DEPENDENCY INJECTION & DESIGN PATTERNS

AGENDA

F

WHAT IS A DEPENDENCY?

DEPENDENCY

Introduces a level of coupling in your code

- Sometimes your code becomes resistant to change
- You can’t test anything in isolation
- You can’t reuse code
  - Code readability - bad
  - Bigger code source
  - Low code quality

DEPENDENCY SIMPTOMS

- Rigidity
  - Difficult to change – if affect other parts
- Fragility
  - One change breaks the hell loose
- Immobility
  - Difficult to reuse code

TYPES OF DEPENDENCIES

VISIBLE

```
public class CustomerAccount
{
   private BankAccount currentAccount;

   public CustomerAccount(BankAccount currentAccount)
   {
         this.currentAccount = currentAccount;
    }
}
```

HIDDEN

```
public class SurvivalAccount
{
   private IBankAccount shoesAccount;
   private IBankAccount handBagsAccount;

    public SurvivalAccount()
    {
      this.shoesAccount = new ShoesAccount();
      this.handBagsAccount = new HandBagsAccount();
    }
}
```

VOLATILE

- Require setup or a configuration
- Implementation of dependency hasn’t been created yet.
- Can be a third party library that requires a license.
- Have non-deterministic behavior => can’t be tested

- Looked at from the environment perspective

TIGHT COUPLING

```
public class BankAccount
 {
   private SavingsAccount savingsAccount = new SavingsAccount();
   private CurrentAccount currentAccount = new CurrentAccount();

  public decimal GetTotalForAccount(Guid accountNumber)
  {
    decimal currentAccountMoney = this.currentAccount.GetMoneyByAccountNumber(accountNumber);
    decimal savingsAccountMoney = this.savingsAccount.GetMoneyByAccountNumber(accountNumber);
    return currentAccountMoney + savingsAccountMoney;
   }
 }
```

LOOSE COUPLING

```
public class BankAccount
 {
   private IBankAccount currentAccount;
   private IBankAccount savingsAccount;

   public BankAccount (IBankAccount currentAccount, IBankAccount savingsAccount)
   {
     this.currentAccount = currentAccount;
     this.savingsAccount = savingsAccount;
   }
}
```

LOOSE COUPLING – how is it achieved?

- Through interfaces because
  - you can inject any implementation you want

- Is it bad?

  Let’s see

```
public class SurvivalAccount
{
   private ShoesAccount shoesAccount;
   private HandBagsAccount handBagsAccount;

    public SurvivalAccount(ShoesAccount shoesAccount, HandBagsAccount handBagsAccount )
    {
      this.shoesAccount = shoesAccount;
      this.handBagsAccount = handBagsAccount;
    }
}
```

```
public class SurvivalAccount
{
   private IAccessoriesAccount shoesAccount;
   private IAccessoriesAccount handBagsAccount;

    public SurvivalAccount(IAccessoriesAccount shoesAccount, IAccessoriesAccount handBagsAccount )
    {
      this.shoesAccount = shoesAccount;
      this.handBagsAccount = handBagsAccount;
    }
}
```

DI ANTIPATTERN - CONTROL FREAK

DI ANTIPATTERN - CONTROL FREAK

```
public class ProductService
{
  private ProductRepository repository;
  private FoodProcessor foodProcessor;
  public ProductService()
    {
       //OMG I need something, so I'll get it by myself
      this.repository = new ProductRepository(connectionString);
      this.foodProcessor=  new FoodProcessor();
    }
}
```

DI ANTIPATTERN - CONTROL FREAK

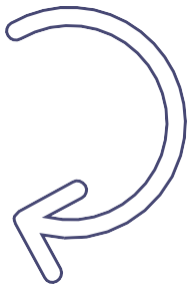
- Most common DI – antipattern
- Default way of creating instances
- No effort to introduce abstractions
- We can’t change implementations
- We can’t develop in parallel
- The Most Problematic in terms of coupling
- Every time when directly or indirectly use the new keyword !

INVERSION OF CONTROL

Inversion of Control

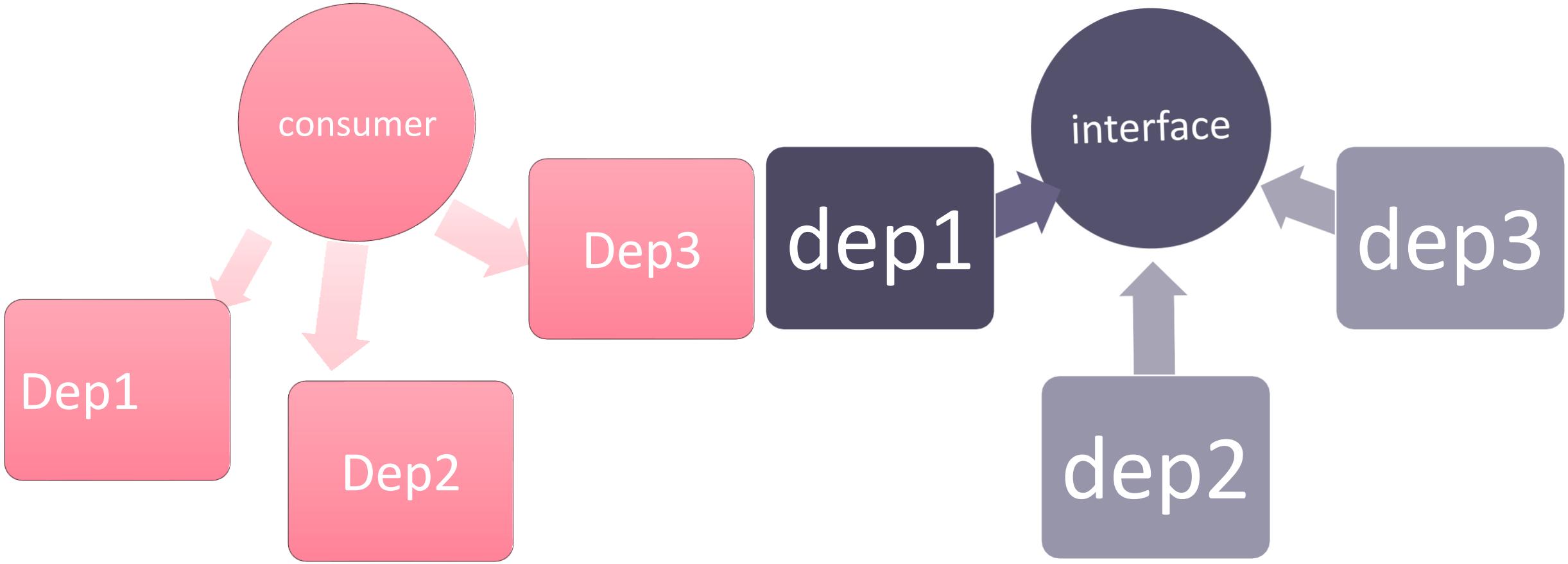
INVERSION OF CONTROL

- Programming style where a framework or runtime controls the program flow
- You let a framework to take care of instance creation
- You move somewhere else the decisions of which concrete class to use



Flexibility

INVERSION OF CONTROL



DEPENDENCY INJECTON

DEPENDENCY INJECTION

- Subset of Inversion of Control
- Refers to dependency management
- The idea is to have a mechanism that provides concrete implementation over an abstraction
- Helps with Single Responsibility (SR) and Separation of Concerns (SoC).

WHY - Benefits

- Loose coupling
  - Extensibility
  - Testability
  - Reusability
- DRY – write less boiler plate code
- Mockability (yes, that’s a word)
- You don’t pull your dependencies, they are pushed

DEPENDENCY INJECTION MINDSET

- It’s not a goal
- It’s one of the best ways to enable loose coupling
  - If used right, gives you more maintainable code
- It’s more a way of thinking and designing code than a collection of tools and techniques
- Not the supreme approach, but should be the minim

DESIGN PATTERNS

What is a design pattern?

- A software design pattern is a general, reusable solution to a commonly occurring problem within a giving context in software design.
- Strategy
- Builder


THANK YOU