Agile Design

敏捷设计

2022/6/9

摘要

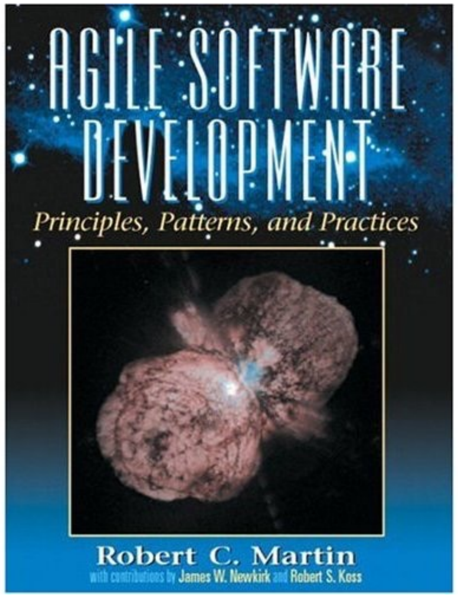
- Introduction
- Agile Development
- Agile Design

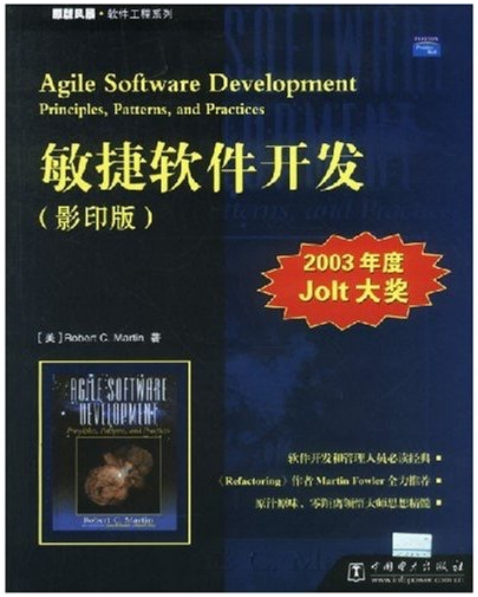
摘要

- **Introduction**
- Agile Development
- Agile Design

Introduction

- Robert C. Martin: “Agile Software Development Principles, Patterns, and Practices”, Prentice Hall (October 25, 2002)


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss



Jolt Award

2003

Introduction

- Robert C. Martin

Uncle Bob

“Designing Object Oriented C++ Applications using the Booch Method ”, 1995

“Pattern Languages of Program Design 3”, 1997

“More C++ Gems”, 1999

“Extreme Programming in Practice”, 2003

“UML for Java Programmers ”,2003

http://cleancoder.com/products

Agile 敏捷

- 敏捷开发是一种面临迅速变化的需求快速开发软件的能力。
  - 提供必要的纪律和反馈的实践 -- practice
  - 保持软件灵活、可维护的设计原则 -- principle
  - 针对特定问题的设计模式 -- pattern
- 适应变化和以人为中心，迭代、循序渐进

Agile Processes

- SCRUM
- Crystal
- Feature Driven Development 特征驱动软件开发
- Adaptive Software Development 自适应软件开发
- eXtreme Programming (XP) 极限编程
- …

摘要

- Introduction
- **Agile Development**
- Agile Design

Agile Development

- Extreme Programming（XP，极限编程） 是一种轻量级的软件开发方法，它使用快速的反馈，大量而迅速的交流，经过保证的测试来最大限度的满足用户的需求。
  - XP强调用户满意，开发人员可以对需求的变化作出快速的反应。
  - XP强调team work。项目管理者，用户，开发人员都处于同一个项目中，他们之间的关系不是对立的，而是互相协作的，具有共同的目标：提交正确的软件。

Extreme Programming

- XP强调4个因素：
  - 交流（communication），XP要求程序员之间以及和用户之间有大量而迅速的交流
  - 简单（simplicity），XP要求设计和实现简单和干净
  - 反馈（feedback），通过测试得到反馈，尽快提交软件并根据反馈修改
  - 勇气（courage），勇敢的面对需求和技术上的变化

Extreme Programming

- XP特别适用于需求经常改变的领域，客户可能对系统的功能并没有清晰的认识，可能系统的需求经常需要变动。
- XP也适用于风险比较高的项目，当开发人员面对一个新的领域或技术时，XP可以帮助降低风险
- XP适用于小的项目（人员上），人员在2-12人之间，XP不适用于人员太多的项目

Practices of XP

- 客户作为团队成员
- 用户素材 user stories
- 短周期交付
  - 迭代计划
  - 发布计划
- 验收测试
- 结对编程 pair programming

Practices of XP

- 测试驱动的开发方法 **Test-Driven Development**
- 集体所有权
- 持续集成
- 可持续的开发速度
- 开放的工作空间
- 计划游戏 **planning game**
- 简单的设计
- 重构 **Refactoring**
- 隐喻 Metaphor：将整个系统联系在一起的全局视图

摘要

- **Introduction**
- Agile Development
- **Agile Design**

What is Design?

- "*After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings*."

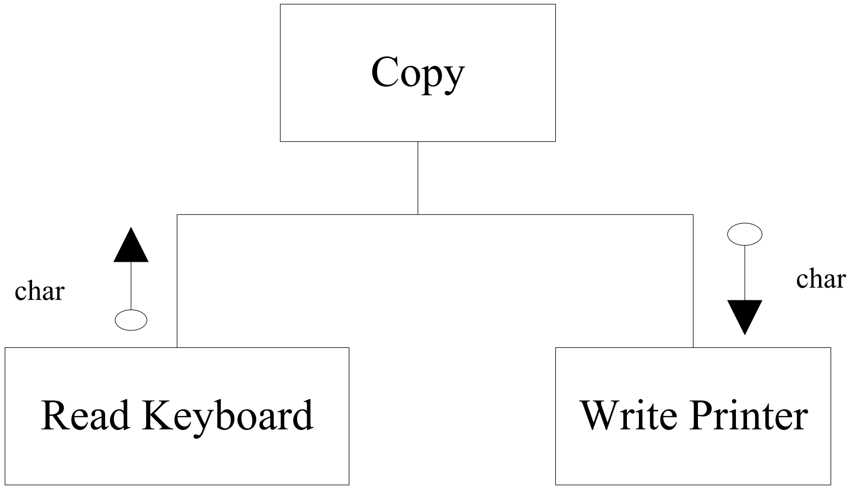
-- Jack Reeves

Design Smells

- Rigidity 僵化性 → 难于修改
- Fragility 脆弱性 → 一改便乱
- Immobility 牢固性 → 难于重用
- Viscosity 粘滞性 → 做好事难
- Needless Complexity 不必要的复杂性
- Needless Repetition 不必要的重复
- Opacity 晦涩性

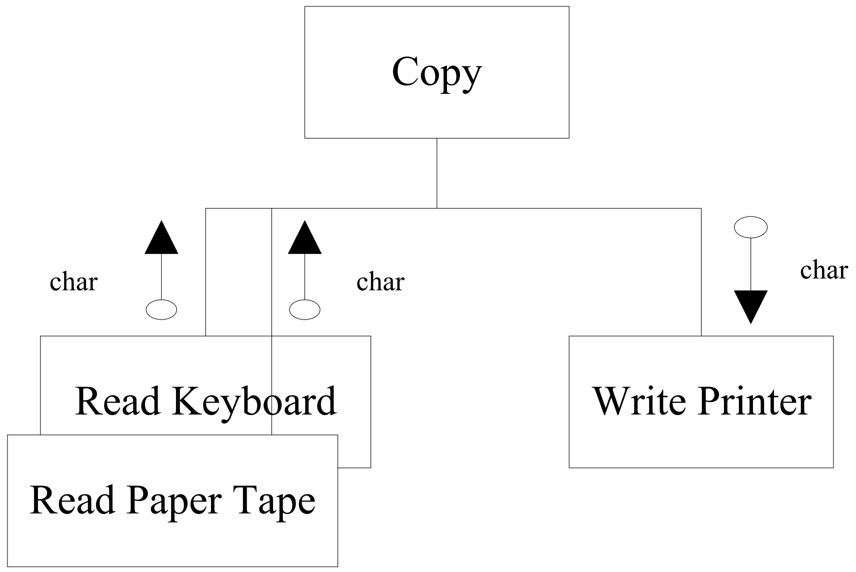
The “Copy” Program

- Initial Design



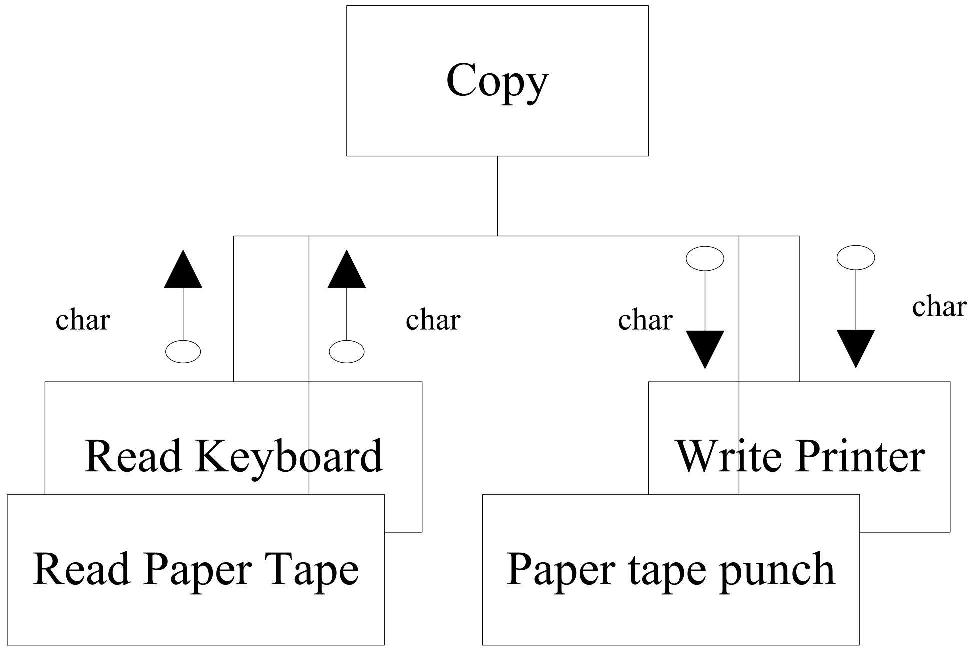
```
void copy(){
   int c;
   while ((c=RdKbd())!= EOF)
      WrtPrt(c);
}
```

Requirement changes



```
bool ptFlag = false;
//remember to reset this flag
void Copy(){
    int c;
    while ((c=(ptFlag?RdPt():
     RdKbd())!=EOF)
      WrtPrt(c);
}
```

Requirement changes again!



```
bool ptFlag = false;
bool punchFlag = false
//remember to reset these flags
void Copy(){
    int c;
    while ((c=(ptFlag?RdPt():
     RdKbd())!=EOF)
       punchFlag ? WrtPunch(c) :
     WrtPrt(c);
}
```

Agile design of the “copy” example

```
int RdKbd();
void WrtPrt(int);
const int EOF = -1;
class Reader{
  public: virtual int read() = 0;
};
class KeyboardReader : public Reader{
  public: virtual int read() {return RdKbd();}
};

KeyboardReader GdefaultReader;
void Copy(reader& reader = GdefaultReader){
  int c;
  while ((c=reader.read()) != EOF)
    WrtPrt(c);
}
```

Agile developers

- 知道要做什么
  - 遵循敏捷实践去发现问题;
  - 应用设计原则去诊断问题;
  - 应用适当的设计模式去解决问题
- 软件开发的这三个方面间的相互作用就是设计

Agile Design

- 结论：敏捷设计是一个过程，不是一个事件。它是一个持续的应用原则、模式以及实践来改进软件的结构和可读性的过程。它致力于保持系统设计在任何时候都尽可能地简单、干净以及富有表现力。

A more complex example: Multi-panel interactive systems

- 问题
- 简单方案
- 结构化的方案
- 面向对象的方案
- 讨论

Multi-panel interactive systems

- 问题：
  - 业务流程
    - 每个会话（session）须经历多个步骤
  - 当前步骤
    - 显示panel（对话框），获取用户输入（选择），若输入错，给提示，直至正确；依据输入进行处理并转入下一步骤（转入哪个步骤可能依赖于用户的输入）；
  - 对话界面
  - 例如
    - 航空订票

– Enquiry on Flights –

Flight sought from: *Santa Barbara* To: *Paris*

Departure on or after: *21 Nov* On or before: *22 Nov*

Preferred airline (s):

Special requirements:

AVAILABLE FLIGHTS: **1**

Flt# AA 42 **Dep 8:25** **Arr 7:45** **Thru: Chicago**

Choose next action:

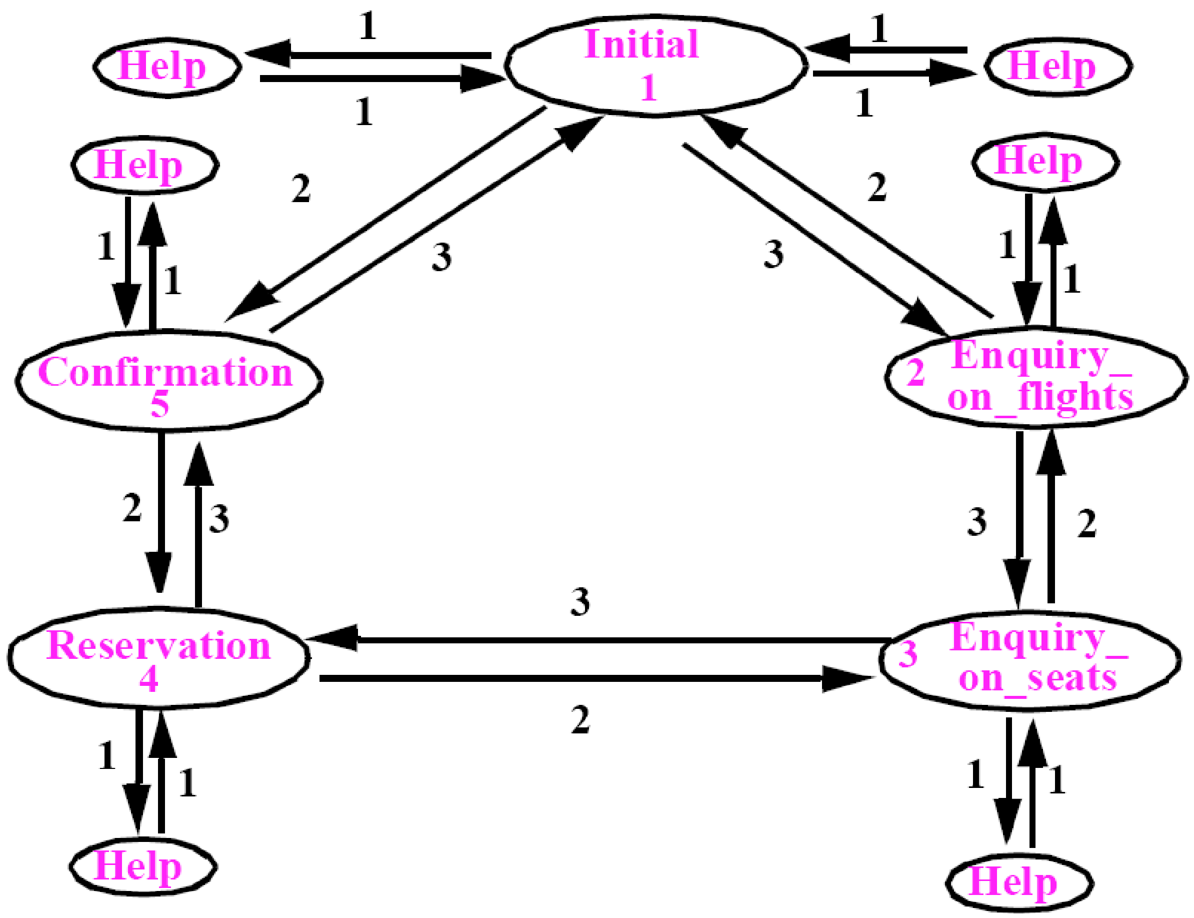
0 — **Exit**

1 — **Help**

2 — **Further enquiry**

3 — **Reserve a seat**

状态转换图


Initial
1
Help
Help
Help
Help
Help
Help
Confirmation
5
2 Enquiry_
on_flights
Reservation
4
3 Enquiry_
on_seats
1
2
3

设计要点

- 图可能很大
- 图可能变化
- 要考虑复用

软件设计之难在于多种（可能冲突）的需求之间的均衡取舍

A Simple-minded solution

```
$B_{Enquiry}$:
    "Display *Enquiry on flights* panel"
    **repeat**
        "Read user's answers and choice $C$ for the next step"
        **if** "Error in answer" **then** "Output appropriate message" **end**
    **until not** error in answer **end**
    "Process answer"
    **case** $C$ **in**
        $C_0$: **goto** $Exit$,
        $C_1$: **goto** $B_{Help}$,
        $C_2$: **goto** $B_{Reservation}$,
        ...
    **end**
```

A Simple-minded solution

- 问题
  - “Goto”!
  - 本质：
    - 转换图结构分散地hardwired到各个模块的算法中
    - 若增加状态或改动流程?
    - 如何复用?

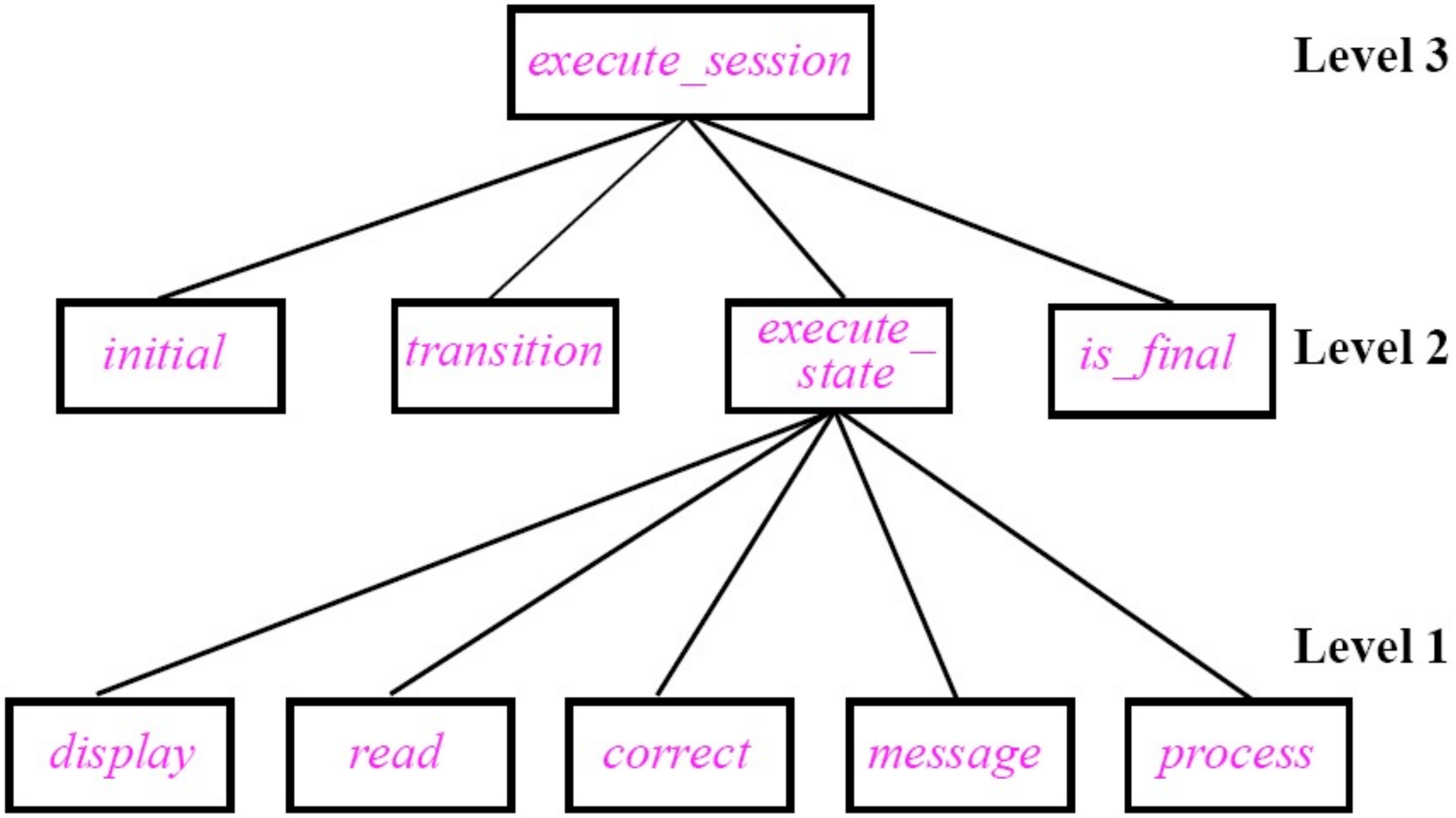
A functional, top-down solution

- 好，我们消除goto,并将流程独立出来，放到一个函数中

  transition(state,choice)

Choice → **↓ State**	**0**	**1**	**2**	**3**
1 (*Initial*)	**–1**	**0**	**5**	**2**
2 (*Flights*)		**0**	**1**	**3**
3 (*Seats*)		**0**	**2**	**4**
4 (*Reserv.*)		**0**	**3**	**5**
5 (*Confirm*)		**0**	**4**	**1**
0 (*Help*)		*Return*		
–1 (*Final*)				

Top-down decomposition

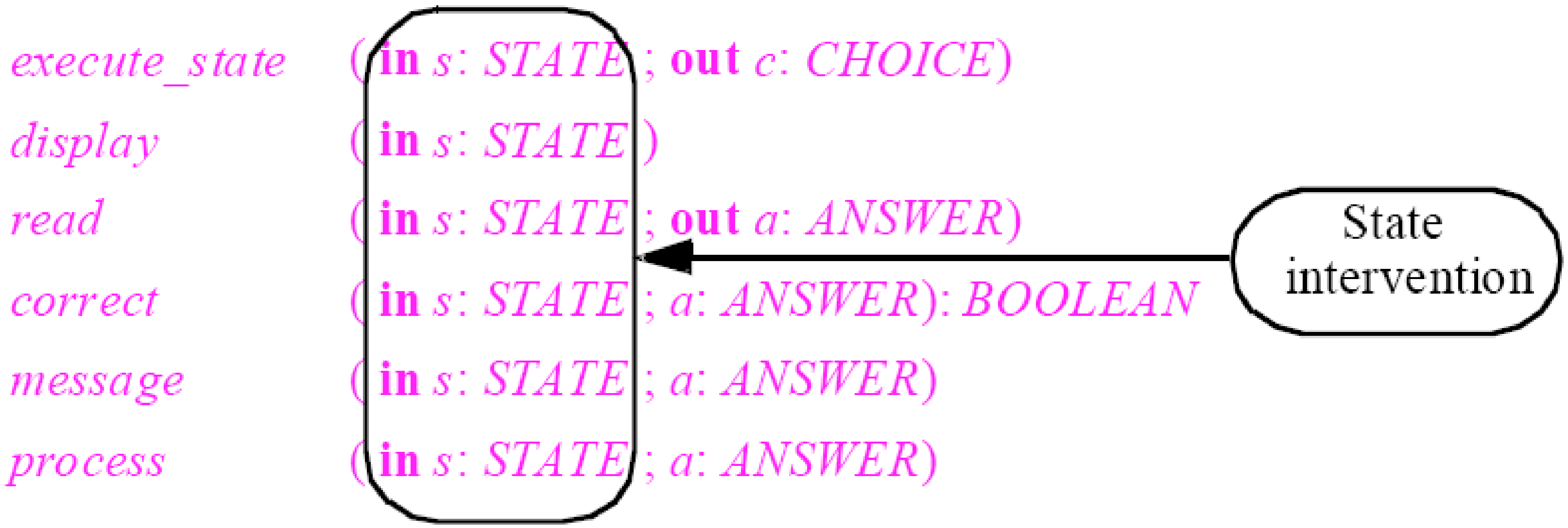


The top

```
execute_session is
        -- Execute a complete session of the interactive system
    local
        state, choice: INTEGER
    do
        state := initial
        repeat
            execute_state (state, →next)
                -- Routine execute_state updates the value of next.
            state := transition (state, next)
        until is_final (state) end
    end
```

```
execute_state (in s: INTEGER; out c: INTEGER) is
        -- Execute the actions associated with state s,
        -- returning into c the user's choice for the next state.
    local
        a: ANSWER; ok: BOOLEAN
    do
        repeat
            display (s)
            read (s, →a)
            ok := correct (s, a)
            if not ok then message (s, a) end
        until ok end
        process (s, a)
        c := next_choice (a)
    end
```

Critique



execute_state (**in** *s*: *STATE*; **out** *c*: *CHOICE*)

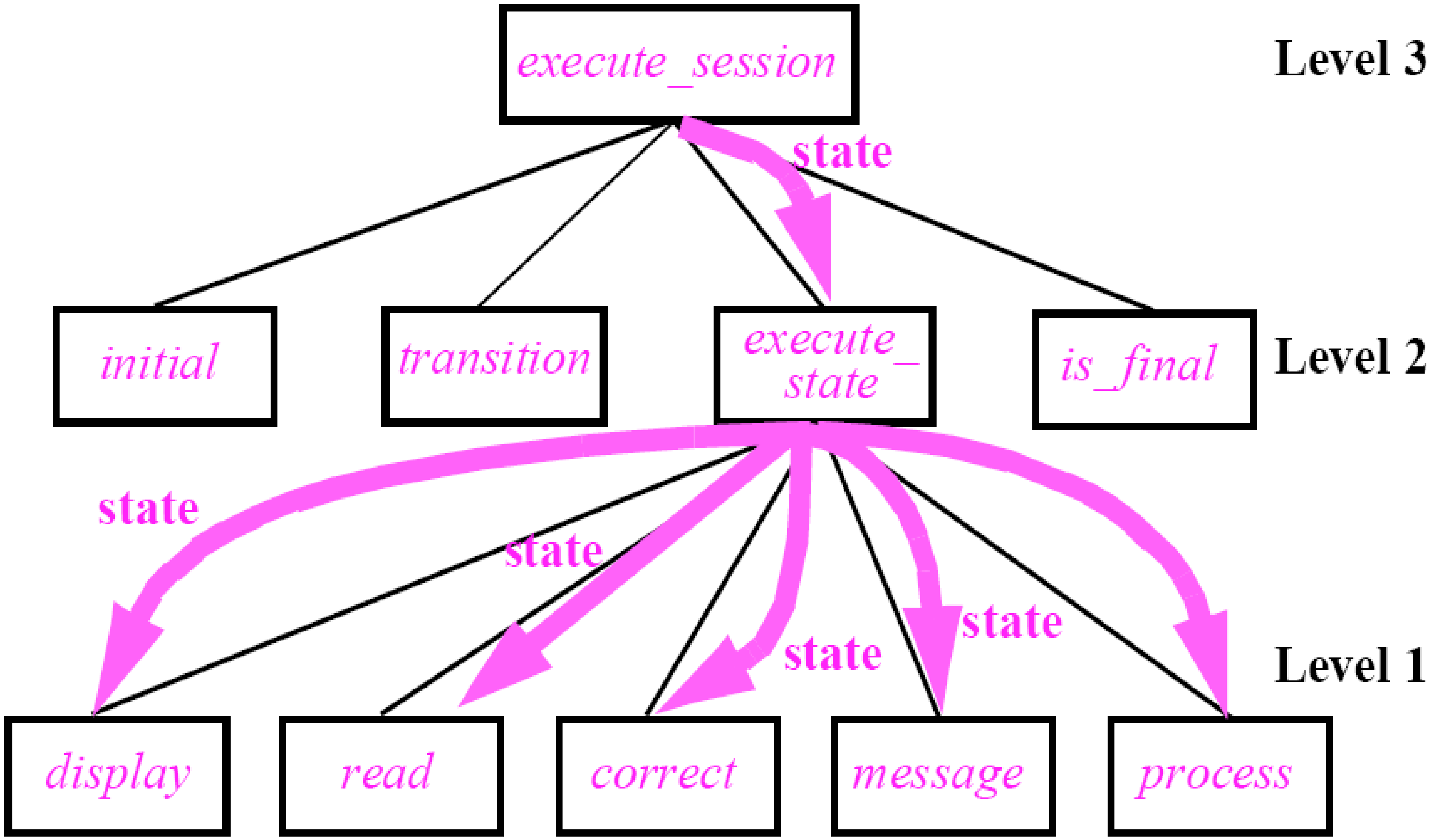
display (**in** *s*: *STATE*)

read (**in** *s*: *STATE*; **out** *a*: *ANSWER*)

correct (**in** *s*: *STATE*; *a*: *ANSWER*): *BOOLEAN*

message (**in** *s*: *STATE*; *a*: *ANSWER*)

process (**in** *s*: *STATE*; *a*: *ANSWER*)


execute_session
Level 3
state
initial
transition
execute_state
is_final
Level 2
state
state
state
state
state
Level 1
display
read
correct
message
process

Fragileness

```
inspect
        s
when Initial then
        ...
when Enquiry_on_flights then
        ...
...
end
```

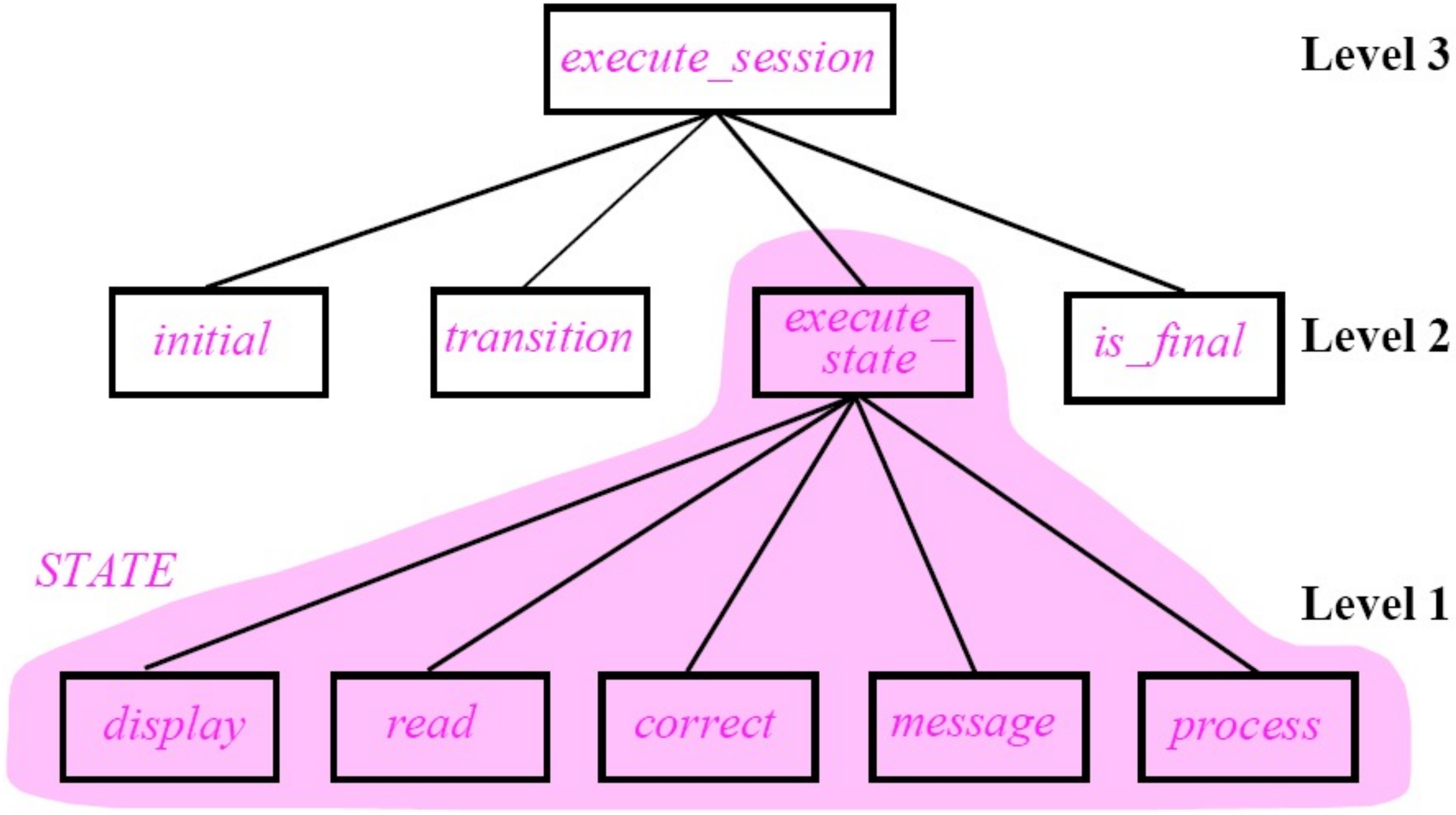
增加状态怎样？

如何在不同应用间复用?

object-oriented architecture

- **Law of inversion**
  - **If your routines exchange too many data, put your routines in your data.**

State as a class



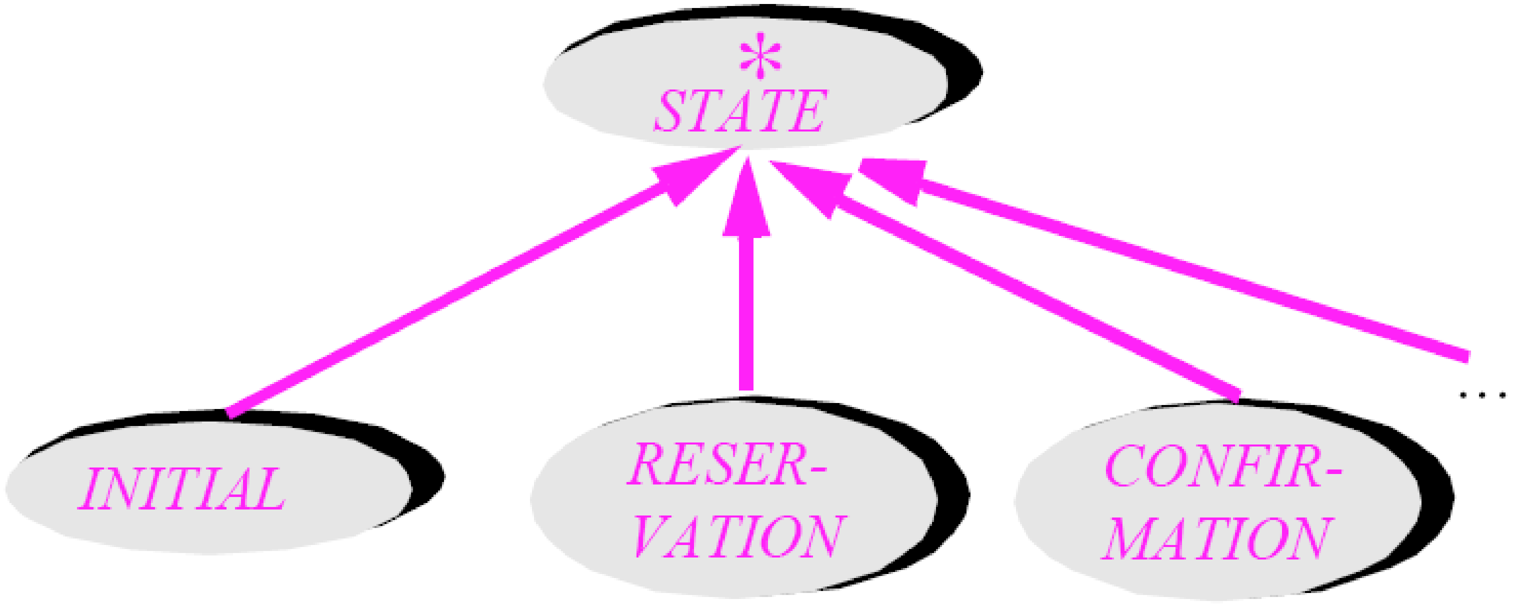
```
… class STATE feature
        input: ANSWER
        choice: INTEGER
        execute is do … end
        display is …
        read is …
        correct: BOOLEAN is …
        message is …
        process is …
end
```

Use inheritance and deferred class

```
… class STATE feature
        input: ANSWER
        choice: INTEGER
⇨       execute is do … end
⇨       display is …
⇨       read is …
⇨       correct: BOOLEAN is …
⇨       message is …
⇨       process is …
  end
```

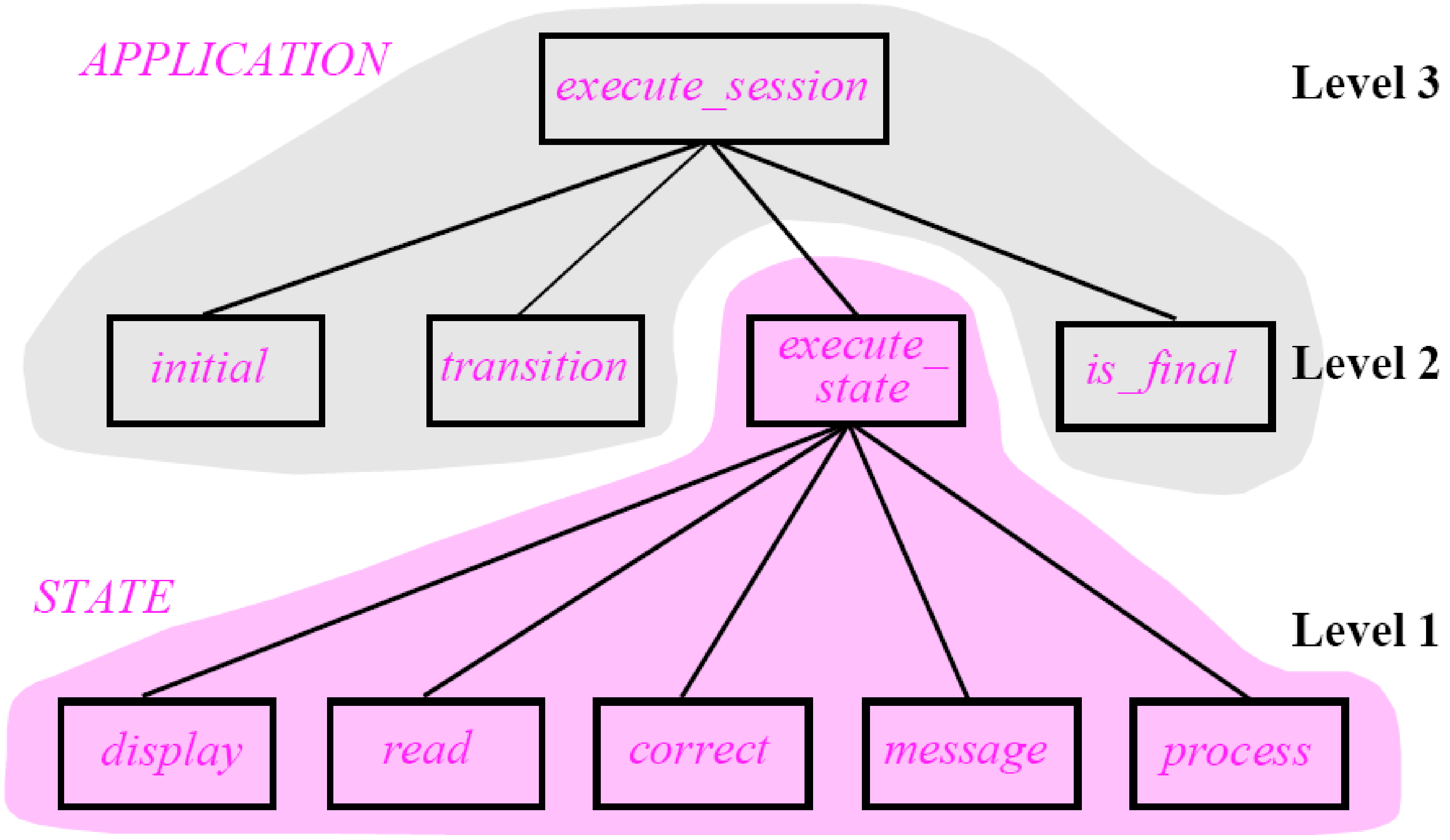
```
indexing
        description: "States for interactive panel-driven applications"
deferred class
        STATE
feature -- Access
        choice: INTEGER
                -- User's choice for next step
        input: ANSWER
                -- User's answer to questions asked in this state.
feature -- Status report
        correct: BOOLEAN is
                        -- Is input a correct answer?
                deferred
                end
feature -- Basic operations
        display is
                        -- Display panel associated with current state.
                deferred
                end
```

```
execute is
        -- Execute actions associated with current state
        -- and set choice to denote user's choice for next state.
    local
        ok: BOOLEAN
    do
        from ok := False until ok loop
            display; read; ok := correct
            if not ok then message end
        end
        process
    ensure
        ok
    end
    message is
            -- Output error message corresponding to input.
        require
            not correct
        deferred
        end
    read is
            -- Obtain user's answer into input and choice into next_choice.
        deferred
        end
    process is
            -- Process input.
        require
            correct
        deferred
        end
end -- class STATE
```


*
STATE
INITIAL
RESER-
VATION
CONFIR-
MATION
...

```
class ENQUIRY_ON_FLIGHTS inherit
        STATE
feature
        display is
                do
                        … Specific display procedure …
                end
        … And similarly for read, correct, message and process …
end -- class ENQUIRY_ON_FLIGHTS
```

The system? – An ADT, not a "main" function



讨论

- Focus on data abstraction
  - "Forget" the "main" function of the system, resist the constant temptation to ask "What does the system do?"
- Law of inversion
- Realworldliness is not a significant difference between OO and other approaches; what counts is how we model the world