





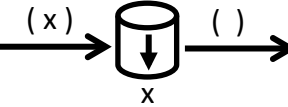
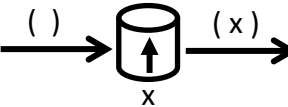
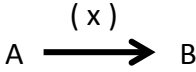

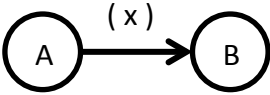
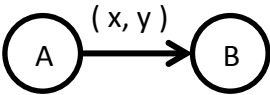
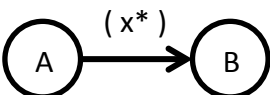
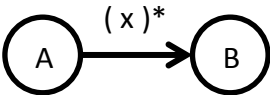
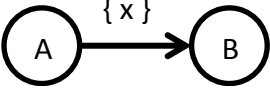
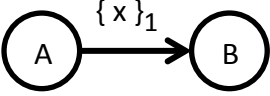
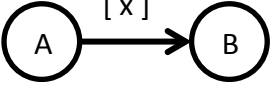
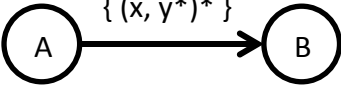


# CheatSheet Flow Design

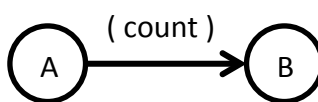
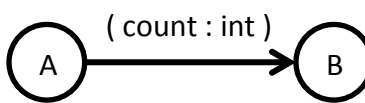
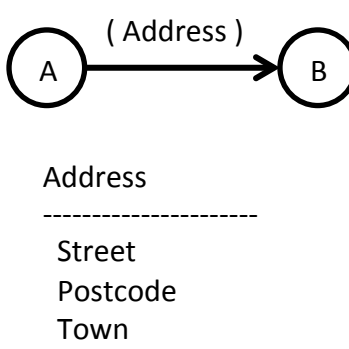
## Symbole

<b>Functional Units / Funktionseinheiten</b>	
Funktionseinheit ist der Überbegriff für Methode, Klasse, Bibliothek, etc.	
	Eine Funktionseinheit A, die <b>Domänenlogik</b> enthält.
	Eine Funktionseinheit, die einen <b>Ressourcenzugriff</b> darstellt.
	Ebenfalls ein Ressourcenzugriff. Dieses Symbol lässt sich meist leichter beschriften.
	Eine Funktionseinheit, die ein <b>Portal</b> darstellt. Portale sind zuständig für den Zugriff des Clients auf das System (UI, GUI, Webservice, etc.).
<b>Functional Units with State / Funktionseinheiten mit Zustand</b>	
Jede Funktionseinheit kann Zustand halten, ohne dass dies speziell gekennzeichnet werden muss. Wenn es der Verständlichkeit dient, kann die „Tonne“ ergänzt werden.	
	Eine Funktionseinheit, die Zustand hält. Dient der Präzisierung, insbesondere bei gemeinsamem Zustand (shared state).
	Präzisierung des Zustands. Die Funktionseinheit A greift auf den Zustand x zu.
Zustand kann auch in den Flow gestellt werden, statt ihn in Funktionseinheiten zu integrieren.	
	Setzen des Zustands.
	Lesen des Zustands

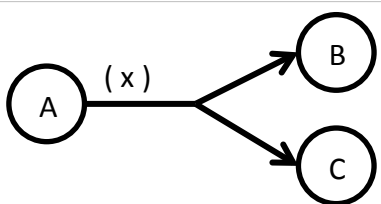
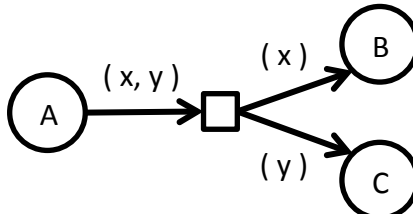
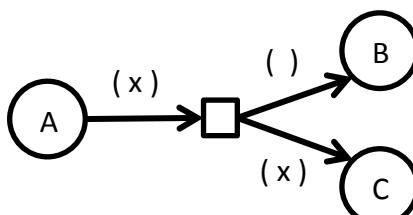
Connections / Verbindungen	
	Datenfluss: ein x fließt von A nach B.
	Abhängigkeit: A hängt ab von B.
Data Flows / Datenflüsse	
	Ein x fließt von A nach B.
	Ein Tupel bestehend aus einem x und einem y fließt von A nach B.
	Es fließen viele x von A nach B. Der Stern steht für die Wiederholung. In der Implementation kann dies ein Array, eine Liste, etc. sein.
	Es fließt mehrfach ein x von A nach B (0..n).
	Präzisierung von (x)* Es fließt mehrfach ein x von A nach B (0..n).
	Präzisierung von (x)* Es fließt mehrfach ein x von A nach B, mindestens einmal (1..n).
	Präzisierung von (x)* Optional fließt ein x von A nach B (0..1).
	Beispiel: <ul style="list-style-type: none"> <li>- es fließt mehrfach (geschweifte Klammern)</li> <li>- eine Liste von Tupeln (äußerer Stern)</li> <li>- jedes Tupel besteht aus einem x und einer Liste von y</li> </ul>

### Data Types / Datentypen

Per Konvention gilt: ist die Nachricht klein geschrieben, handelt es sich um einen Standardtyp wie string, int, bool, etc. Großschreibung bedeutet, dass es sich um einen eigenen Datentyp handelt.

	<p>count wird per Konvention in einen int übersetzt.</p>
	<p>Ist der Typ nicht klar erkennbar, kann er explizit ausgeschrieben werden.</p>
	<p>Großbuchstabe am Anfang bedeutet per Konvention: es handelt sich um einen eigenen Typ. Dieser wird tabellarisch beschrieben.      In der tabellarischen Typbeschreibung können die Typen der Felder weggelassen werden, wenn sie sich aus dem Kontext ergeben.</p>

### Split

	<p>Bei der einfachsten Form eines Splits fließen die Daten unverändert zu mehreren Funktionseinheiten.</p>
	<p>Durch einen Split kann ein Datenstrom aufgeteilt werden. Hier wird das Tupel aufgeteilt.</p>
	<p>In diesem Beispiel fließt das x nur an C, B erhält keinen Input.</p>

	<p>A produziert ein x. Dieses wird allerdings nicht als Input an B weitergereicht, sondern durch den Split ignoriert.</p>
--	---

	<p>Kurzschreibweise für den Split. Alternativ zum Schrägstrich „/“ kann auch der Pipeslash „ “ verwendet werden.</p>
--	--

**Alternative**

	<p>Entweder A oder B produziert ein x. Dieses fließt an C.</p>
--	--

**Join**

	<p>A produziert ein x, B produziert ein y. Beide fließen zusammen als Tupel an C.</p>
--	---

	<p>Per Konvention wird das Tupel auf dem Datenfluss weggelassen.</p>
--	--

	<p>A produziert ein x, das aber nicht an B fließt (Split). B produziert ein y. Sowohl x als auch y werden mittels Join zu einem Tupel zusammengefasst und an C geliefert.</p>
--	---

	<p>Kurzschreibweise für den Join aus x und y. B produziert ein y. An C wird das y sowie das von A produzierte x als Tupel weitergereicht.</p>
--	---

**Iterations / Aufzählungen**

	<p>Schleifen sollen in Flows nicht vorkommen. Eine Aufteilung in die Schleife und eine Funktionseinheit, welche für ein einzelnes Element verantwortlich ist, kann als Abhängigkeit modelliert werden.</p>
	<p>Kurznotation für die Aufteilung in zwei Funktionseinheiten wie oben.</p>

## Start der Anwendung

	<p>Start der Anwendung durch das Betriebssystem. Es liefert die Kommandozeilenargumente.</p>
--	--

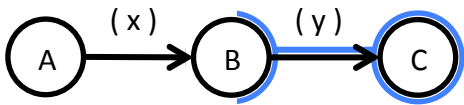
## Konstruktoraufruf

	<p>Der Konstruktor der Klasse <i>MyType</i> wird mit <i>x</i> als Argument aufgerufen. Ergebnis ist eine Instanz vom Typ <i>MyType</i>.</p>
--	---

## Exception / Ausnahme

	<p>Die Funktionseinheit A löst im Fehlerfall eine Exception aus.</p>
	<p>Die Funktionseinheit A liefert entweder ein <i>y</i> oder löst im Fehlerfall eine Exception aus.</p>

## Multithreading

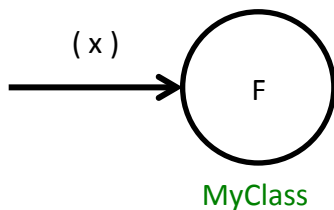


Die Funktionseinheit B produziert ein y. Dies geschieht allerdings auf einem anderen Thread, hier in blau eingefärbt. Die Ausführung von B beginnt somit auf dem einen Thread und wird auf dem anderen Thread fortgesetzt.

## Implementation

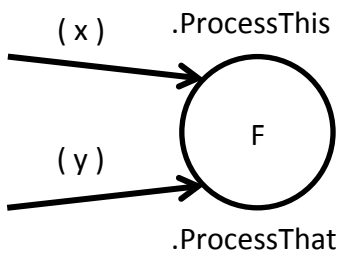
### Inputs

#### Single Path Input



```
public class MyClass
{
    public void F(int x) {
        // ...
    }
}
```

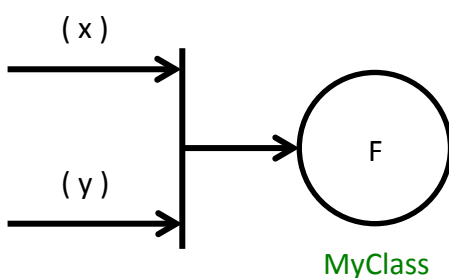
#### Multiple Path Input



```
public class F
{
    public void ProcessThis(int x) {
        // ...
    }

    public void ProcessThat(string y) {
        // ...
    }
}
```

#### Joined Input



```
public class MyClass
{
    public void F(int x, int y) {
        // ...
    }
}
```



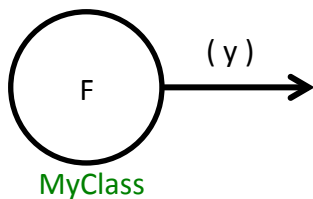


# Clean Code Developer School

Saubere Softwareentwicklung üben  
regelmäßig · fokussiert · individuell · angeleitet

## Outputs

### Single Path Output



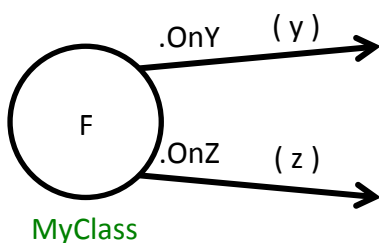
```
public class MyClass
{
    public int F() {
        // ...
        return 42;
    }
}
```

```
public class MyClass
{
    public void F(Action<int> onResult) {
        // ...
        onResult(42);
    }
}
```

```
public class MyClass
{
    public event Action<int> OnResult;

    public void F() {
        // ...
        OnResult(42);
    }
}
```

### Multiple Path Output



```
public class MyClass
{
    public void F(
        Action<int> onY, Action<int> onZ) {
        // ...
        onY(56);
        onZ(42);
    }
}
```

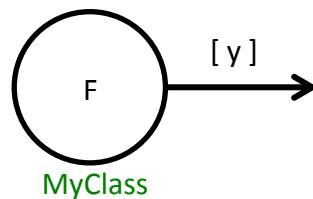
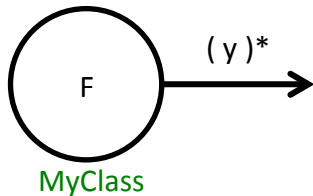


```
public class MyClass
{
    public event Action<int> OnY;

    public event Action<int> OnZ;

    public void F() {
        // ...
        OnY(56);
        OnZ(42);
    }
}
```

## Optional Output

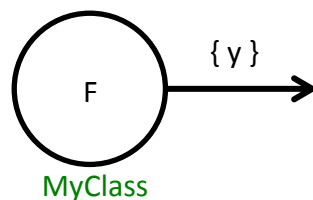
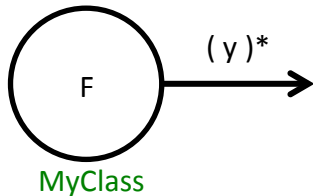


```
public class MyClass
{
    public void F(Action<int> onResult) {
        // ...
        onResult(42);
    }
}
```

```
public class MyClass
{
    public event Action<int> OnResult;

    public void F() {
        // ...
        OnResult(42);
    }
}
```

## Streamed Output



```
public class MyClass
{
    public void F(Action<string> onResult) {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            onResult(s);
        }
        onResult(null); // end-of-stream
    }
}
```

```
public class MyClass
{
    public event Action<string> OnResult;

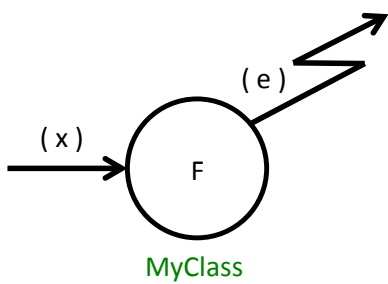
    public void F() {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            OnResult(s);
        }
        OnResult(null); // end-of-stream
    }
}
```

```
public class MyClass
{
    public IEnumerable<string> F() {
        // ...
        foreach(var s in new[]{"1", "2"}) {
            yield return s;
        }
    }
}
```

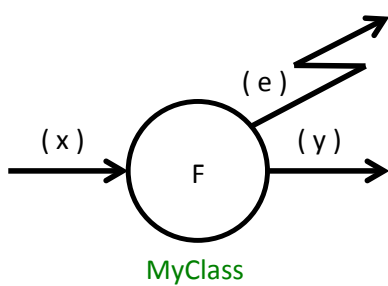
## Exceptions / Ausnahmen

# Clean Code Developer School

Saubere Softwareentwicklung üben  
regelmäßig, fokussiert, individuell, angeleitet



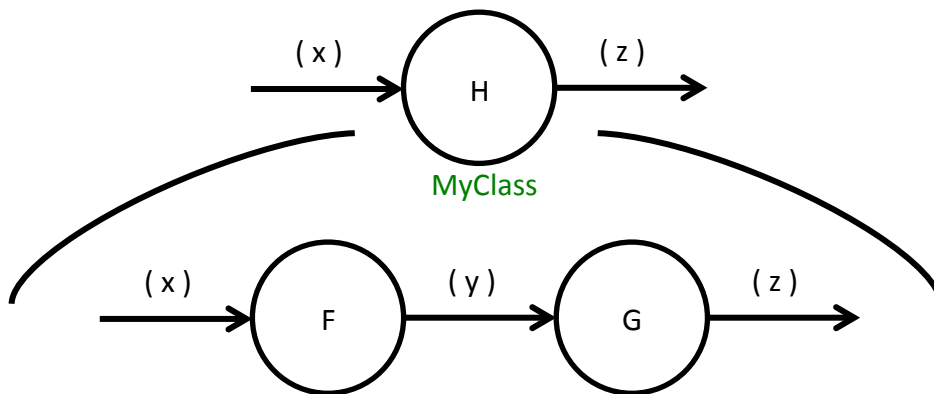
```
public class MyClass
{
    public void F(int x) {
        // ...
        if(...) {
            throw new Exception("Error");
        }
    }
}
```



```
public class MyClass
{
    public int F(int x) {
        // ...
        if(...) {
            throw new Exception("Error");
        }
        else {
            return y;
        }
    }
}
```

## Integration

### Hierarchical Data Flow / Hierarchischer Datenfluss



```
public class MyClass
{
    public int H(int x) {
        var y = F(x);
        var z = G(y);
        return z;
    }

    public int F(int x) {
        return x + 1;
    }

    public int G(int y) {
        return y * 2;
    }
}
```

# Clean Code Developer School

Saubere Softwareentwicklung üben  
regelmäßig, fokussiert, individuell, angeleitet

```
public class MyClass
{
    public void H(int x, Action<int> continueWith) {
        F(x, y => G(y, continueWith));
    }

    public void F(int x, Action<int> continueWith) {
        var y = ...
        continueWith(y);
    }

    public void G(int y, Action<int> continueWith) {
        var z = ...
        continueWith(z);
    }
}

public class Integration
{
    public void H(int x) {
        var operations = new Operations();

        operations.Result_of_F += operations.G;
        operations.Result_of_G += z => Result_of_H(z);

        operations.F(x);
    }

    public event Action<int> Result_of_H;
}

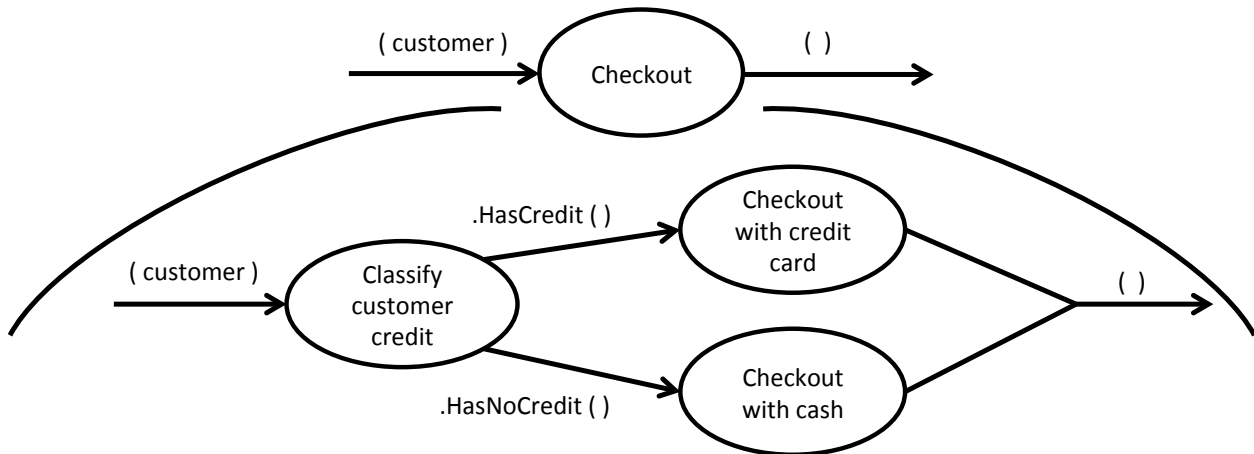
public class Operations {
    public void F(int x) {
        var y = ...
        Result_of_F(y);
    }

    public event Action<int> Result_of_F;

    public void G(int y) {
        var z = ...
        Result_of_G(z);
    }

    public event Action<int> Result_of_G;
}
```

## Branching Data Flow / Verzweigender Datenfluss



Implementation mittels Continuations.

```
public class Checkout_Processor
{
    public void Checkout(Customer customer) {
        Classify_customer_credit(customer,
            Checkout_with_credit_card,
            Checkout_with_cash);
    }

    public void Classify_customer_credit(
        Customer customer, Action has_credit, Action has_no_credit) {
        if(customer.Balance >= 1000.0) {
            has_credit();
        } else {
            has_no_credit();
        }
    }

    public void Checkout_with_credit_card() {
        // ...
    }

    public void Checkout_with_cash() {
        // ...
    }
}
```



# Clean Code Developer School

Saubere Softwareentwicklung üben  
regelmäßig, fokussiert, individuell, angeleitet

```
public class Customer
{
    public string Name { get; set; }

    public double Balance { get; set; }

    // ...
}
```

Implementation mittels if in der Integration.

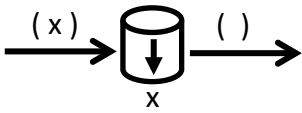
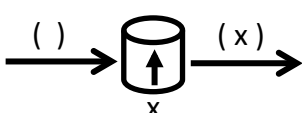
```
public class Checkout_Processor
{
    public void Checkout(Customer customer) {
        if(Classify_customer_credit(customer)) {
            Checkout_with_credit_card();
        }
        else {
            Checkout_with_cash();
        }
    }

    public bool Classify_customer_credit(Customer customer) {
        return customer.Balance >= 1000.0;
    }

    public void Checkout_with_credit_card() {
        // ...
    }

    public void Checkout_with_cash() {
        // ...
    }
}
```

## State / Zustand

 <p>A diagram showing a cylinder representing a stateful object. An arrow labeled '(x)' points into the cylinder from the left. Inside the cylinder, a downward-pointing arrow indicates the state. An arrow labeled '()' points out of the cylinder to the right. Below the cylinder is the letter 'x'.</p>	<pre>public class State&lt;T&gt; {     private T state;      public void Put(T state) {         this.state = state;     }      public T Get() {         return state;     } }</pre>
 <p>A diagram showing a cylinder representing a stateful object. An arrow labeled '()' points into the cylinder from the left. Inside the cylinder, an upward-pointing arrow indicates the state. An arrow labeled '(x)' points out of the cylinder to the right. Below the cylinder is the letter 'x'.</p>	