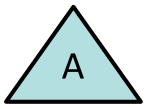
CHEATSHEET FLOW DESIGN

SYMBOLE

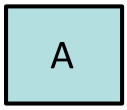
Functional Units / Funktionseinheiten
Funktionseinheit ist der Überbegriff für Methode, Klasse, Bibliothek, etc

Eine Funktionseinheit A, die **Domänenlogik** enthält.



Eine Funktionseinheit, die einen **Ressourcenzugriff** darstellt.

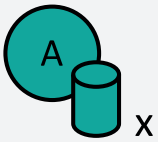
Ebenfalls ein **Ressourcenzugriff.** Dieses Symbol lässt sich meist leichter beschriften.



Eine Funktionseinheit, die ein **Portal** darstellt. Portale sind zuständig für den Zugriff des Clients auf das System (UI, GUI, WebService, etc.).

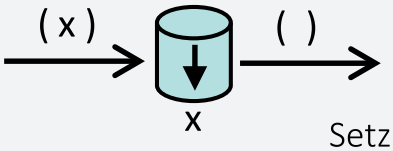
Functional Units with State / Funktionseinheiten mit Zustand
Jede Funktionseinheit kann Zustand halten, ohne dass dies speziell gekennzeichnet werden muss.
Wenn es der Verständlichkeit dient, kann die „Tonne“ ergänzt werden.

Eine Funktionseinheit, die Zustand hält. Dient der Präzisierung, insbesondere bei gemeinsamem Zustand (shared state).

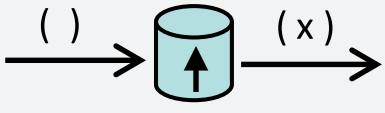


Präzisierung des Zustands. Die Funktionseinheit A greift auf den Zustand x zu.

Zustand kann auch in den Flow gestellt werden, statt ihn in Funktionseinheiten zu integrieren.

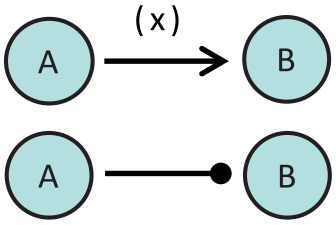


Setzen des Zustands



Lesen des Zustands

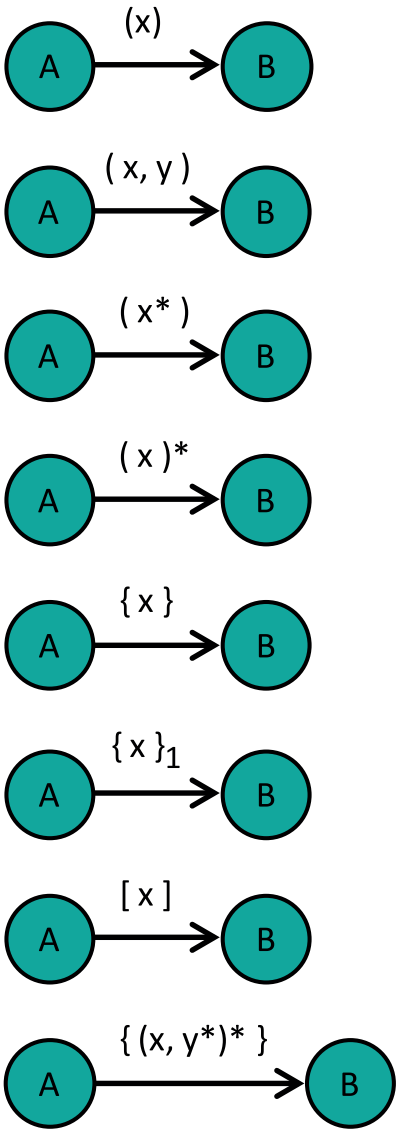
Connections / Verbindungen



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).

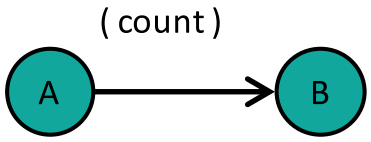
Präzisierung von (x)* Es fließt mehrfach ein x von A nach B, mindestens einmal (1..n).

Präzisierung von (x)* Optional fließt ein x von A nach B (0..1).

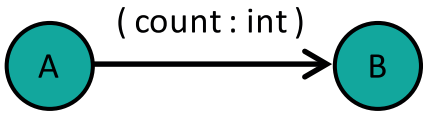
Beispiel:
- es fließt mehrfach (geschweifte Klammern)
- eine Liste von Tupeln (äußerer Stern)
- jedes Tupel besteht aus einem x und einer Liste von y

Data Types / Datentypen

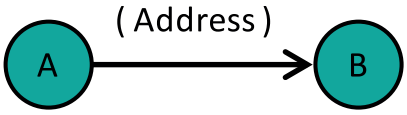
Per Konvention gilt: ist die Nachricht klein geschrieben, handelt es sich um einen Standardtyp wie string, int, bool, etc. Großschreibung bedeutet, dass es sich um einen eigenen Datentyp handelt.



count wird per Konvention in einen int übersetzt.



Ist der Typ nicht klar erkennbar, kann er explizit ausgeschrieben werden.

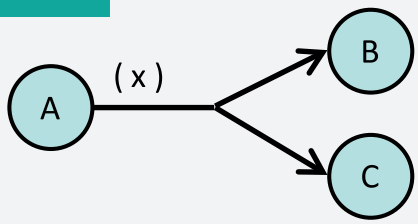


Address

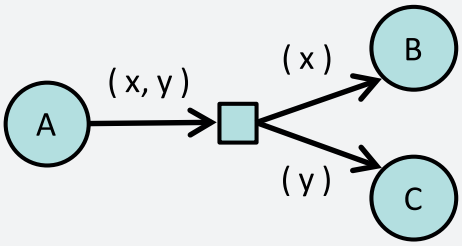
- Street
- Postcode : string
- Town

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.

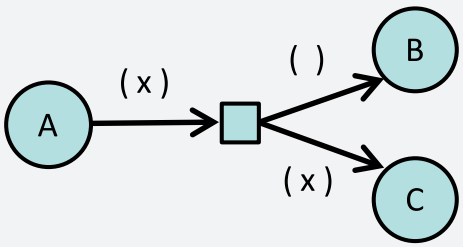
Split



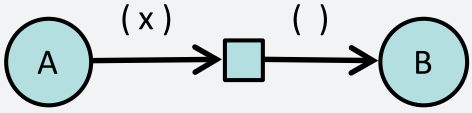
Bei der einfachsten Form eines Splits fließen die Daten unverändert zu mehren Funktionseinheiten.



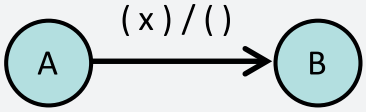
Durch einen Split kann ein Datenstrom aufgeteilt werden. Hier wird das Tupel aufgeteilt.



In diesem Beispiel fließt das x nur an C, B erhält keinen Input.

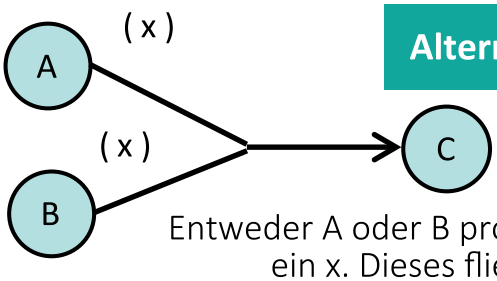


A produziert ein x. Dieses wird allerdings nicht als Input an B weitergereicht, sondern durch den Split ignoriert.



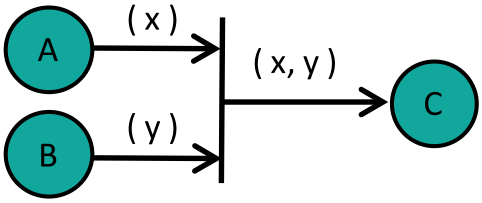
Kurzschreibweise für den Split. Alternativ zum Schrägstrich „/“ kann auch der Pipeslash „|“ verwendet werden.

Alternative

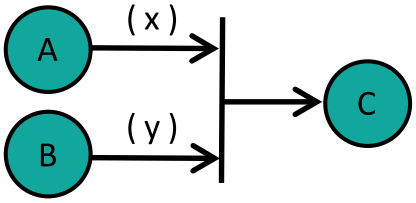


Entweder A oder B produziert ein x. Dieses fließt an C.

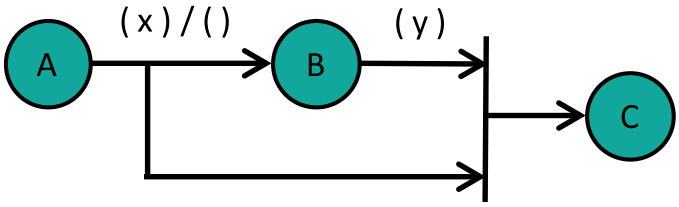
Join



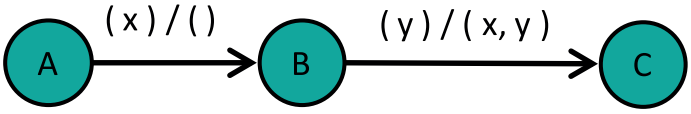
A produziert ein x, B produziert ein y. Beide fließen zusammen als Tupel an C.



Per Konvention wird das Tupel auf dem Datenfluss weggelassen.

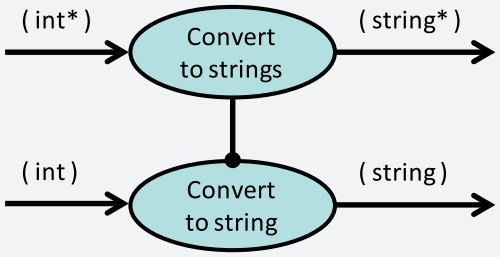


A produziert ein x, das aber nicht an B fließt (Split). B produziert ein y. Sowohl x als auch y werden mittels Join zu einem Tupel zusammengefasst und an C geliefert.

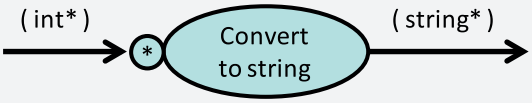


Kurzschreibweise für den Join aus x und y. B produziert ein y. An C wird das y sowie das von A produzierte x als Tupel weitergereicht.

Iterations / Aufzählungen

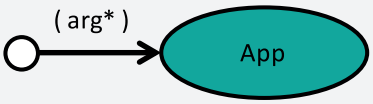


Schleifen sollen in Flows nicht vorkommen. Eine Aufteilung in die Schleife und eine Funktionseinheit, welche für ein einzelnes Element verantwortlich ist, kann als Abhängigkeit modelliert werden.



Kurznotation für die Aufteilung in zwei Funktionseinheiten wie oben.

Start der Anwendung

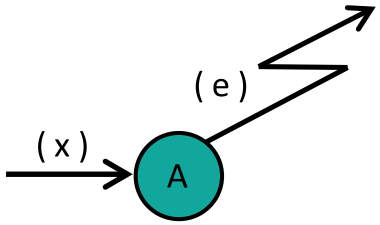


Start der Anwendung durch das Betriebssystem. Es liefert die Kommandozeilenargumente.

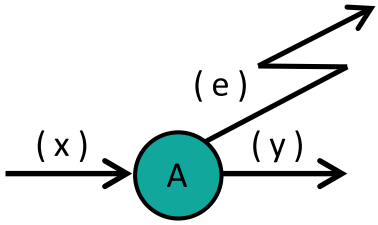
Konstruktoraufruf

Der Konstruktor der Klasse MyType wird mit x als Argument aufgerufen. Ergebnis ist eine Instanz vom Typ MyType.

Exception / Ausnahme

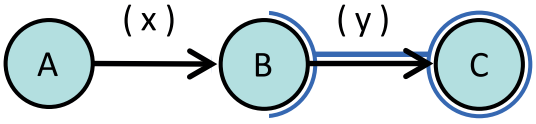


Die Funktionseinheit A löst im Fehlerfall eine Exception aus.



Die Funktionseinheit A liefert entweder ein y oder löst im Fehlerfall eine Exception aus.

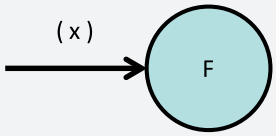
Multithreading



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

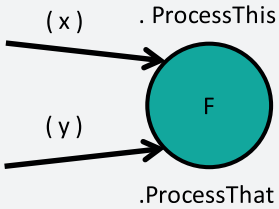
Single Path Input



MyClass

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

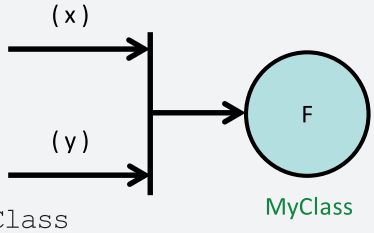
Multiple Path Input



.ProcessThat

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

Joined Input

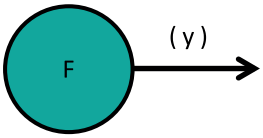


MyClass

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

Outputs

Single Path Output

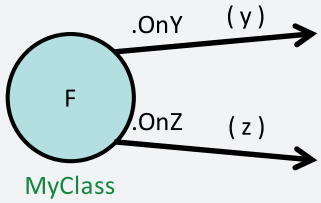



```
public class MyClass
{
    public int F( ) {
        // ...
        return 42;
    }
}
```

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

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

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

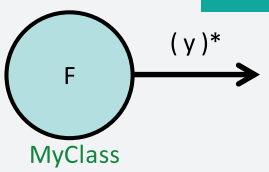
Multiple Path Output

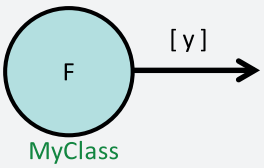



```
public class MyClass
{
    public void F(
        Action<int> onY, Action<int> onZ) {
            // ...
            onY(56);
            onZ(42);
    }
}
```

```
public class MyClass
{
    public event Action<int> OnY;
    public event Action<int> OnZ;
    public void F( ) {
        // ...
        OnY(56);
        OnZ(42);
    }
}
```

Optional Output

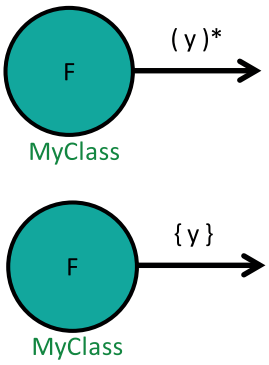





```
public class MyClass
{
    public void F(Action<int> onResult) {
        // ...
        onResult(42);
    }
}
```

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

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

Streamed Output

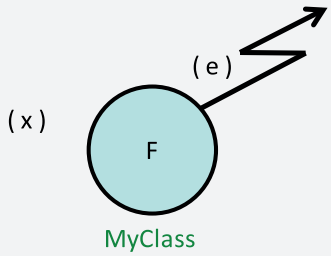



```
public class MyClass
{
    public void F(Action<string> onResult) {
        // ...
        foreach(var s in new [ ] {"1", "2"}) {
            onResult(s);
        }
        onResult(null); // end-of-stream
    }
}
```

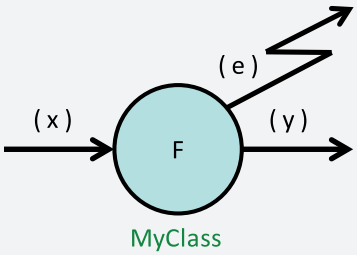
```
public class MyClass
{
    public event Action<string> OnResult;

    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 sin new [ ] {"1", "2"}) {
            yield return s;
        }
    }
}
```

Exceptions / Ausnahmen



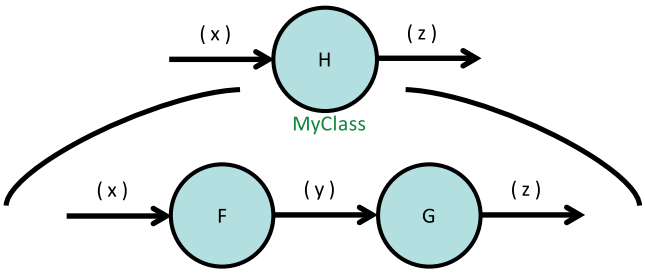

```
public class MyClass {
    public void F(int x) {
        // ...
        if(...) {
            throw new Exception("Error");
        }
    }
}
```




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

Integration

Hierarchical Data Flow / Hierarchischer Datenfluss



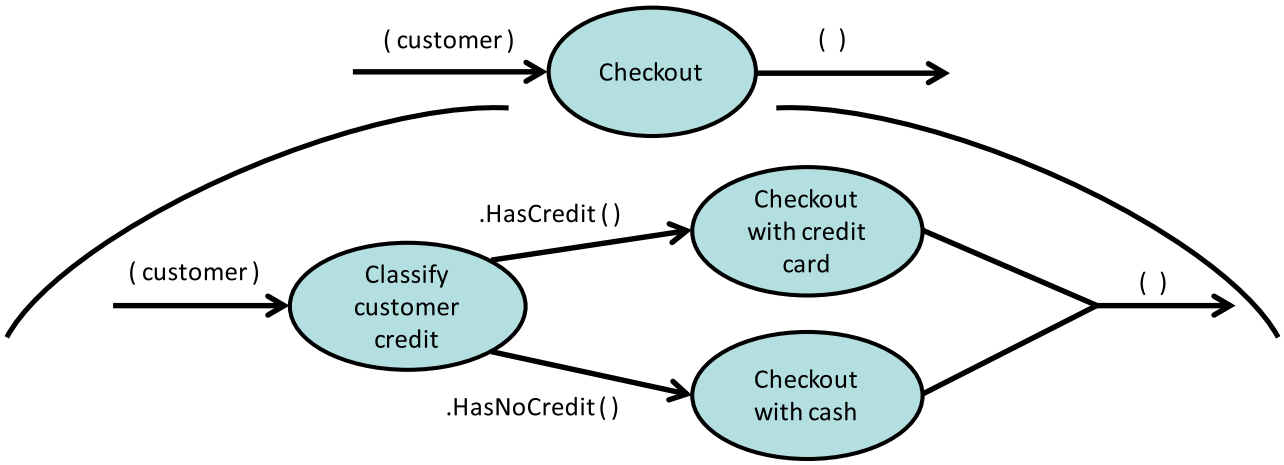

```
public class MyClass
{
    public void H(int x, Action<int> continueWith) {
        F(x, y = > G(y, continueWith));
    }
    public void F(int x, Action<int> continueWith) {
        var y = ...
        continueWith (y);
    }
    public void G(int y, Action<int> continueWith) {
        var z = ...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

```
public class 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;
    }
}
```

```
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.

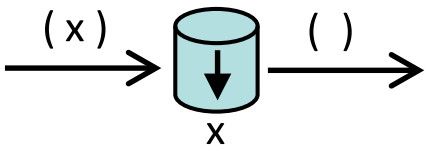
```
public class Checkout_Processor
{
    public void Checkout(Customer customer) {
            Classify_customer_credit(customer,
                Checkout_with_credit_card,
                 Checkout_with_cash);
    }
    public void Classify_customer_credit(
        Customer customer, Action has_credit,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

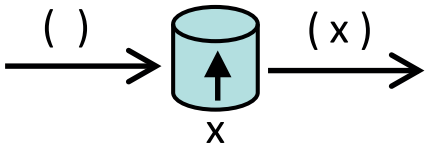
```
public class Customer
{
    public string Name { get; set; }

    public double Balance { get; set; }

    // ...
}
```

Implementation mittels if in der Integration.

```
public class Checkout_Processor
{
    public void Checkout (Customк 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






```
public class State <T>
{
    private T state;

    public void Put(T state) {
        this. state = state;
    }

    public T Get( ) {
        return  state;
    }
}
```