Design Class Diagrams

Lecture 8

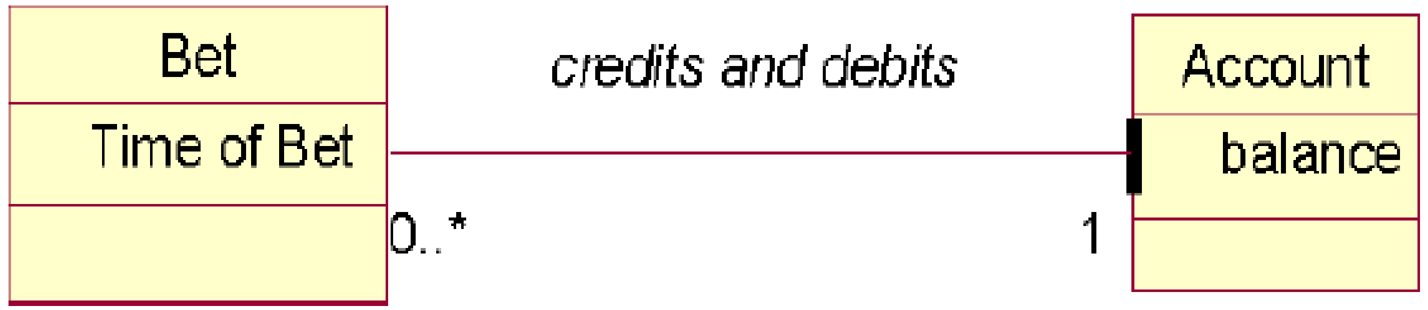
Recall that at the elaboration stage, we built a conceptual model. The conceptual model contained details about the customer's problem, and concentrated on the customer's concepts, and the properties of those concepts. We did not allocate behaviour to any of those concepts.

Now that we have begun to create collaboration diagrams, we can progress the conceptual model, and build it into a true **Design Class Diagram**. In other words, a diagram which we can base our final program code upon.

Producing the design class diagram is a fairly mechanical process. In this chapter, we'll examine an example Use Case, and how the conceptual model is modified because of it.

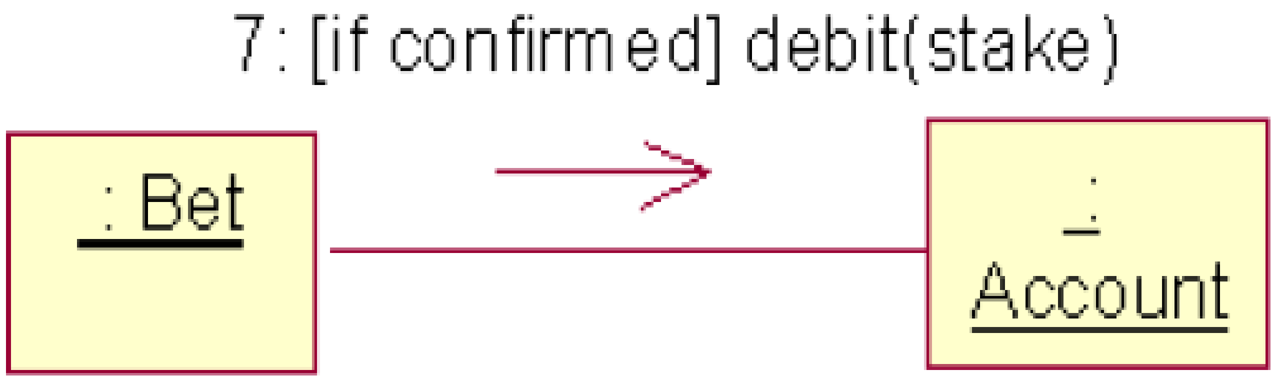
Crediting and Debiting Accounts

At the end of the “place bet” Use Case, the “bet” object sends a message to the customer’s “Account” object, to tell it that it must be decremented. The following conceptual model was the basis of this design:



Conceptual Model for this example

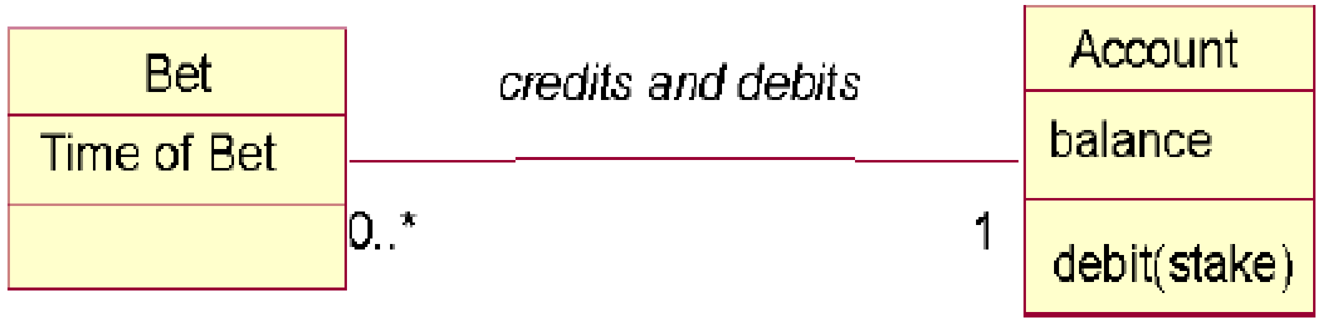
From the conceptual model, the following (portion of a) collaboration was developed:



Part of the Collaboration for "Place Bet"

Step 1 : Add Operations

From the collaboration diagram, we can see that the “Account” class needs to provide the behaviour “debit”. So we add this operation to the lower half of the class symbol.

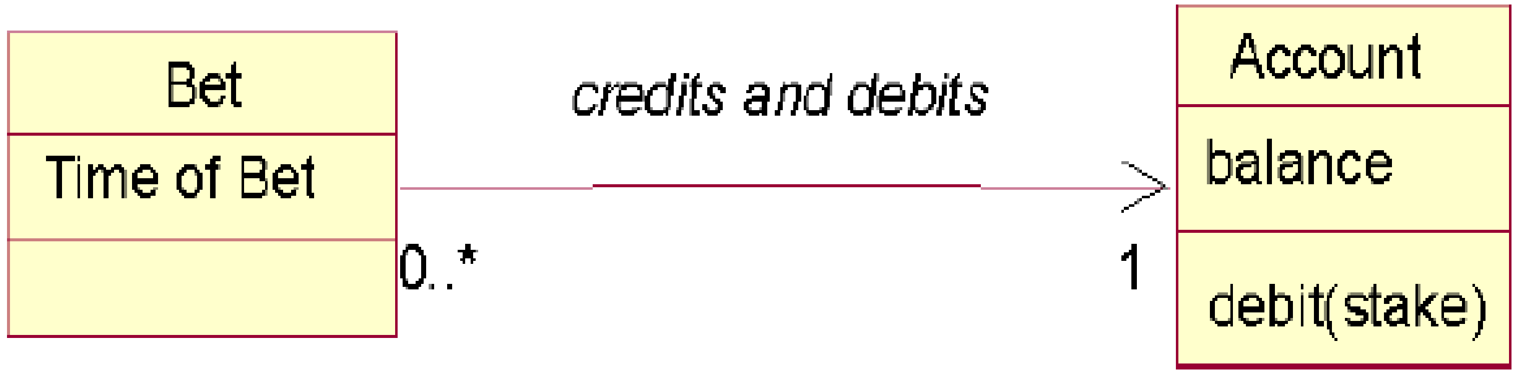


Classes with Operation Added

Note that most people don’t bother to add the *create* operations, as this will clutter the diagram (most classes need one anyway).

Step 2 : Add Navigability

The direction of the messages which are being passed through an association are also added. In this case, the message is being sent from the bet class to the account class, so we orient the association from the caller to the receiver:



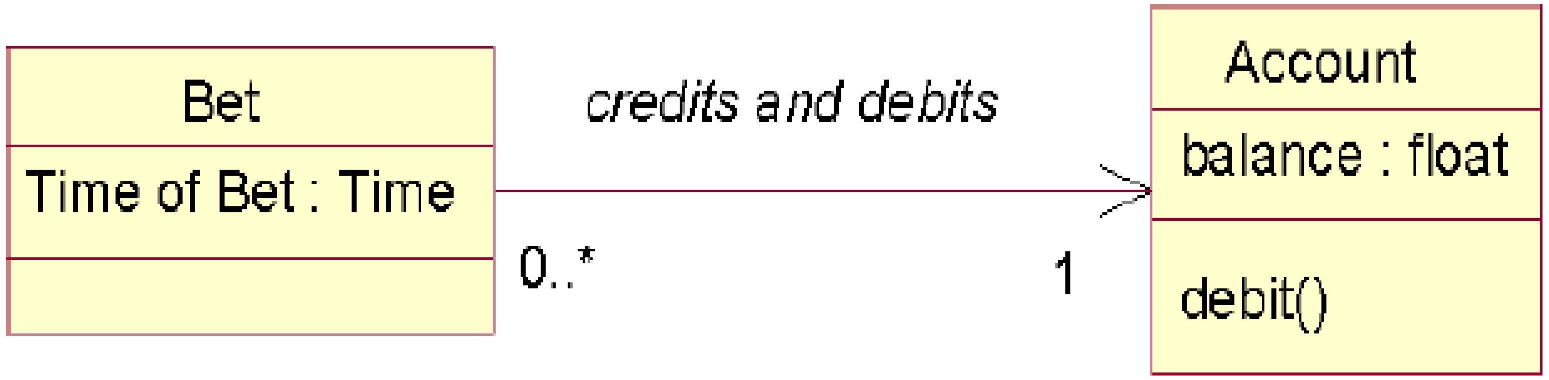
Account Class with Navigability Added

Sometimes, a situation will arise where messages need to be passed both ways across an association. What to do in this case? The UML notation for this situation is to simply leave the arrow head off the association – a **bi-direction association**.

Many modellers believe that bi-directional associations are erroneous and need to be removed from the model somehow. In actual fact, there is nothing fundamentally wrong with a bi-directional relationship, but it does *suggest* a bad design. We’ll explore this problem in a later chapter.

Step 3 : Enhance Attributes

We can also decide upon the datatype of the attributes at this stage. Here, we have decided to store the balance of an account as a **float**. Of course, this is dependent upon the choice of language.



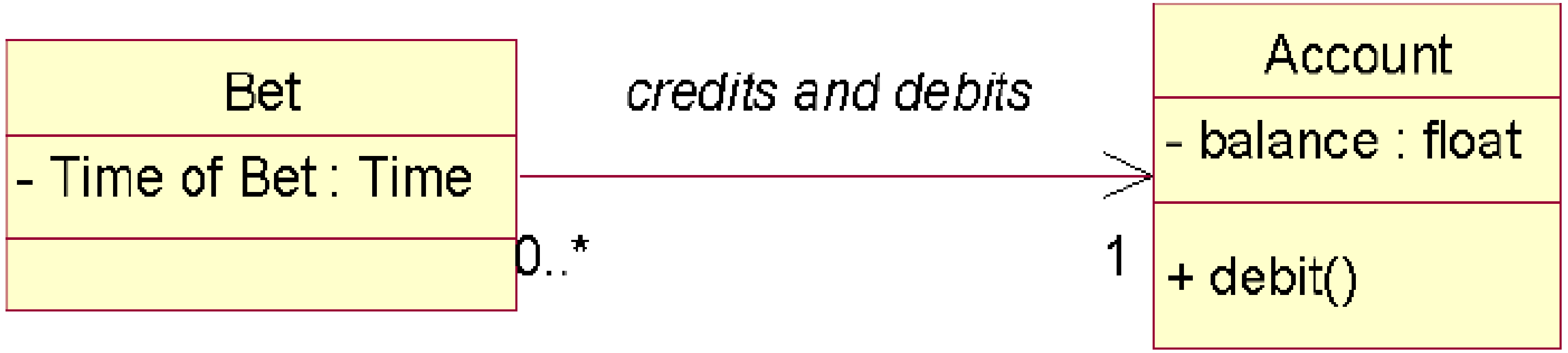
Datatypes added

Step 4 : Determine Visibility

A fundamental concept of Object Orientation is encapsulation – the idea that the data held by an object is kept private from the outside world (ie from other objects).

We can signal which attributes and operations are public or private on a UML Class diagram by preceding the attribute/operation name by a plus sign (for public) and a minus sign (for private).

All attributes will be private, unless there is an extremely good reason (and there rarely is). Usually, operations will be public, unless they are helper functions, only to be used by operations contained within the class.



The Class Diagram, complete!

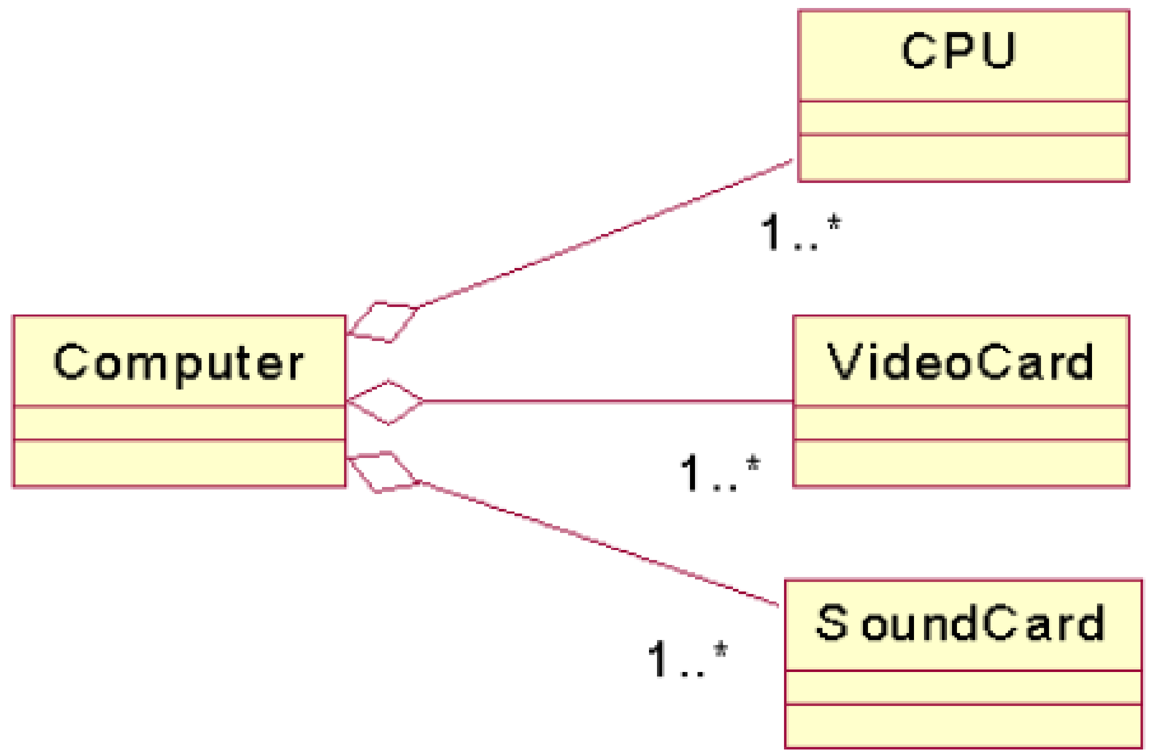
Now that the class diagram is complete, we now have enough information to produce the code. We'll examine the transition to code in a later chapter.

Aggregation

An important aspect of object oriented design is the concept of **Aggregation** – the idea that one object can be built from (aggregated from) other objects.

For example, in a typical computer system, the computer is an aggregation of a CPU, a graphics card, a soundcard and so on.

We can denote aggregation in UML using the aggregation symbol – a diamond at the end of an association link.



Computer built from other objects

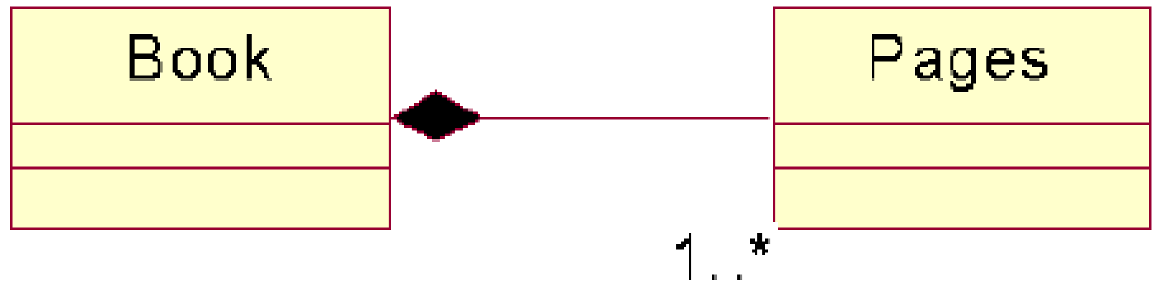
If you spot aggregation on your conceptual model, it may be clearer to explicitly notate the fact, using the aggregation symbol.

Composition

A very similar concept to aggregation is composition. Composition is stronger than aggregation, in the sense that the relationship implies that the whole cannot exist without the parts.

In the aggregation example above, for example, if we removed the soundcard, the Computer would still be a computer. However, a book isn’t a book without its pages, so we say that a book is **composed of** the pages.

The notation for this is similar to aggregation, except this time the diamond is filled, as follows:



A book is composed of 1 or more pages

Finding Aggregation and Composition

Finding these relationships on your class diagram is useful, but it is far from crucial to the success of your design. Some UML practitioners go further, and claim that these relationships are redundant, and should be removed (aggregation and composition can be modelled as an association with a name like “is composed from”).

I believe, however, that as aggregation is one of the key concepts of Object Orientation, it is definitely worth explicitly notating its presence.

Summary

In this chapter, we looked at how to progress the class model, based on our work on collaborations. The transition from conceptual to design class model is actually fairly trivial and mechanical, and shouldn’t cause too many headaches.

Responsibility Assignment Patterns

In this section, we are going to take time out from the development process, and look closely at the skills involved in building good Object Oriented Designs.

Some of the advice given in this chapter may appear to be quite obvious and trivial. In fact, it is the violation of these simple guidelines that causes most of the problems in object oriented design.

The GRASP Patterns

To improve the way in which we produce our collaboration diagrams, we'll study the so-called "GRASP" patterns, as described by Larman

What is a pattern?

A pattern is a well used, extremely general, solution to a common occurring problem. The pattern movement began as an internet-based discussion community, but was popularised through the classic textbook "Design Patterns" , written by the so called "Gang of Four".

To aid communication, each design pattern has an easy to remember name (such as Factory, Flywheel, Observer), and there are at least a handful of design patterns that every self respecting designer should be familiar with.

We’ll be looking at some of the “Gang of Four’s” patterns later, but first we’ll study the GRASP patterns.

GRASP stands for “General Responsibility Assignment Software Patterns”, and they help us to ensure we allocate behaviour to classes in the most elegant way possible.

The patterns are called **Expert, Creator, High Cohesion, Low Coupling** and **Controller**. Let’s look at them in turn:

Grasp 1 : Expert

This is, on the face of it, a very simple pattern. It is also the one which is most commonly broken! So, this pattern should be at the forefront of your mind whenever you are building collaboration diagrams or creating design class diagrams.

Essentially, the Expert pattern says “given a behaviour, which class should the behaviour be allocated to?”.

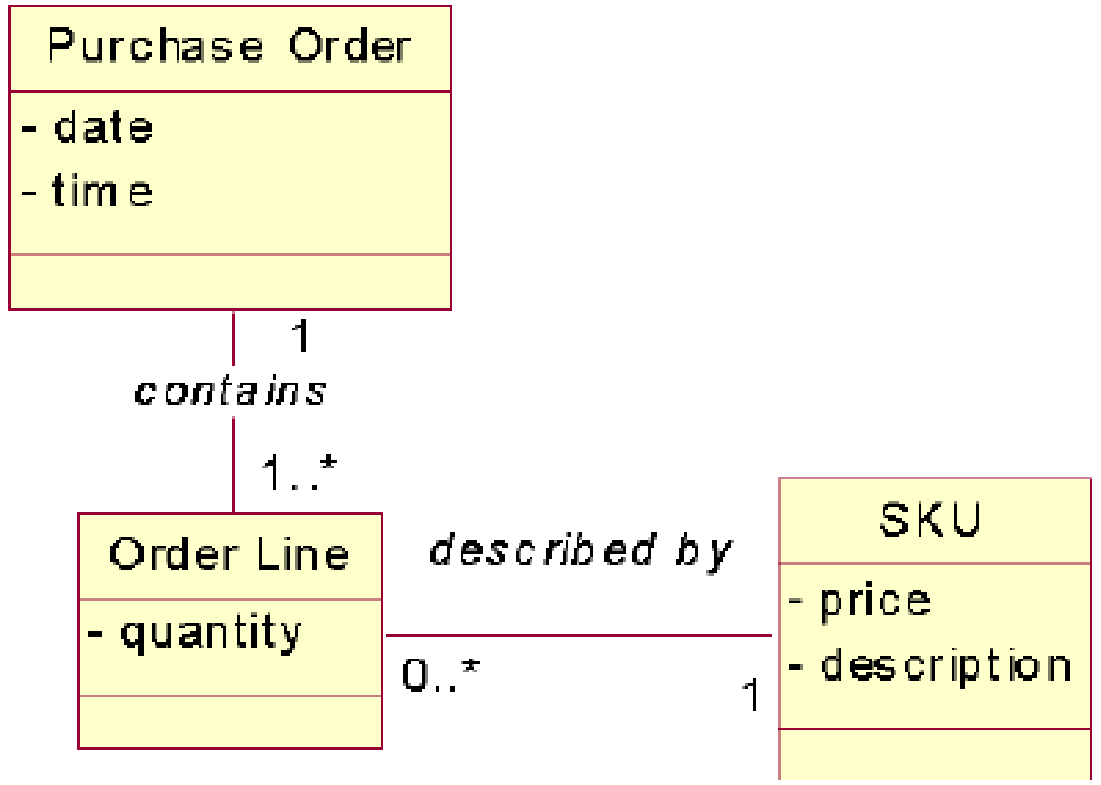
Wise allocation of behaviour leads to systems which are:

- Easier to understand
- More easily extendible
- Reusable
- More Robust

Let's look at a simple example. We have three classes, one representing a Purchase Order, one for an Order Line, and finally, one for a SKU

(Stock Keeping Unit)

Here is a fragment from the conceptual model:



Fragment from the conceptual model

Now, imagine that we are building collaboration for one of the use cases. This use case demands that the total value of a selected purchase order is presented to the user. Which class should be allocated the behaviour called “calculate_total()”?

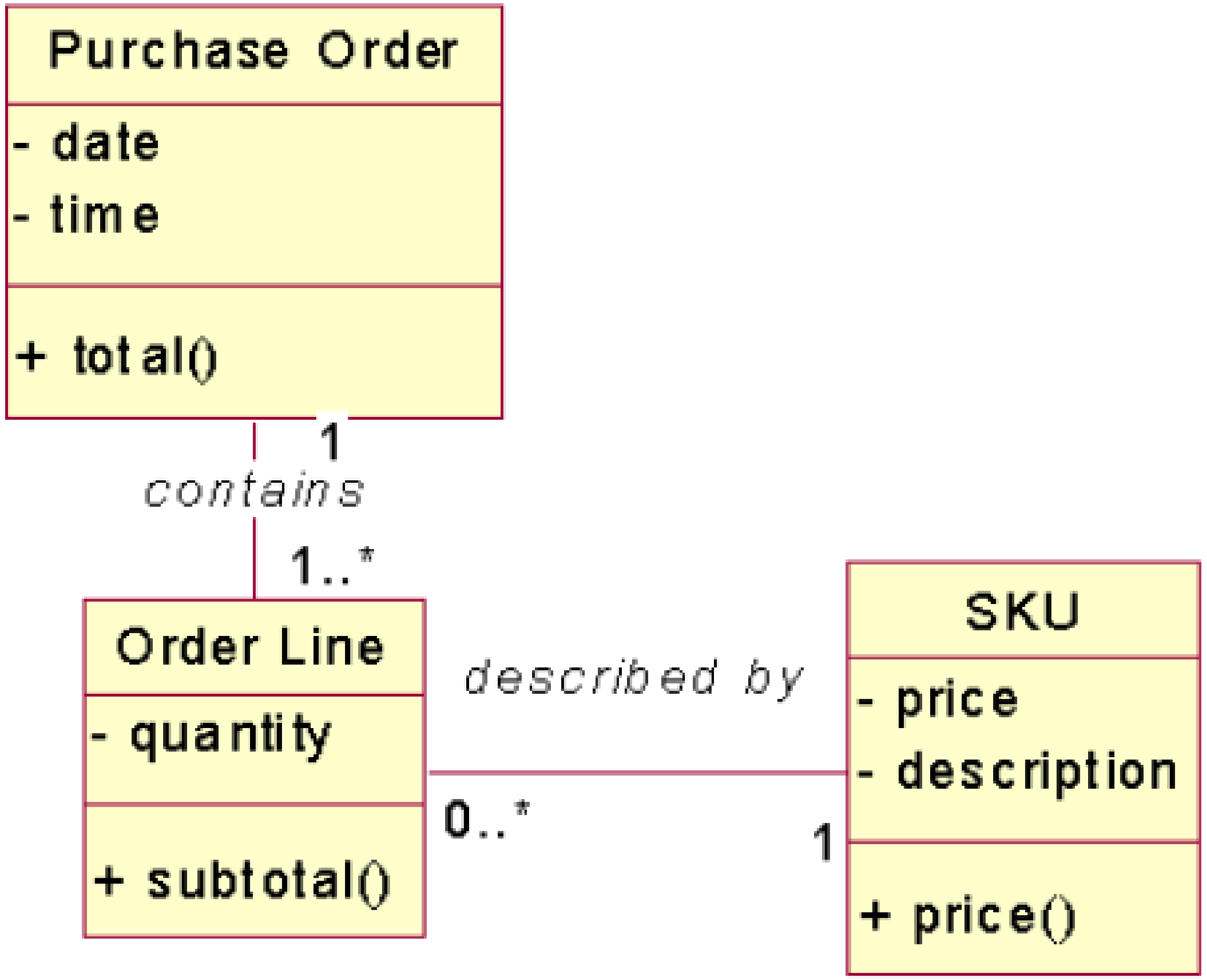
The expert pattern tells us that the only class who should be allowed to deal with the total cost of purchase orders is the purchase order class itself – because that class should be an expert about all things to do with purchase orders.

So, we allocate the “calcuate_total()” method to the Purchase Order class.

Now, to calculate the total of a purchase order, the purchase order needs to find out the value of all of the order lines.

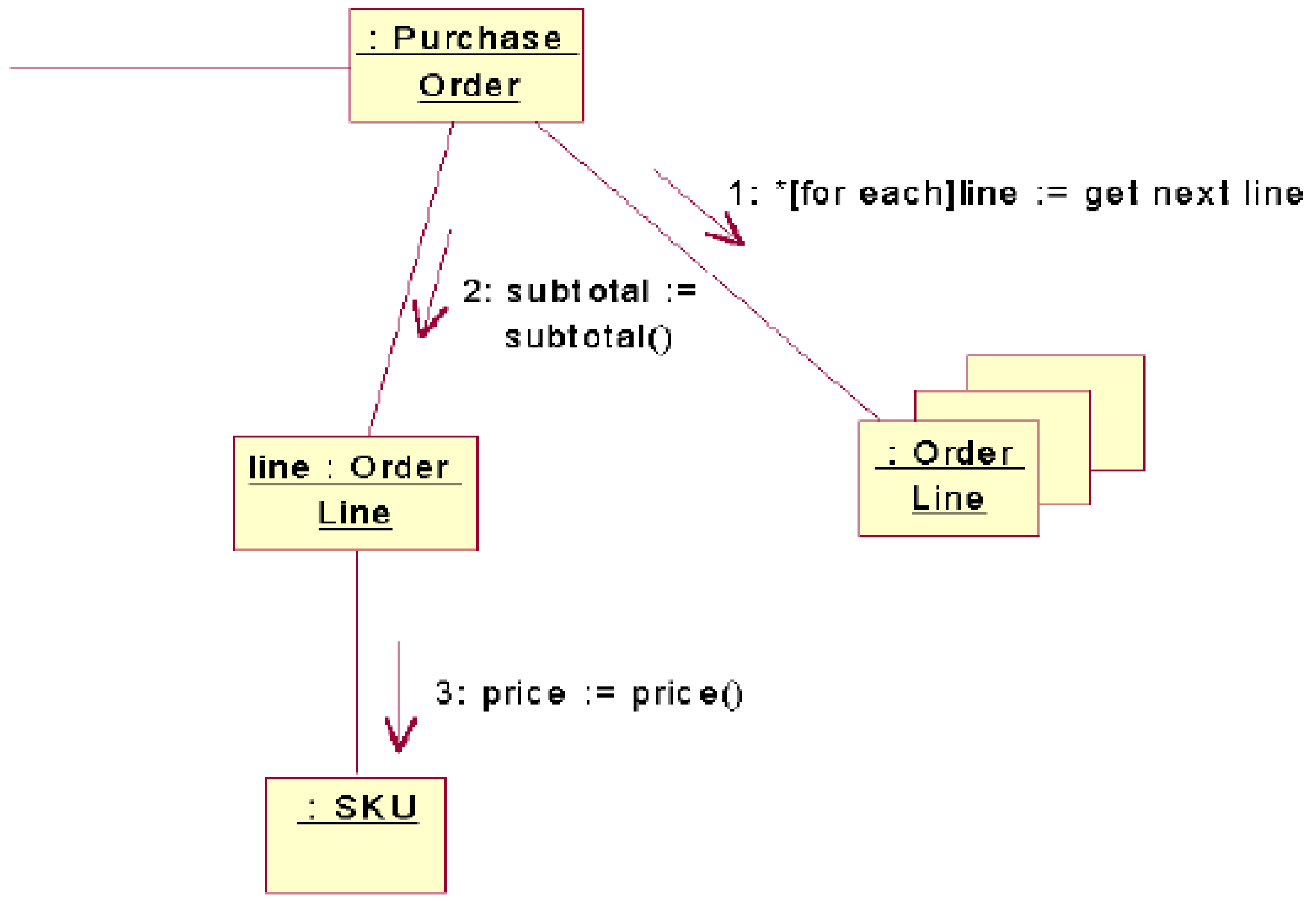
A poor design for this would be to let the purchase order see the contents of every order line (via accessor functions), and then sum up the total. This is breaking the expert pattern, because the only class who should be allowed to calculate the total of an order line is the order line class itself.

So we allocate a further behaviour to the order line class, called subtotal(). This method returns the total cost of the single order line. To achieve this behaviour, the order line class needs to find out the cost of a single SKU through another method (this time, an accessor) called price() in the SKU class.



Behaviours allocated by observing the expert pattern

This leads to the following collaboration diagram:



three classes of objects collaborating to provide the total cost of a purchase order

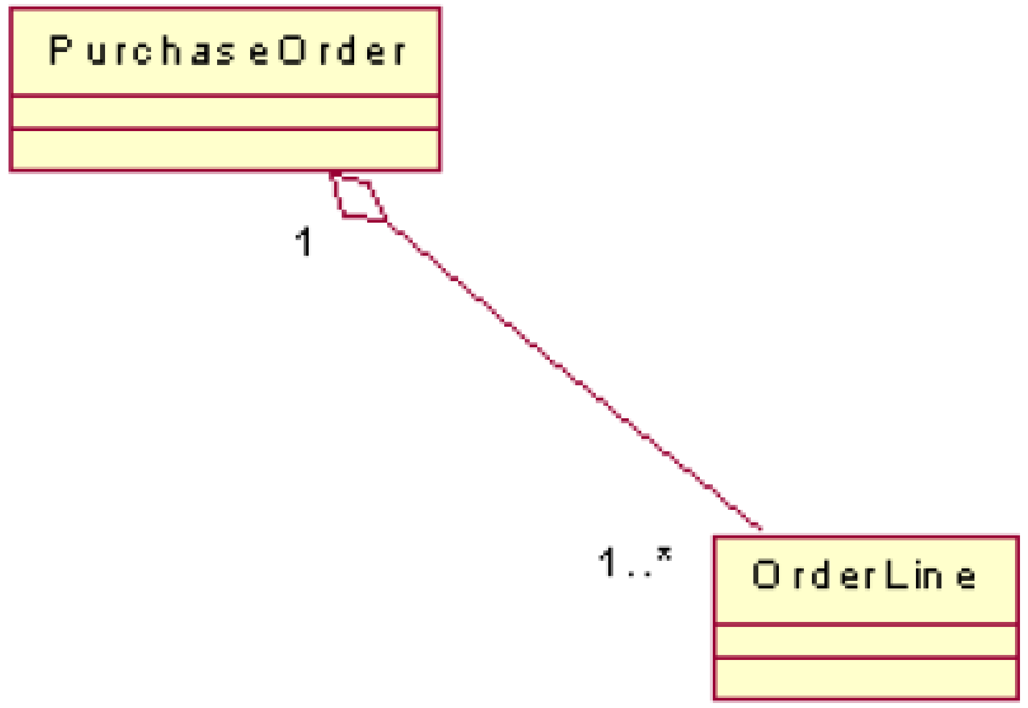
Grasp 2 : Creator

The Creator pattern is a specific application of the Expert pattern. It asks the question "who should be responsible for creating instances of a particular class?"

The answer is that Class A should be responsible for creating objects from Class B if:

- A Contains B Objects
- A *closely uses* B Objects
- A *has the initialising data* that will be passed to B Objects

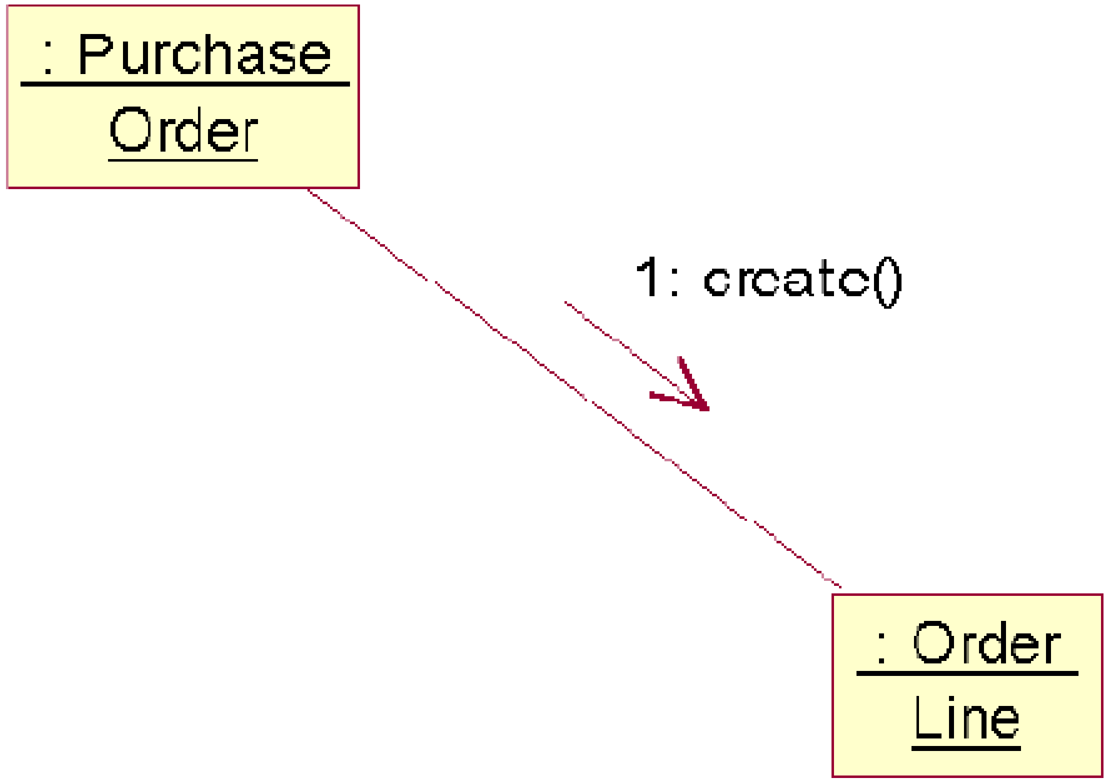
For example, lets return to the purchase order example. Let’s say that a new purchase order has been created. Which class should be responsible for creating the corresponding purchase order lines?



Which class should create purchase orders?

The solution is that, as a purchase *contains* purchase order lines, then the purchase order class (and only that class) should be responsible for creating order lines.

Here is the collaboration diagram for this situation:

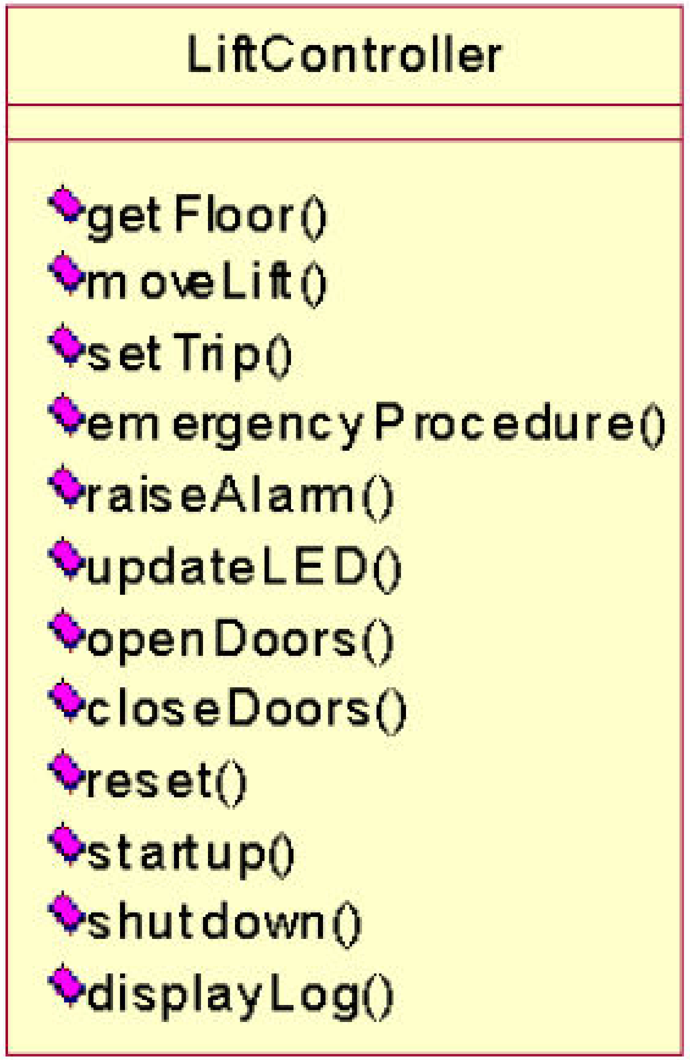


The Purchase Order creating Order Lines

Grasp 3 : High Cohesion

It is extremely important to ensure that the responsibilities of each class are focussed. In a good object oriented design, each class should not do too much work. A sign of a good OO design is one where every class has only a small number of methods.

Consider the following example. In the design for a Lift Management system, the following class has been designed:



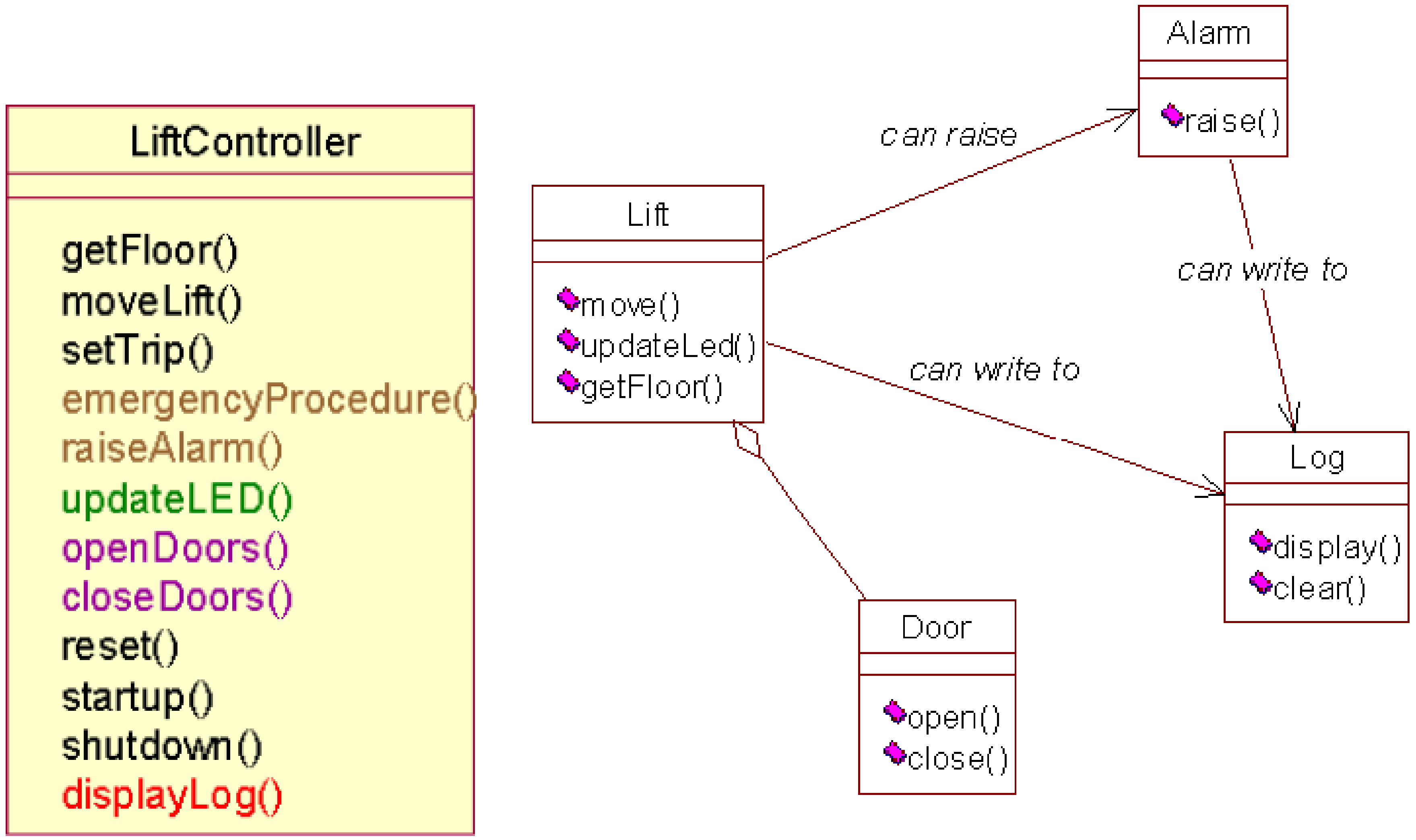
A class from the Lift Management System

Is this a good design? Well, the class obviously does a lot of work – raising alarms, starting up/shutting down, moving the lift and updating the display indicator. This is a bad design because the class is not cohesive.

This class would be difficult to maintain, as it isn't obviously clear what the class is supposed to be doing.

The rule of thumb to follow when building classes is that each class should capture only one **key abstraction** – in other words, the class should represent one "thing" from the real world.

Our Lift controller is trying to model at least three separate key abstractions – an Alarm, the lift Doors and the Fault Log. So a better design is to break the Lift Controller into separate classes.



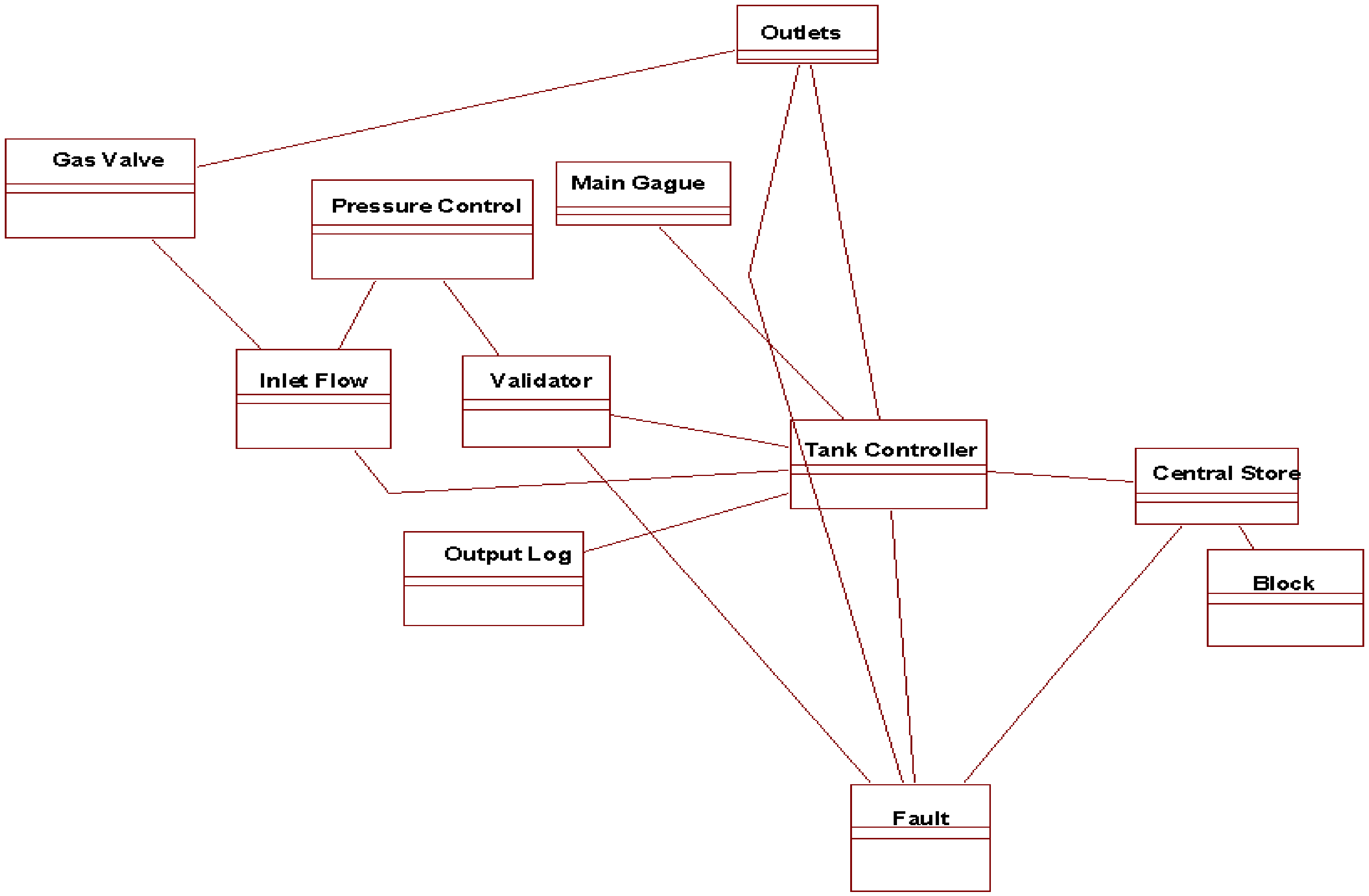
The Lift Controller class modelled as four separate, more cohesive, classes

Grasp 4 : Low Coupling

Coupling is a measure of how dependent one class is on other classes. High Coupling leads to code that is difficult to change or maintain – a single to change to just one class could lead to changes "rippling" throughout the system.

The Collaboration Diagram provides an excellent means for spotting coupling, and as a result, high coupling can be also be spotted through the Class Diagram.

The following is an example extract from a Class Diagram that is showing clear signs of high coupling:



High Coupling in a Class Diagram

Are all of the associations in Figure really necessary? The designer of this system should ask some serious questions about the design. For example:

- Why is the Fault class associated to the Outlets class directly, when an indirect association exists through the Tank Controller Class? This link may have been put in place for performance reasons, which is fine, but more likely the link has appeared due to sloppiness in the Collaboration Modelling
- Why does Tank Controller have so many associations? This class is probably incohesive and doing too much work.

Following the conceptual model is an excellent way of reducing coupling. Only send a message from one class to another class if an association was identified at the conceptual modelling stage. This way, you are restricting yourself to introducing coupling only if the customer agreed that the concepts are coupled in real life.

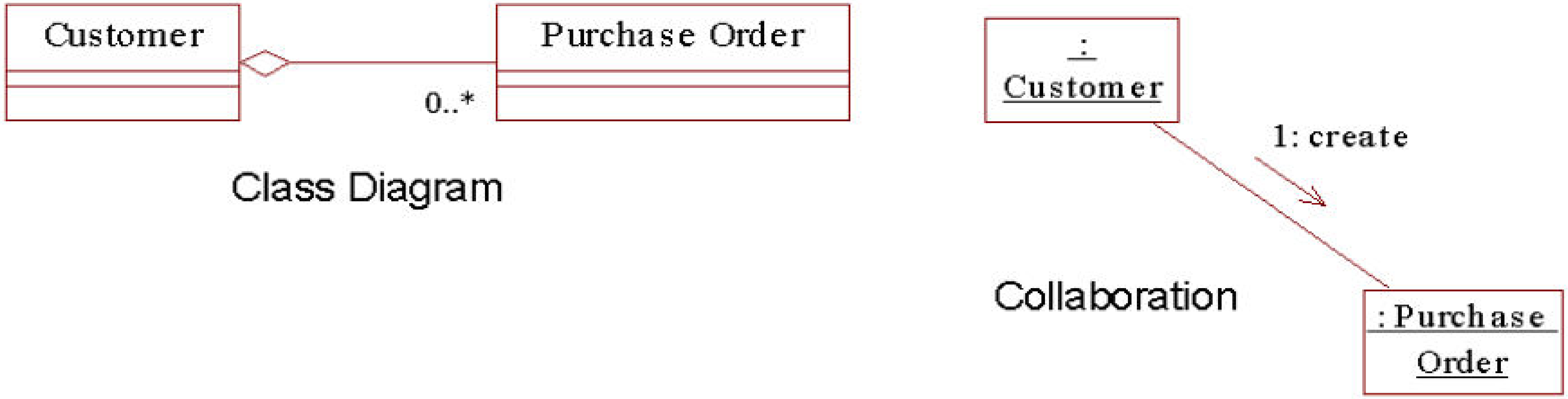
If, at the Collaboration stage, you realise that you wish to send a message from one class to another and they are NOT associated on the conceptual model, then ask a very serious question about whether or not the coupling exists in the real world. Talking to the customer may help here - perhaps the association was overlooked when you built the conceptual model.

So the rule of thumb here is : **Keep coupling to a minimum - the conceptual model is an excellent source of advice on what that minimum should be. It is fine to raise the level of coupling, as long as you have thought very carefully about the consequences!**

Worked Example

Let us look at a Purhcase Ordering System. In the conceptual model, it was identified that Customers own Purchase Orders (because they raise them):

For the "Create Purchase Order" Use Case, which class should be responsible for creating new purchase orders? The Creator Pattern suggests that the Customer Class should be responsible:

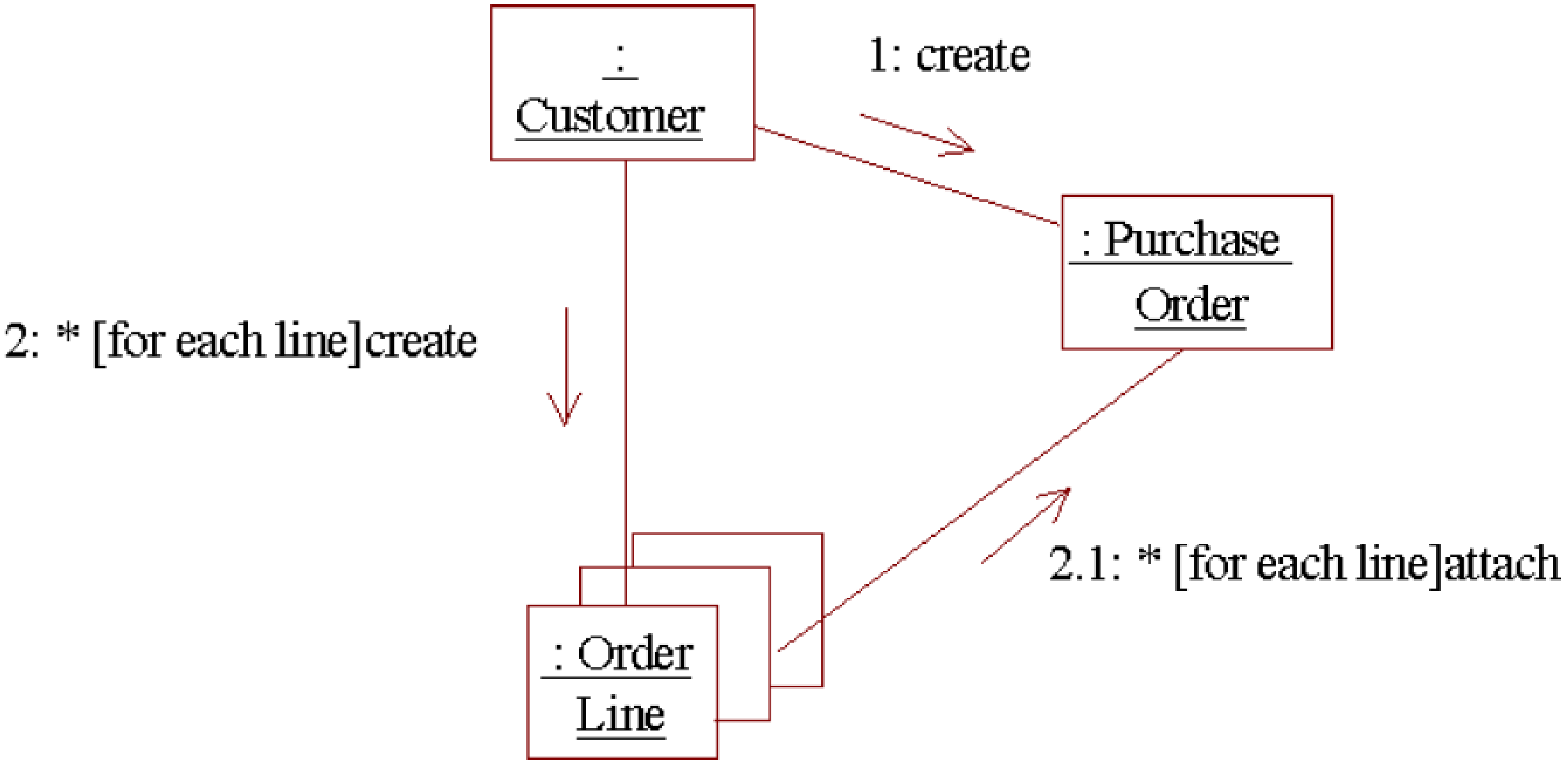


Class Diagram and Collaboration for "Create Purchase Order"

So, we have now coupled Customer and Purchase Order together. That is fine, because they are coupled together in real life too.

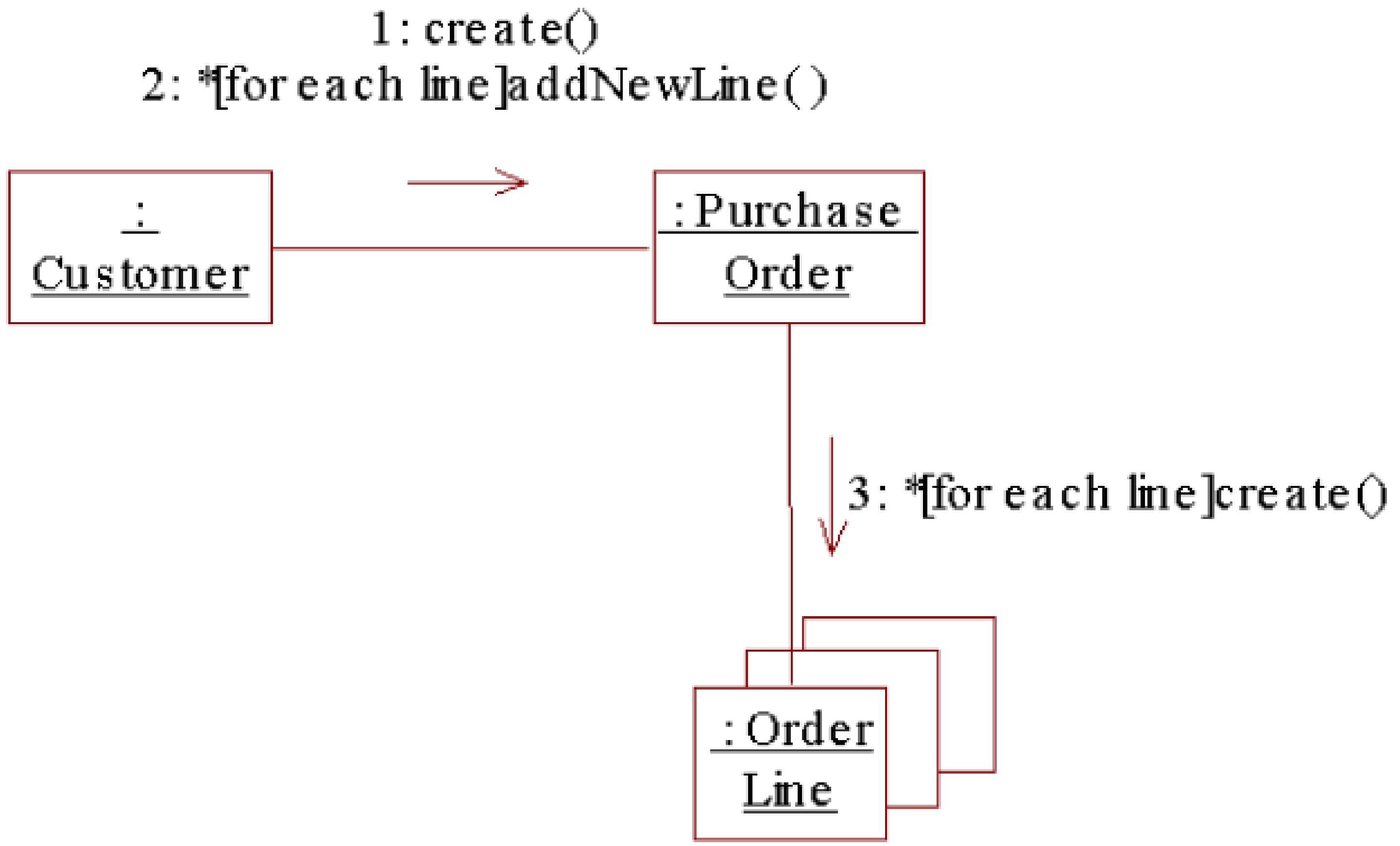
Next, once the Purchase Order has been created, the Use Case needs to add lines to the Order. Who should be responsible for adding lines?

One approach is to let the Customer class do the work (after all, it does have the required initialising data - how many lines, what products, what quantities, etc).



First attempt. The customer objects creates the lines because it holds the initialising data

This approach has raised coupling artificially however. We now have all three class dependent upon each other, whereas if we had made the Purchase Order responsible for creating the lines, we would have the situation where Customers have no knowledge of Order Lines:



Adding Lines, with reduced coupling

So now, if the implementation of the Order Line class changed for any reason, the only class affected would be the Purchase Order Class. All the coupling that exists on this design was coupling that was identified at the conceptual stage. Customers own Purchase Orders; Purchase Orders own Lines. So it makes sense that this coupling should exist!

The Law of Demeter

This Law, also known as *Don't Talk to Strangers* is an effective method of combating coupling. The Law states that any method of an object should only call methods belonging to:

- Itself
- Any parameters that were passed in to the method
- Any objects it created
- Any directly held component objects

Make your objects "shy" and coupling will be reduced!

Final Words on Coupling

Some more issues to consider:

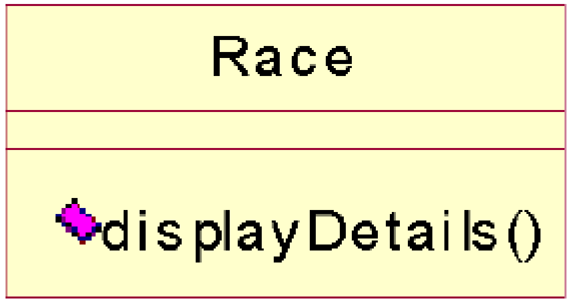
- **Never** make an attribute of a class public - a public attribute instantly opens up the class to abuse (the exception is constants held by the class)
- Only provide get/set methods when strictly necessary
- Provide a minimum public interface (ie only make a method public if it has be accessed by the outside world)
- Don’t let data flow around the system - ie minimise data passed as parameters
- Don’t consider coupling in isolation - remember High Cohesion and Expert! A completely uncoupled system will have bloated classes doing too much work. This often manifests itself as a system with a few “active objects” that don’t communicate.

Grasp 5 : Controller

The final GRASP pattern we will look at in this section is the controller pattern. Let's return to the bookmaking system, and return to the Place Bet Use Case. When we built the collaborations for this Use Case (see page 72), we realised that we hadn't really considered how the input would be entered by the user, and how the results are displayed to the user.

So, for example, we need to display the details of a race to the user. Which class should be responsible for satisfying this requirement?

Application of the **Expert** pattern suggests that as the details pertain to a Race, then the Race class should be the expert in displaying the relevant details.



Should "Race" be responsible for displaying its details?

This sounds ok at first, but actually, we have **violated** the expert pattern! The Race class should indeed be an expert at everything to do with races, but it must not be an expert on other matters – like Graphical User Interfaces!

In general, adding information about GUI's (and databases, or any other physical object) into our classes is poor design – imagine if we have five hundred classes in our system, and many of them read and write to the screen, perhaps to a text based console. What would happen if the requirement changed, and we wanted to replace the text-based screen for a windows-based GUI? We would have to trawl through all of our classes, and laboriously work out what needs to be altered.

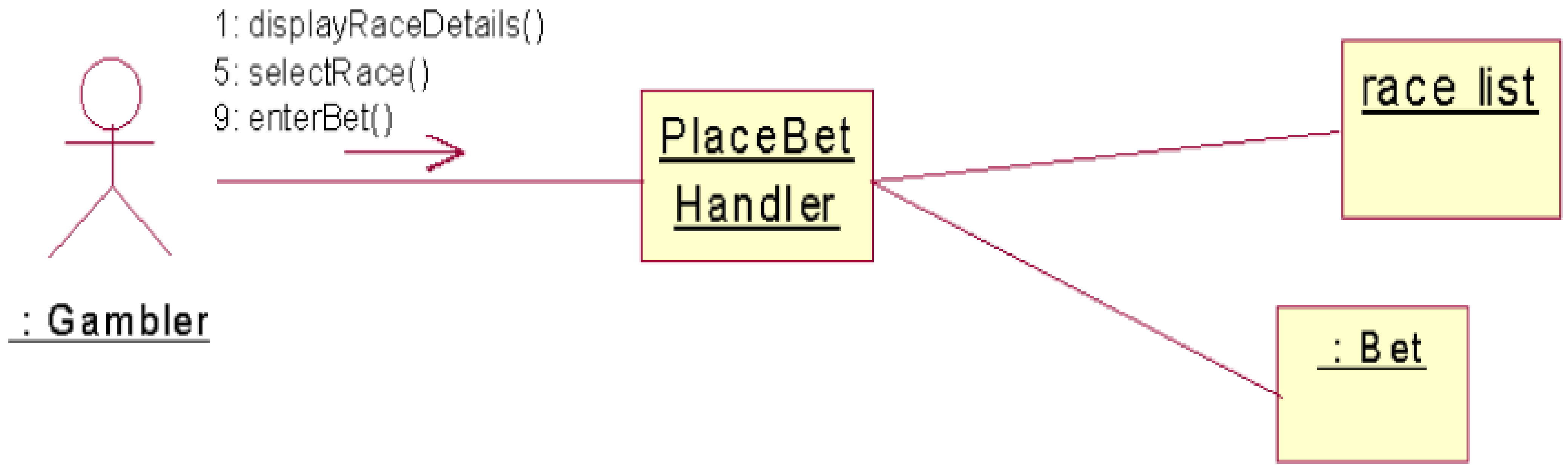
It is much better to keep all of the classes from the conceptual model (I'll call them "Business Classes") pure, and remove all reference to GUI's, Databases and the like. But how our Race class can display the race details?

Solution – Controller Pattern

One possible solution is the use of a Controller Pattern. We can introduce a new class, and make it sit between the actor and the business classes.

The name of this controller class is usually called <UseCaseName>Handler. So in our case, we need a handler called “PlaceBetHandler”.

The handler reads the commands from the user, and then decides which classes the messages should be directed to. The handler is the only class that will be allowed to read and write to the screen.



Use Case Controller added to the design

In the event of us needing to replace the user interface, the only classes we have to modify are the controller classes.

Summary

In this chapter, we explored the “GRASP” Patterns. Careful application of the five GRASP patterns leads to more understandable, modifiable and robust Object Oriented Designs.