ThoughtWorks®

组合式设计

低成本演进式设计的基石

Jieying Yuan, ThoughtWorks 高级咨询师

演进式设计

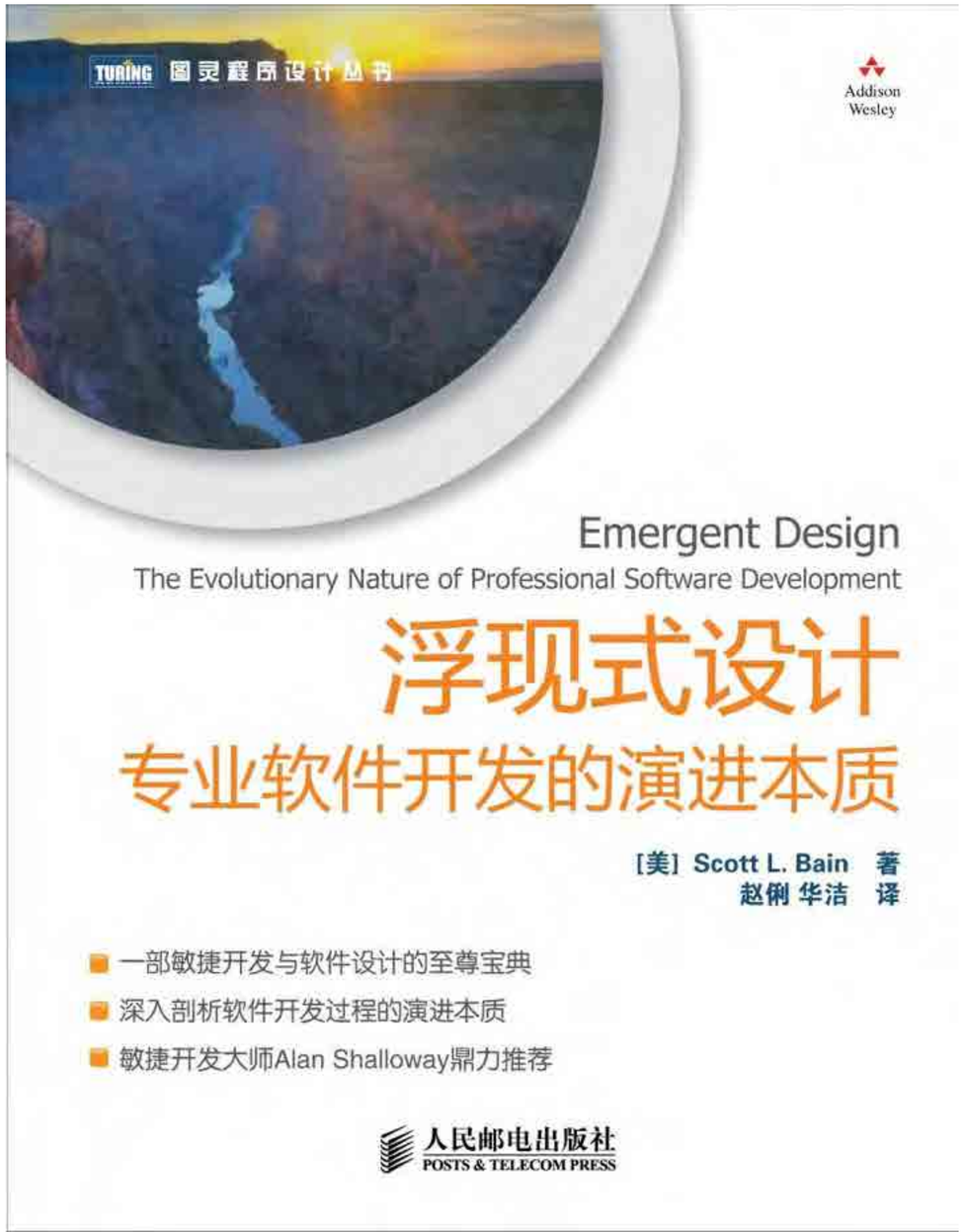
简介

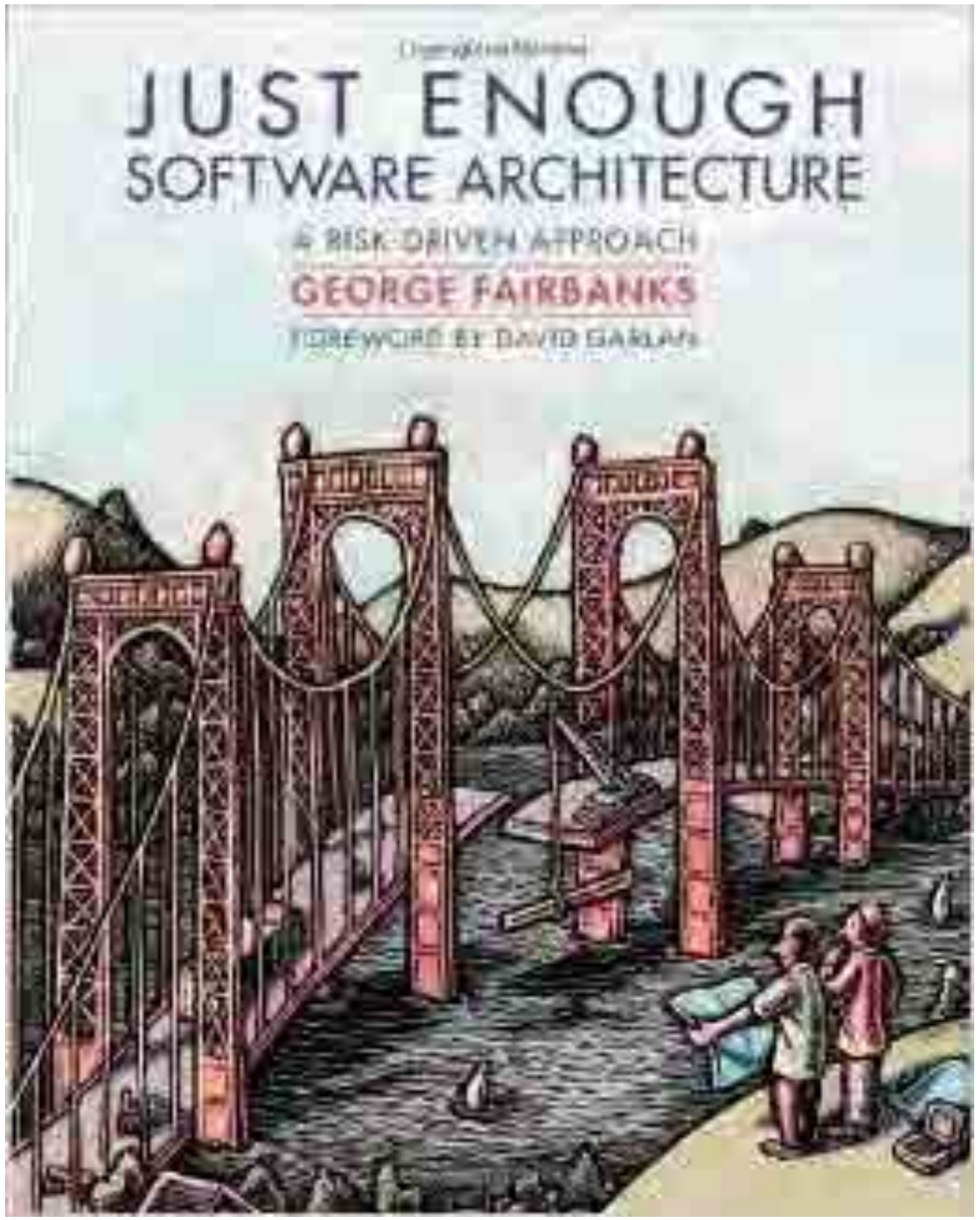
ThoughtWorks®

演进式设计

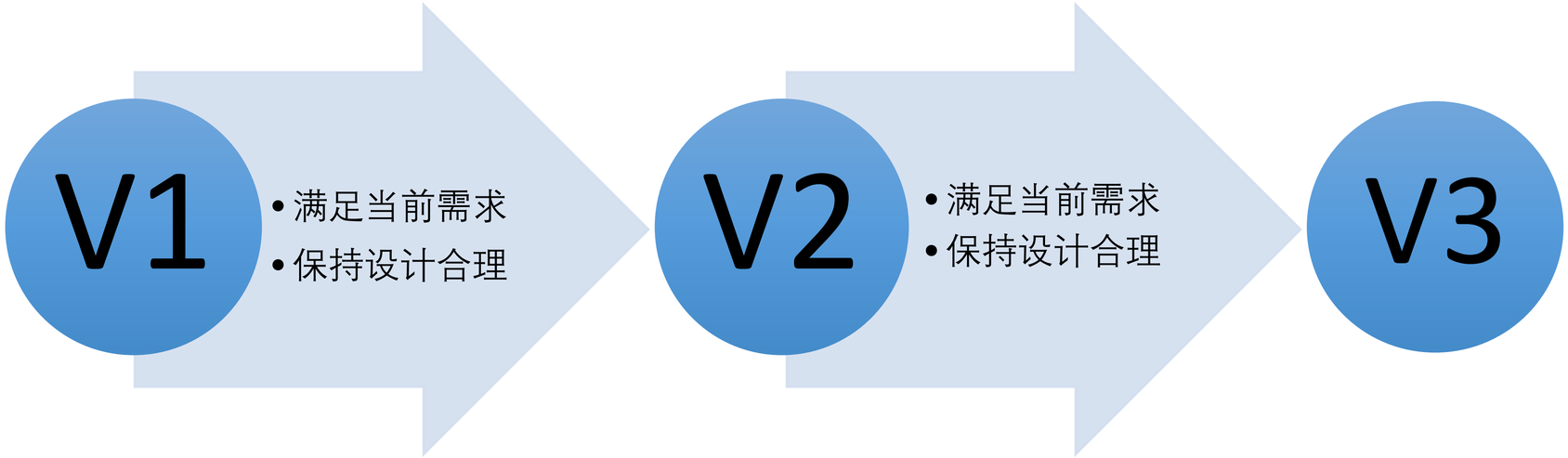
简介

推荐书籍

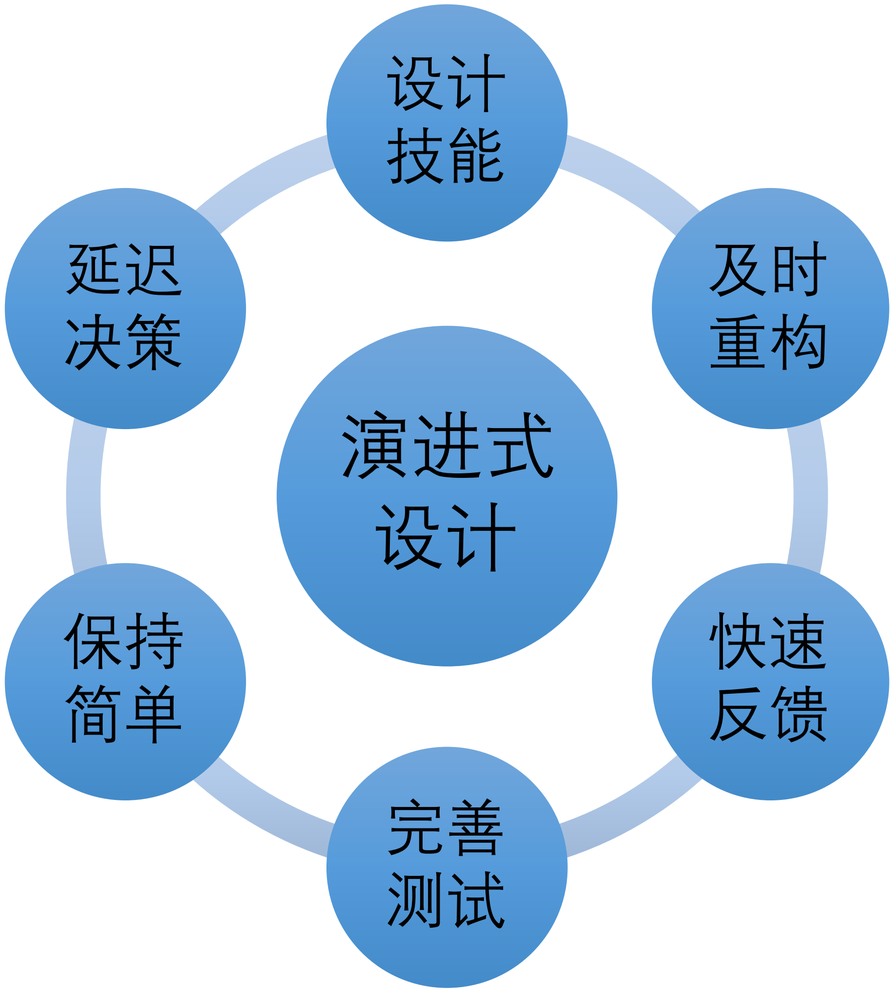

TURING 图灵程序设计丛书
Addison Wesley
Emergent Design
The Evolutionary Nature of Professional Software Development
浮现式设计
专业软件开发的演进本质
[美] Scott L. Bain 著
赵俐 华洁 译
一部敏捷开发与软件设计的至尊宝典
深入剖析软件开发过程的演进本质
敏捷开发大师Alan Shalloway鼎力推荐
人民邮电出版社
POSTS & TELECOM PRESS


JUST ENOUGH
SOFTWARE ARCHITECTURE
A RISK-DRIVEN APPROACH
GEORGE FAIRBANKS
FOREWORD BY DAVID GARLAN

演进式设计



演进式设计



ThoughtWorks®

设计

设计的作用

“Design is there to enable you to keep changing the software easily in the long term” —— Kent Beck

保持软件容易变更：正交四原则

1.最小化重复

2.分离不同的变化方向

3.缩小依赖范围

4.向着稳定的方向依赖

组合式设计

正交原则

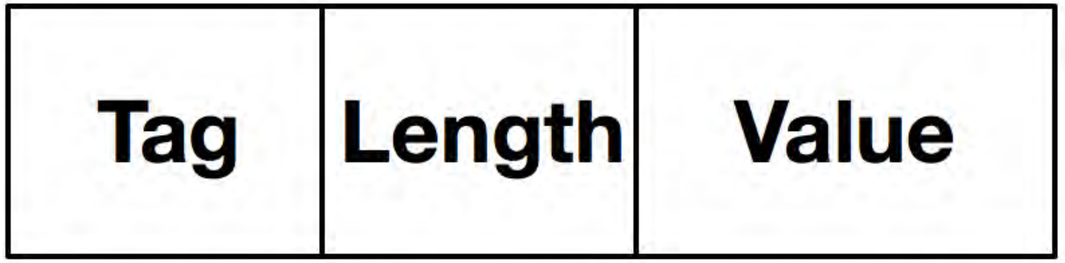
ThoughtWorks®

组合式设计

案例一

非标准的TLV

- 子系统和周边多个子系统进行复杂的异步消息交互
- 消息格式名义上为TLV，但事实上，
    1. 即不是标准的TLV；
    2. 也没有任何规范；
    3. 完全是团队的随机定义
- 不可更改
    1. 已经是历史事实
    2. 牵扯到太多团队



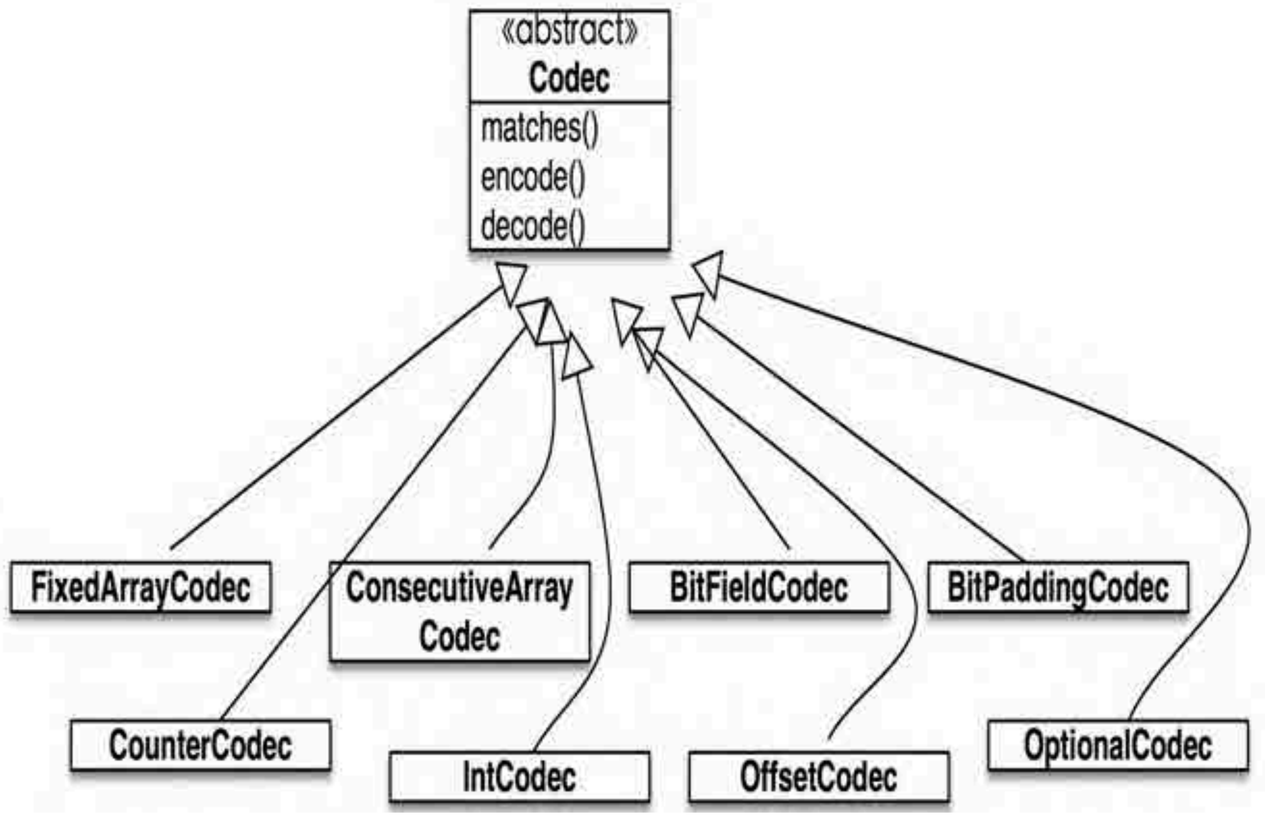
两难的抉择

- 在缺乏规律的情况下，写一个自动的Code Generator维护工作量太大。
- 手工编解码，工作量太大，容易出错

编解码接口

```
DEFINE_INTERFACE(Codec)
{
    DEFAULT(bool, matches(CodecBuffer&) const);

    ABSTRACT(Status encode(const MsgBuffer& msg, CodecBuffer& buf) const);
    ABSTRACT(Status decode(CodecBuffer& buf, const MsgBuffer& msg) const);
};
```

定义一组具体的CODEC



- AutoCounterCodec.h
- BitFieldCodec.h
- BitPaddingCodec.h
- ConsecutiveArrayCodec.h
- CountUnitSeries.h
- CounterCodec.h
- FailSkipSizeCalculator.h
- FixedArrayCodec.h
- FixedSkipSizeCaculator.h
- FixedValueCodec.h
- IntCodec.h
- NilCodec.h
- OffsetCodec.h
- OptionalCodec.h
- ScatteredArrayCodec.h
- SimpleMsgCodec.h
- SingleIeCodec.h
- TlvCodec.h
- TlvSkipSizeCalculator.h
- TvCodec.h

职责足够小

- 每个Codec都仅仅完成一个非常小的具体信息的编解码
    1. 只有足够小，才能在变化剧烈的环境下具备更高的可复用价值
    2. 如果发生某个细节的特殊变化需求，只需要编写一个新的小Codec

组合

```
template <_UC TAG, typename T, size_t OFFSET, size_t SIZE_OF_LEN = 1>
struct ABIS_TLV_IE_OPT_CODEC :
  GenericOffsetCodec < OFFSET,
    GenericOptionalCodec <
      GenericTlvCodec < TAG, ABIS_TAG_SIZE, SIZE_OF_LEN,
        SingleIeCodec<T> > > >{};

template <_UC TAG, typename T, size_t OFFSET, _UL MAX_IES, size_t SIZE_OF_LEN = 1>
struct ABIS_TLV_ARRAY_CODEC :
   GenericOffsetCodec<OFFSET,
     GenericTlvCodec<TAG, ABIS_TAG_SIZE, SIZE_OF_LEN,
        AbisSimpleArrayCodec<T, MAX_IES> > >{};
```

组合

- 组合之后，依然是一个Codec。
    1. 事实上是一个装饰模式，或职责链；
    2. 能够被抽象为具备闭包性质的组合模式，都会带来巨大的灵活性。
- 一个组合后的Codec也仅能解析单个信元；

更进一步的组合

- 用户通过这样的形式定义消息，在一个消息内，所有针对单个信元的Codec会按照顺序序列化；
- 通过预处理时多态，还会自动生成消息多对应的C语言数据结构。

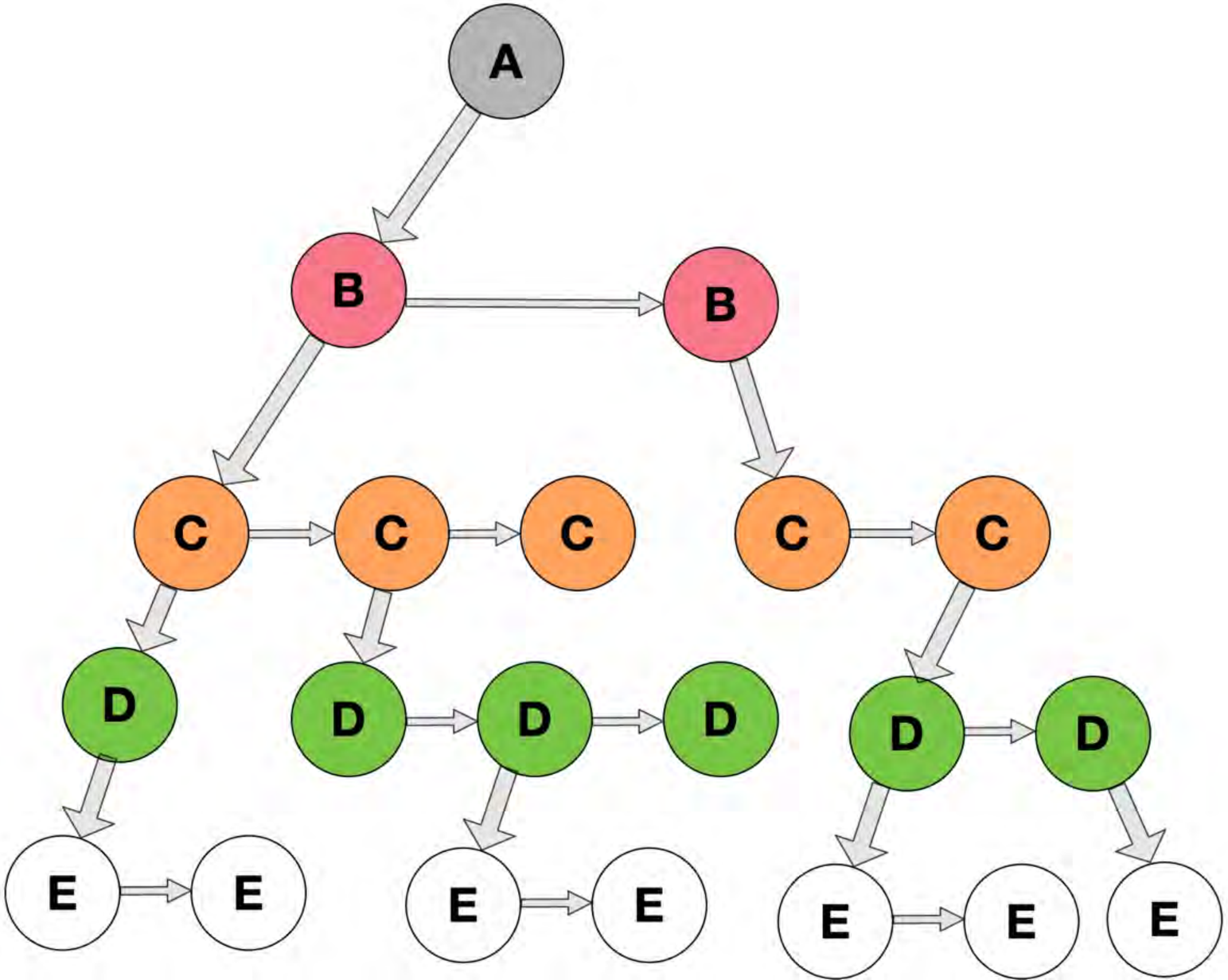
```
BEGIN_CBTS_STRUCT(Msg1)
    CBTS_IE  (TAG_1, _UC, f1)
    CBTS_IE  (TAG_2, _UL, f2)
END_CBTS_STRUCT()

BEGIN_CBTS_STRUCT(Msg2)
    CBTS_IE  (TAG_3, _UL,  f1)
    CBTS_IE  (TAG_4, Msg1, f2)
END_CBTS_STRUCT()

BEGIN_CBTS_STRUCT(Msg3)
    CBTS_ARRAY  (TAG_5, _UC,  f1, 3)
    CBTS_ARRAY  (TAG_6, Msg1, f2, 3)
END_CBTS_STRUCT()
```

ThoughtWorks®

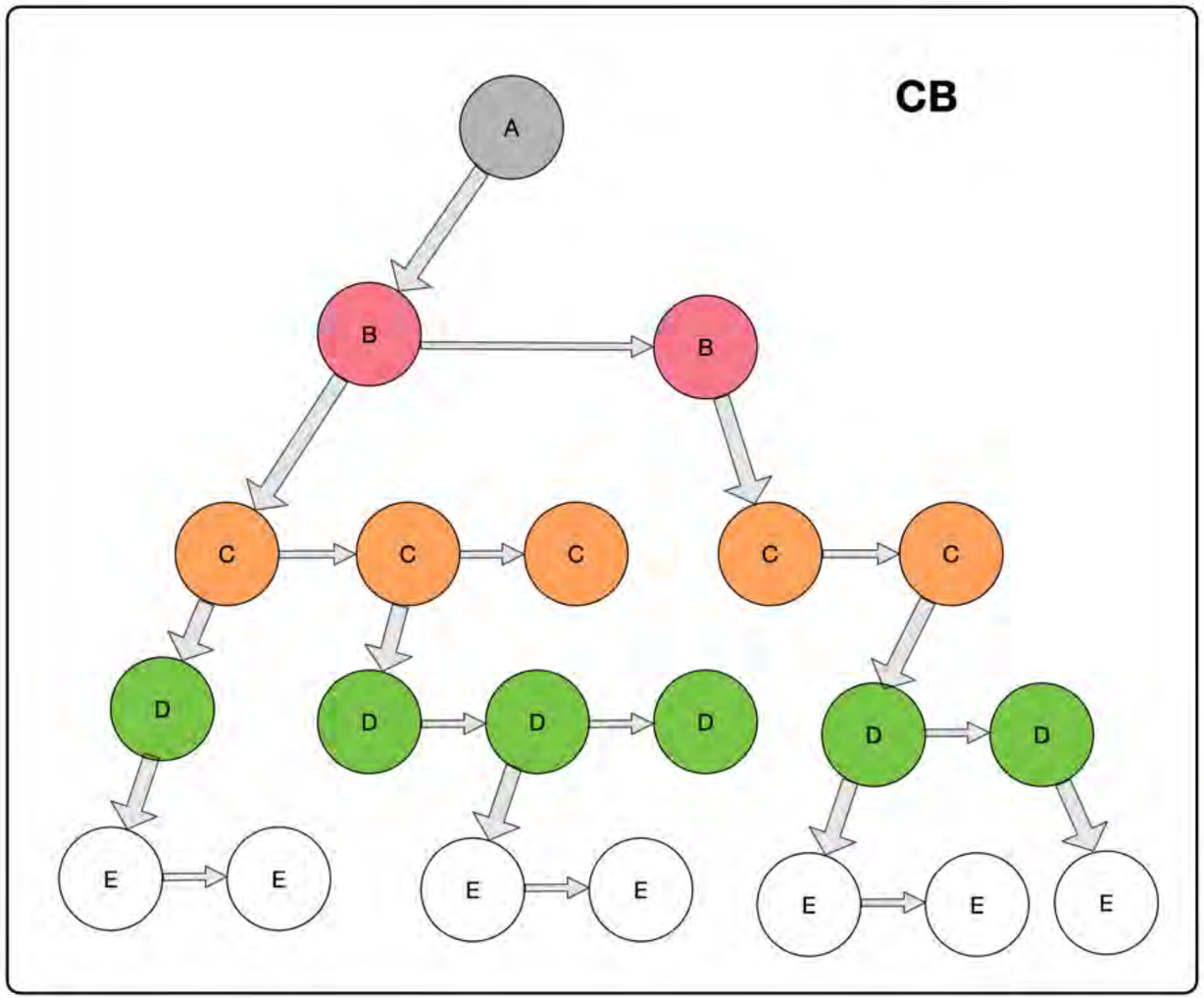
组合式设计

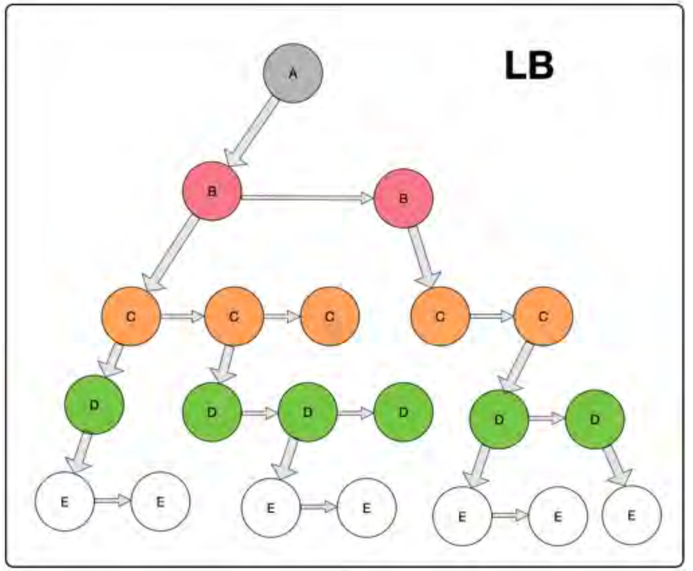
案例二

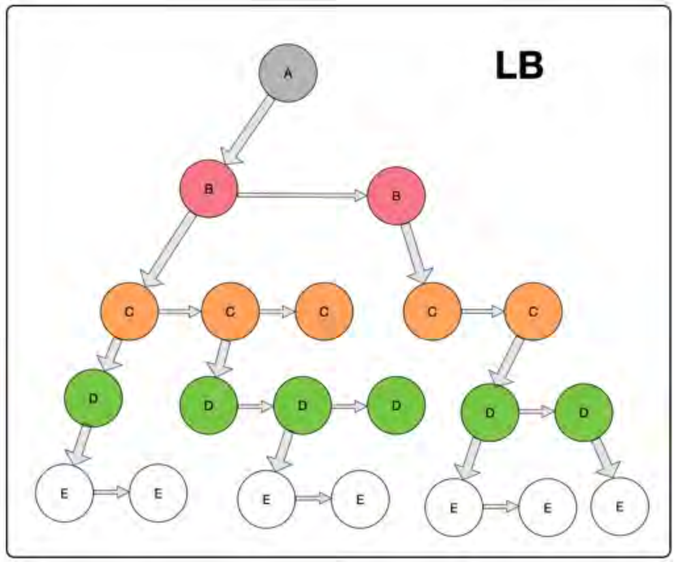
背景：受管对象树



背景：CB & LB (主控板和业务板)



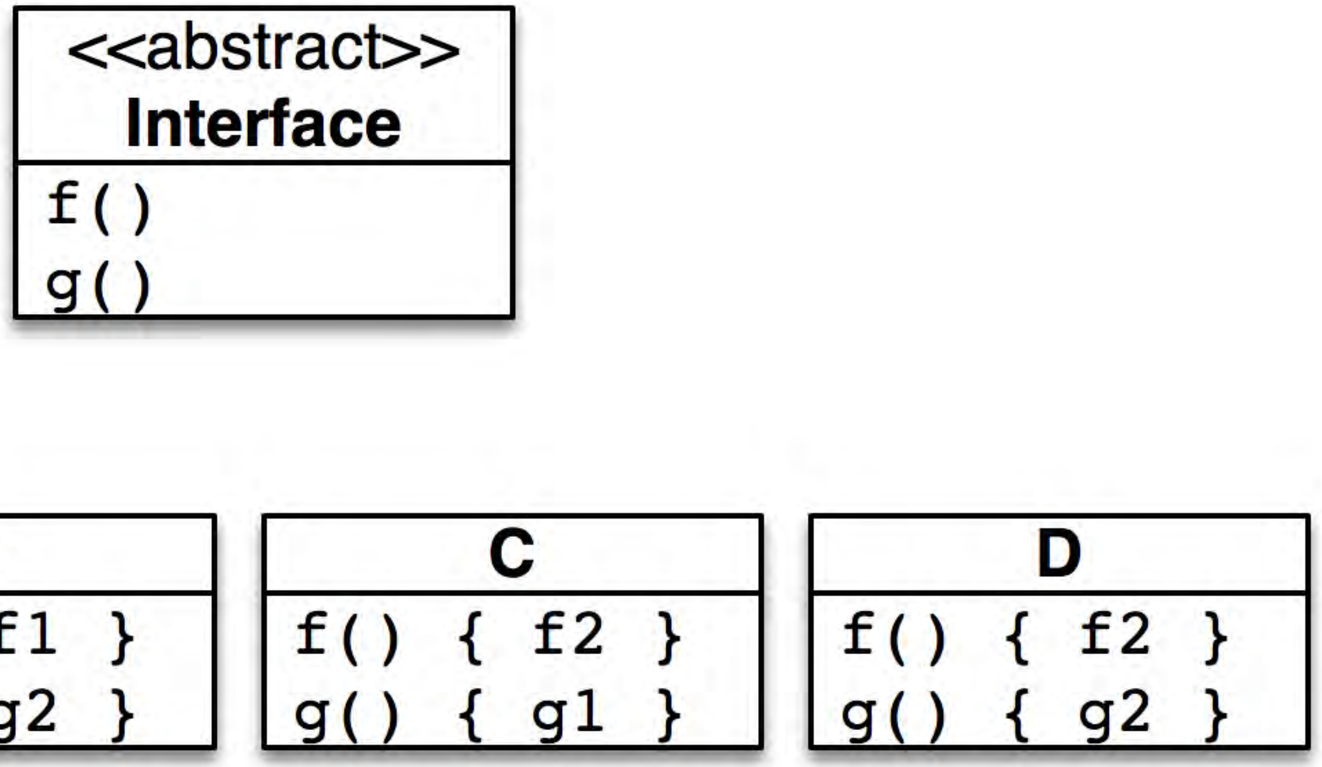




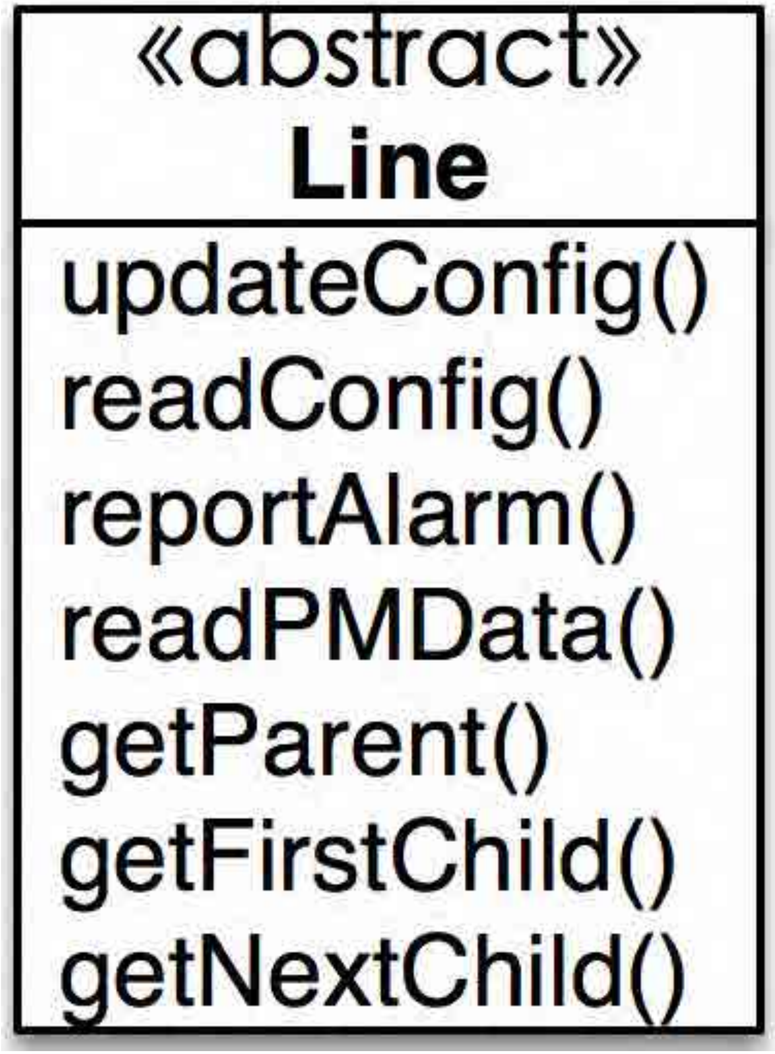
背景：过多的变化方向

- CB, LB上的同类对象，虽然概念上有相同的行为，但实现上却存在很大差异；
- 在CB或LB上，不同类对象间概念上也有相同的行为，但实现上也各有差异；
- 有些对象从概念上并不应该具备某些行为。

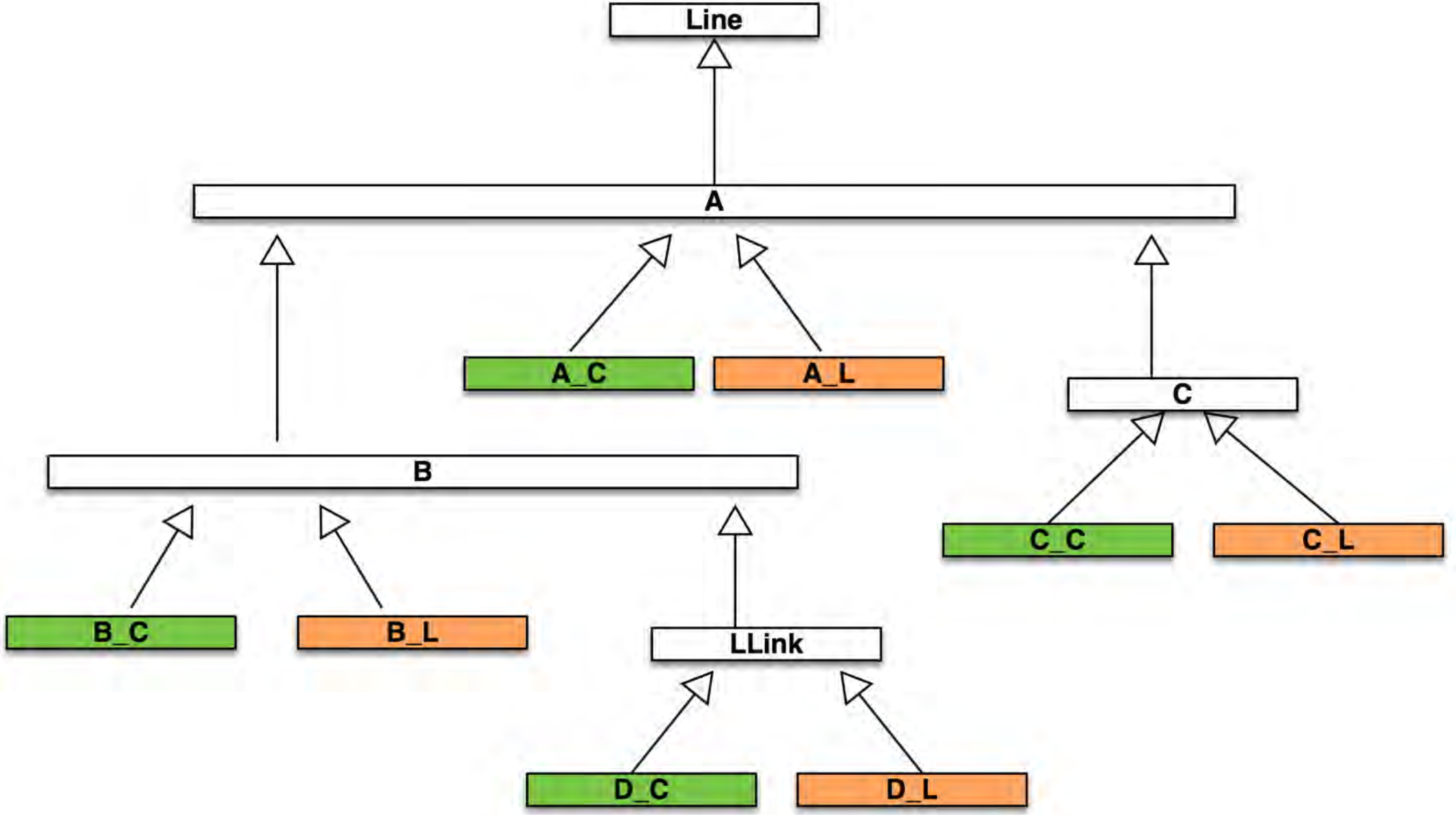
继承只能处理单一方向变化问题



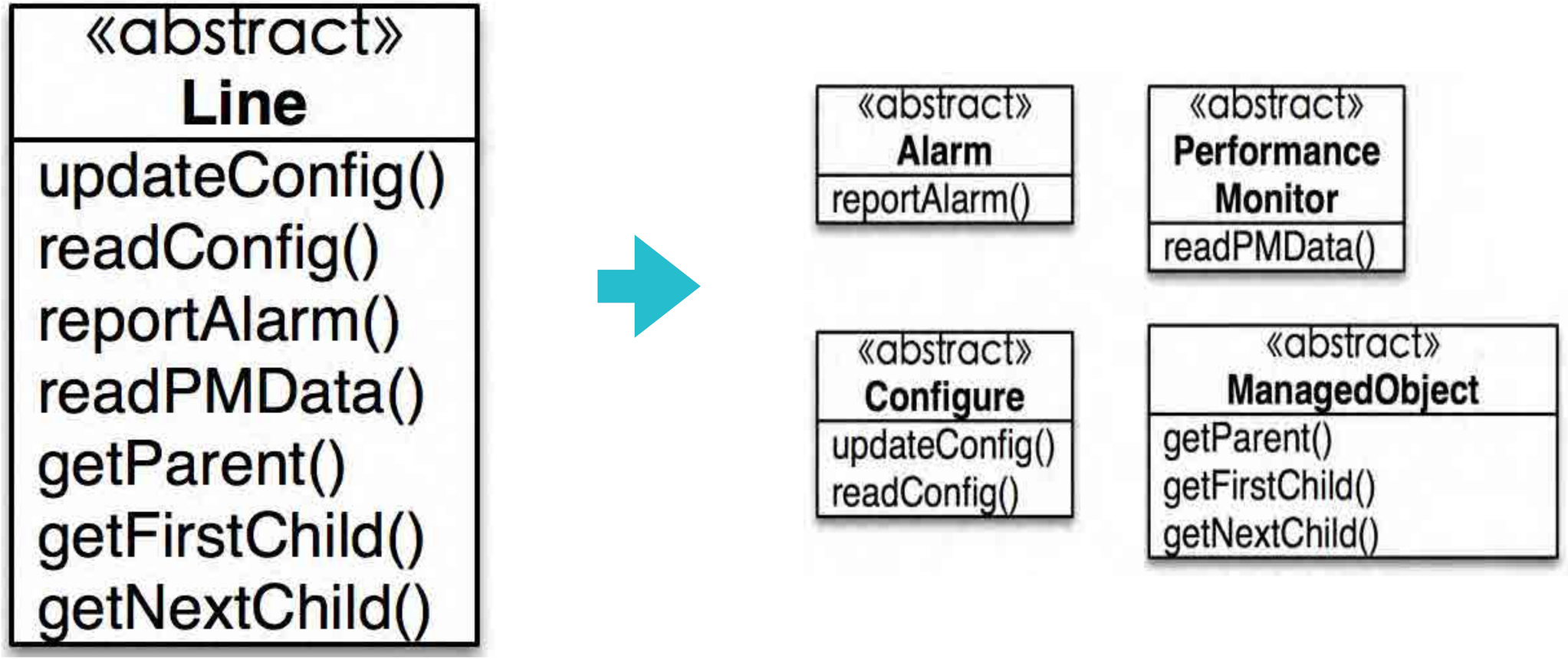
多重职责


«abstract»
Line
updateConfig()
readConfig()
reportAlarm()
readPMData()
getParent()
getFirstChild()
getNextChild()

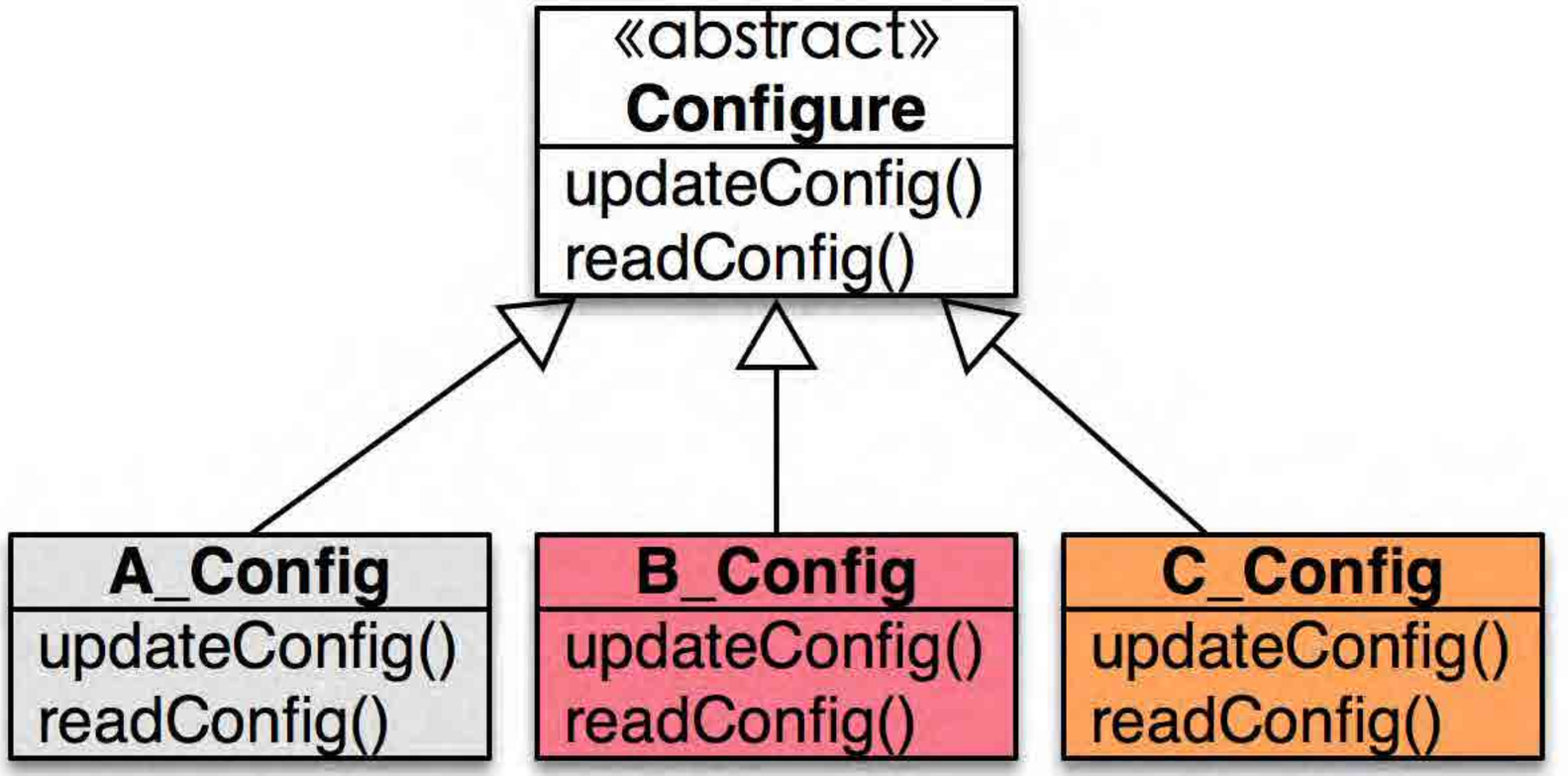
继承树



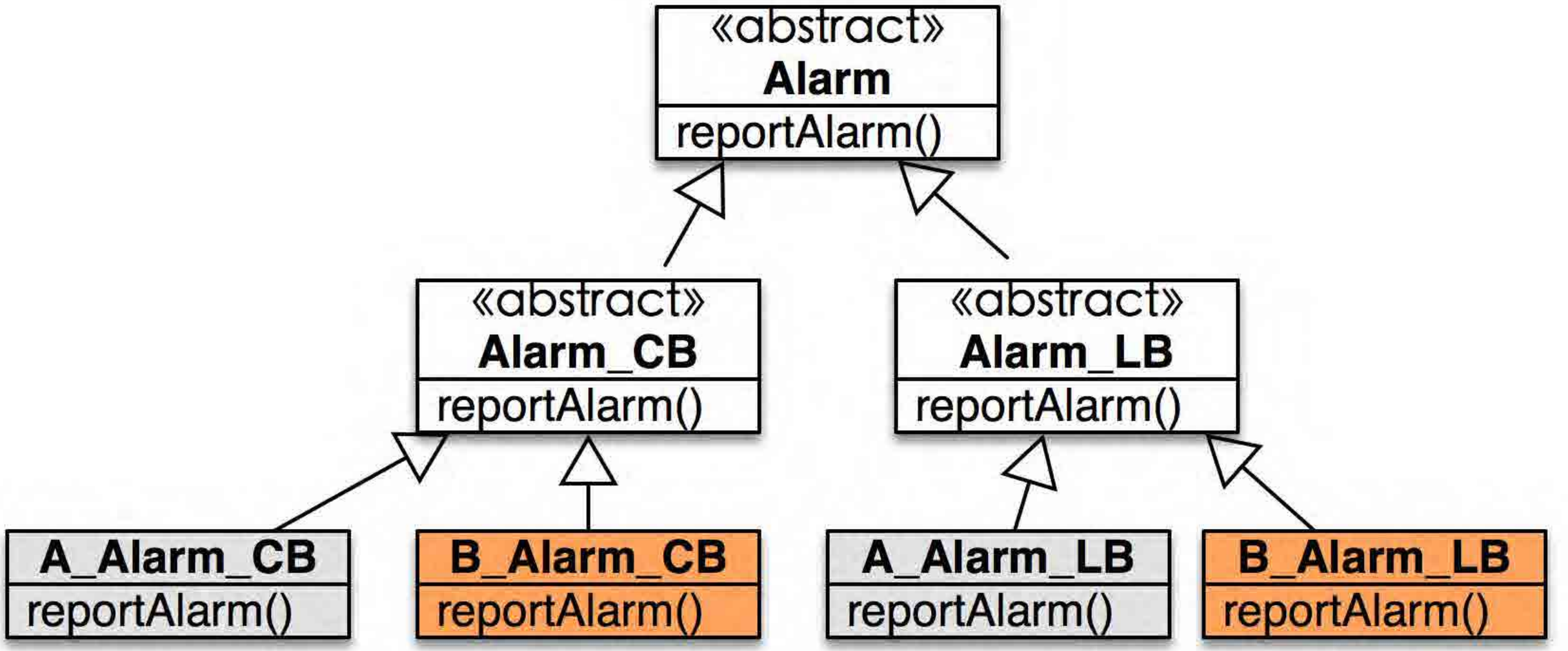
职责分离


«abstract»
Line
updateConfig()
readConfig()
reportAlarm()
readPMData()
getParent()
getFirstChild()
getNextChild()
«abstract»
Alarm
reportAlarm()
«abstract»
Performance Monitor
readPMData()
«abstract»
Configure
updateConfig()
readConfig()
«abstract»
ManagedObject
getParent()
getFirstChild()
getNextChild()

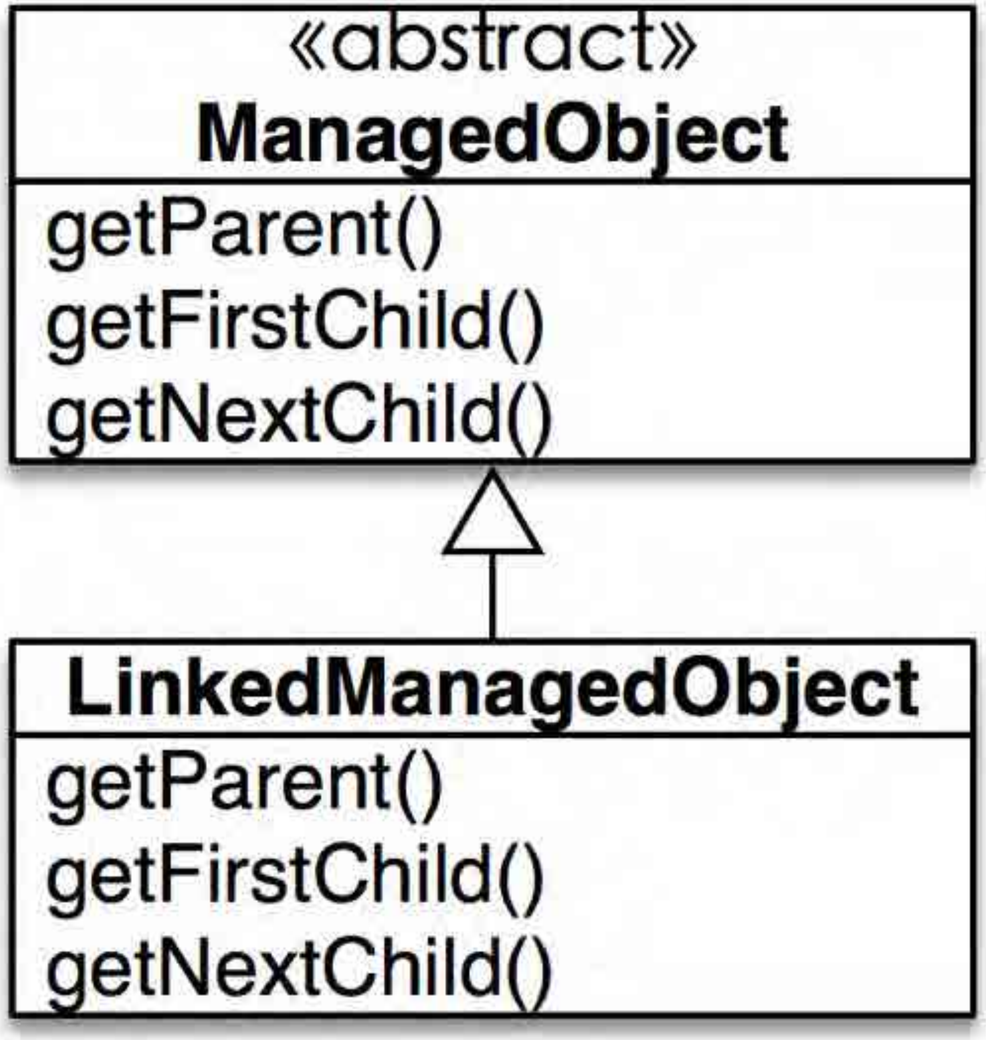
CONFIG的多个实现



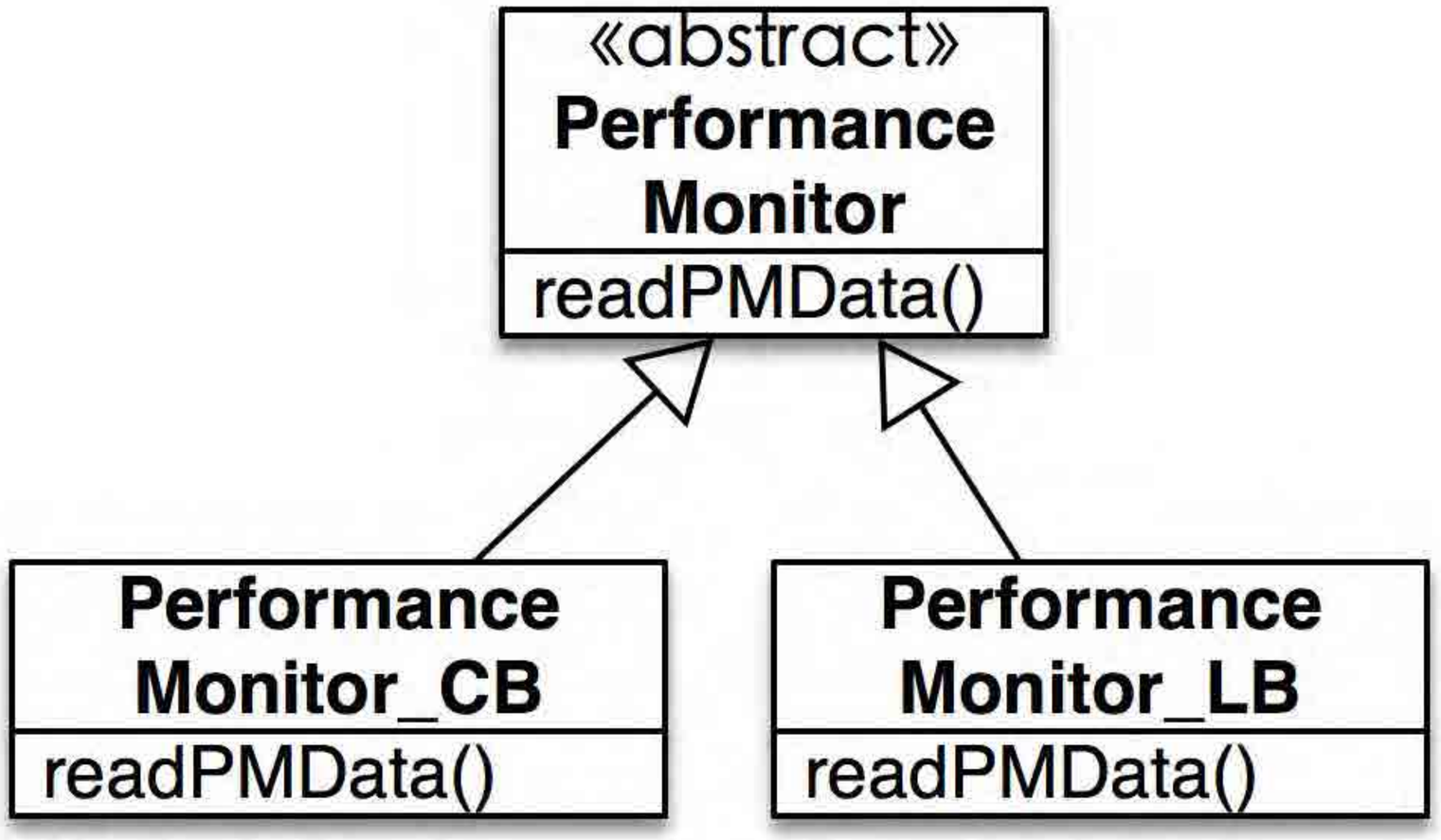
ALARM的多个实现


«abstract»
Alarm
reportAlarm()
«abstract»
Alarm_CB
reportAlarm()
«abstract»
Alarm_LB
reportAlarm()
A_Alarm_CB
reportAlarm()
B_Alarm_CB
reportAlarm()
A_Alarm_LB
reportAlarm()
B_Alarm_LB
reportAlarm()

MANAGED OBJECT共享同一个实现



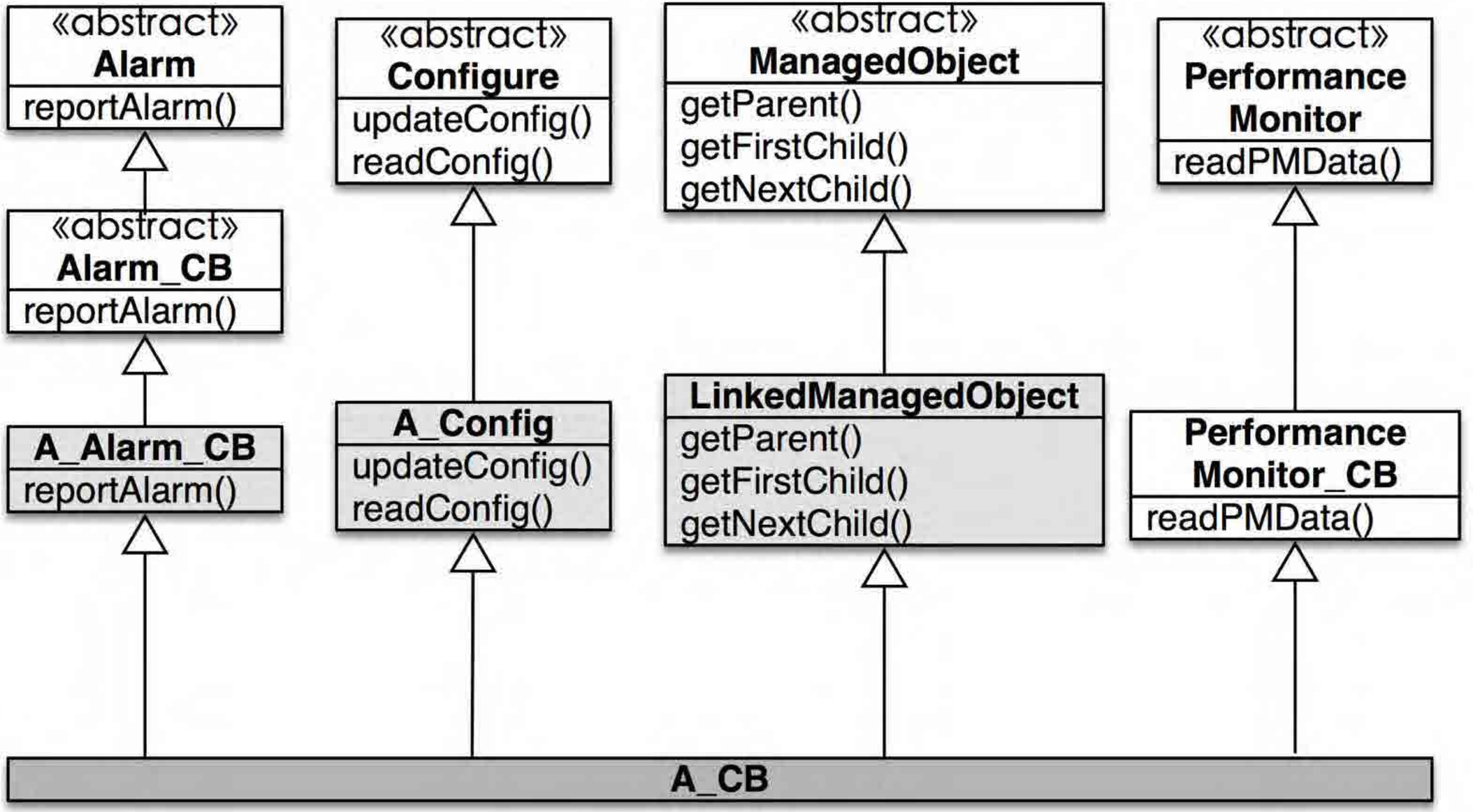
PERFORMANCE MONITOR的多个实现



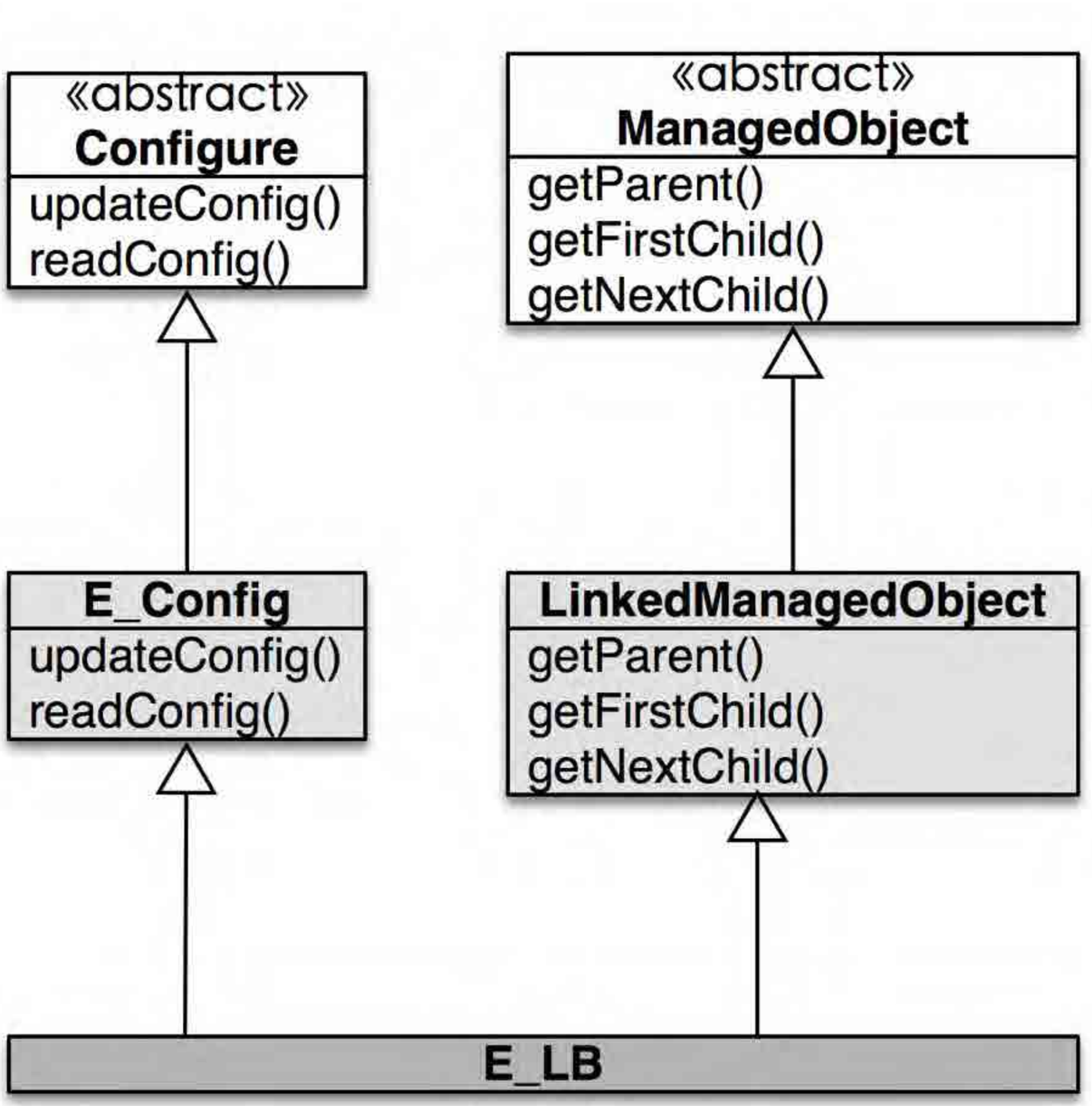
约束

- 尽管接口被分成了多个类，但在树上的节点依然应该是单个对象
- 通过依赖注入进行组合是一个常规选择，但是
  1. 并不是所有的对象从概念上都应该具备所有的接口
  2. 我们希望不希望外部依赖一个胖接口
- 在嵌入式环境下：
  1. 依赖管理所需的指针会增大内存开销；
  2. 增加程序员内存管理的负担；
  3. 增加程序员编写Factory代码的负担；

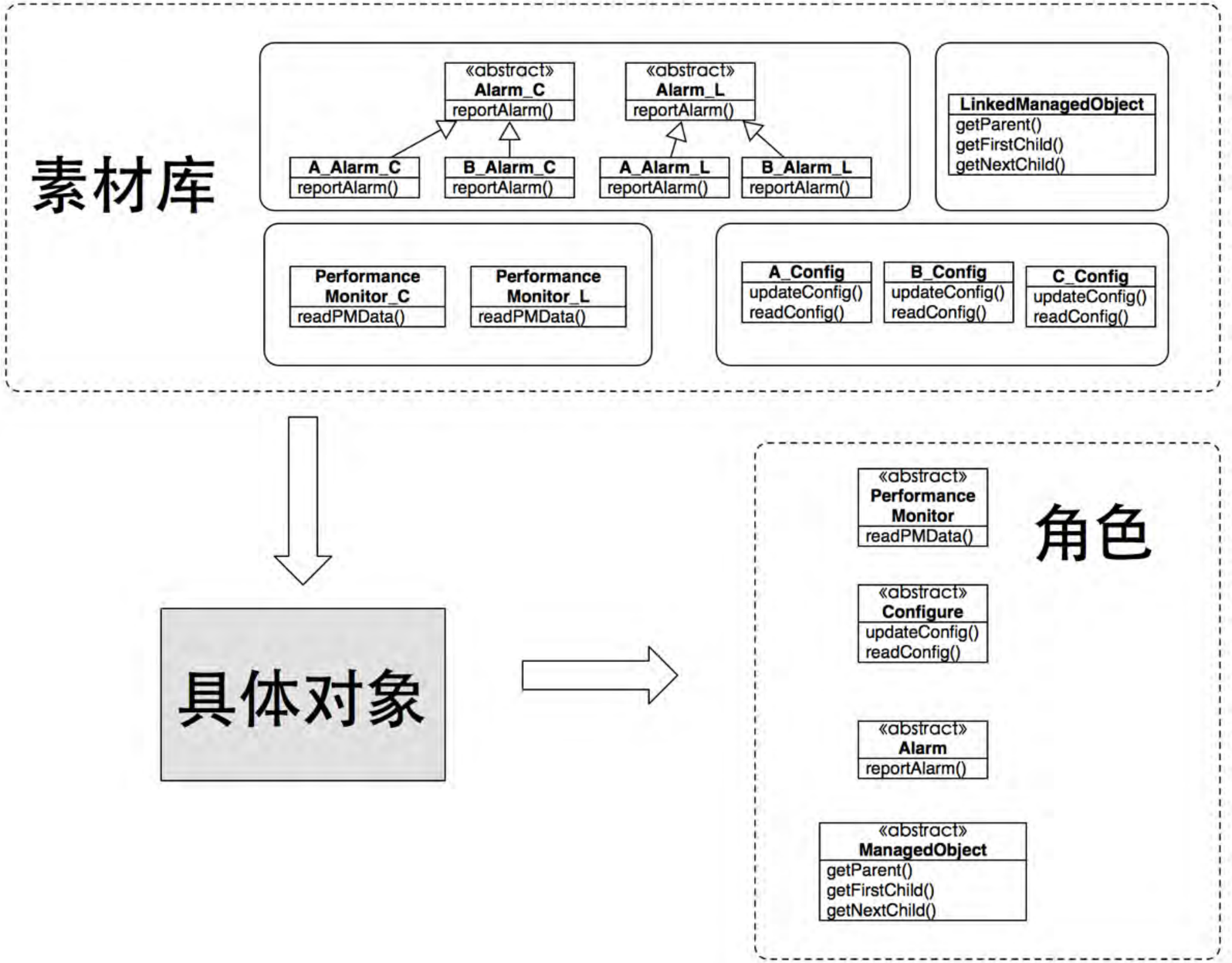
用继承进行组合（A_CB）


«abstract»
Alarm
reportAlarm()
«abstract»
Alarm_CB
reportAlarm()
A_Alarm_CB
reportAlarm()
«abstract»
Configure
updateConfig()
readConfig()
A_Config
updateConfig()
readConfig()
«abstract»
ManagedObject
getParent()
getFirstChild()
getNextChild()
LinkedManagedObject
getParent()
getFirstChild()
getNextChild()
«abstract»
Performance
Monitor
readPMData()
Performance
Monitor_CB
readPMData()
A_CB

用继承进行组合（E_LB）

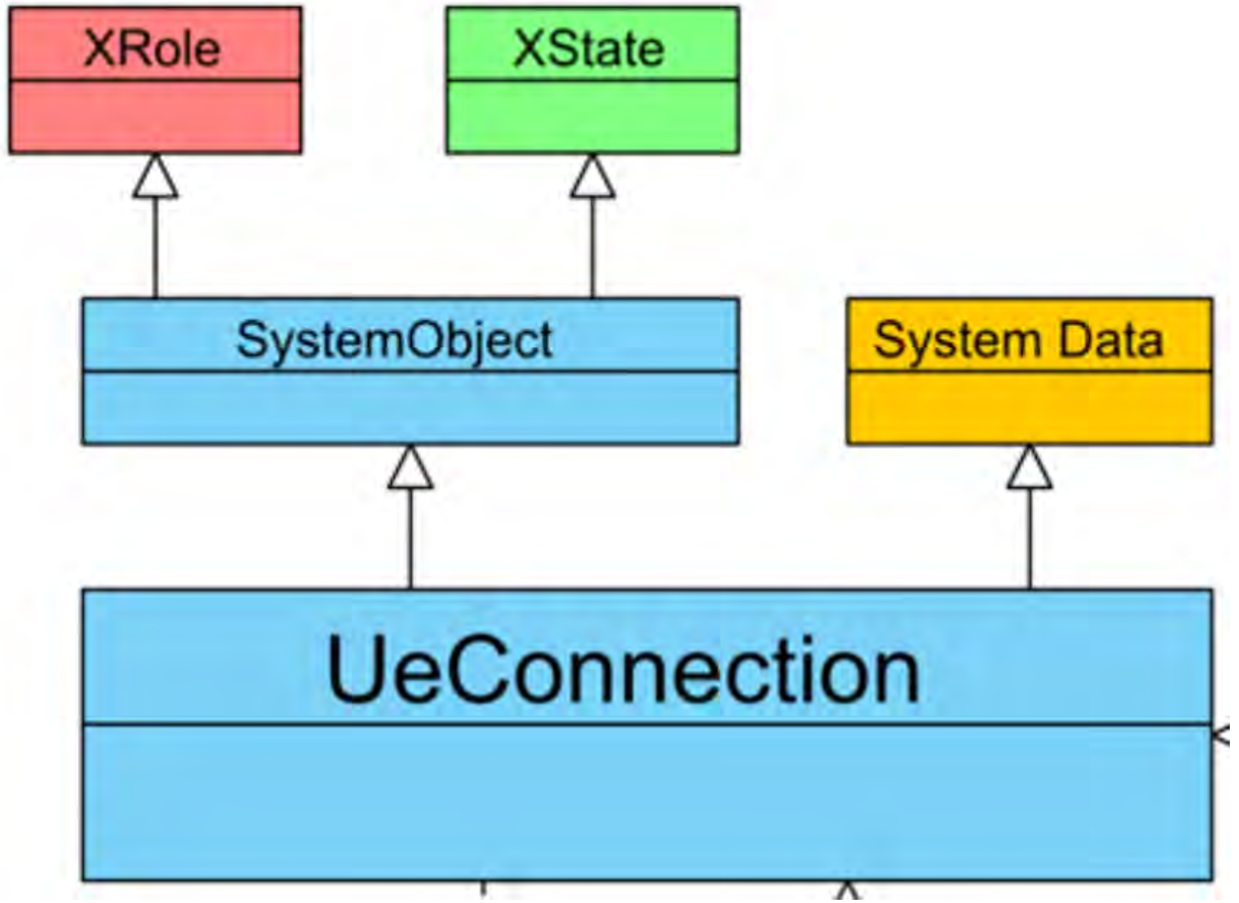

«abstract»
Configure
updateConfig()
readConfig()
«abstract»
ManagedObject
getParent()
getFirstChild()
getNextChild()
E_Config
updateConfig()
readConfig()
LinkedManagedObject
getParent()
getFirstChild()
getNextChild()
E_LB

灵活的组合关系


素材库
«abstract»
Alarm_C
reportAlarm()
«abstract»
Alarm_L
reportAlarm()
A_Alarm_C
reportAlarm()
B_Alarm_C
reportAlarm()
A_Alarm_L
reportAlarm()
B_Alarm_L
reportAlarm()
LinkedManagedObject
getParent()
getFirstChild()
getNextChild()
Performance Monitor_C
readPMData()
Performance Monitor_L
readPMData()
A_Config
updateConfig()
readConfig()
B_Config
updateConfig()
readConfig()
C_Config
updateConfig()
readConfig()
具体对象
角色
«abstract»
Performance Monitor
readPMData()
«abstract»
Configure
updateConfig()
readConfig()
«abstract»
Alarm
reportAlarm()
«abstract»
ManagedObject
getParent()
getFirstChild()
getNextChild()

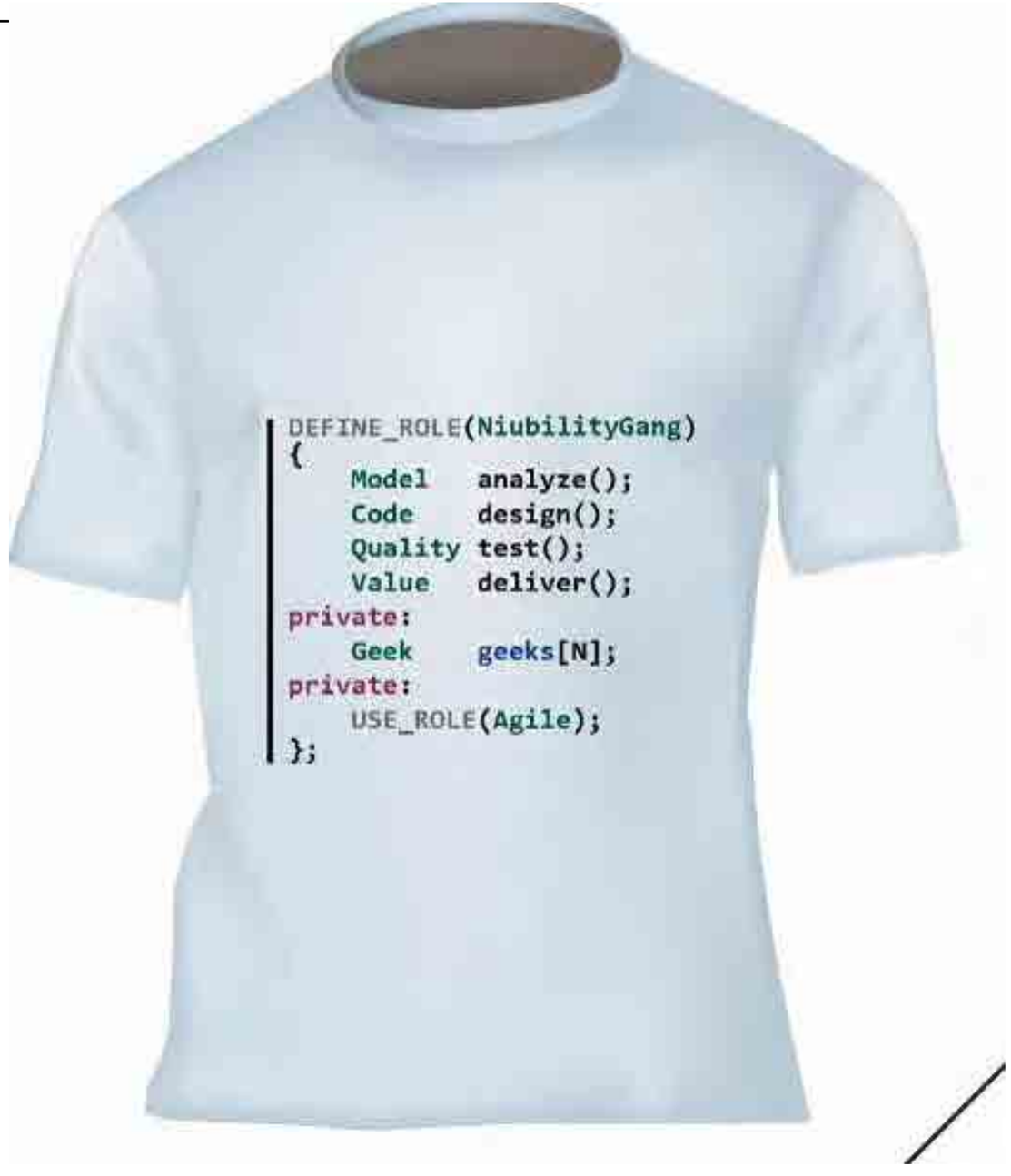
多角色对象

- 继承树倒置
- 没有违背继承的IS-A语义；
- 不同对象可以根据自己的真实概念逻辑选择实现部分角色
- 没有在嵌入式环境下引入额外的管理成本和内存开销；



A TYPICAL ROLE

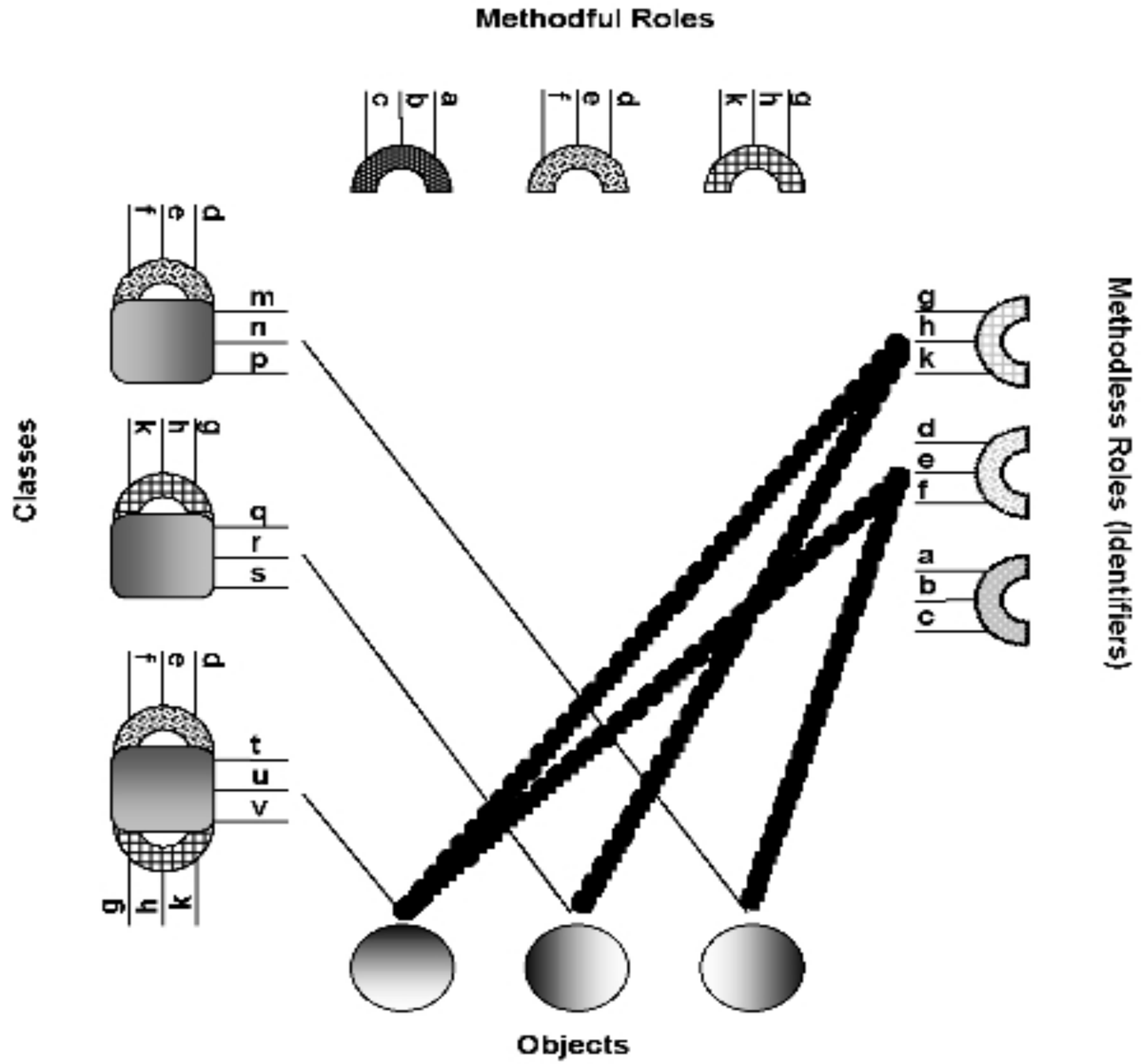
角色对其他角色的依赖，是依赖注入的语义，但实现成本更低。



小类大对象

- 类应该是小的
    1. 单一职责
    2. 上帝类是糟糕的设计
- 对象可以是大的
    1. 对象不应该和类有对应关系，而是应该和领域模型对应
    2. 但上帝对象没有任何问题

和DCI殊途同归

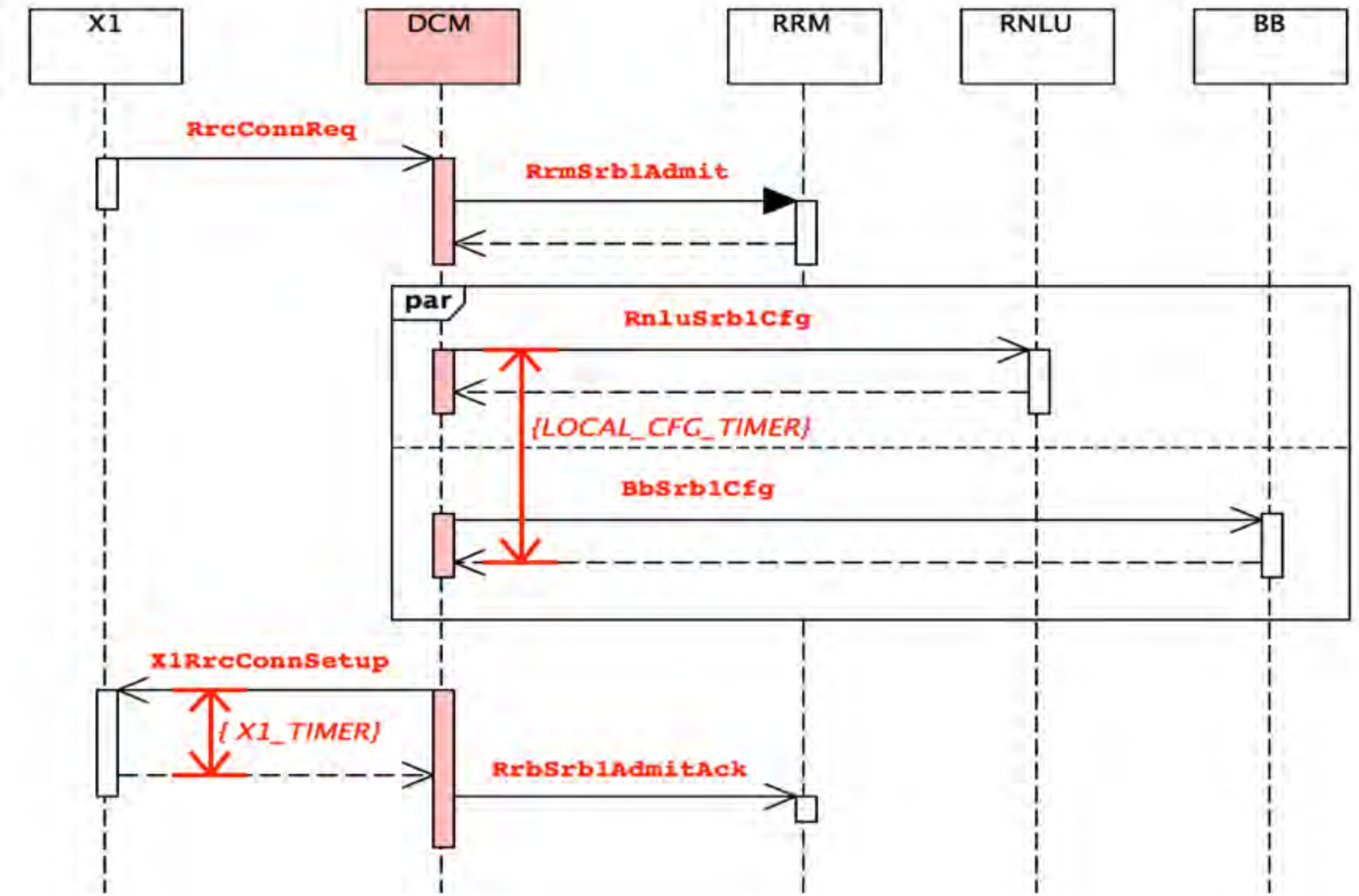

Methodful Roles
Classes
Methodless Roles (Identifiers)
Objects

ThoughtWorks®

组合式设计

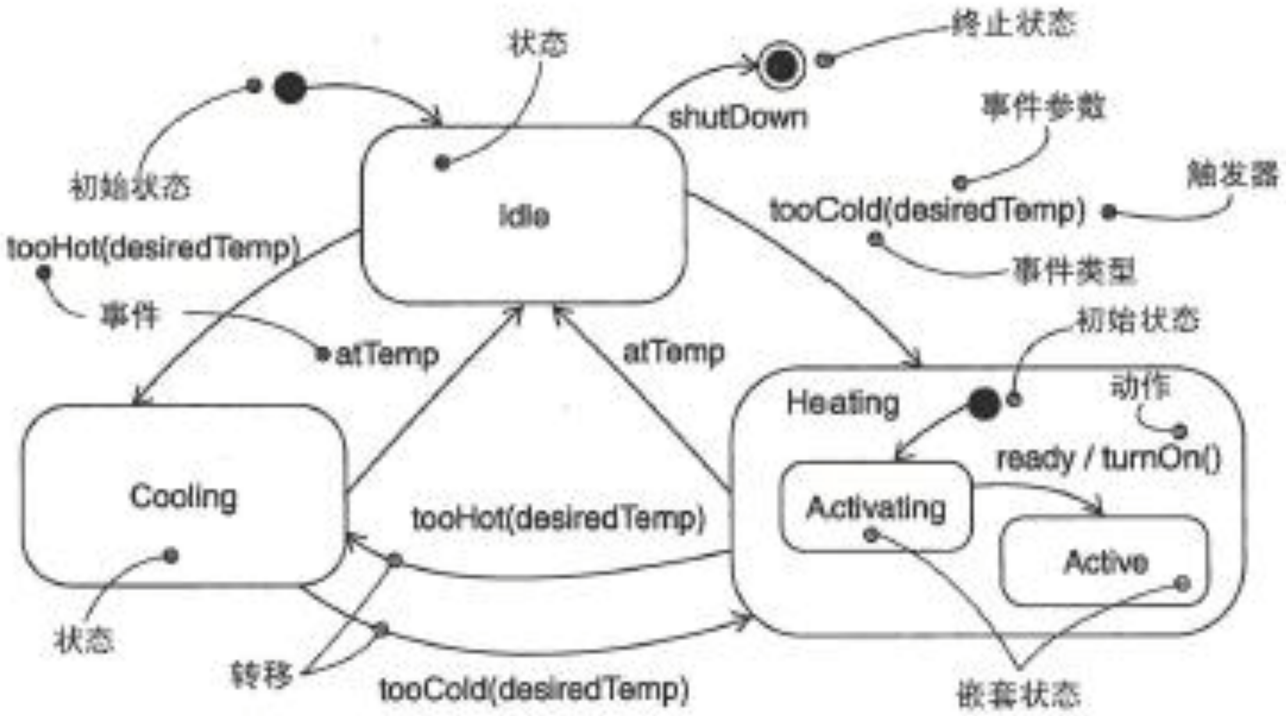
案例三

背景：目标系统由大量复杂的异步交互构成



背景：状态机是个糟糕的选择

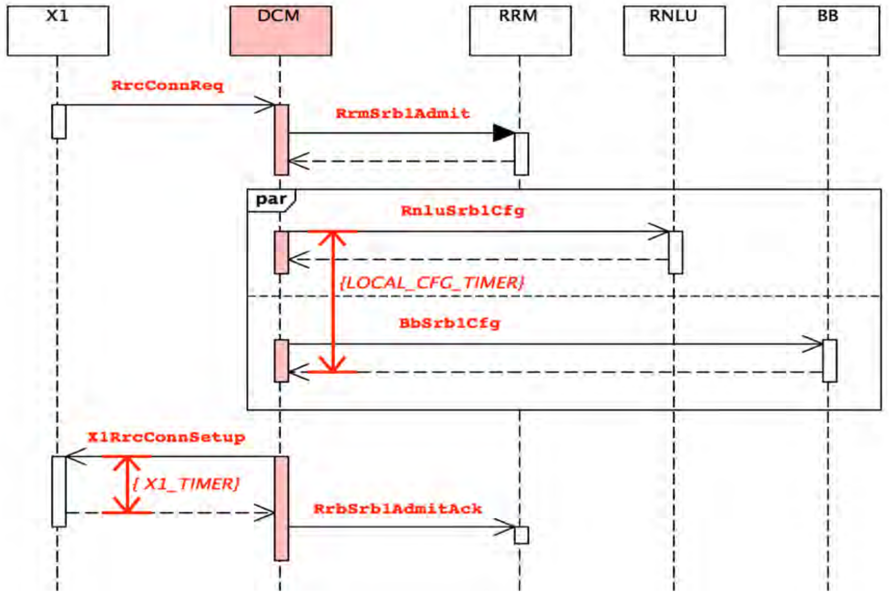
1. 非常多的临时状态
2. 状态机受流程的变化而变化
3. 状态机的代码与业务代码的耦合（goto state）



ACTION

```
DEF_SIMPLE_ASYNC_ACTION(Action3)
{
    Status exec(const TransactionInfo&)
    {
        // 构建并发送请求消息
        Reqeust3 request;
        request.build();
        Status status = sendRequestTo(OTHER_SYSTEM3_PID, request);
        if(status != SUCCESS) return status;

        // 声明自己要等待的应答消息类型，以及对应的处理函数
        WAIT_ON(EV_ACTION3_RSP, handleAction3Rsp);

        // 由于要等待应答消息，所以返回 CONTINUE
        return CONTINUE;
    }

private:
    // 定义事件处理函数
    Status handleAction3Rsp(const TransactionInfo&, const Event& event)
    {
        // 处理应答消息
        handleRsp(event);

        // 返回成功，代表此 Action 成功处理结束
        return SUCCESS;
    }
};
```

将ACTION进行组合




```
__def_transaction
( __sequential
    ( __req(RrcConnectRequest)
    , __call(RrmAdmitSrb1, __actor(TargetRrm))
    , __timer_prot
        ( LOCAL_CFG_TIMER
        , __concurrent
            ( __asyn(RnluConfigSrb1, __actor(TargetRnlu, __with(TargetCell)))
            , __asyn(BbConfigSrb1, __actor(TargetBb))))
    , __timer_prot(X1_CFG_TIMER, __asyn(RrcConnectSetup, __actor(TargetCell)))
    , __call(RrmAck, __actor(TargetRrm)))
) RrcConnect;
```

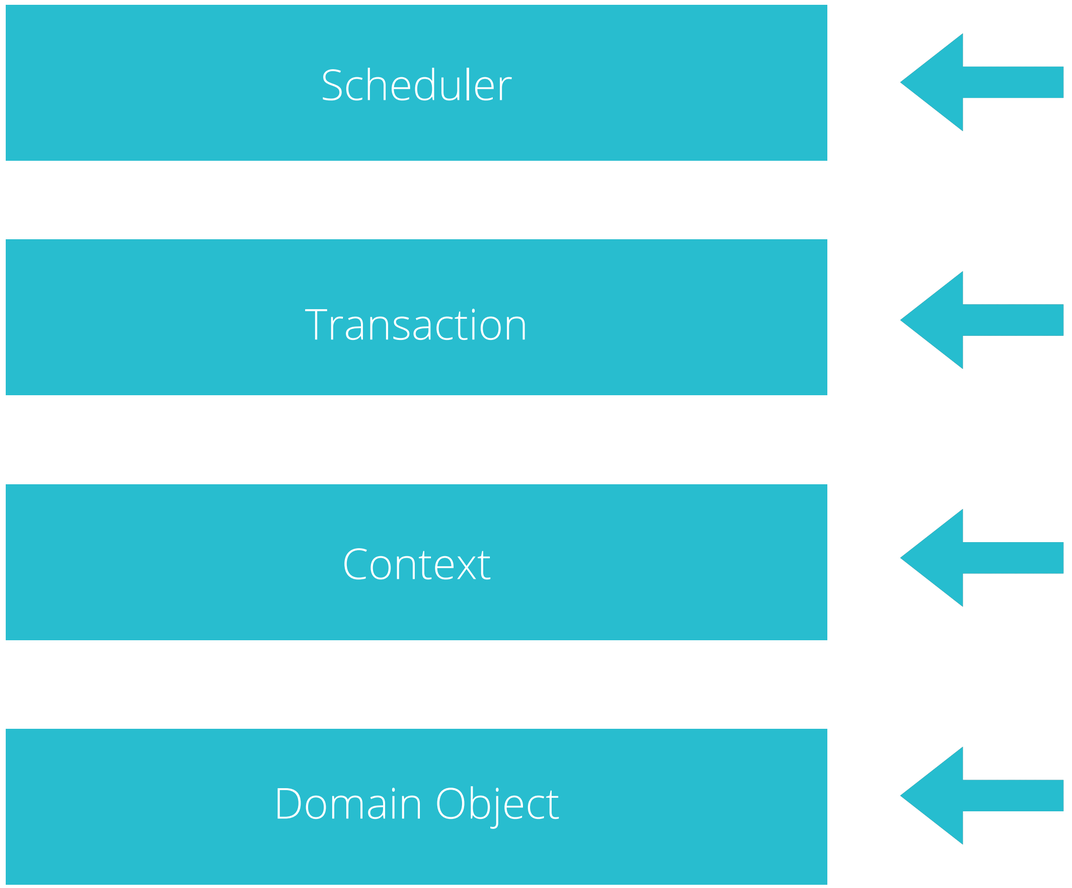
结果

- 对异步过程的直接描述
- 节省了大量的状态机维护工作
- 业务代码和流程各自独立的变化，互不影响
- 绝大多数Action之间也处于正交关系
- Action被当作一种组合原子，可以在任意流程里，以组合的方式被复用。

EVEN BETTER

- 大量对流程进行监控的的工具类需求，亦通过组合的方式完成；
    1. 4天之内完成了近2000项KPI Counter；
    2. 这些工具类需求和Action以及Transaction完全正交

四层架构



状态模型框架

1. 只包含真正的状态
2. 状态之间的转换控制

定义事务

1. 使用Transaction DSL 定义事务

定义基本操作

1. 从环境中找对象
2. 将对象Cast成角色
3. 角色之间交互，完成算法

定义领域模型

1. 领域对象
2. 对象间的关系
3. 对象所实现的角色

领域层

- 领域层使用“小类大对象”的方式进行角色组合

ThoughtWorks®

组合式设计

总结

组合式设计

为了应对复杂的变化方向：

- 单一职责的组合元素
- 低成本的组合手段

简单的问题简单处理，避免过度设计

以下四个原则的重要程度依次降低。

1. 通过所有测试（Passes its tests）
2. 尽可能消除重复 (Minimizes duplication)
3. 尽可能清晰的表达 (Maximizes clarity)
4. 尽可能的减少代码元素的的数量 (Has fewer elements)

THANK YOU

For questions or suggestions:
Yingjie Yuan 18123980818
darwin_yuan@hotmail.com

ThoughtWorks®