Composite Reuse Principle

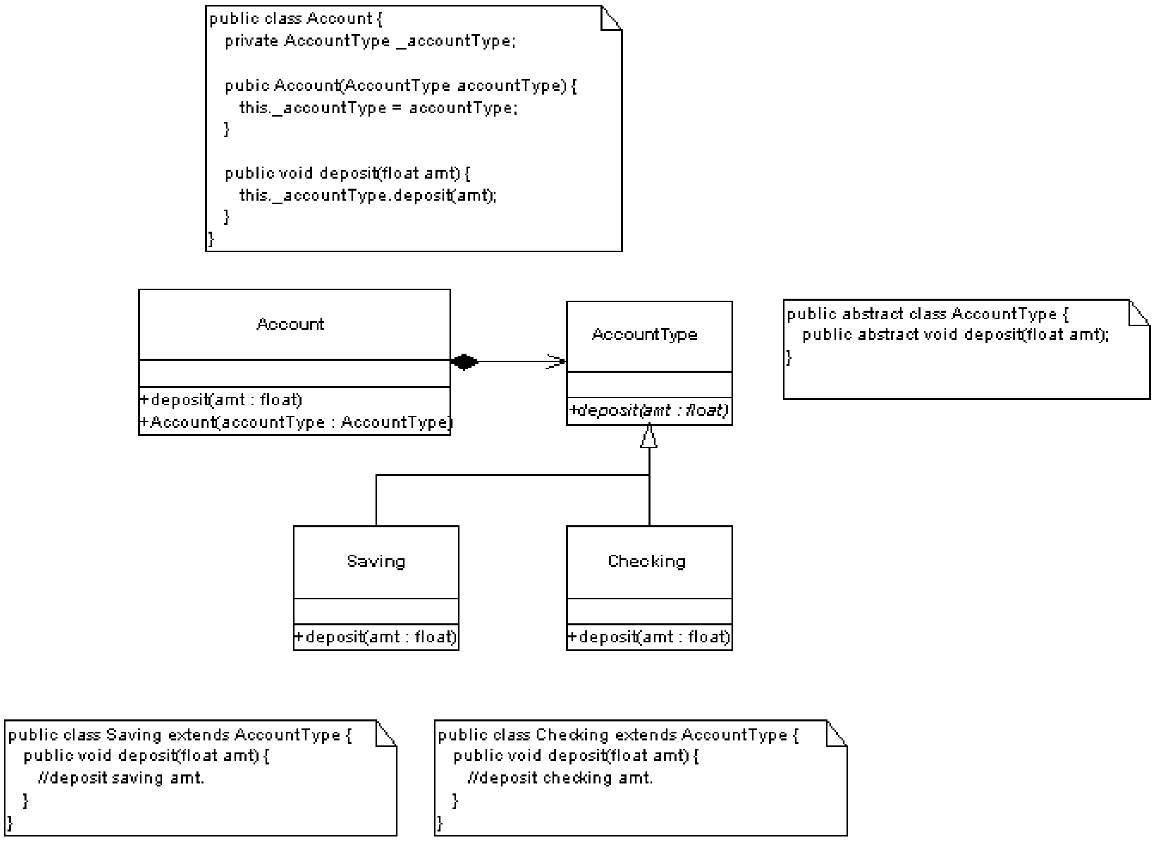
Favor polymorphic composition of objects over inheritance.

Principle Foundation

Reuse has long been touted as the Holy Grail of object-orientation. Seen as the mechanism through which we achieve high degrees of reusability, inheritance is often used in an attempt to realize reuse. In this effort, however, inheritance typically causes more problems than solutions. Brittle architectures are often the result of bloated ancestor classes attempting to define perceived default behaviors in future descendent classes. The result is a necessity to override the ancestor code in descendents. Worse yet, we find that we copy code from other classes.

The Composite Reuse Principles focuses on achieving reuse through delegation (or more appropriately forwarding), instead of defining default behaviors in ancestor classes. Inheritance is only used in the Composite Reuse Principle to help us flexibly manage implementations.

Sample Illustration



```
public class Account {
  private AccountType _accountType;

  pubic Account(AccountType accountType) {
    this._accountType = accountType;
  }

  public void deposit(float amt) {
    this._accountType.deposit(amt);
  }
}
```

```
public abstract class AccountType {
  public abstract void deposit(float amt);
}
```

```
public class Saving extends AccountType {
  public void deposit(float amt) {
    //deposit saving amt.
  }
}
```

```
public class Checking extends AccountType {
  public void deposit(float amt) {
    //deposit checking amt.
  }
}
```

Figure 1

In Figure 1, our `Account` class delegates behaviors specific to a type of account to the `AccountType` hierarchy. Doing so allows us to flexibly extend our application by defining new `AccountType` descendents, which ensures compliance with the Open-Closed Principle. More closely pertaining to the Composite Reuse Principle, this structure also allows client classes other than `Account` to reuse the `AccountType` hierarchy, as `AccountType` and all descendents are not coupled to the `Account` class.

Solving this design challenge using inheritance results in our `Saving` and `Checking` classes to inherit directly from the `Account` class. Doing so limits the likelihood that other clients wishing to take advantage of `Saving` and `Checking` functionality cannot do so outside the context of `Account`.

Key Implementation Considerations

• Delegation	This principle relies heavily upon delegation, or more appropriately forwarding. Classes with a desire to reuse behavior call methods on other classes instead of calling methods on an ancestor.
• Interface Class	If it's unlikely that the reuse implementation will change, the ancestor class may be omitted. This results in a simpler design, but also results in possible future challenges as it will be difficult to comply with the Open-Closed Principle, which is a natural by-product of the Composite Reuse Principle.
• Implicit `this`	When reusing through delegation, additional coding is required as we are lacking the implicit `this` we receive when using inheritance. Therefore, instead of simply coding `this.deposit(100)` were we to use inheritance, we are forced to first instantiate the desired concrete class, then reference explicitly that variable to call the desired method, as follows: `AccountType a = new Saving();` `a.deposit(100);` This obviously results in more coding as well as introducing instance creation challenges, as discussed in the Open-Closed and Dependency Principles.

Consequences

The granularity of the classes to which we delegate is extremely important when applying this principle. While structurally similar to the Open-Closed Principle, there is a separation of concerns. Open-Closed Principle emphasizes extending the application through inheritance based on existing clients. Composite Reuse Principle emphasizes the ability to reuse the hierarchy across clients. Subsequently, the behaviors defined in our supplier classes are the emphasis here. This will likely result in refactoring the supplier classes as new clients are introduced.

When refactoring the supplier classes, applicability of the Composite Reuse Principle is cyclic. Behaviors common to one context, but not another are factored out into another, separate hierarchy. As such, the configuration of objects present is certainly context specific.

Related Principles

Composite Reuse Principle helps us comply with the Open Closed Principle.

Composite Reuse Principle is typically used in conjunction with Dependency Inversion, though this is not a requirement.

References

[OOSC88] *Object-Oriented Software Construction.* Bertrand Meyer. Prentice-Hall, 1988.

[MARTIN00] *Design Principles and Design Patterns*. Robert C. Martin, 2000.

[JOUP02] *Java Design: Objects, UML, and Process.*. Kirk Knoernschild. Addison-Wesley, 2002.