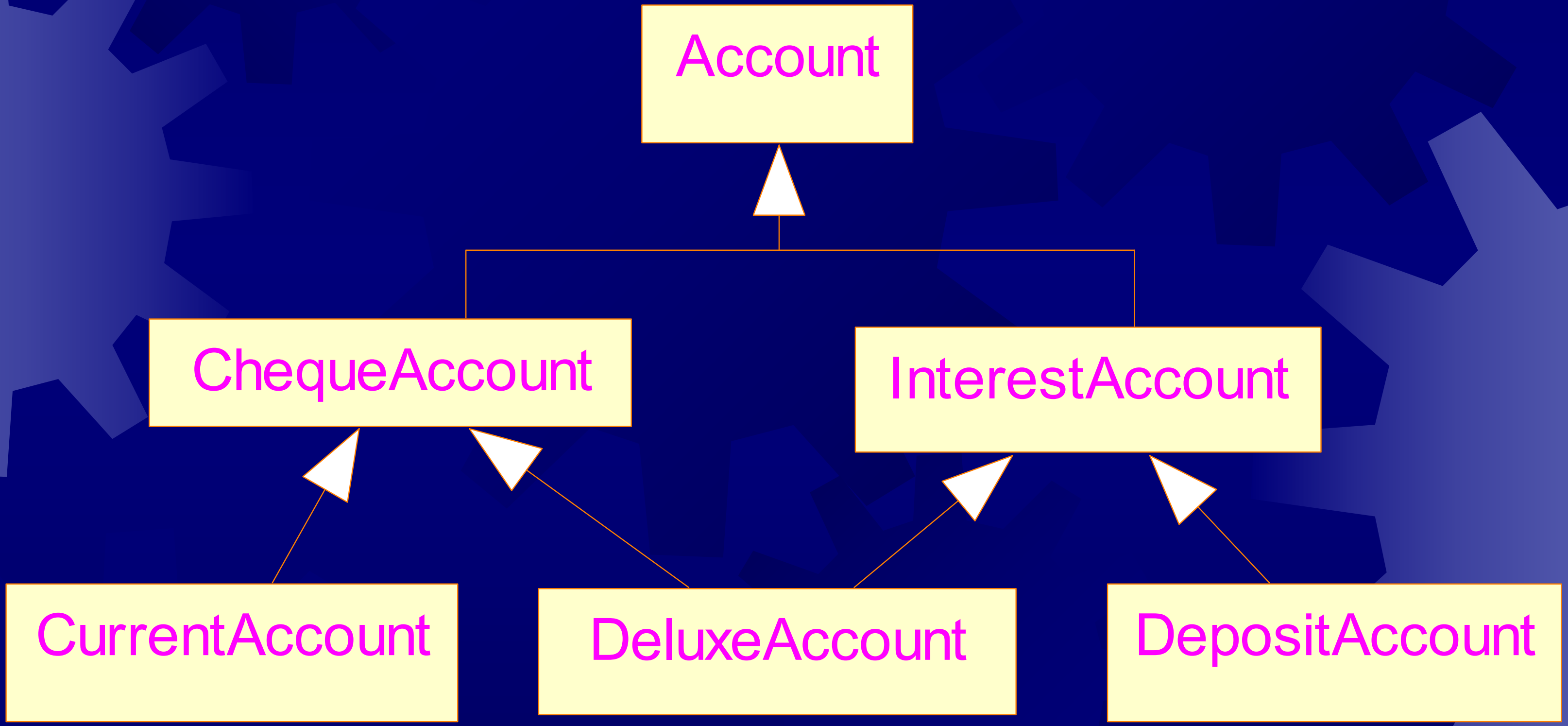
- A mixin class: is intended to provide an optional interface or functionality to other classes.
  The functionality should be independent with that of the existing classes.

- It is similar to an abstract classes in that it’s not intended to be instantiated.
- Mixin classes require multiple inheritance.

Illustration of mixin classes



If without mixin class



Account

ChequeAccount

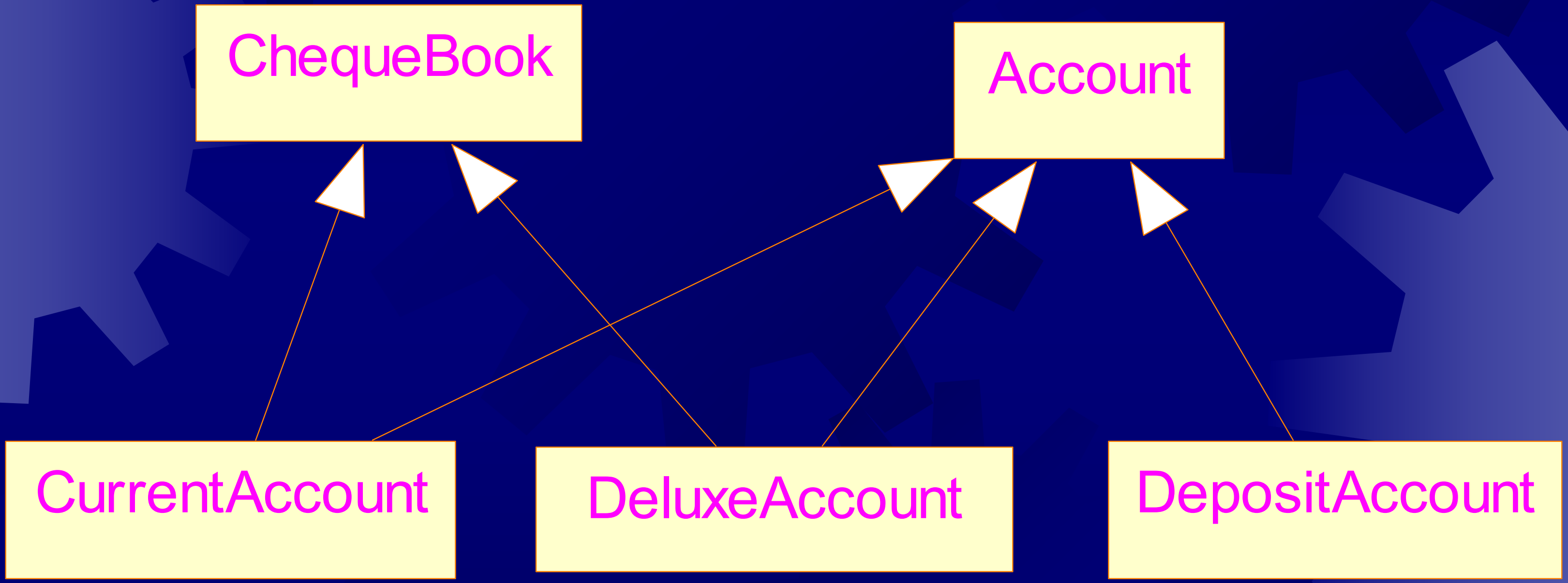
InterestAccount

CurrentAccount

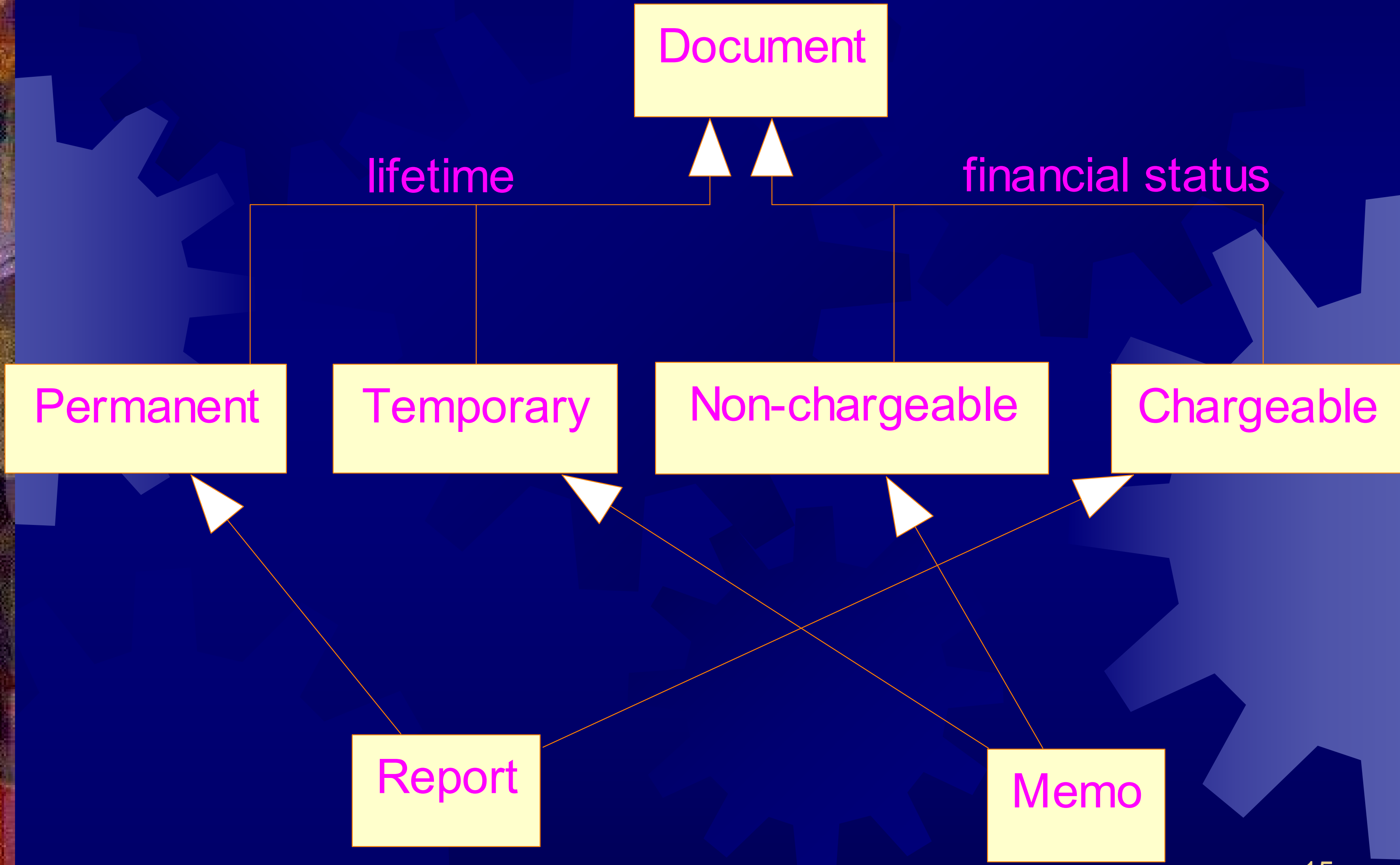
DeluxeAccount

DepositAccount

With Mixin class



Discriminators

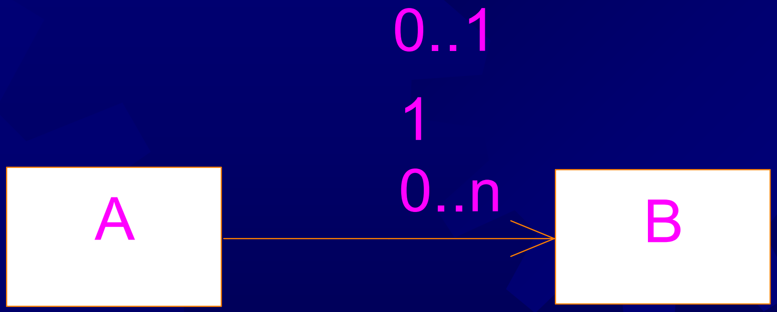


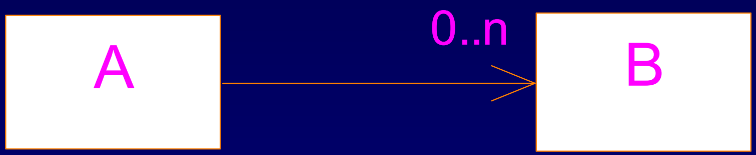
Implementing associations

- Uni-directional links
  - simple to implement
  - Will lead to problems if they are later modified to bi-directional links
- Bi-directional links
  - Complex to implement → consistency
  - With no risk
- Both styles of implementation should be hidden from client code

Uni-directional implementations

- The multiplicity at the tail of the arrow has no effect on the implementation
- Mutable and immutable associations
- To implement the semantic correctly, we need:
  - Proper declarations of data members, *and*
  - Proper definitions of member functions

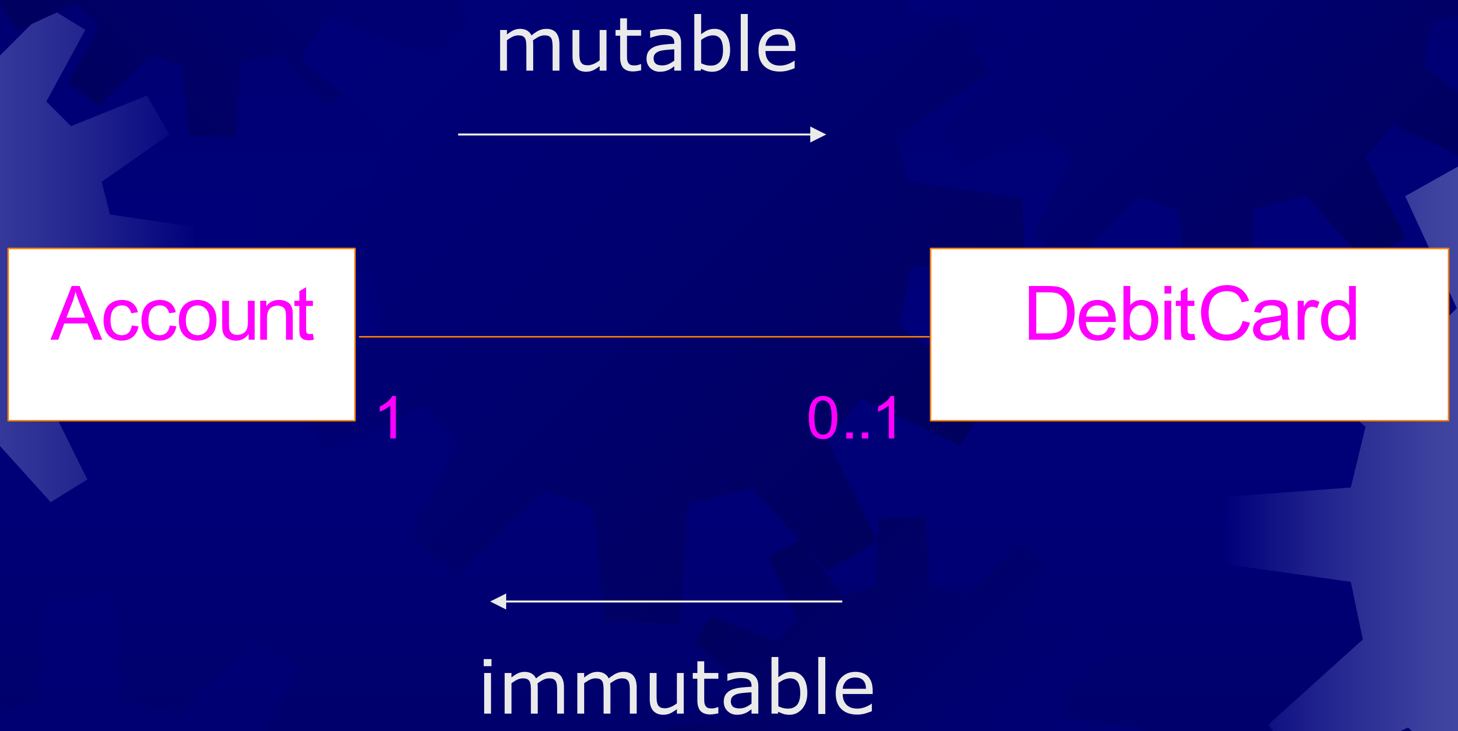




```
#include <iostream>
#include <vector>
using namespace std;
class B {
        //...
public:
        operator ==( B & r) {};
};
class A {
      vector<B> links;
public:
        addLink(B & b) {
                links.push_back( b );
        };
```

```
    removeLink(B & b ){
        vector<B>::iterator it;
        for ( it=links.begin(); it!=links.end();
            it++) {
            if ( *it == b) break;
        }
        if ( it != links.end() )
          links.erase(it);
    };

}
main()
{
}
```

Bidirectional implementation



mutable

Account

DebitCard

1

0..1

immutable

```
class DebitCard;
class Account {
    DebitCard * card;
public:
    DebitCard& getCard();
    void setCard(DebitCard & p_card );
    void removeCard();
};
class DebitCard{
        Account * theAccount;
public:
        DebitCard(Account & a);
        getAccount();
}
```

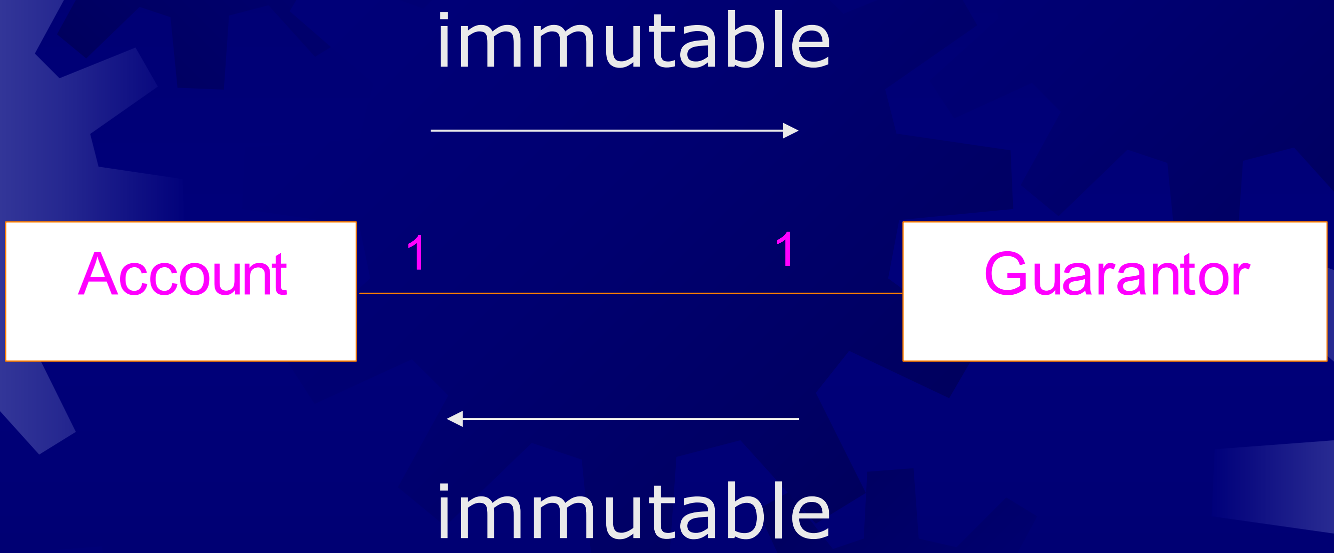
```
main()
{
        Account a1;
        Account a2;
        DebitCard card1(a1);
        a1.setCard(card1);
        //a2.setCard(card1); → cause problem!
}
```

A better solution

```
class Account;
class DebitCard{
        Account * theAccount;
private:
        DebitCard(Account & a);
public:
        getAccount();
        friend class Account;
};
class Account {
        DebitCard * card;
public:
        void addCard() {
                card = new DebitCard(*this);
        }
```

```
    DebitCard& getCard();
    void removeCard();
};
main()
{
        Account a1;
        a1.addCard();
}
```

Bidirectional implementation: both directions are immutable



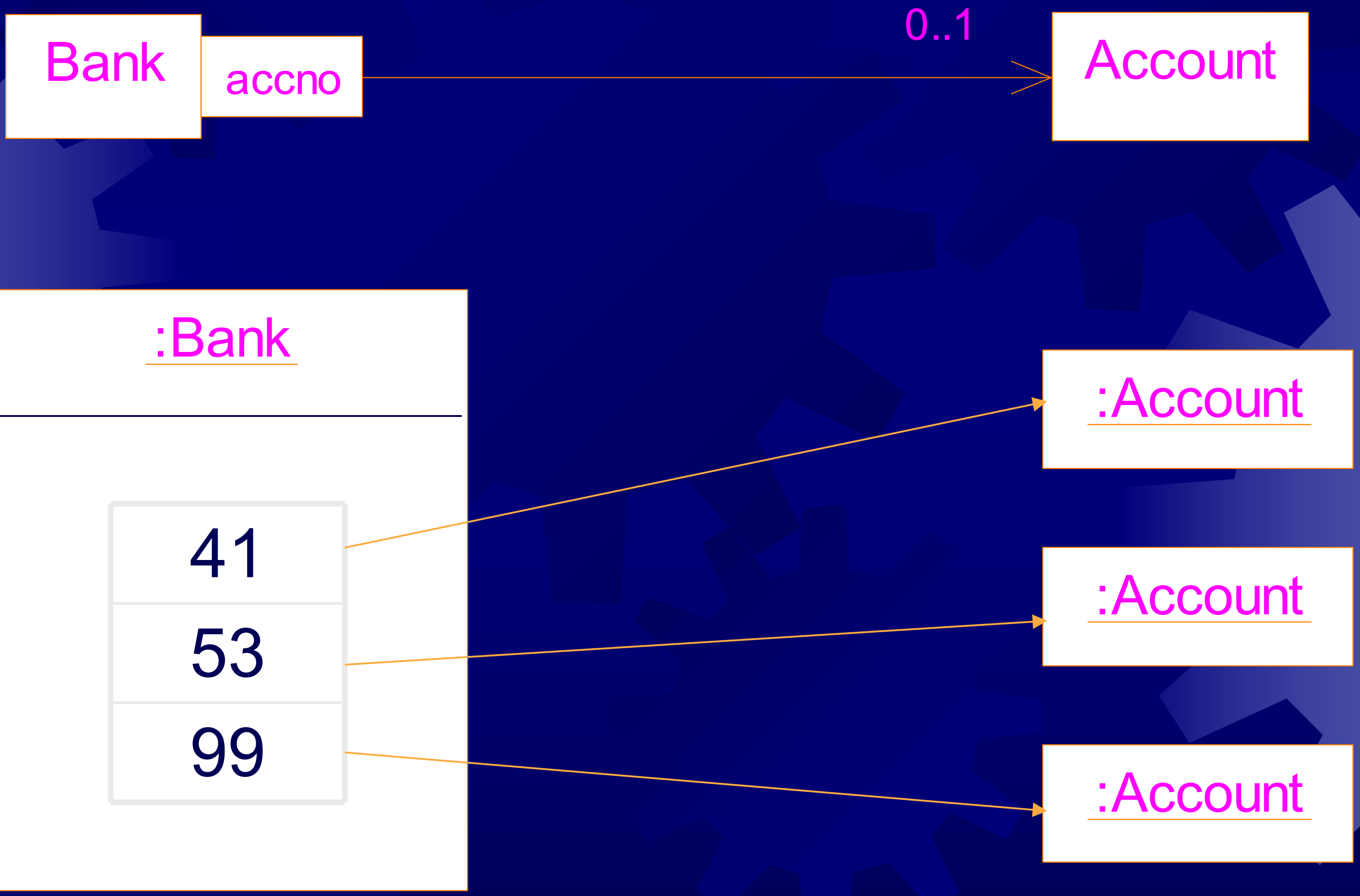
A typical mistake: want to create the link within a single step.

```
#include <iostream>
using namespace std;
class Guarantor;
class Account {
  public:
        Guarantor * pGuarantor;
        int i;
        Account( Guarantor * g) {
                pGuarantor = g;
                i = 100;
               f();
        }
        virtual f()  {cout << "f\n"; }
};
```

```
class Guarantor {
public: Account * pAccount;
        int j;
        Guarantor(Account * a)     {
              pAccount = a;   j = 200;
              g();
              //a->f();
              //cout << a->i;
      }
      virtual g(){ cout << "g\n"; }
};
main ()
{  Account *a = new Account(new Guarantor(a) );
   cout << a->pGuarantor->j;
   //cout << a->pGuarantor->pAccount->i;
}
```

uncertain

Implementing qualifiers



```
#include <iostream>
#include <string>
#include <map>
using namespace std;
main( )
{
   // a map of expenditure
   map<string, double> exp;
   string item;
   double cost;
   while (cin>>item >>cost)
       exp[ item ] += cost;
   double total=0;
   map<string,double>::iterator p;
for (p=exp.begin(); p!=exp.end(); p++){
    total += p->second;
    cout << p->first << ": "
          << p->second << endl;
  }
  cout << "total expenditure: " << total;
}
```

```
Input:
food  100
book 150
tax    30
book  100
food  180
```

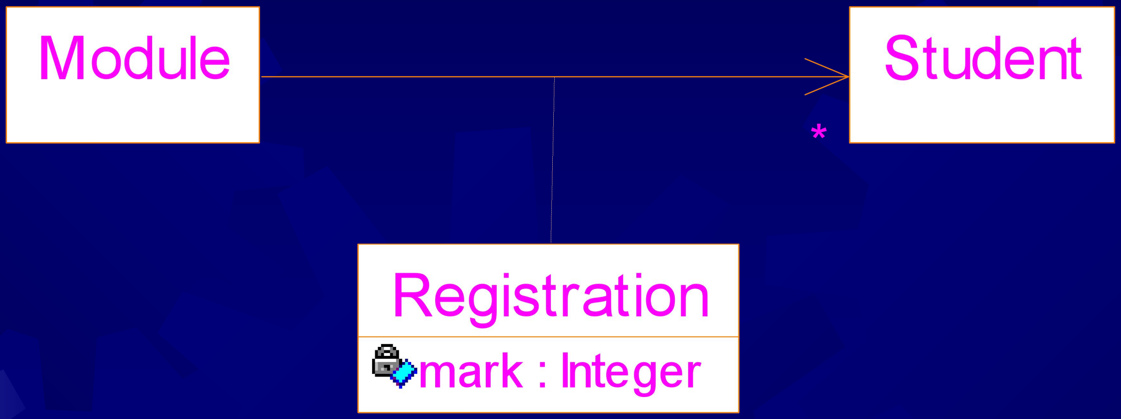
```
output:
book: 250
food: 280
tax: 30

total expenditure: 560
```

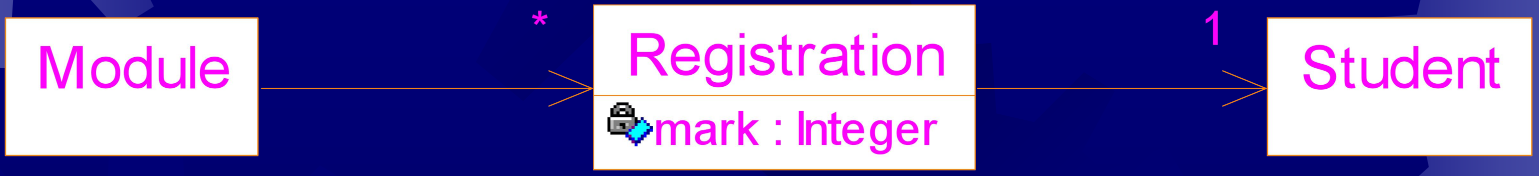
Implementation of association classes

- Solution: Transform the association class into a simple class linked to the two original classes

- The interface of the two original classes should be kept unchanged.

- The implementation changed the meaning of the original class diagram. Further constraints should be imposed on the implementation.


Module
Student
*
Registration
mark : Integer


Module
*
Registration
mark : Integer
1
Student

```
class Registration {
  Student * pStudent;
  int mark; // the attribute of the link
public:
  Registration(Student* st) {
    pStudent = st;   mark=0; }
};
class Module {
  vector<Registration*> registrations;
public:
  void enrol(Student* st) { // interface remains
   registrations.push_back(new Registration(st));
  }
}
```