DDD分层架构的三种模式

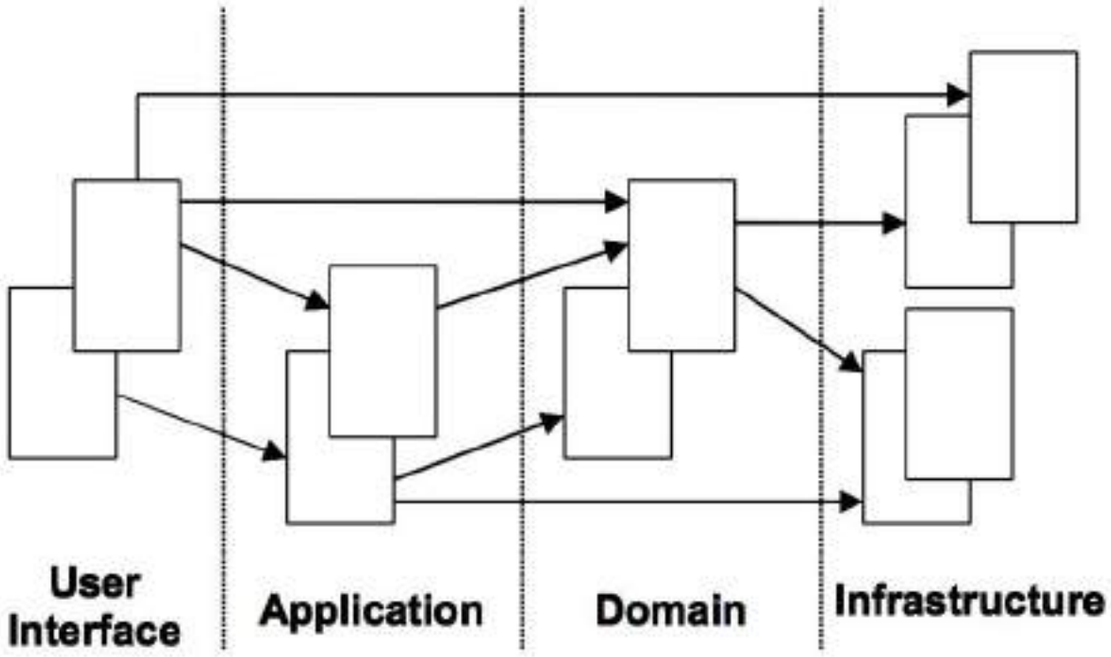
中兴通讯虚拟化架构师　张晓龙

内容大纲

- **DDD分层架构介绍**
- DDD分层：L型架构模式
- DDD分层：L型＋DCI架构模式
- DDD分层：DIP架构模式

DDD分层架构介绍

- 定义
  - 将BC的解决方案分隔到不同的层中，每一层的子解决方案应保持内聚性，并且在同一个抽象级别，每一层都应与它下方的各层保持松散耦合



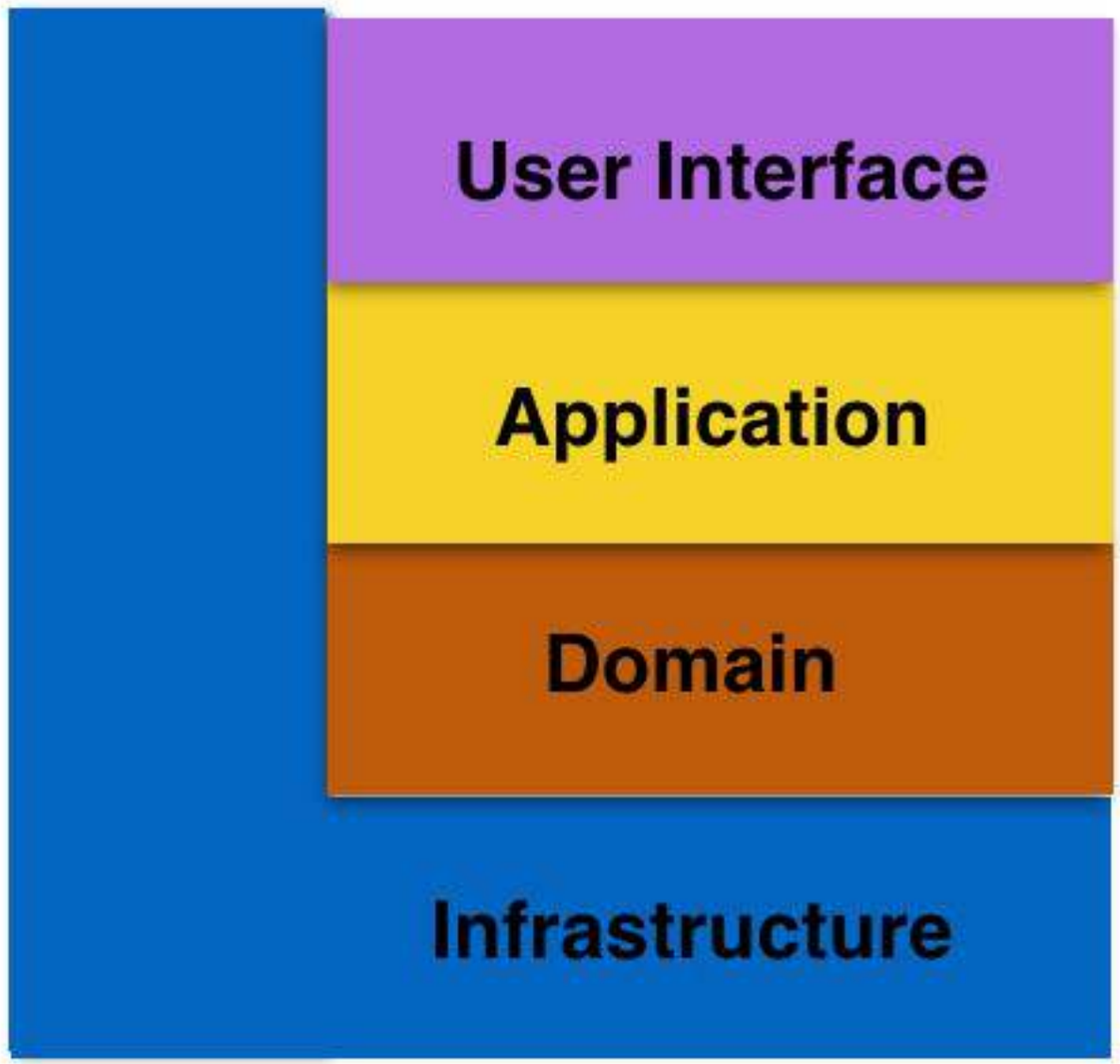
DDD分层架构介绍

- 我们在实践中总结出了三种模式：
  - DDD分层：L型架构模式
  - DDD分层：L型＋DCI架构模式
  - DDD分层：DIP架构模式

内容大纲

- DDD分层架构介绍
- **DDD分层：L型架构模式**
- DDD分层：L型＋DCI架构模式
- DDD分层：DIP架构模式

L型四层架构



内容大纲

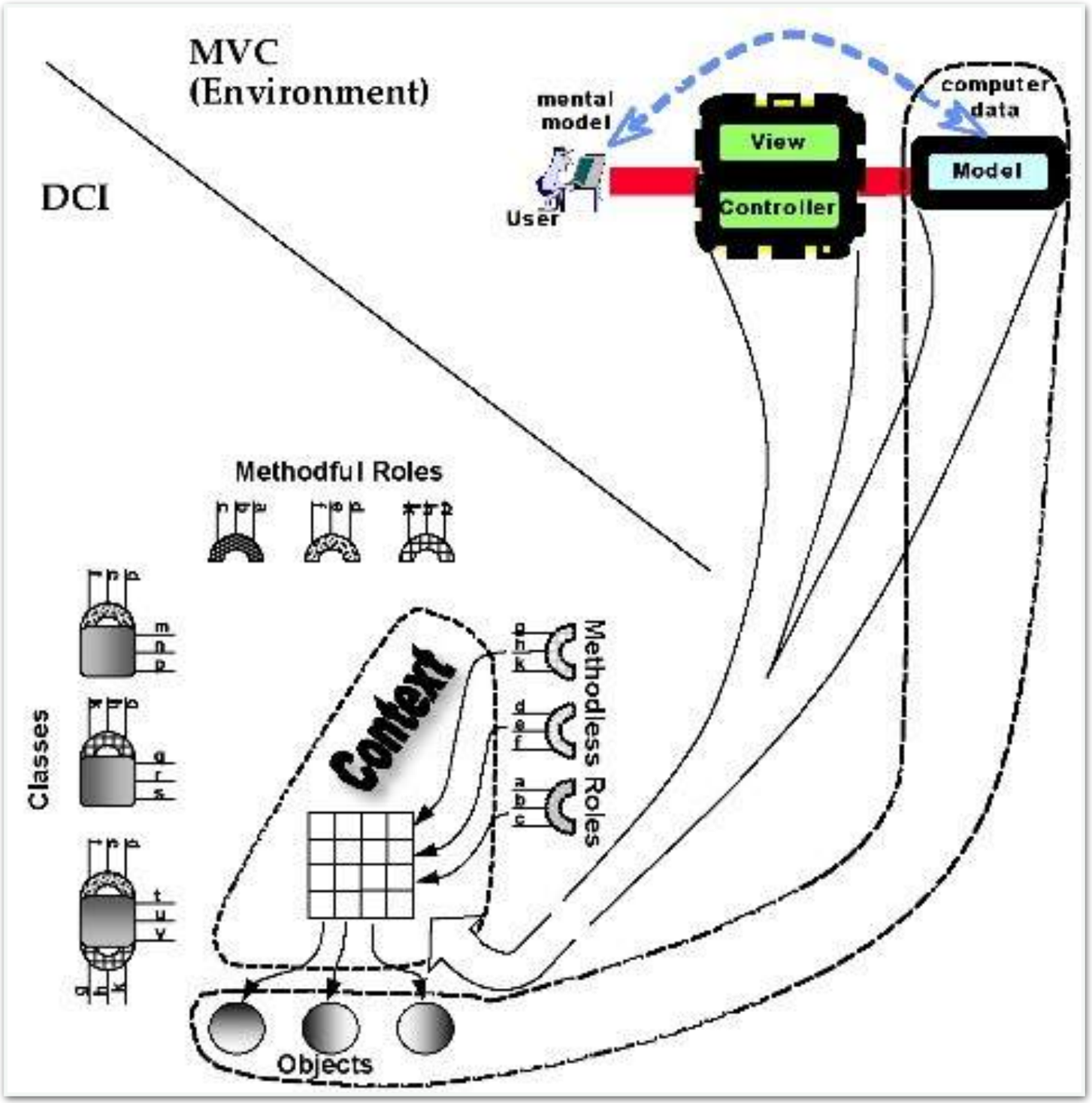
- DDD分层架构介绍
- DDD分层：L型架构模式
- **DDD分层：L型＋DCI架构模式**
- DDD分层：DIP架构模式

两种模型

- 贫血模型：
  - 数据和行为的完全分离
  - 面向过程

- 充血模型：
  - 高频变化的行为和低频变化的数据的耦合
  - 上帝类

DCI



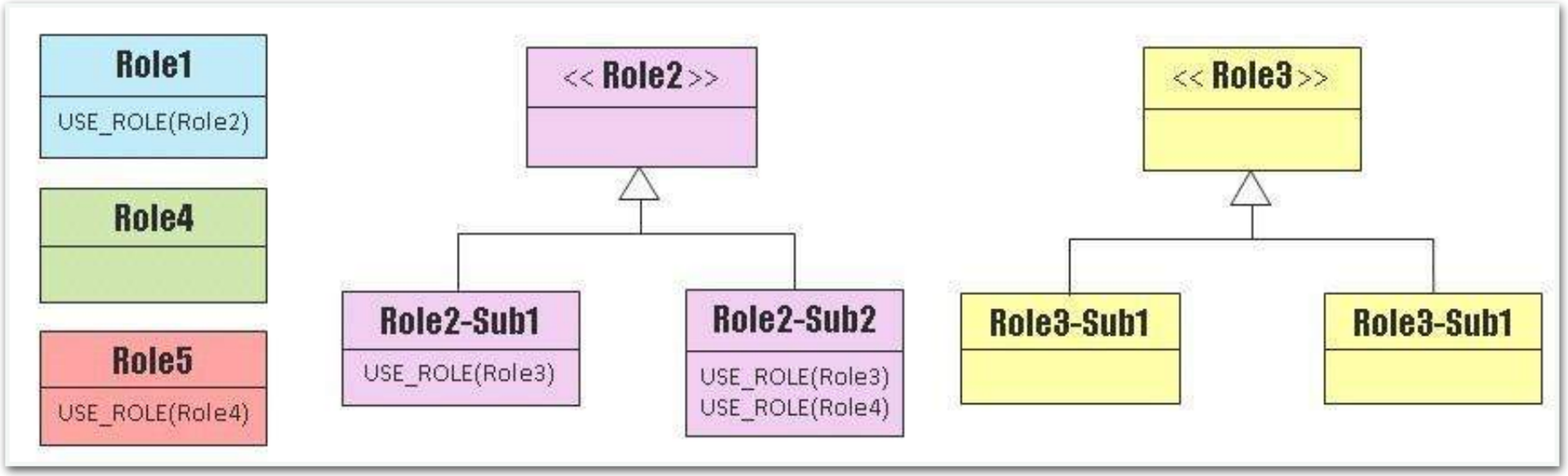
Data, Context and Interaction :

A New Architectural Approach

by James O. Coplien and

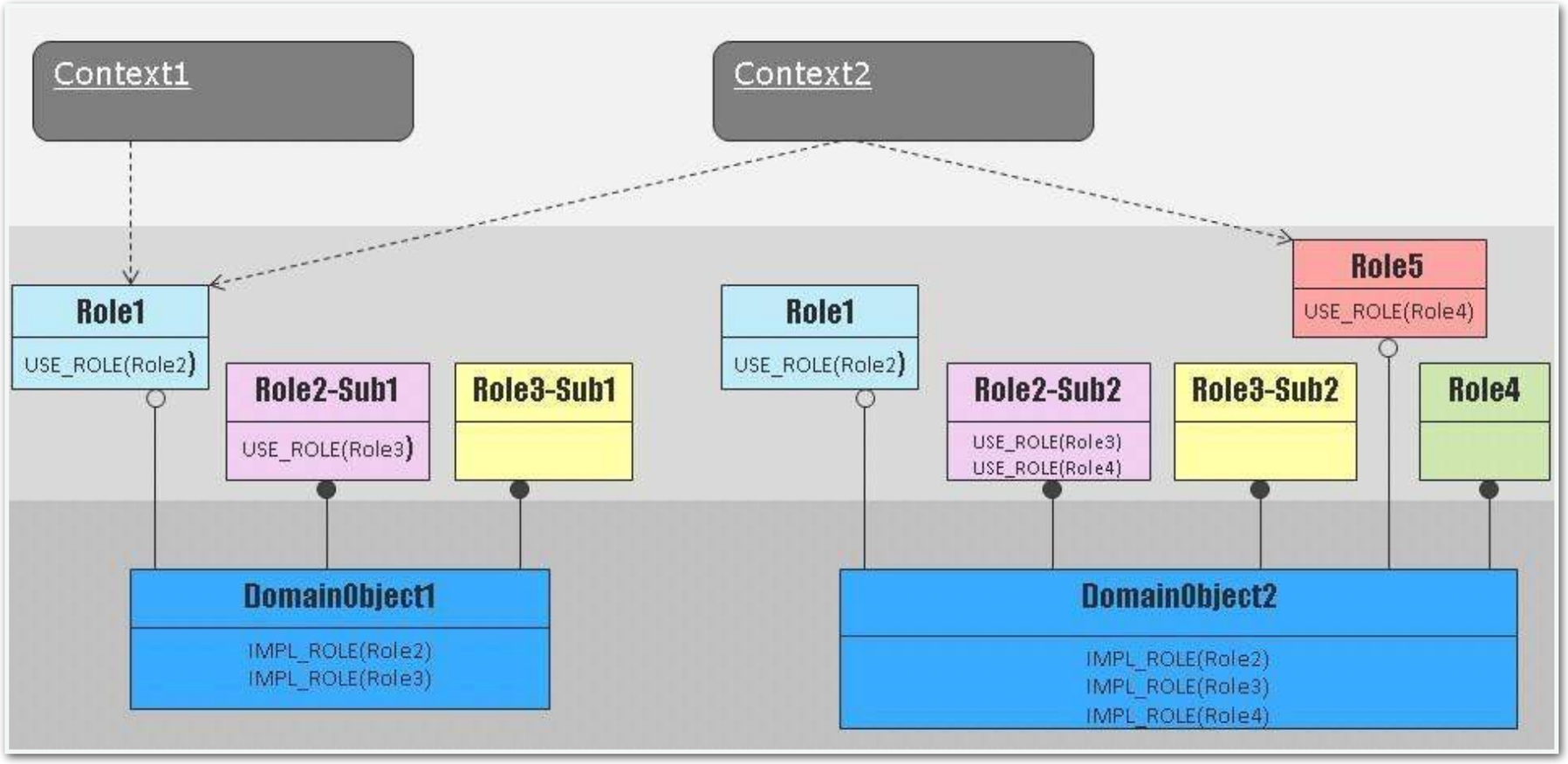
Trygve Reenskau

DCI

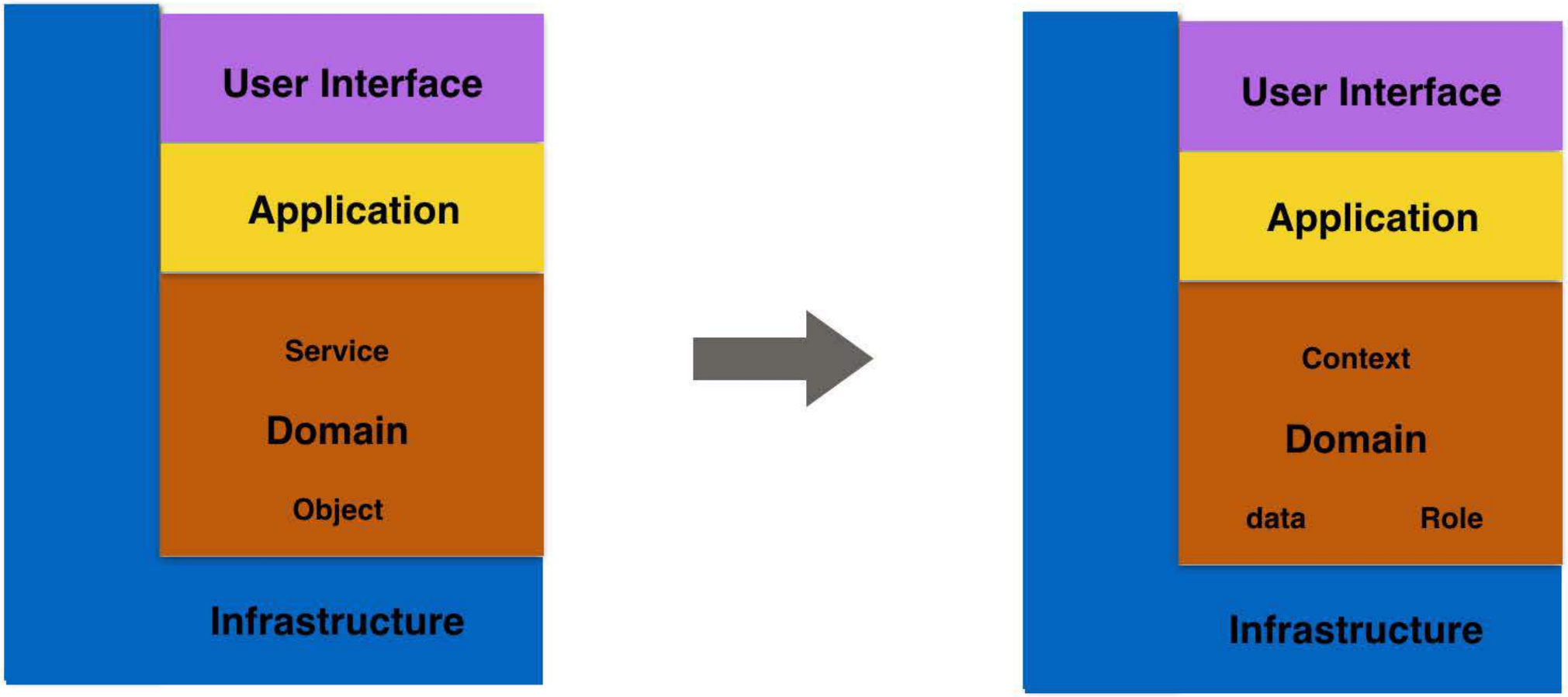


类视图

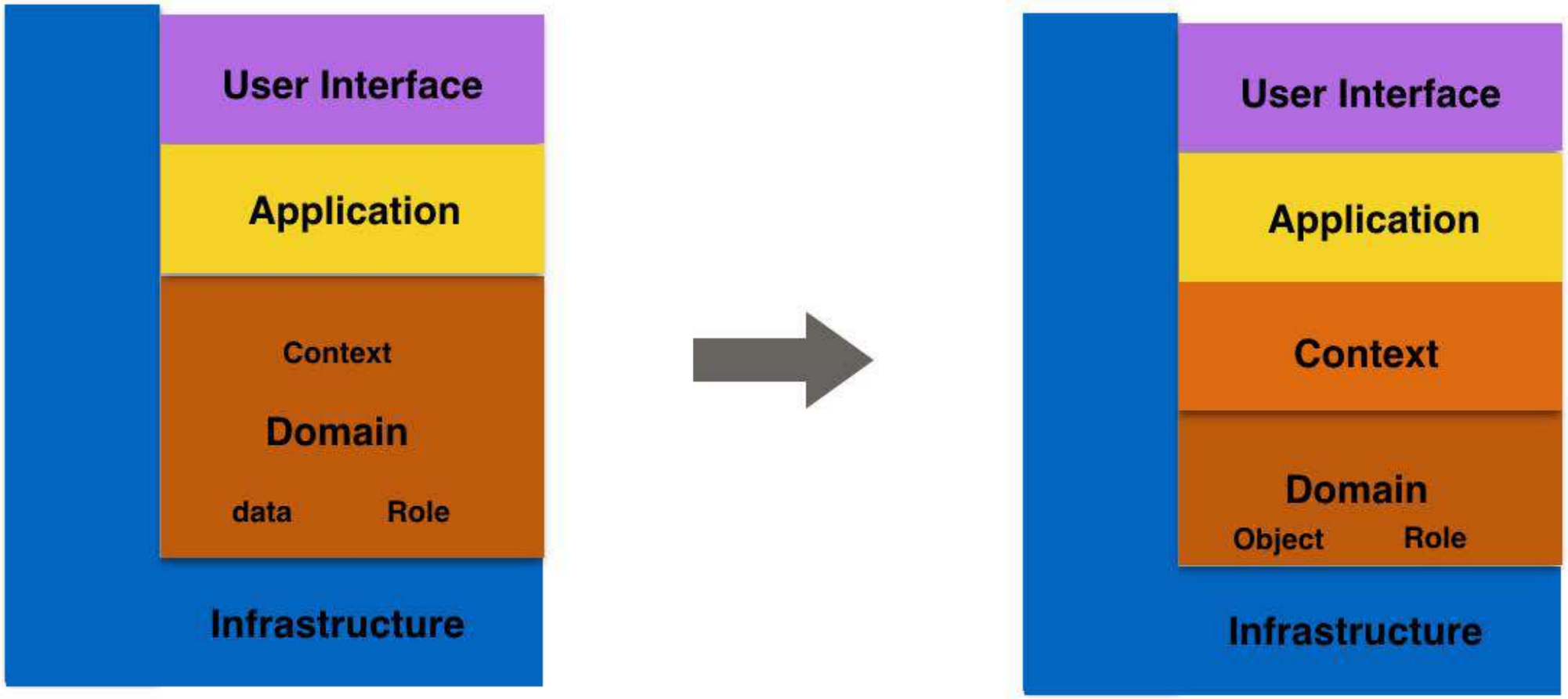
DCI视图



L型＋DCI模式：四层


User Interface
Application
Service
Domain
Object
Infrastructure
User Interface
Application
Context
Domain
data
Role
Infrastructure

L型＋DCI模式：五层


User Interface
Application
Context
Domain
data
Role
Infrastructure
User Interface
Application
Context
Domain
Object
Role
Infrastructure

特殊领域

- 在一次业务中消息交互比较多的领域：
  - 电信领域的控制面
  - 网络领域的管理面

特殊领域

- 在一次业务中消息交互比较多的领域：
  - 电信领域的控制面
  - 网络领域的管理面

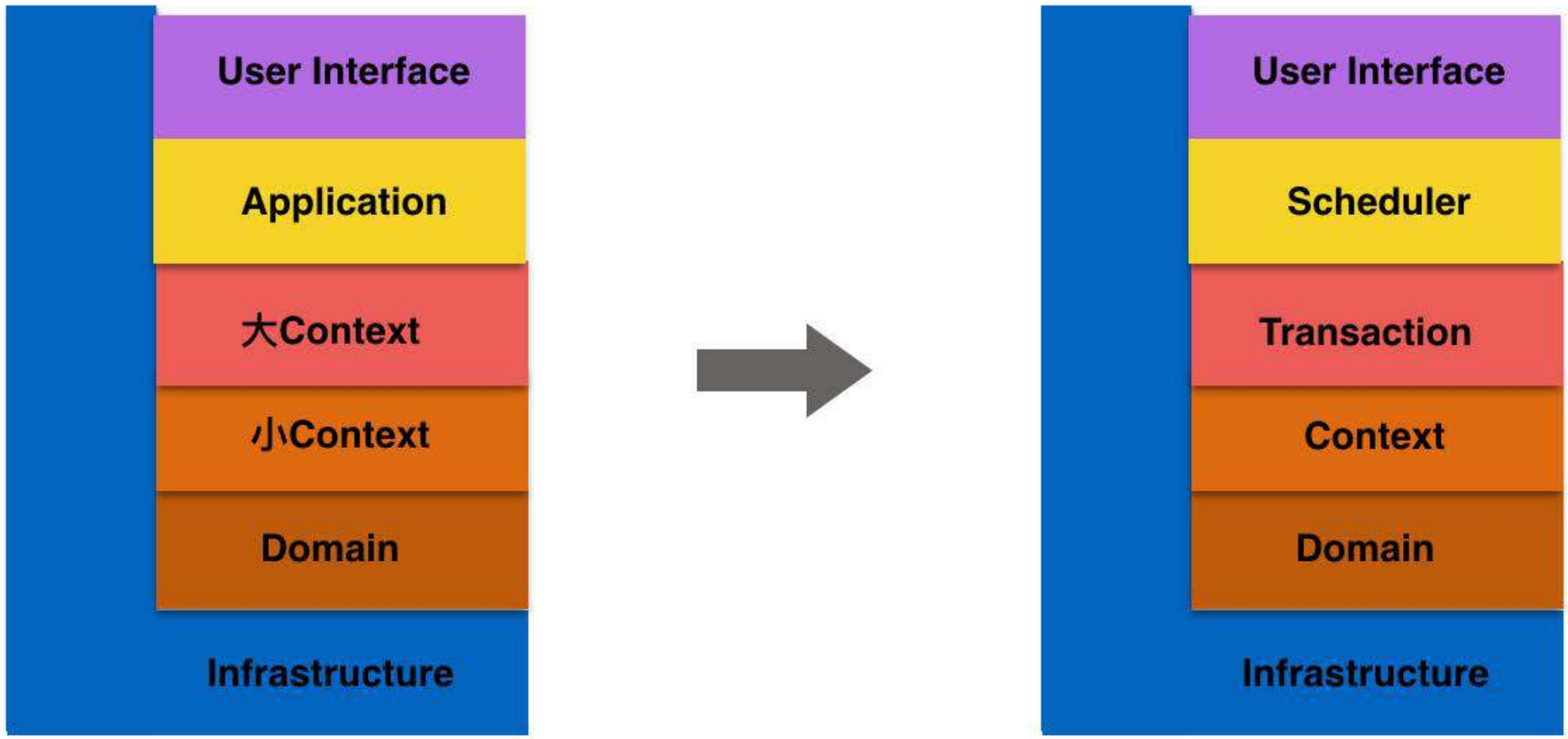
➤ 在这些领域中，Context具有哪些特征?

特殊领域

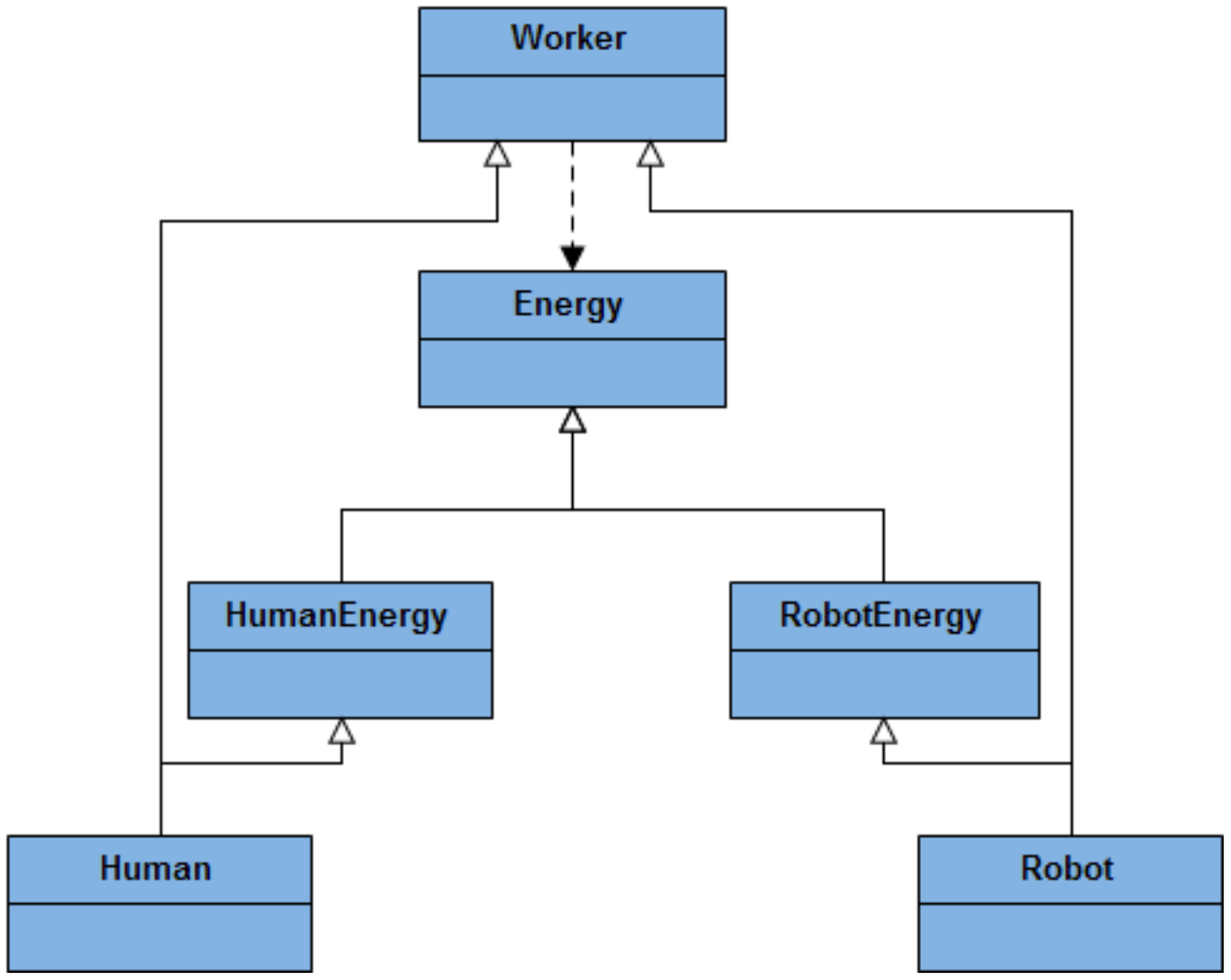
- 在一次业务中消息交互比较多的领域：
  - 电信领域的控制面
  - 网络领域的管理面

- 在这些领域中，Context具有哪些特征?
  1. 对象交互多
  2. 业务流程长
  3. 事务操作

L型＋DCI模式：六层

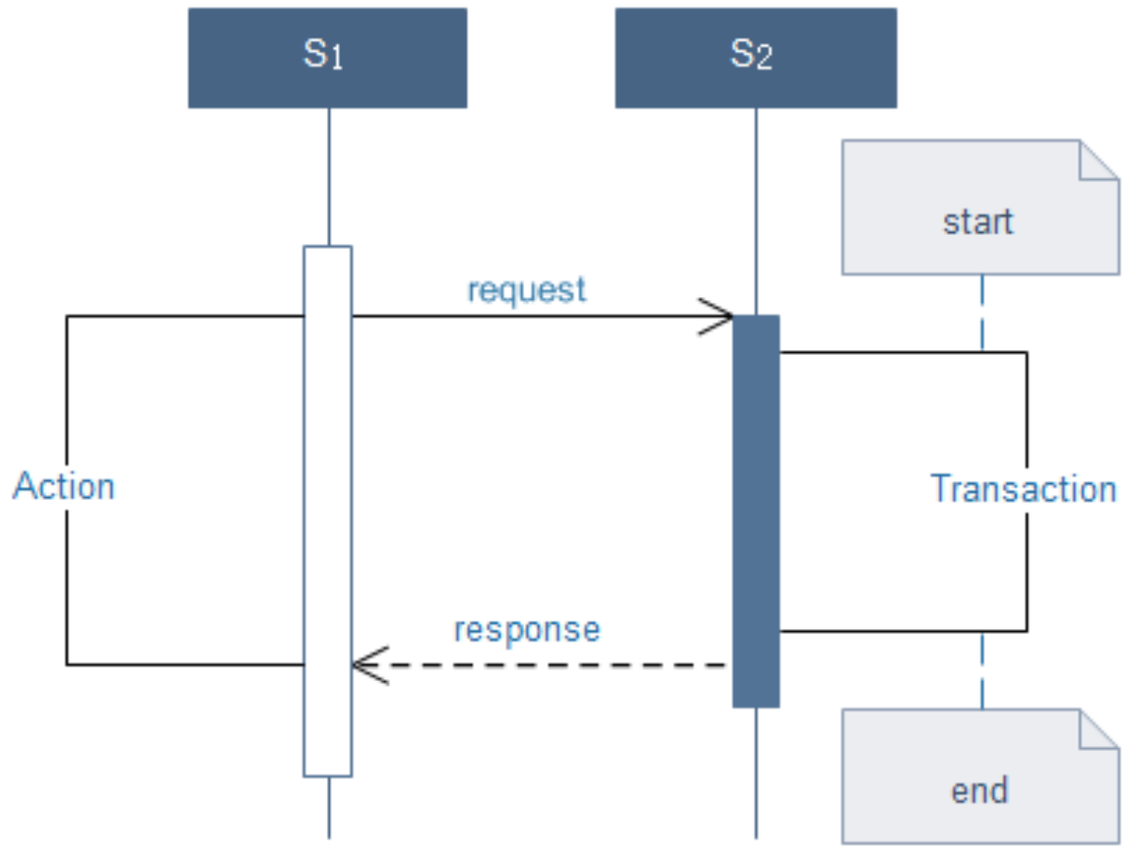

User Interface
Application
大Context
小Context
Domain
Infrastructure
User Interface
Scheduler
Transaction
Context
Domain
Infrastructure

领域对象案例


Worker
Energy
HumanEnergy
RobotEnergy
Human
Robot


domain
object
role

事务模型案例



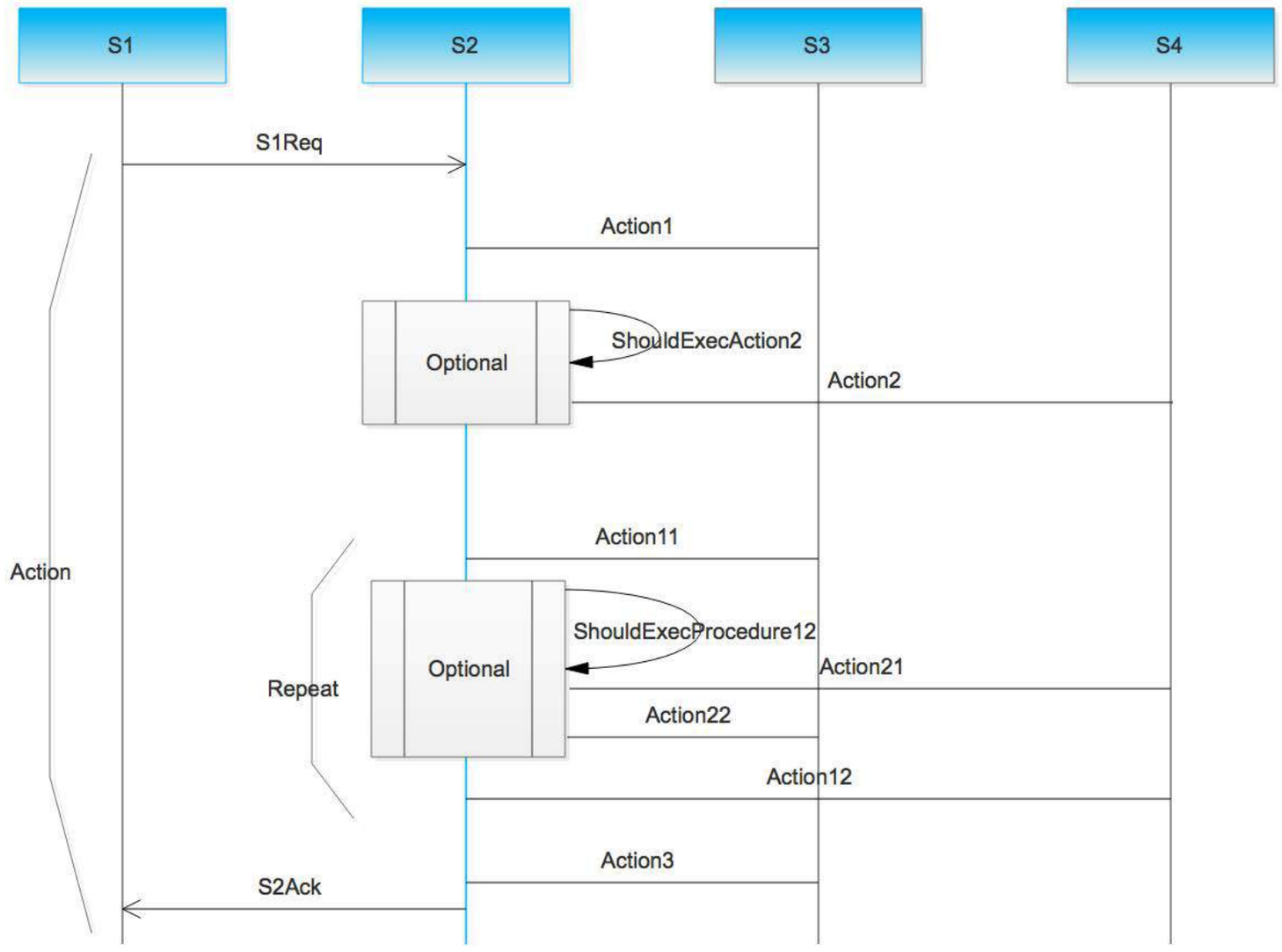

```
func scheduleS1ReqTrans(req []byte) error {
    transInfo := &transdsl.TransInfo{AppInfo: &context.S2Info{}}
    s1ReqTrans := trans.NewS1ReqTrans()
    err = s1ReqTrans.Exec(transInfo)
    if err != nil {
        s1ReqTrans.RollBack(transInfo)
    }
    return err
}
```

事务模型案例

```
func NewS1ReqTrans() *transdsl.Transaction {
    trans := &transdsl.Transaction {
        Fragments: []transdsl.Fragment {
            new(context.Action1),
            &transdsl.Optional {
                Spec: new(context.ShouldExecAction2),
                Fragment: new(context.Action2),
            },
            &transdsl.Repeat {
                FuncVar: newProcedure1,
            },
            new(context.Action3),
        },
    }
    return trans
}
```

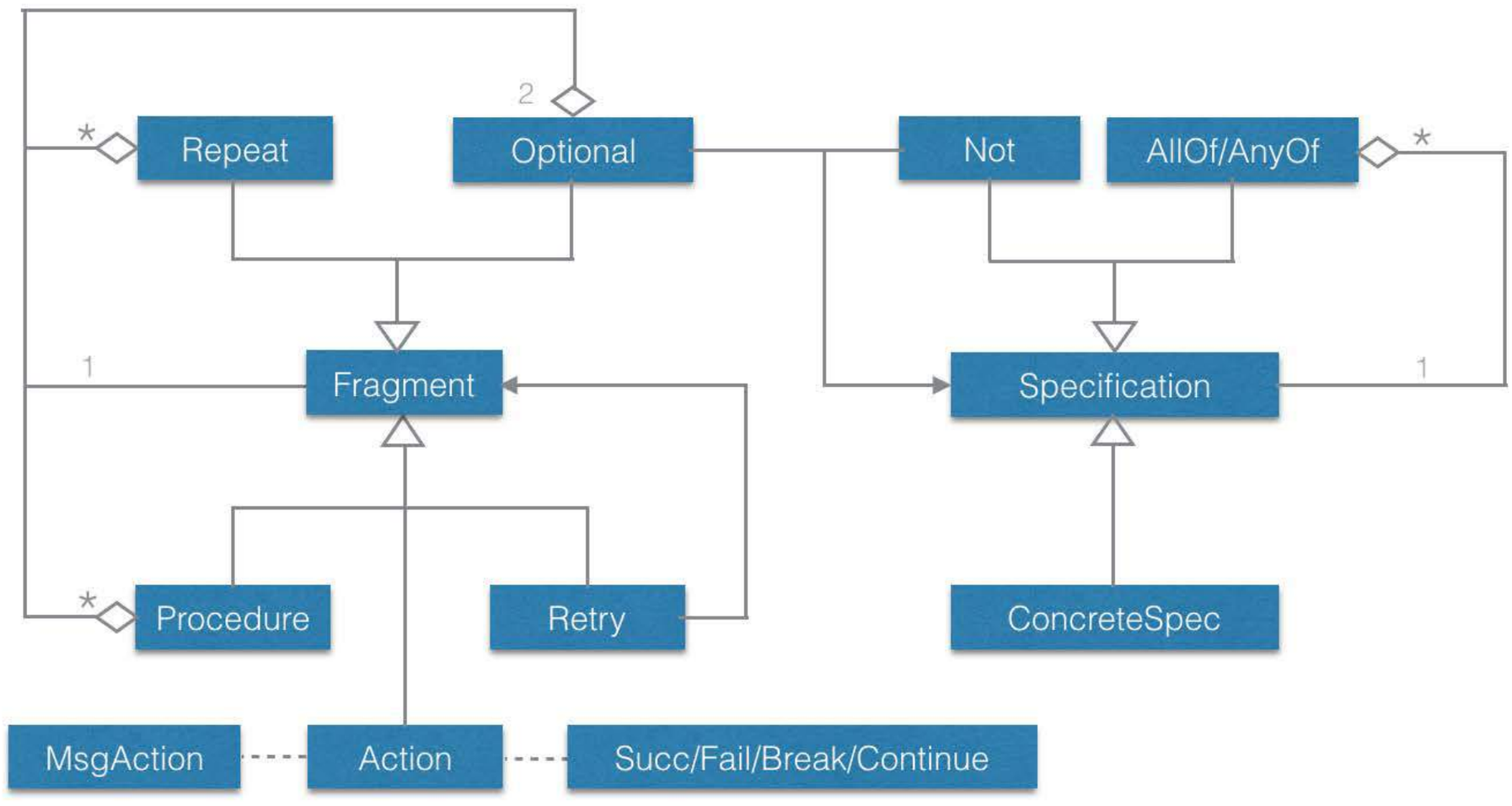
```
func newProcedure1() transdsl.Fragment {
    procedure := &transdsl.Procedure {
        Fragments: []transdsl.Fragment {
            new(context.Action11),
            &transdsl.Optional {
                Spec: new(context.ShouldExecProcedure2),
                Fragment: newProcedure2(),
            },
            new(context.Action12),
        },
    }
    return procedure
}

func newProcedure2() transdsl.Fragment {
    procedure := &transdsl.Procedure {
        Fragments: []transdsl.Fragment {
            new(context.Action21),
            new(context.Action22),
        },
    }
    return procedure
}
```



事务模型案例

```
type Fragment interface {
    Exec(transInfo *TransInfo) error
    RollBack(transInfo *TransInfo)
}
```



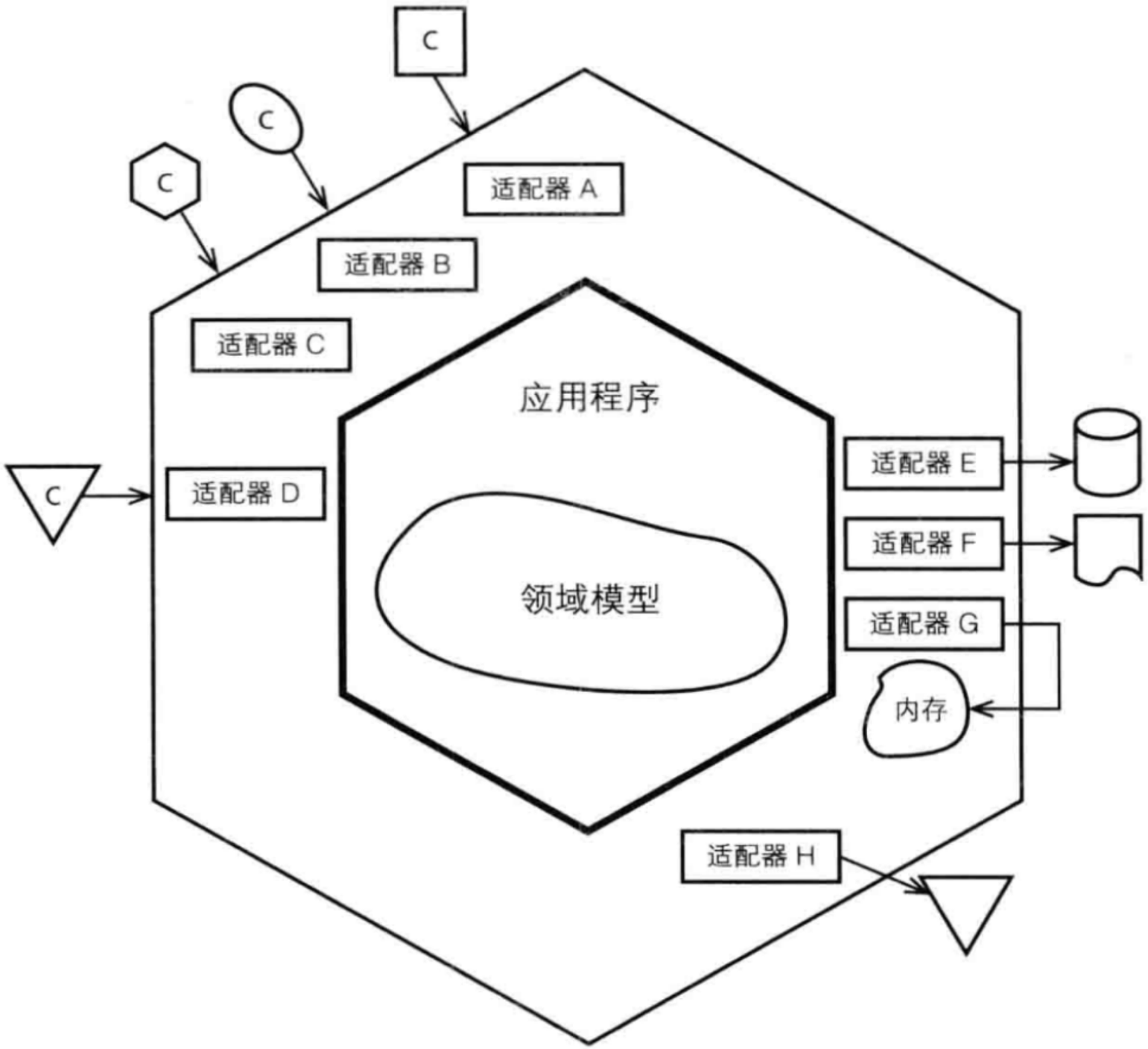
内容大纲

- DDD分层架构介绍
- DDD分层：L型架构模式
- DDD分层：L型＋DCI架构模式
- **DDD分层：DIP架构模式**

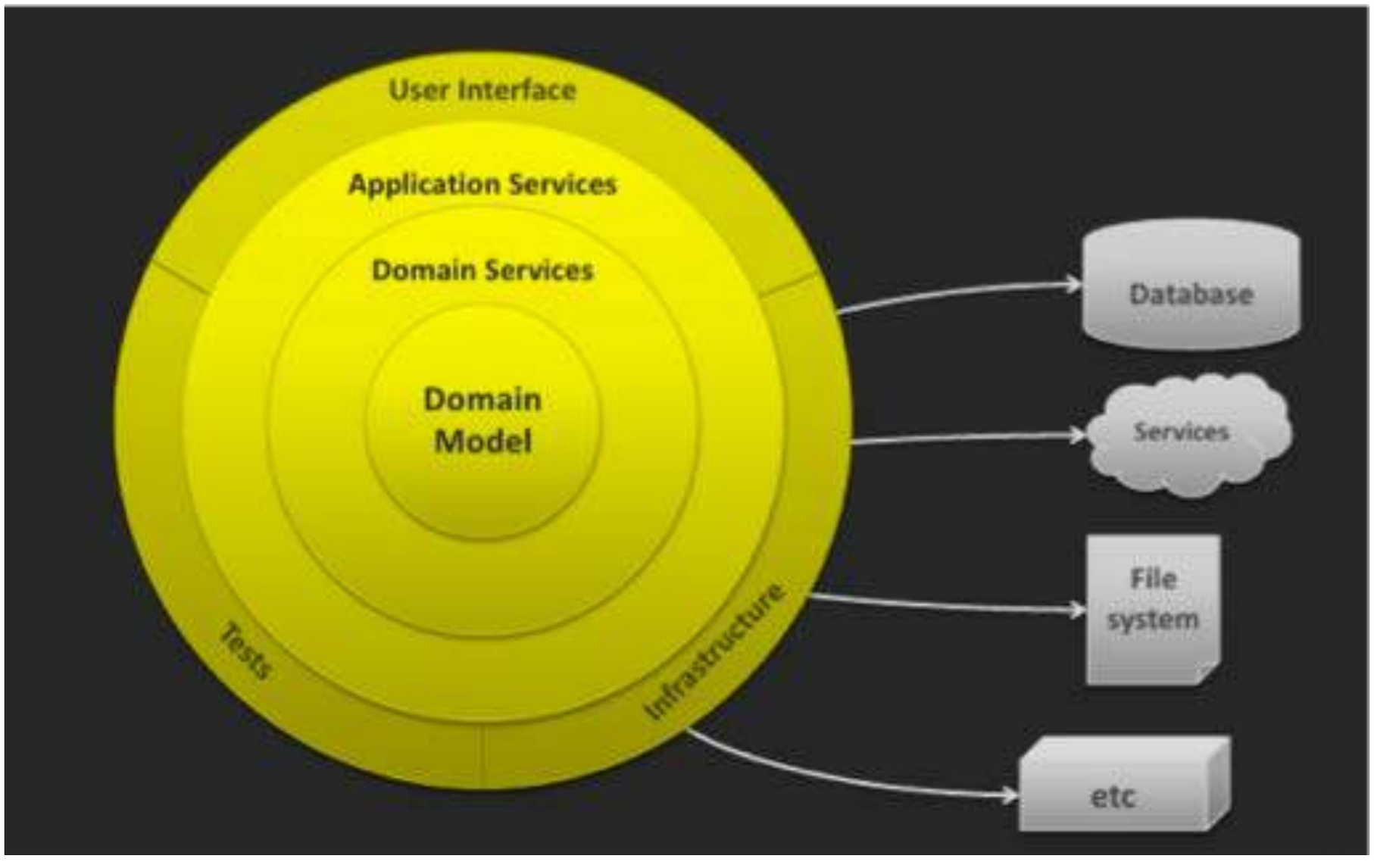
DIP

- 依赖倒置原则（DIP）由Robert C. Martin提出：
  - 高层模块不应该依赖于底层模块，两者都应该依赖于抽象
  - 抽象不应该依赖于细节，细节应该依赖于抽象

六边形架构

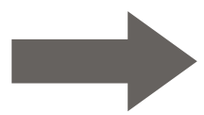


洋葱圈架构


User Interface
Application Services
Domain Services
Domain
Model
Tests
Infrastructure
Database
Services
File
system
etc

Repository案例

- 以前：L型分层架构
  - 抽象类CargoRepository定义在Domain层
  - 实现类CargoRepositoryImpl定义在Domain层
  - 抽象类CargoRepository和实现类CargoRepositoryImpl的绑定在Domain层



- 现在：洋葱圈架构
  - 抽象类CargoRepository定义在Domain层
  - 实现类CargoRepositoryImpl定义在Infra层
  - 抽象类CargoRepository和实现类CargoRepositoryImpl的绑定在main包的init函数中完成

Repository案例

```
package repository

import (
    "domain/object"
)

type CargoRepo interface {
    Save(cargo *object.Cargo)
    Find(id int) *object.Cargo
}

var c CargoRepo

func SetCargoRepoInstance(c CargoRepo) {
    c = c
}

func GetCargoRepoInstance() CargoRepo {
    return c
}
```

```
package main

import (
    "domain/repository"
    "infra/repository-impl"
)

func init() {
    repository.SetCargoRepoInstance(repositoryimpl.NewCargoRepoImpl())
}
```

```
package repositoryimpl

import (
    "domain/object"
    "infra/etcdclient"
    "encoding/json"
    "strconv"
)

type CargoRepoImpl struct {
    dir string
    etcd EtcdClient
}

func NewCargoRepoImpl() *CargoRepoImpl {
    return &CargoRepoImpl{CertainDir, NewEtcdClient()}
}

func (c *CargoRepoImpl) Save(cargo *object.Cargo) {
    bytes, err := json.Marshal(cargo)
    if err != nil {
        return
    }
    c.etcd.SaveLeaf(c.dir + strconv.Itoa(cargo.GetId()), string(bytes))
}

func (c *CargoRepoImpl) Find(id int) *object.Cargo {
    bytes, err := c.etcd.ReadLeaf(c.dir + strconv.Itoa(id))
    if err != nil {
        return nil
    }
    var cargo object.Cargo
    err = json.Unmarshal(bytes, &cargo)
    if err != nil {
        return nil
    }
    return &cargo
}
```

Send案例

- 以前：业务和平台同时开发，业务依赖于平台
  - 平台一开始就提供Send函数作为API，业务使用
  - Send函数在稳定之前，业务跟着变化

- 现在：业务和平台同时开发，业务不依赖于平台
  - 平台开发Send函数，业务不使用，而是自行定义Sender接口
  - 在业务开发阶段，对Sender接口进行打桩完成测试
  - 在业务和平台集成阶段，业务在适配层实现Sender接口，并完成依赖注入

Send案例

```
// in infra layer
package plat

import ...

fun Send(payload []byte, ip, port string) {
    ...
}
```

```
// in infra layer
package adapter

import ...

func NewSenderImpl() *SenderImpl {
    return &SenderImpl{}
}

type SenderImpl struct {

}

func (s *SenderImpl) SendMsg(payload []byte, serviceType, serviceInst string) {
    cache := GetCacheInstance()
    ip, port := cache.Retrieve(serviceType, serviceInst)
    plat.Send(payload, ip, port)
}
```

```
// in domain layer
package business

type Sender interface {
    SendMsg(payload []byte, serviceType, serviceInst string)
}

var s Sender

func SetSenderInstance(s Sender) {
    s = s
}

func GetSenderInstance() CargoRepo {
    return s
}
```

```
package main

import (
    "domain/business"
    "infra/adapter"
)

func init() {
    business.SetSenderInstance(adapter.NewSenderImpl())
}
```

小结

- DDD分层架构介绍
  - 经典四层架构
  - 实践中的三种模式
- DDD分层：L型架构模式
  - L型四层架构
- DDD分层：L型＋DCI架构模式
  - L型＋DCI模式：四层
  - L型＋DCI模式：五层
  - L型＋DCI模式：六层
- DDD分层：DIP架构模式
  - 六边形架构
  - 洋葱圈架构


Q
A

谢谢大家！

- 张晓龙 @ 中兴通讯
- 架构师，技术教练，DDD实践布道者
- zhangxiaolong1980@126.com
- 简书个人主页：
  http://www.jianshu.com/u/1381dc29fed9