

Chapter 13

When and How to Use Inheritance

The mechanism of inheritance is provided in some form by most modern programming languages. However, its improper use can lead to brittle, unnecessarily inflexible architectures that sacrifice encapsulation for little or no gain. This should not, however, lead a developer to conclude that inheritance is bad, or even that it should be used in a minimal, last-resort way. The real question is: what is inheritance good for, and when should it be used?

The Gang of Four

In their seminal book on Design Patterns¹, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (who are often affectionately referred to as the “Gang of Four”) issued several important pieces of general advice on software design. In a sense, each pattern can be thought of as, among other things, examples of following this advice when presented with a particular sort of problem or problem domain.

One such piece of advice was “favor aggregation over inheritance”. Sometimes the word “aggregation” is replaced with “composition”, or even “delegation”, but the implication is pretty clear: don’t inherit, instead delegate from one object to another.

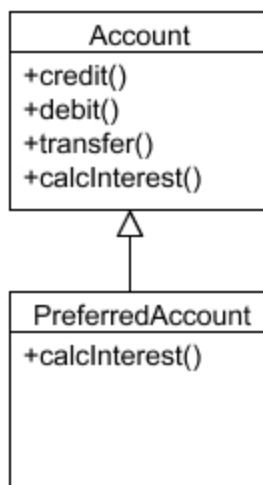
As an example, let’s consider the following design decision: We have a class that represents a bank account. We’ll leave most of the implementation out, but you can imagine the likely interface of such an entity (debit, credit, transfer, etc...). As part of its responsibility, however, it must apply an algorithm for calculating the interest to be paid on the account.

Let’s say the actual algorithm varies, depending on whether this is standard bank account, or one that belongs to a “preferred customer”. Most likely, the preferred customer is given a higher rate.

A fairly traditional approach would be to take the Account class, and add a method to it, called “calcInterest()” or something similar. In the very early days of object orientation (OO), we would have considered it reasonable to then “reuse”² the Account class by subclassing it as “PreferredAccount” and overriding the calcInterest() method with the preferred Algorithm:

¹ Gamma, Eric, et al. Design Patterns: Elements of Reusable Object - Oriented Software. Reading, MA, Addison-Wesley Professional, 1995

² In fact, there are many who have said that the value of OO is precisely this form of reusing existing objects.

**Figure 13.1***Use of Direct Inheritance*

This is what is often termed “direct inheritance” or “inheritance for specialization”. It violates the advice of the Gang of Four. The problem is that any change one might make to Account can affect PreferredAccount, even if this is not desired, because they are coupled through inheritance.

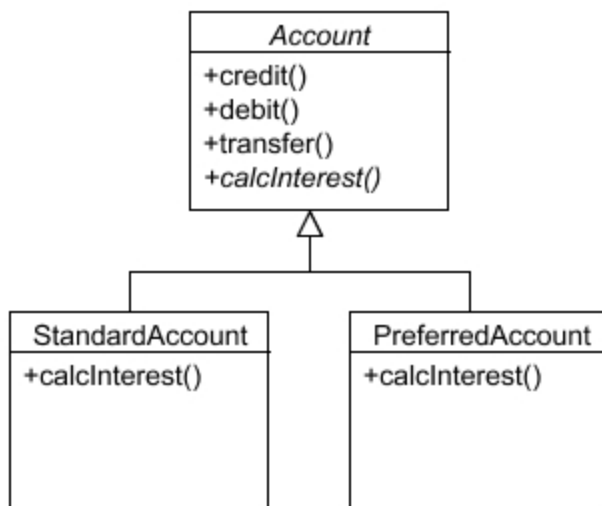
Interestingly, the tendency to use inheritance this way comes from the original notion of “Open-Closed”. Even today we’d say, as a principle, that we would like our systems to be “open for extension, but closed to modification” because we much prefer accommodating change by writing new code, rather than changing existing code. This notion of direct inheritance would seem to follow this principle, because we’ve created this new PreferredAccount and have left the existing code in Account alone entirely. Furthermore, if we upcast PreferredAccount to Account wherever it is used, we won’t have to change that using code either.³

Nevertheless, this approach can give us problems.

What if we need to make some sort of code change to Account that is not intended to also change PreferredAccount? The structure in Figure 3.3.1 requires care and investigation to ensure that PreferredAccount does not accept the unwanted change, but rather overrides or shadows it. This issue of “unwanted side-effects” can be even more complicated if the language makes a distinction between virtual (late-bound) and non-virtual (early-bound) methods. Therefore, the more often inheritance is used in this way, the more difficult and dangerous it is to make changes.

However, inheritance can be used in a slightly different way to avoid much of this:

³ It is also interesting to note that the keyword for inheritance in Java, which was invented in 1995 when this form of inheritance was quite popular, is “extends.”

**Figure 13.2***Use of Abstract Inheritance*

The advantage here is one of control. If a developer wishes to make a change that *could only* affect the StandardAccount, the change is made to that class and we note there is no coupling between it and PreferredAccount. The same is true in the reverse; a change to PreferredAccount *can only* affect that one single class.

If, on the other hand, the developer in fact wishes to make a change that affects them both, then the change would be made to Account. The developer is in the driver’s seat, and it’s much clearer to everyone which changes will affect which entities.

Still, while this is certainly better, is it good enough? If we want the effort we make to create software (which is the major cost of software in most cases) to continue to deliver value for a long time into the future, we have to pay attention not only to what a design is, but also what it is likely to become.

Initial Vectors, Eventual Results

The way you begin can have a big influence on where you end up. In space flight, they call this “the initial vector”, and engineers are very circumspect about it. When a spacecraft is leaving earth’s orbit and heading to the moon, for instance, if the angle of thrust is off by a fraction of a percent you will find yourself hundreds of miles off-target by the time you reach your destination. The initial vector change is small, but the difference in the eventual result is large and significant.

Design decisions can be like this.

In other words, the larger point here is about what we do *now* vs. what we will be *able to do later* and how well early solutions scale into later ones.

For example, the design in figure 3.3.2 is not very dynamic. If we instantiate StandardAccount, we are committed to the calcInterestRate() implementation it contains. If we wanted, at runtime, to switch to the preferred algorithm, we will have to build an instance of PreferredAccount, and transfer all the state that is currently held by our instance of StandardAccount into it. That means the state of StandardAccount will have to be exposed (breaking encapsulation), or we’ll have to give StandardAccount a way to create an instance of PreferredAccount (and vice versa), passing the needed state through its constructor. These would

be known as “cloning methods” and would couple these subtypes to each other, which we were trying to avoid in the first place.

The business rules before us *today* might say “standard accounts never change into preferred accounts”, and so we might think we’re okay. Then *tomorrow* the bank holds an unexpected marketing promotion that requires this very capability. This sort of thing happens all the time.

Another example: in the problem currently before us, we have a single varying algorithm, the one which calculates interest. What if, at some point in the future, some other aspect of Account starts to vary? This is an unpredictable⁴ thing, and so it’s certainly credible that a non-varying issue today could be a varying one tomorrow.

For instance, let’s say the transfer mechanism also contains a variation. The rules have changed, and now there are two versions; immediate transfer, and delayed transfer. If we try to scale the non-preferred solution above, we’d likely do something like this:

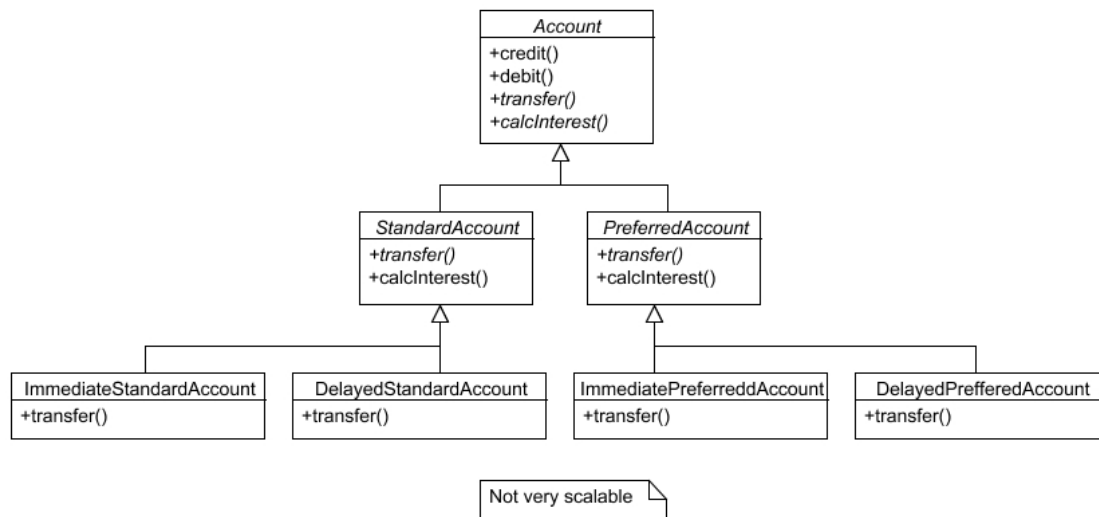


Figure 13.3
Attempting to scale the solution

All of the problems we had initially have become worse. We cannot make either varying issue dynamic (runtime-changeable) without degrading coupling or encapsulation issues, we’ve created more coupling through all the inheritance, and so forth.

Furthermore, given the notion of the Separation of Concerns (See the summary of Section 2), we note that the farther down the inheritance hierarchy we go, the more concerns are becoming coupled together inside single classes. When we began, Account was about one thing: modeling a bank account. Then, PreferredAccount and StandardAccount were each about two things: modeling a bank account, and varying the interest calculation in a particular way. Now, all these various accounts are about three things, modeling an account, varying the interest, and varying the transfer mechanism. At each stage along the way, we are putting more concerns into a single place in each class.

⁴ Generally, most approaches to software development that require prediction are destined to fail. The world is too complex, the future is too hard to envision, and changes are too rapid for prediction to be in any sense reliable.

A pragmatic comment on this example

The above example may look contrived, but the authors have seen this result in many ways.

Imagine that we started with was the delayed transfer, but now it is determined PreferredAccounts need to have an option to transfer immediately. It would not be unusual for someone using direct inheritance to just make a derivation of the PreferredAccount with this type of calculation. If later, we need to add this calculation for standard accounts, it will become clear that we should pull out the calculator. But consider the situation the developer will find himself in if he didn't write the code for the PreferredAccount or if the code doesn't have a full set of tests.

In this case, there is some danger to refactoring and retesting the code and the developer may just chose to copy and paste the calculator into a new subclass of StandardAccount.

It's just too easy to fall into this trap.

Favoring Delegation

The Gang of Four's advice about favoring delegation would seem to argue for something more like this:

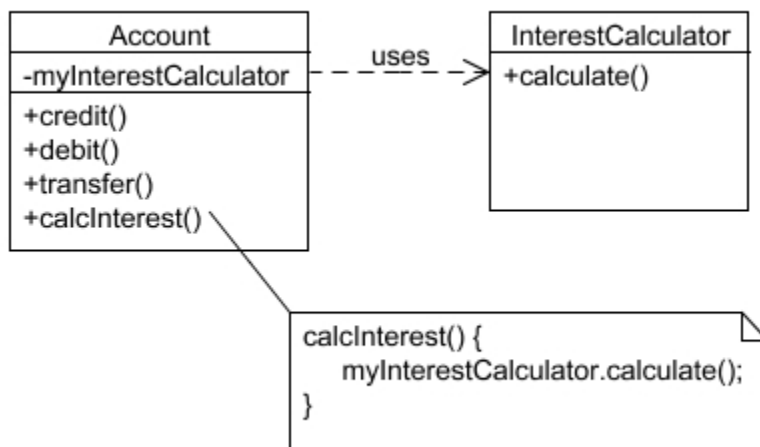


Figure 13.4

Delegating through a pointer

Note that the `calcInterest()` method is still implemented (not abstract), but it's implementation consists merely of calling the `calculate()` method on the service object `InterestCalculator`. This "passing the buck" behavior is what we mean by delegation.

The very act of pulling the interest calculation out of the `Account` feels right, in terms of the separation of concerns, because it removes one concern from the `Account`.

Almost immediately, however, in seeking to create a variation in what we’re delegating *to*, we end up putting inheritance back in again. Almost all of the patterns do this in various ways, and it would seem at first glance that the Gang of Four is warning against inheritance, and then using it repeatedly throughout the patterns they illustrate.

In this case, the variation is of a single algorithm, and would likely be enabled through the use of the Strategy Pattern (see <http://www.netobjectivesrepository.com/TheStrategyPattern>):

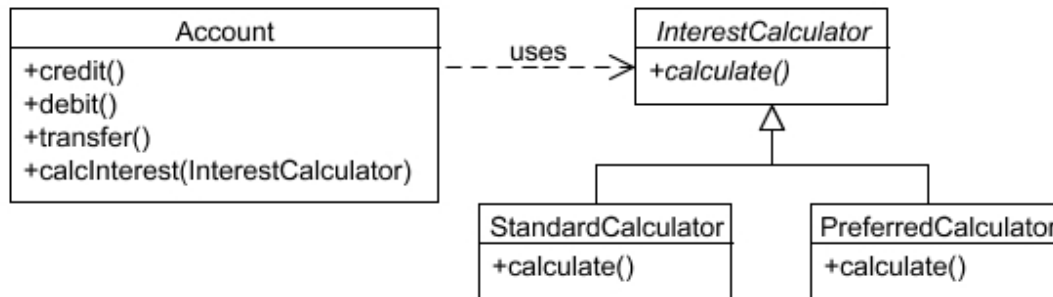


Figure 13.5
The Strategy Pattern

So what’s the real recommendation? What are we favoring over what? Comparing Figures 3.3.2 and 3.3.5 we see inheritance at work in both cases, and in fact it is abstract inheritance each time.

But notice that from Account’s “point of view” the Gang of Four approach uses delegation to handle the variation of the interest calculation while in the older approach inheritance is directly used on the Account class to achieve this variation. The difference is not whether inheritance is used; inheritance is just a mechanism, and it can be used advantageously and disadvantageously just as any mechanism can be. The difference is in what we’re using inheritance *for*.

We sometimes forget that the code is not the software. It’s an abstract representation that gets converted, by the compiler, into the actual software, which exists at runtime, not when we’re writing it. All our work is at some abstract level, whether it is UML, code, or whatever representation we use.

Examining inheritance vs. delegation at runtime can be very revealing.

The Use of Inheritance vs. Delegation

Let’s step back and consider these two mechanisms, delegation and inheritance, in and of themselves.

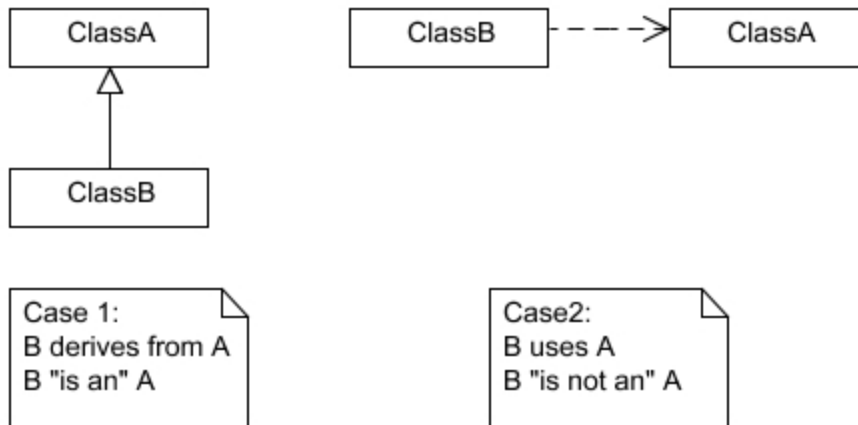


Figure 13.6
Inheritance and Delegation

An interesting thing to note is that the relationships are in some ways more similar than they seem at first glance, when you examine them at run-time.

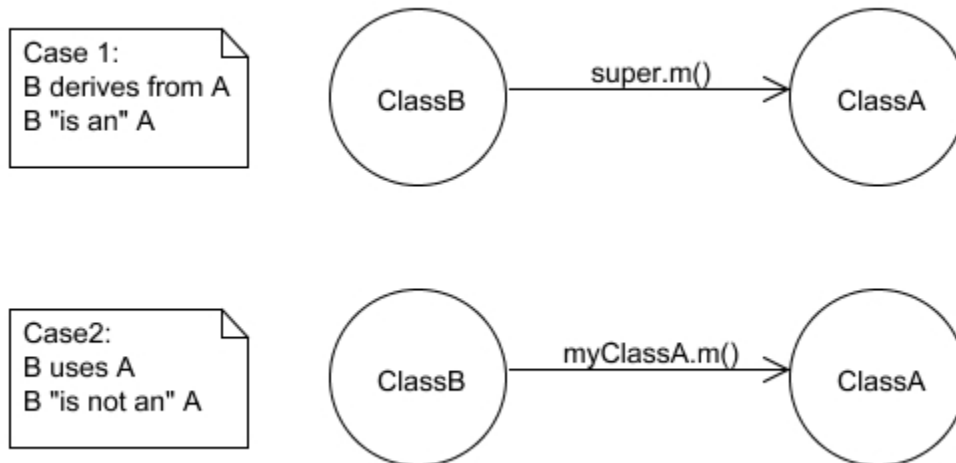


Figure 13.7
Run-time Similarities

In Case 1, when ClassB gets loaded by the class loader, and an instance is created, in fact ClassA will also load, and an instance of it will be created as well (this will happen first, in fact). This is to allow the instance of B to access the instance of A through an inherent pointer (or accessible via a keyword such as “super” or “base”, etc...).

In Case 2, very similarly, an instance of ClassB will be accompanied by an instance of ClassA, which it can access via a pointer. The immediately obvious difference is how and when the instances get created and how the pointer is provided and accessed.

These two cases, in other words, have a very similar, seemingly almost identical run-time relationship. So why should we favor one over the other? Here are some of the differences:

Table 13.1
Inheritance vs. Delegation

Inheritance	Delegation
A reference of ClassB can be cast to ClassA. There is an implied substitutability between them.	ClassB and ClassA are distinct; there is no implication of sameness.
Any change to ClassA can potentially effect ClassB. Care must be taken to ensure that unwanted side-effects in ClassB are overridden. Also, in some languages the issue of virtual/non-virtual methods must be considered, and whether an overridden method replaces or shadows the original, and whether casting can change the method implementation being bound to at runtime.	Changes to ClassA can only effect ClassB insofar as those changes propagate through ClassB's interface. It is much easier to control the potential for unwanted side-effects from a change, because we have stronger encapsulation.
The instance of ClassA that ClassB has access to cannot be changed after the object is created. Also, the inherent instance will always be of type ClassA, concretely.	The addition of a setter() method in ClassB would allow us to change the instance of ClassA that ClassB uses, and in fact can give ClassB an instance of something other than ClassA (a subclass, for example), so long as it passes type-check.
The instance of ClassA that ClassB uses cannot be shared.	The instance of ClassA that ClassB uses can be shared.

Uses of Inheritance

Let's examine the two options for containing our interest variation in Account, side by side:

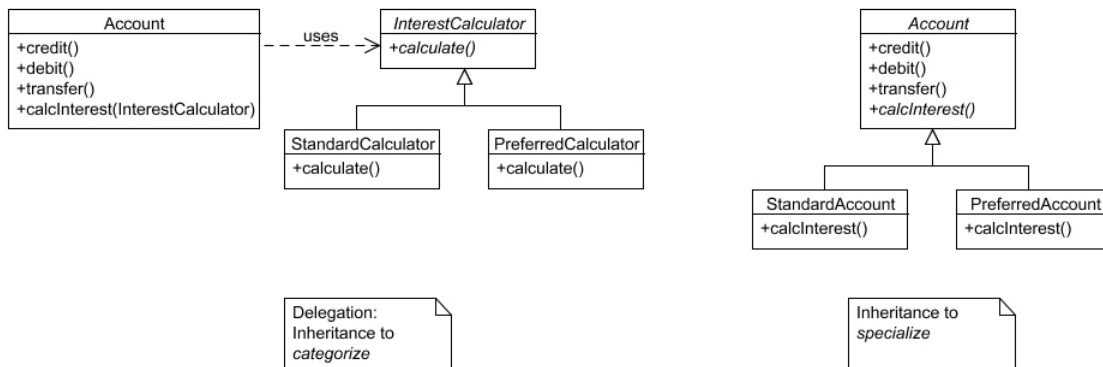


Figure 13.8
Side by side comparison

What the gang of four was recommending was *to use inheritance to hide variation, not to achieve reuse of code.*

In their preferred solution (the one on the left) inheritance is used to create a *category*, or substitutability/pluggability, around the concept “InterestCalculator”. InterestCalculator itself is not “a thing” but rather “an idea”. The “real” things are StandardCalculator and PreferredCalculator, which have been made interchangeable through this use of inheritance⁵.

⁵ If you're familiar with .Net delegates, you know that a very similar end could be achieved using them, or with “interface” types in (.Net, Java, and other languages) which could have multiple implementations. The point is the pluggability, the way you get it will vary depending on the language and platform you are using, and other forces operating on your decision.

What the gang of four is recommending against is the approach on the right, where the Account (which is a “real” thing) is being *specialized* through the inheritance mechanism.

One clue for us that this is perhaps not a natural fit to the problem is in the actual code for the right-hand design:

```
abstract class Account {
    public void credit() { //some implementing code }
    public void debit() { //some implementing code }
    public void transfer() { //some implementing code }
    public abstract void calcInterest();
}
```

In order to make this work, we’ve had to give Account an abstract method (calcInterest()), and therefore Account itself must be made abstract. But Account is not an abstract idea, it’s a real thing. Compare this to InterestCalculator, which is conceptual and which would therefore seem to be a more natural, logical thing to make abstract.

This is just a clue, of course. We could artificially implement calcInterest() in Account to avoid being forced to make it abstract, but we’d be chasing our tails a bit, if you revisit table 3.3.1 and see all the advantages we’d be giving up.

Pay particular attention to the third point in the table – if we need to change the calculation algorithm at runtime, the preferred solution is much easier on us. In the solution on the right, if we had built StandardAccount, the only way to switch to the preferred algorithm is to build an instance of PreferredAccount, and *somehow transfer any state from one to the other*. This would require us to either:

1. Break encapsulation on the state, so we could retrieve it from the old instance to give it to the new one.
2. Create a “cloning” method that allows each to remake itself as the other.

This is the “dynamism problem” we encountered earlier. In the first case, we give up encapsulation, in the second, we create unnecessary coupling between these two classes.

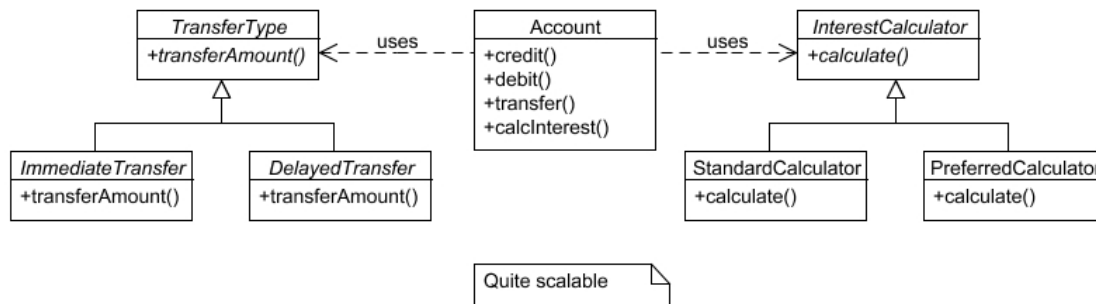
On the other hand, if we choose to follow the gang of four’s recommendation and use the preferred solution (the “Strategy Pattern” as we mentioned above), then we need only add a method like setCalculator(InterestCalculator aCalculator) to the Account class, and voila, we can change the interest algorithm whenever we like, without rebuilding the Account object.

This also means that we don’t have to add that method now, if there is no such requirement. Good design often does this: we know it will be easy to add this capability later if we ever need it, and so we don’t overdesign the initial solution now. See our chapter “Don’t Build Something Unless You Know You Need To” for more details on the virtues of this.

Scalability

We saw earlier that scaling the design, where inheritance was used to specialize the Account object, lead to ever-decreasing quality and a resulting increase in the difficulty to accommodate changes.

On the other hand, if we try to scale the Strategy Patterns solution, we can simply do what we did before – pull the new variation (transfer speed) out:

**Figure 13.9***Much more scalable*

Note how many aspects of the design are actually improved by this change.

For instance, many (perhaps yourself) would have argued initially that the account had too many responsibilities. Some would say that interest calculation and transfer from account to account really were always separate responsibilities, even before they started to vary. It’s a fair point, but it’s difficult to know how far to go⁶ with the instinct to “pull stuff apart”.

But note how the evolution of our design gets us to this realization, by its very nature. Our design is neither fighting us in terms of being able to accommodate change, nor is it leading us down the primrose path in terms of overall quality. We don’t have to “get everything right” in the beginning, because our initial attention to good design gives us the power to make changes efficiently, with low risk and very little waste.

Applying the Lessons from the Gang of Four to Agile Development

People often think of “Design Patterns” as a kind of design up front. One needs to remember that the design patterns came into vogue when that was mostly how you did things. However, if you explore the thought process under the patterns, you can apply that thought process to agile development just as well as you can when you are doing a big design up front. Perhaps better – because the lessons of patterns tell us what to be looking at – not so much as what to do.

Let’s look at where you are at the start of the project:

- You know of multiple behaviors you need to have and you need them right away
- You know of multiple behaviors you need to have but you only need the first one at the start
- You know of only one behavior you will need to have

Agile development tells us we should actually deal with all three of these items in a similar manner. In other words, even if we know of more than one case, we should only build the first one anyway, then go onto the second. In other words, YAGNI becomes YAGNIBALYW (You Are Gonna Need It But Act Like You Won’t). The second case is actually how we’re saying to handle the first case. The third case, of course becomes the second case when something new comes up. Thus, we can say, in the Agile world, the Gang of Four’s advice to “pull out what is varying and encapsulate it” becomes “when something starts to vary, pull it out into its own class

⁶ We do have more to say on this, however. Please read the chapter “Define Tests Up Front” for a discussion of how tests can help us to see how far to go in separating behaviors into objects.

and encapsulate it.” This compliments the ideas expressed in the “Refactor to the Open-Closed Principle” chapter.

This approach allows us to follow another powerful principle of software design – “A class should only have one reason to vary.” While this too has often been used as a principle for up-front designs, we can now see that it tells us to extract variation from classes as it starts to occur.

Testing Issues

We have another chapter on testing, but it’s interesting to note here that the use of inheritance in the way the Gang of Four is recommending plays well into testing scenarios.

In short, our definition of a good test is one that fails for a single, reliable reason. Our definition of a good test suite is one where a single change or bug can only ever break a single test. These are ideals, of course, and cannot be perfectly achieved, but it is always our goal to get as close as we can. Taken together, these bit of guidance make our tests more useful, and keeps our suites from become maintenance problems in and of themselves.

We note that the use of a mock object in our Strategy Pattern achieves these goals perfectly:

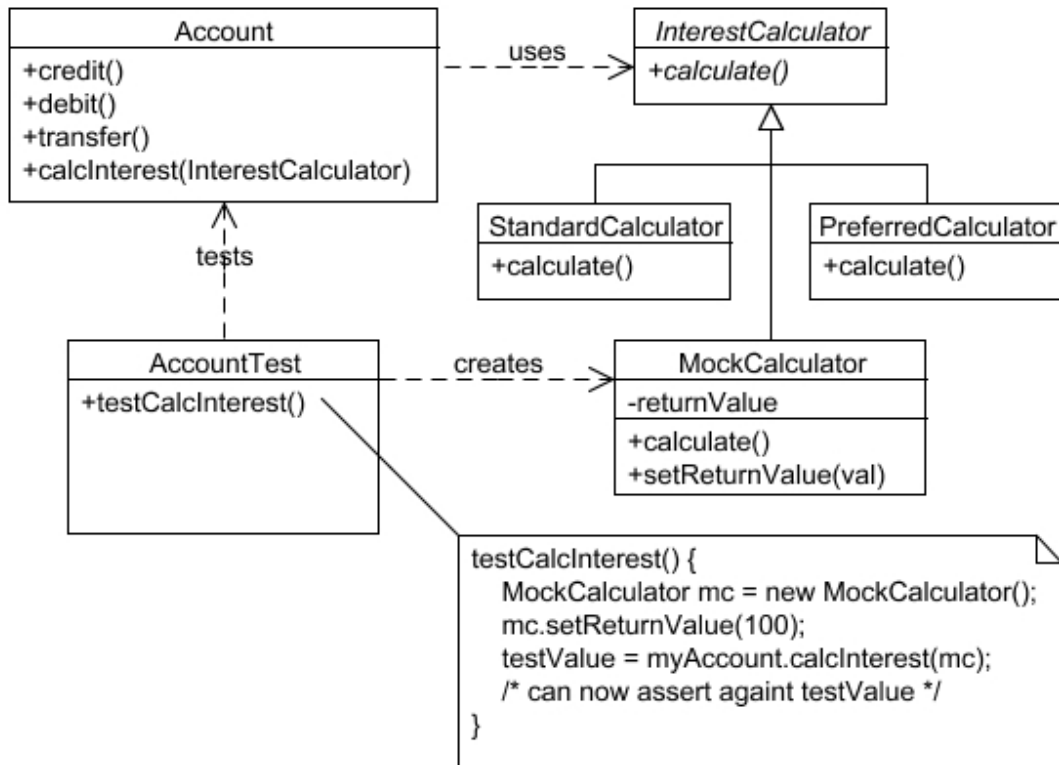


Figure 13.10
Mocking for testing separation

The test for `Account` would instantiate `MockCalculator`, and hold it as `MockCalculator`, which gives it access to `setReturnValue()` and any other for-testing-only method we care to add (there are many possibilities, this is an intentionally simple example). In handing it to `Account`, however, note it will be upcast to `InterestCalculator` (because of the parameter type) and thus `Account` cannot become coupled to the testing methods, or the existence of the mock at all.

Testing each calculator requires a single test per implementation, which can only fail if the calculator is broken or changed.

Testing the Account is done in the context of the MockCalculator, and so breaking or changing either of the “real” calculators will not cause the test for Account to fail. The Account test can only fail if the Account is actually broken or changed.

There’s more

This example is simple, intentionally so to make the point clear. There are many, many more examples available that show how inheritance can be used in this preferred way (to create categories for pluggability).

They are all Design Patterns.

Visit www.netobjectivesrepository.com for more examples of how inheritance can be used to create pluggability and dynamism.

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality

Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

<p>Assessments</p> <p>See where you are, where you want to go, and how to get there.</p> <p>Business and Management Training</p> <p>Lean Software Development Product Portfolio Management Enterprise Release Planning</p>	<p>Productive Lean-Agile Team Training</p> <p>Team training in Kanban, Scrum Technical Training in ATDD, TDD, Design Patterns</p> <p>Roles Training</p> <p>Lean-Agile Project Manager Product Owner</p>
--	---

