Kapitel 7

Das Liskovsche Substitutionsprinzip

Am Ende dieses Kapitels werden Sie in der Lage sein, die folgenden Aufgaben durchzuführen:

- Beschreiben der Bedeutung des Liskovschen Substitutionsprinzips
- Vermeiden, die Regeln des Liskovschen Substitutionsprinzip zu brechen
- Verfestigen Ihrer Gewohnheiten zum Einhalten von Single-Responsibility- und Open/Closed-Prinzip
- Erstellen abgeleiteter Klassen, die Verträge ihrer Basisklassen erfüllen
- Verwenden von Codeverträgen, um Vorbedingungen, Nachbedingungen und Dateninvarianz zu implementieren
- Schreiben richtigen Codes zum Auslösen von Ausnahmen
- Beschreiben von Kovarianz, Kontravarianz und Invarianz und wo sie angewendet werden

Einführung in das Liskovsche Substitutionsprinzip

Das Liskovsche Substitutionsprinzip (Liskov Substitution Principle, LSP) ist eine Sammlung von Richtlinien zum Erstellen von Vererbungshierarchien, bei denen ein Client jede Klasse oder abgeleitete Klasse zuverlässig benutzen kann, ohne dass sich das erwartete Verhalten ändert.

Werden die Regeln des LSPs nicht eingehalten, kann es sein, dass nach einer Erweiterung der Klassenhierarchie (das heißt nach Erstellung einer neuen abgeleiteten Klasse) Änderungen an allen Clients der Basisklasse oder Schnittstelle erforderlich sind. Wird das LSP dagegen eingehalten, brauchen Clients nichts von den Änderungen in der Klassenhierarchie zu erfahren. Solange sich nichts an der Schnittstelle ändert, sollte es keinen Grund geben, irgendwelchen vorhandenen Code zu ändern. Das LSP hilft also dabei, sowohl das Open/Closed-Prinzip als auch das Single-Responsibility-Prinzip zu erzwingen.

Formale Definition

Die Definition des LSP durch die bekannte Informatikerin Barbara Liskov ist etwas trocken, daher sollte sie ausführlicher erklärt werden. Hier die offizielle Definition:

> *Sei S ein von T abgeleiteter Typ, dann können Objekte des Typs T durch Objekte des Typs S ersetzt werden, ohne das Programm zu beschädigen.*
>
> *– Barbara Liskov*

Im LSP kommen drei Codezutaten vor:

- **Basistyp (engl. base type)**
  Der Typ (T), auf den Clients verweisen. Clients rufen verschiedene Methoden auf, die alle vom abgeleiteten Typ überschrieben (oder teilweise spezialisiert) werden können.
- **Abgeleiteter Typ (engl. subtype)**
  Jede Klasse aus einer möglichen Familie von Klassen (S), die vom Basistyp (T) abgeleitet sind. Clients sollten nicht wissen, welchen konkreten abgeleiteten Typ sie aufrufen, und das sollte auch gar nicht erforderlich sein. Der Client sollte sich immer gleich verhalten, unabhängig davon, welchen abgeleiteten Typ die Instanz hat, die ihm zur Verfügung gestellt wird.
- **Kontext**
  Die Art, wie der Client mit dem abgeleiteten Typ interagiert. Falls der Client nicht mit einem abgeleiteten Typ interagiert, kann das LSP weder eingehalten noch missachtet werden.

LSP-Regeln

Sie müssen mehrere »Regeln« einhalten, um das LSP zu befolgen. Diese Regeln können in zwei Kategorien untergliedert werden: Vertragsregeln (engl. contract rules) betreffen die Erwartungen an Klassen und Varianzregeln (engl. variance rules) die Typen, die im Code ersetzt werden können.

Vertragsregeln

Diese Regeln beziehen sich auf den Vertrag des Basistyps und die Einschränkungen, die für Verträge festgelegt werden können, die zum abgeleiteten Typ hinzugefügt werden.

- Vorbedingungen (engl. preconditions) dürfen in einem abgeleiteten Typ nicht verschärft werden
- Nachbedingungen (engl. postconditions) dürfen in einem abgeleiteten Typ nicht geschwächt werden
- Invarianten (engl. invariants) des Basistyps (das heißt Bedingungen, die wahr bleiben müssen) müssen in einem abgeleiteten Typ erhalten bleiben

Um diese Vertragsregeln zu verstehen, müssen Sie erstens das Konzept von Verträgen kennen und zweitens wissen, auf welche Weise Sie sicherstellen, dass Sie diese Regeln befolgen, wenn Sie abgeleitete Typen erstellen. Der Abschnitt »Verträge« weiter unten in diesem Kapitel behandelt diese beiden Themen genauer.

Varianzregeln

Diese Regeln beziehen sich auf die Varianz (engl. variance) von Argumenten und Rückgabetypen.

- Es ist Kontravarianz der Methodenparameter im abgeleiteten Typ erforderlich
- Es ist Kovarianz der Rückgabetypen im abgeleiteten Typ erforderlich
- Der abgeleitete Typ darf nur dann neue Ausnahmen auslösen, wenn sie Teil der vorhandenen Ausnahmehierarchie sind

Das Konzept der Typvarianz in den Sprachen der Common Language Runtime (CLR) im Microsoft .NET Framework beschränkt sich auf generische Typen und Delegaten. Es lohnt sich aber, wenn Sie sich mit der Varianz in diesen Szenarien vertraut machen; Sie verfügen dann über die Grundlagen, um Code zu schreiben, der in Bezug auf die Varianz LSP-kompatibel ist. Dieses Thema wird im Abschnitt »Kovarianz und Kontravarianz« weiter unten in diesem Kapitel detailliert behandelt.

Verträge

Es wird oft gesagt, Entwickler sollten *»auf Schnittstellen hin programmieren«* oder *»auf einen Vertrag hin programmieren«*. Abgesehen von den Methodensignaturen liefern Schnittstellen aber nur sehr wenige Informationen über einen Vertrag. Eine Methodensignatur verrät wenig über die tatsächlichen Anforderungen und Garantien der Methodenimplementierung (Abbildung 7–1). In einer streng typisierten Sprache wie C# gibt es zumindest das Prinzip, den korrekten Typ als Argument zu übergeben, aber dort endet die Schnittstelle auch schon. Daran muss sich das Konzept des Vertrags anschließen.

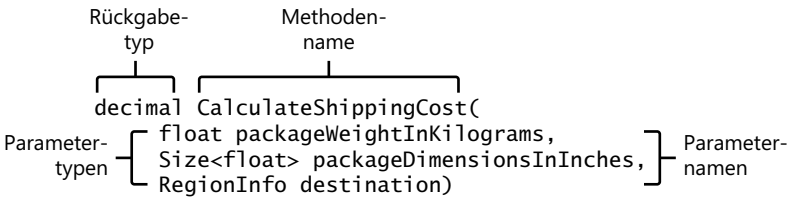

Abb. 7–1 Methodensignaturen verraten wenig über die Erwartungen der Implementierung

Alle Methoden haben zumindest einen optionalen Rückgabetyp, einen Namen und eine optionale Liste formeller Parameter. Jeder Parameter besteht aus einer Typangabe und einem Namen. Wenn Sie die Methode aus Abbildung 7–1 aufrufen, wissen Sie aus der Sig-

natur, dass Sie drei Argumente übergeben müssen, eines vom Typ `float`, eines vom Typ `Size<float>` und das dritte vom Typ `RegionInfo`. Sie wissen außerdem, dass Sie den Rückgabewert, der den Typ `decimal` hat, in einer Variablen speichern oder anderweitig weiterverarbeiten können, nachdem der Aufruf zurückgekehrt ist.

HINWEIS Anders als in Abbildung 7–1 gezeigt ist es nicht sinnvoll, den Typ `decimal` zu verwenden, um Geldbeträge darzustellen. Stattdessen sollten Sie den Werttyp `Money`[1] benutzen. Es wurde zwar erheblicher Aufwand betrieben, um sicherzustellen, dass die Beispiele in diesem Buch möglichst praxisnah sind, aber damit der Code nicht zu lang wird, waren einige Kompromisse notwendig.

Als Programmierer der Methode können Sie festlegen, welche Namen die Parameter und Methoden erhalten. Bemühen Sie sich um Methodennamen, die den Zweck der Methode erkennen lassen, und um aussagekräftige Parameternamen. Der Name der Funktion `CalculateShippingCost` (»Berechne Versandkosten«) folgt der Verb-Substantiv-Form. Das Verb, also die Aktion, die von der Methode durchgeführt wird, ist `Calculate` und das Substantiv, also das Objekt des Verbs, ist `ShippingCost`. Dieses Substantiv ist in gewisser Weise der Name des Rückgabewerts. Auch die Parameter haben aussagekräftige Namen: `packageDimensionsInInches` (»Paketabmessungen in Zoll«) und `packageWeightInKilograms` (»Paketgewicht in Kilogramm«) sind selbsterklärend, insbesondere im Kontext der Methode. Sie bilden den Ausgangspunkt zum Dokumentieren der Methode.

HINWEIS Weitere Informationen über gute Variablen- und Methodennamen sowie andere Best Practices finden Sie in *Code Complete* von Steve McConnell (Microsoft Press Deutschland, 2005).

HINWEIS Verträge, wie sie in diesem Kapitel beschrieben werden, bieten zwar während der Laufzeit Schutz vor vielen ungültigen Methodenaufrufen, aber die Bedeutung guter Methoden- und Parameternamen lässt sich gar nicht genug betonen. Würden die formalen Parameter der Methode `CalculateShippingCost` nicht darauf hinweisen, dass die Werte in der Einheit Zoll beziehungsweise Kilogramm interpretiert werden, könnten Clients der Methode beispielsweise Werte in den Einheiten Zentimeter beziehungsweise Pfund übergeben.

1. *http://moneytype.codeplex.com*

Vorbedingungen

Vorbedingungen sind definiert als die Gesamtheit aller Bedingungen, die eingehalten werden müssen, damit eine Methode zuverlässig und fehlerfrei läuft. Jede Methode setzt voraus, dass einige Vorbedingungen zutreffen, bevor sie aufgerufen wird. In der Standardeinstellung erzwingen Schnittstellen keine Garantien von den Implementierern ihrer Methoden. Listing 7–1 zeigt, wie Sie eine Vorbedingung mithilfe einer Wächterbedingung am Anfang einer Methode implementieren können.

```
public decimal CalculateShippingCost(
  float packageWeightInKilograms,
  Size<float> packageDimensionsInInches,
  RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f) throw new Exception();

    return decimal.MinusOne;
}
```

List. 7–1 Eine Ausnahme auszulösen, ist eine effektive Möglichkeit, Vorbedingungen zu erzwingen

Die `if`-Anweisung ganz am Anfang der Methode ist eine Möglichkeit, eine Vorbedingung zu erzwingen, in diesen Fall die Anforderung, dass das Gewicht eine positive Zahl sein muss. Trifft die Bedingung `packageWeightInKilograms <= 0f` zu, wird eine Ausnahme ausgelöst und die Methode beendet sofort ihre Ausführung. Das verhindert zweifellos, dass eine Methode ausgeführt wird, falls nicht alle Parameter gültige Werte haben. Indem Sie eine aussagekräftigere Ausnahme verwenden, können Sie dem Aufrufer mehr Kontext zur Verfügung stellen (Listing 7–2).

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms",
           "Paketgewicht muss positiv und ungleich null sein");

    return decimal.MinusOne;
}
```

List. 7–2 Es ist wichtig, möglichst viel Kontext darüber zu liefern, warum die Vorbedingung einen Fehler verursacht

Das ist eine Verbesserung gegenüber der ersten Ausnahme. Abgesehen davon, dass diesmal eine Ausnahme verwendet wird, die explizit für Argumente außerhalb des erlaubten

Wertebereichs gedacht ist, wird der Client darüber informiert, welcher Parameter falsch ist, und bekommt eine Beschreibung des Problems.

Indem Sie mehrere solche Wächterbedingungen verketten, können Sie weitere Bedingungen hinzufügen, die erfüllt sein müssen, damit die Methode aufgerufen werden kann, ohne eine Ausnahme auszulösen. In Listing 7–3 kommt eine Ausnahme hinzu, die ausgelöst wird, falls die Paketabmessungen außerhalb des erlaubten Bereichs liegen.

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms",
          "Paketgewicht muss positiv und ungleich null sein");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches",
          "Paketabmessungen müssen positiv und ungleich null sein");

    return decimal.MinusOne;
}
```

List. 7–3 Sie können so viele Vorbedingungen hinzufügen, wie erforderlich sind, um zu verhindern, dass die Methode mit ungültigen Argumenten aufgerufen wird

Stehen diese Vorbedingungen bereit, müssen die Clients sicherstellen, dass die Argumente, die sie übergeben, im erlaubten Wertebereich liegen, bevor sie die Methode aufrufen. Das hat unter anderem die Folge, dass der gesamte Zustand, der in einer Vorbedingung überprüft wird, öffentlich für Clients verfügbar sein *muss*. Falls der Client nicht genau feststellen kann, unter welchen Bedingungen die Methode, die er aufruft, aufgrund einer ungültigen Vorbedingung eine Ausnahme auslöst, kann er auch nicht sicherstellen, dass der Aufruf erfolgreich verläuft. Daher sollte ein privater Zustand niemals Ziel einer Vorbedingung sein. Nur Methodenparameter und öffentliche Eigenschaften der Klasse sollten Vorbedingungen haben.

Nachbedingungen

Nachbedingungen prüfen, ob ein Objekt in gültigem Zustand hinterlassen wird, wenn die Methode verlassen wird. Immer wenn der Zustand in einer Methode verändert wird, besteht die Möglichkeit, dass er aufgrund von Logikfehlern ungültig wird.

Nachbedingungen werden auf dieselbe Weise wie Vorbedingungen implementiert, das heißt mit Wächterbedingungen. Statt die Bedingungen an den Anfang der Methode zu legen, müssen Sie die Wächterbedingungen für Nachbedingungen allerdings ans Ende der Methode schreiben, nachdem alle Änderungen am Zustand durchgeführt wurden (Listing 7–4).

```
public virtual decimal CalculateShippingCost(float packageWeightInKilograms,
  Size<float> packageDimensionsInInches, RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms",
          "Paketgewicht muss positiv und ungleich null sein");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches",
          "Paketabmessungen müssen positiv und ungleich null sein");

    // Berechnen der Versandkosten

    var shippingCost = decimal.One;

    if(shippingCost <= decimal.Zero)
        throw new ArgumentOutOfRangeException("return",
          "Rückgabewert ist ungültig");

    return shippingCost;
}
```

List. 7–4 Die Wächterbedingung am Ende der Methode ist eine Nachbedingung, die sicherstellt, dass der Rückgabewert im gültigen Bereich liegt

Indem Sie prüfen, ob sich der Zustand innerhalb eines gültigen Bereichs befindet, und eine Ausnahme auslösen, falls das nicht zutrifft, können Sie in der Methode eine Nachbedingung erzwingen. Die Nachbedingung bezieht sich hier nicht auf den Zustand des Objekts, sondern auf den Rückgabewert. Ähnlich wie die Werte der Methodenparameter anhand von Vorbedingungen geprüft werden, wird die Gültigkeit der Rückgabewerte mithilfe von Nachbedingungen überprüft. Wird der Rückgabewert irgendwo innerhalb der Methode auf null oder einen negativen Wert gesetzt, erkennt die Nachbedingung das und hält die Ausführung am Ende der Methode an. Auf diese Weise erhalten Clients der Methode niemals einen ungültigen Wert und können die Verarbeitung unter der Annahme fortsetzen, dass der Wert auf jeden Fall gültig ist. Beachten Sie aber, dass die Schnittstelle der Methode nicht mittteilt, dass der Rückgabewert immer ungleich null und positiv ist. Das ist Teil des Vertrags, den die Schnittstelle mit Clients abschließt.

Dateninvarianten

Ein dritter Vertragstyp ist die Dateninvariante. Eine *Dateninvariante* (engl. data invariant) ist ein Prädikat, das über die gesamte Lebensdauer eines Objekts hinweg wahr bleibt. Es ist wahr unmittelbar nach der Erstellung und muss wahr bleiben, bis das Objekt beseitigt wird. Dateninvarianten beziehen sich auf den erwarteten internen Zustand des Objekts. Um eine Dateninvariante an unserer Beispielklasse `ShippingStrategy` zu demonstrieren, nehmen wir an, dass eine Flatrate mit Pauschalversandkosten angeboten wird. Der Wert dieser Flatrate

muss positiv und ungleich null sein. Wird die Flatrate wie in Listing 7–5 bei der Objekterstellung festgelegt, verhindert eine simple Wächterbedingung im Konstruktor, dass ein ungültiger Wert eingetragen wird.

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        if (flatRate <= decimal.Zero)
            throw new ArgumentOutOfRangeException("flatRate",
              "Flatrate muss positiv und ungleich null sein");

        this.flatRate = flatRate;
    }

    protected decimal flatRate;
}
```

List. 7–5 Eine Vorbedingung im Konstruktor hilft, eine Dateninvariante zu schützen

Weil `flatRate` eine geschützte Membervariable ist, können Clients diesen Wert einzig über den Konstruktor festsetzen. Hat `flatRate` an diesem Punkt einen gültigen Wert, bleibt er garantiert auch für die restliche Lebensdauer des Objekts gültig, weil Clients keine Möglichkeit haben, ihn zu ändern.

Ist die Variable `flatRate` stattdessen eine öffentlich änderbare Eigenschaft, muss die Wächterbedingung in den Setter-Block verlegt werden, um die Dateninvariante zu schützen. Listing 7–6 zeigt, wie die Flatrate als öffentliche Eigenschaft mit einer zugehörigen Wächterbedingung refaktoriert wurde.

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        FlatRate = flatRate;
    }

    public decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            if (value <= decimal.Zero)
                throw new ArgumentOutOfRangeException("value",
                  "Flatrate muss positiv und ungleich null sein");
```

```
            flatRate = value;
        }
    }

    protected decimal flatRate;
}
```

List. 7–6 Wenn eine Dateninvariante eine öffentliche Eigenschaft ist, gehört die Wächterbedingung in den Setter

Jetzt können Clients zwar den Wert der Eigenschaft `FlatRate` verändern, aber wegen der `if`-Anweisung und der Ausnahme kann die Invariante nicht beschädigt werden.

Kapselung und Verträge

Die in diesem Beispiel implementierten Verträge sind sinnvoll, sie sind aber nur deshalb notwendig, weil für jeden Wert ein ungeeigneter Typ gewählt wurde. Der Vorbedingungsvertrag, der sicherstellen soll, dass der Parameter für das Paketgewicht ungleich null und positiv ist, ist untrennbar mit dem Typ der Variablen verknüpft: Das Gewicht sollte niemals null oder negativ sein. Das legt den Gedanken nahe, das Gewicht in einem eigenen Typ zu kapseln. Falls (und das ist wahrscheinlich) eine andere Klasse oder Methode ebenfalls einen Gewichtswert benötigt, müssen Sie diese Vorbedingung auch in den neuen Code hinüberbringen. Das ist ineffizient, schwierig zu warten und fehlerträchtig. Viel sinnvoller ist es, einen neuen Typen zu erstellen und die Vorbedingung so zu definieren, dass jede Nutzung des Typs `Weight` einen positiven Wert ungleich null haben muss. Es handelt sich hier um eine Invariante des Typs, nicht um eine Vorbedingung der Methode `CalculateShippingCost`.

Ähnliches gilt für die Flatrate, für die sich der Typ `decimal` nicht gut eignet. Stattdessen sollten Sie dafür einen eigenen Werttyp erstellen und die erforderliche Invariante, dass sie ein positiver Wert ungleich null sein muss, auf diesen Typ anwenden.

Liskovsche Vertragsregeln

Die bisherigen Erläuterungen zu Methodenverträgen sind lediglich die Einleitung für einige Kernaspekte des Liskovschen Substitutionsprinzips. Das LSP setzt Regeln fest, nach denen Typen Verträge erben müssen. Hier noch einmal die Definition des LSPs:

> *Sei S ein von T abgeleiteter Typ, dann können Objekte des Typs T durch Objekte des Typs S ersetzt werden, ohne das Programm zu beschädigen.*

Soweit dies Verträge betrifft, ergeben sich daraus die bereits erwähnten Richtlinien:

- Vorbedingungen dürfen in einem abgeleiteten Typ nicht verschärft werden
- Nachbedingungen dürfen in einem abgeleiteten Typ nicht geschwächt werden
- Invarianten des Basistyps müssen in einem abgeleiteten Typ erhalten bleiben