Which class design is better? [closed]

Asked 16 years, 3 months ago Modified 10 years, 4 months ago

Viewed 17k times

47

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 10 years ago.

Improve this question

Which class design is better and why?

```
public class User
{
    public String UserName;
    public String Password;
    public String FirstName;
    public String LastName;
}

public class Employee : User
{
    public String EmployeeId;
    public String EmployeeCode;
    public String DepartmentId;
}
```

```
public class Member : User
{
    public String MemberId;
    public String JoinDate;
    public String ExpiryDate;
}
```

OR

```
public class User
{
    public String UserId;
    public String UserName;
    public String Password;
    public String FirstName;
    public String LastName;
}

public class Employee
{
    public User UserInfo;
    public String EmployeeId;
    public String EmployeeCode;
    public String DepartmentId;
}

public class Member
{
    public User UserInfo;
    public String MemberId;
    public String JoinDate;
    public String ExpiryDate;
}
```

oop **class-design**

Share

edited Sep 2, 2008 at 4:46

Improve this question

Follow

Sander

26.3k ● 3 ● 54 ● 88

asked Sep 2, 2008 at 4:34

Ramesh Soni

16.1k ● 28 ● 96 ● 117

`User`, `Employee` and `Member` are all the roles in systems. However, the fields composing these classes are intended for and related with many other aspects such as identification, logging, naming and authorization, which are the contexts of usage. Roles can be performed jointly (inclusively) or exclusively. This feature can be expressed by association with the performer rather than via inheritance as no one role means another in common case as they can be performed independently. Inheritance is just a way of composing a class by including features into it. Thus, your question is incorrect. – Aleksey F. Oct 7, 2021 at 0:17

11 Answers

Sorted by: Highest score (default)

63

The question is simply answered by recognising that inheritance models an "IS-A" relationship, while membership models a "HAS-A" relationship.

- An employee IS A user
- An employee HAS A userinfo

Which one is correct? This is your answer.

Share Improve this answer

Follow

edited Sep 2, 2008 at 4:46

answered Sep 2, 2008 at 4:41

1800 INFORMATION

135k ● 30 ● 163 ● 242

54 Sorry, while common, this is a terrible way to reason about design. Human languages are vague and imprecise, programming languages can't afford to be. – CurtainDog Jan 21, 2011 at 0:08

3 I believe it's good practice to favor aggregation over inheritance. I think that makes sense here. – gsgx Jun 26, 2012 at 13:40

4 I'd extend that a bit further to say that an `employee` HAS a `userinfo` *in the context of this domain*. If you find that your `employee` has a `postcode` simply because a `user` does, but for the business purposes of an `employee`, the `postcode` isn't needed (it is forced on you by the contract of a `user`) then composition may be more appropriate than inheritence in such a case. – 8bitjunkie Apr 15, 2014 at 13:16

21

I don't like either one. What happens when someone is both a member and an employee?

Share Improve this answer

Follow

answered Sep 2, 2008 at 4:39

Brad Wilson

70.4k ● 9 ● 77 ● 85

7 If each of the classes implemented interfaces, and you extracted the common attributes out of each one, you could have your new class implement Member and Employee, contain the right instances of the common classes plus the extra ones you need, and delegate. – moffdub Jan 24, 2009 at 1:09

4 Your concern, which is valid, is an argument for the 2nd design, not an argument against both. – Jonah Oct 3, 2015 at 15:44

`public class EmployedMember: Employee, Member`, assuming your language supports multiple ancestors. Of course, trouble starts when both classes use `Id` instead of `EmployeeId` and `MemberId` respectively...
– Tobias Kienzler Feb 16, 2017 at 9:59

Someone who is both a member and an employee is like someone who is both a moderator and an admin. It's not a very bright idea. – bit2shift May 20, 2017 at 14:37

19

Ask yourself the following:

- Do you want to model an Employee *IS* a User? If so, chose inheritance.
- Do you want to model an Employee *HAS* a User information? If so, use composition.
- Are virtual functions involved between the User (info) and the Employee? If so, use inheritance.
- Can an Employee have multiple instances of User (info)? If so, use composition.

- Does it make sense to assign an Employee object to a User (info) object? If so, use inheritance.

In general, strive to model the reality your program simulates, under the constraints of code complexity and required efficiency.

Share Improve this answer

Follow

answered Sep 2, 2008 at 5:03

wilhelmtell

58.6k • 20 • 97 • 131

Could you please explain more parts 3, 4 and 5? Thanks in advance – Peyman Mohamadpour May 30, 2016 at 12:31

14

Nice question although to avoid distractions about *right* and *wrong* I'd consider asking for the pros and cons of each approach -- I think that's what you meant by which is better or worse and why. Anyway ....

The First Approach aka Inheritance

Pros:

- Allows polymorphic behavior.
- Is *initially* simple and convenient.

Cons:

- *May* become complex or clumsy over time *if* more behavior and relations are added.

The Second Approach aka Composition

Pros:

- Maps well to non-oop scenarios like relational tables, structured programing, etc
- Is straightforward (if not necessarily convenient) to *incrementally* extend relations and behavior.

Cons:

- No polymorphism therefore it's less convenient to use related information and behavior

Lists like these + the questions Jon Limjap mentioned will help you make decisions and get started -- then you can find what the *right* answers should have been ;-)

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Sep 9, 2008 at 4:07

maccullt

2,829 ● 1 ● 19 ● 15

It's simple to make the second approach polymorphic. Both member and employee expose a userinfo, this behaviour could be pulled up to a parent. – CurtainDog Jan 20, 2011 at 23:58

Could you please explain "polymorphic behaviour" section of your answer? – Peyman Mohamadpour May 30, 2016 at 12:35

12

I don't think composition is always better than inheritance (just usually). If Employee and Member really are Users, and they are mutually exclusive, then the first design is better. Consider the scenario where you need to access the UserName of an Employee. Using the second design you would have:

```
myEmployee.UserInfo.UserName
```

which is bad (law of Demeter), so you would refactor to:

```
myEmployee.UserName
```

which requires a small method on Employee to delegate to the User object. All of which is avoided by the first design.

Share Improve this answer

Follow

answered Sep 2, 2008 at 5:46

liammclennan

5,368 ● 3 ● 35 ● 31

7

You can also think of Employee as a **role** of the User (Person). The role of a User can change in time (user can become unemployed) or User can have multiple roles at the same time.

Inheritance is much better when there is **real** "is a" relation, for example Apple - Fruit. But be very careful: Circle - Ellipse is not real "is a" relation, because cirlce has less "freedom" than ellipse (circle is a **state** of ellipse) - see: Circle Ellipse problem.

Share Improve this answer Follow

edited Aug 21, 2014 at 19:59

answered Aug 21, 2014 at 16:04

Marcin Raczyński

983 ● 1 ● 10 ● 14

An Apple could have less freedom than a fruit as well. What if the fruit class has a setColor() method? `unknownFruit.setColor('pink')` makes sense, but `bannana.setColor('pink')` does not. – Scotty Jamison Mar 22, 2022 at 17:08

5

The real questions are:

- What are the business rules and user stories behind a user?
- What are the business rules and user stories behind an employee?
- What are the business rules and user stories behind a member?

These can be three completely unrelated entities or not, and that will determine whether your first or second design will work, or if another completely different design is in order.

Share Improve this answer

Follow

answered Sep 2, 2008 at 5:43

Jon Limjap

95.3k ● 15 ● 103 ● 153

4

Neither one is good. Too much mutable state. You should not be able to construct an instance of a class that is in an invalid or partially initialized state.

That said, the second one is better because it favours composition over inheritance.

Share Improve this answer

Follow

answered Sep 2, 2008 at 4:47

Apocalisp

35k ● 8 ● 108 ● 158

What do you mean by first part of your answer? do you mean using Abstract / Interface ? – Peyman Mohamadpour May 30, 2016 at 13:51

1 I mean that any object returned from a constructor should be safe to use and not require further initialization. – Apocalisp May 30, 2016 at 23:11

Stating your requirement/spec might help arrive at the 'best design'.

3

Your question is too 'subject-to-reader-interpretation' at the moment.

Share Improve this answer

Follow

answered Sep 2, 2008 at 7:43

Gishu

137k ● 47 ● 226 ● 311

2

Here's a scenario you should think about:

Composition (the 2nd example) is preferable if the same User can be both an Employee and a Member. Why? Because for two instances (Employee and Member) that represent the same User, if User data changes, you don't have to update it in two places. Only the User instance contains all the User information, and only it has to be updated. Since both Employee and Member classes contain the same User instance, they will automatically both contain the updated information.

Share Improve this answer

Follow

answered Sep 2, 2008 at 7:54

Jonathan

7,401 ● 5 ● 32 ● 36

0

Three more options:

1. Have the `User` class contain the supplemental information for both employees and members, with unused fields blank (the `ID` of a particular `User` would indicate whether the user was an employee, member, both, or whatever).

2. Have an `User` class which contains a reference to an `ISupplementalInfo`, where `ISupplementalInfo` is inherited by `ISupplementalEmployeeInfo`, `ISupplementalMemberInfo`, etc. Code which is applicable to all users could work with `User` class objects, and code which had a `User` reference could get access to a user's supplemental information, but this approach would avoid having to change `User` if different combinations of supplemental information are required in future.

3. As above, but have the `User` class contain some kind of collection of `ISupplementalInfo`. This approach would have the advantage of facilitating the run-time addition of properties to a user (e.g. because a `Member` got hired). When using the previous approach, one would have to define different classes for different combinations of properties; turning a "member" into a "member+customer" would require different code from turning an "employee" into an "employee+customer". The disadvantage of the latter approach is that it would make it harder to guard against redundant or inconsistent attributes (using something like a `Dictionary<Type, ISupplementalInfo>` to hold supplemental information could work, but would seem a little "bulky").

I would tend to favor the second approach, in that it allows for future expansion better than would direct inheritance. Working with a collection of objects rather

than a single object might be slightly burdensome, but that approach may be better able than the others to handle changing requirements.

Share Improve this answer Follow

edited Nov 26, 2013 at 17:28

answered Oct 29, 2012 at 15:36

supercat

80.8k ● 9 ● 174 ● 220

Could you please add some simple diagram to represent your solution? – Peyman Mohamadpour May 30, 2016 at 13:55