

# Functional Domain Modeling

---



## 1 Indirizzo di consegna

[Modifica](#)

Cosa succede se non sono a casa? [Indica dove possiamo lasciare il tuo pacco](#)  
Oppure seleziona un punto di ritiro - [Scegli](#)

## 2 Modalità di pagamento

 Visa / Electron termina cor[Modifica](#)

[Indirizzo di fatturazione:](#) Giovanni .

^ Aggiungi un buono regalo o un codice promozionale

## 3 Rivedi gli articoli e la spedizione



**Giovanni, come cliente Amazon Prime risparmi EUR 5,99 sulle spese di spedizione di questo ordine con la Spedizione 1 giorno senza costi aggiuntivi.**

Ti ringraziamo per essere un cliente Amazon Prime. Approfitta di consegne illimitate in 1 giorno su oltre 2 milioni di prodotti e in 2-3 giorni su molti altri milioni.

**Data di consegna garantita: 6 feb. 2019** Se effettui l'ordine nei prossimi 3 ore e 48 minuti ( [Dettagli](#) )

Articoli di Amazon.it



**BRITA filtri MAXTRA+ Pack 6, Cartucce per caraffe filtranti, 6 filtri x 6 mesi di acqua filtrata**

**EUR 24,56** ✓prime

Q.tà: 1 ▼

Venduto da: Amazon EU S.a.r.L.

**Disponibilità immediata.**

[Aggiungi opzioni regalo](#)

**L'articolo arriva in un imballaggio che rivela il contenuto.** Per nascondere, scegli **Spedizione in imballaggio Amazon.**

**Scegli la tua modalità di spedizione Prime:**

☒ GRATIS 1 giorno — **Consegna garantita entro domani 6 feb.**

☐ GRATIS Standard - **Consegna venerdì 8 feb.**

Confermando il tuo ordine accetti integralmente le nostre [Condizioni generali di uso e vendita](#). L'acquisto sarà completato solo con la conferma di spedizione. Prendi visione della nostra [Informativa sulla privacy](#), della nostra [Informativa sui Cookie](#) e della nostra [Informativa sulla Pubblicità definita in base agli interessi](#).

## Riepilogo Ordine

Articoli: EUR 24,56

Costi di spedizione: EUR 0,00

**Totale ordine: EUR 24,56**

Il totale ordine include l'IVA. [Dettagli](#)

[Come vengono calcolate le spese di spedizione?](#)

[Scopri come modificare le preferenze per le consegne il sabato. Ti ricordiamo che il servizio è disponibile solo in alcune aree..](#)

Tutti gli articoli che possono essere spediti con Amazon Prime saranno inviati con tale opzione.

```
type ShippingAddress = {  
  name: string;  
  street: string;  
  city: string;  
  cap: string;  
  
  isCompanyPickup: boolean;  
  pickupCompany?: string;  
}
```

```
type Step = (info: CheckoutInfo) => CheckoutInfo

function checkout(info: CheckoutInfo): CheckoutInfo {
    return processShippingAddress(info)
        .then(processPaymentMethod)
        .then(processShippingCourier)
        .then(validate)
        .then(confirmCheckout)
}
```

# Assumptions

- FP concepts
  - purity
  - composition
  - immutability
- Functor, Applicative, Monad, Semigroup/Monoid

# Domain Model <sup>1</sup>

1. **Entities**
2. **Behaviours**
3. Ubiquitous language

---

<sup>1</sup> Domain Driven Design

1. Model entities with **immutable** Algebraic Data Types (ADT)
2. Model behaviours as pure functions in modules
3. Behaviours operate on the types of the entities

```
export type ShippingAddress = Readonly<{  
  name: string;  
  street: string;  
  city: string;  
  cap: string;  
  
  isCompanyPickup: boolean;  
  pickupCompany?: string;  

```

```
export type PaymentMethod = Readonly<{  
  paymentMethod: string;  
  cardNumber: string;  
  cardExpiration: string;  
  cardCode: string;  

```

```
export type ShippingCourier = Readonly<{  
  shippingMethod: string;  
  estimate: number;  

```

```
export type CheckoutInfo = ShippingAddress & PaymentMethod & ShippingCourier;
```



```
import { ShippingAddress, PaymentMethod, ShippingCourier } from "../domain.ts";  
  
export function processShippingAddress(info: CheckoutInfo): CheckoutInfo {}  
  
export function processPaymentMethod(info: CheckoutInfo): CheckoutInfo {}  
  
export function processShippingCourier(info: CheckoutInfo): CheckoutInfo {}  
  
export function validate(info: CheckoutInfo): CheckoutInfo {  
    if (!info.cardCode) throw new Error('Invalid card');  
  
    return info;  
}  
  
export function confirmCheckout(info: CheckoutInfo): CheckoutInfo {}
```

# Algebraic Data Types

# Sum type

Union of sets.

```
type PaymentMethod = 'visa' | 'mastercard'
```

```
//: < l1: 'visa', l2: 'mastercard' >
```

# Option

```
type Option<A> =  
  | { type: 'None' }  
  | {  
    type: 'Some',  
    value: A  
  }
```

# When to use sum types?

To model the variations within an entity

```
export type ShippingAddress = Readonly<{
  name: string;
  street: string;
  city: string;
  cap: string;

  isCompanyPickup: boolean;
  pickupCompany: Option<string>;
}>;

type PaymentMethod = 'Visa' | 'Mastercard'
export type PaymentMethod = Readonly<{
  paymentMethod: PaymentMethod;
  cardNumber: string;
  cardExpiration: string;
  cardCode: string;
}>;

type ShippingMethod = 'Prime' | 'Standard';
export type ShippingCourier = Readonly<{
  shippingMethod: ShippingMethod;
  estimate: number;
}>;
```

# Pattern matching

```
function getCourier(shippingMethod: ShippingMethod) {  
    switch (shippingMethod) {  
        case 'Standard':  
            return 3;  
        case 'Prime':  
            return 1;  
        // No default case  
    }  
}
```

# Sum types vs Inheritance

```
interface ShippingMethod {...}  
class PrimeMethod implements ShippingMethod {...}  
class StandardMethod implements ShippingMethod {...}
```



```
type CardType = 'Visa' | 'Mastercard'
type CardPayment = Readonly<{
  type: 'CardPayment';
  card: CardType;
  cardNumber: string;
  cardExpiration: string;
  cardCode: string;
}>;
```

```
type CashPayment = {
  type: 'CashPayment';
}
```

```
type ChequePayment = {
  type: 'ChequePayment';
}
```

```
export type PaymentMethod = CardPayment | CashPayment | ChequePayment
```

# Inheritance cons

- What is the common behaviour?
- State and behaviour coupling
- Data and code is scattered around many locations

# Product type

Product of sets.

```
type ShippingMethod = 'Prime' | 'Standard'  
type CourierTuple = [ShippingMethod, number]
```

```
//: ShippingMethod * number
```

# Product type

```
type CourierRecord = {  
  shippingMethod: 'Prime' | 'Standard';  
  estimate: number;  
};
```

# Tuple <=> Record

There is an isomorphism between the types **tuples** and **record**

```
from . to = to . from = identity
```

```
function toRecord(tuple: CourierTuple): CourierRecord {  
  return { shippingMethod: tuple[0], estimate: tuple[1] }  
}
```

```
function fromRecord(record: CourierRecord): CourierTuple {  
  return [record.shippingMethod, record.estimate]  
}
```

# When to use product types?

To model the related data which form larger abstractions.

# NonEmptyArray

```
type NonEmptyArray<A> = {  
  head: A;  
  tail: Array<A>  
}
```

# Sum types again



# Sum types again - Atomic updates

```
export type ShippingAddress = Readonly<{  
  name: string;  
  street: string;  
  city: string;  
  cap: string;  
  
  isCompanyPickup: boolean;  
  pickupCompany: Option<string>;  
}>;
```

# Sum types again - Atomic updates

```
type HomePickup = { type: 'HomePickup' }  
type CompanyPickup = { type: 'CompanyPickup', pickupCompany: string }  
type PickupType = HomePickup | CompanyPickup
```

```
export type ShippingAddress = Readonly<{  
  name: string;  
  street: string;  
  city: string;  
  cap: string;  
  
  pickup: PickupType;  

```

# Sum types again - Either

```
type Either<L, R> =  
  | {  
    type: 'Left',  
    left: L  
  }  
  | {  
    type: 'Right',  
    right: R  
  }
```

# Either

```
import { left, right } from 'fp-ts/lib/Either';

export function verifyPaymentMethod(payment: PaymentMethod)
  : Either<string, PaymentMethod> {
  if (!payment.cardCode) return left('Invalid card');

  return right(payment);
}
```

# Either Monad - Fail fast

```
export function verifyShippingAddress(address: ShippingAddress)
  : Either<string, ShippingAddress> {}

export function verifyPaymentMethod(payment: PaymentMethod)
  : Either<string, PaymentMethod> {}

export function verifyCheckoutInfo(checkoutInfo: CheckoutInfo)
  : Either<string, CheckoutInfo> {
  // Structural subtyping
  return verifyShippingAddress(checkoutInfo)
    .chain(verifyPaymentMethod)
}
```

# Validation Applicative - Accumulate failures

```
import { validation, failure, success } from 'fp-ts/lib/Validation'
import { traverse } from 'fp-ts/lib/Array'

export function verifyShippingAddress(address: ShippingAddress)
  : Validation<NonEmptyArray<string>, ShippingAddress> {}

export function verifyPaymentMethod(payment: PaymentMethod)
  : Validation<NonEmptyArray<string>, PaymentMethod> {}

export function verifyCheckoutInfo(checkoutInfo: CheckoutInfo)
  : Validation<NonEmptyArray<string>, CheckoutInfo> {
  const validators = [verifyShippingAddress, verifyPaymentMethod];
  return traverse(validation)(validators, f => f(checkoutInfo))
}

// failure(new NonEmptyArray("Invalid address", ["Invalid card"]))
verifyCheckoutInfo({ ... })
```

# UI State

Make illegal states unrepresentable

```

type State = {
  isFillingAddress: boolean;
  isFillingPayment: boolean;
  isFillingCourier: boolean;
} & CheckoutInfo

class CheckoutForm extends React.Component<{}, State> {
  state: State = {
    isFillingAddress: true,
    isFillingPayment: false,
    isFillingCourier: false,

    name: '';
    street: '';
    city: '';
    ...
    paymentMethod: '';
    cardNumber: '';
    cardExpiration: '';
    cardCode: '';
  }

  render() {
    return (
      <form> ... </form>
    );
  }
}

```



```
type FillingAddressState = { type: 'FillingAddressState' }  
  & ShippingAddress
```

```
type FillingPaymentState = { type: 'FillingPaymentState' }  
  & ShippingAddress & PaymentMethod
```

```
type FillingCourierState = { type: 'FillingCourierState' }  
  & ShippingAddress & PaymentMethod & ShippingCourier
```

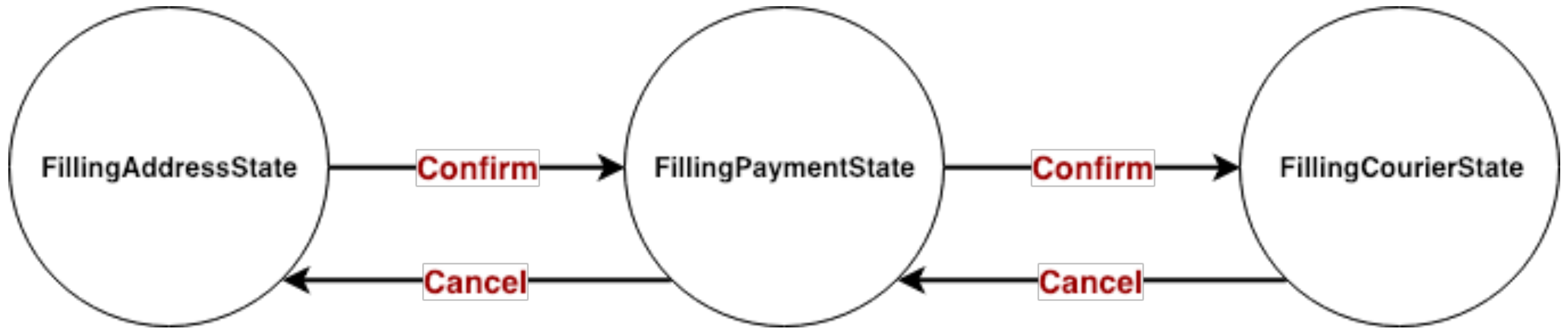
```
type State =  
  | FillingAddressState  
  | FillingPaymentState  
  | FillingCourierState
```

```
class CheckoutForm extends React.Component<{}, State> {
  state: State = {
    type: 'FillingAddressState',

    name: '';
    street: '';
    city: '';
    cap: '';
    pickup: { type: 'HomePickup' }
  }

  render() {
    switch (this.state.type) {
      case 'FillingAddressState':
        return this.renderAddressFields();
      case 'FillingPaymentState':
        return this.renderPaymentFields();
      case 'FillingCourierState':
        return this.renderCourieFields();
    }
  }
}
```

# Finite state machine



# Finite state machine

- Each state has different possible data
- Forces to think about possible states
- States and transitions are types

# Phantom Types

---

```
type FormData<A> = CheckoutInfo;

type Unvalidated = { type: 'Unvalidated' }
type Validated = { type: 'Validated' }

function createFormData(data: CheckoutInfo)
  : FormData<Unvalidated> {}

function setShippingAddress(data: FormData<Unvalidated>, address: ShippingAddress)
  : FormData<Unvalidated> {}

function validate(data: FormData<Unvalidated>)
  : FormData<Validated> {}

function confirmCheckout(data: FormData<Validated>)
  : FormData<Validated> {}
```

# References

- Functional and Algebraic Domain Modeling - Debasish Ghosh - DDD Europe 2018
- Domain Modeling Made Functional - Scott Wlaschin
- Functional and Reactive Domain Modeling
- Algebraic Data Types - Giulio Canti