体系结构模式

2.4 交互式系统

- 主要问题
  - 保持功能内核独立于UI
    - 内核基于系统功能需求，通常保持稳定
    - UI常常要变化或者改建

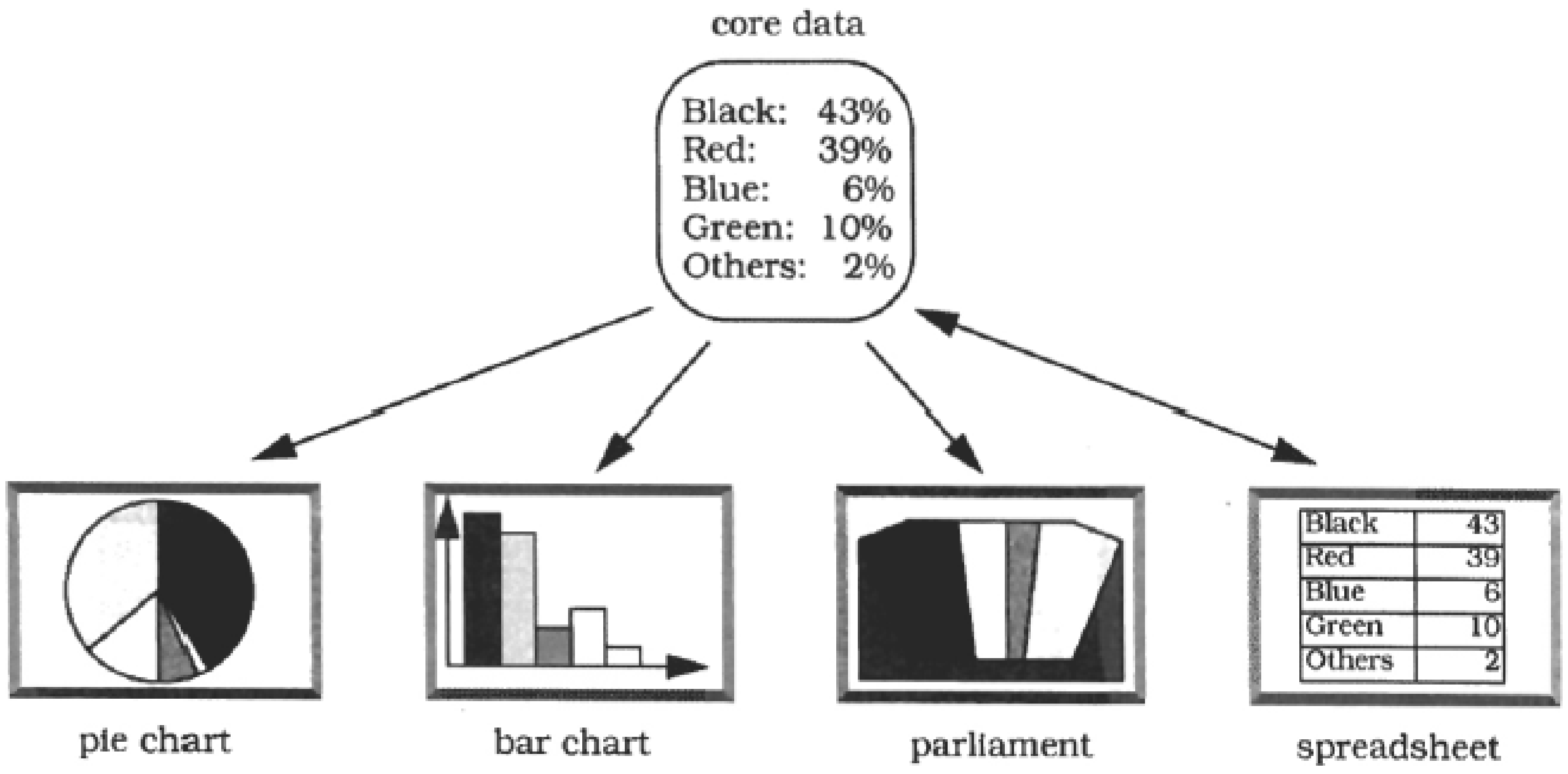
2.4 交互式系统

- 模型-视图-控制器(***Model-View-Controller*** , MVC) 模式将一个交互式应用程序分为是三个组件
  - 模型包含核心功能和数据
  - 视图向用户显示信息
  - 控制器处理用户的输入
  - 视图和控制器共同构成了用户界面。用变更传播机制确保用户界面和模型之间的一致性
- 表示-抽象-控制(***Presentation-Abstraction-Control*** , PAC)模式以合作Agent的层次形式定义了交互式软件的一种结构。每个Agent负责应用程序功能的某一特定方面，由…构成：
  - 表示, 抽象, 控制. 这种划分将…和…分隔开来
    - Agent的人机交互部分
    - Agent的功能内核，Agent和其他Agent的通信

2.4.1 模型-视图-控制器

- The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components.
  - The model contains the core functionality and data.
  - Views display information to the user.
  - Controllers handle user input.
- Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

1. 例子


core data
Black: 43%
Red: 39%
Blue: 6%
Green: 10%
Others: 2%
Black 43
Red 39
Blue 6
Green 10
Others 2
pie chart
bar chart
parliament
spreadsheet

2. 语境

- 具有灵活人-机接口的交互式应用程序
- *Interactive applications with a flexible human-computer interface*

3. 问题

- 用户界面的需求容易发生变化
  - 扩展新功能时，需要修改菜单
  - 系统移植到具有不同“look and feel”标准的另一平台上
  - …
- 不同用户对用户界面提出相互冲突的要求

3. 问题

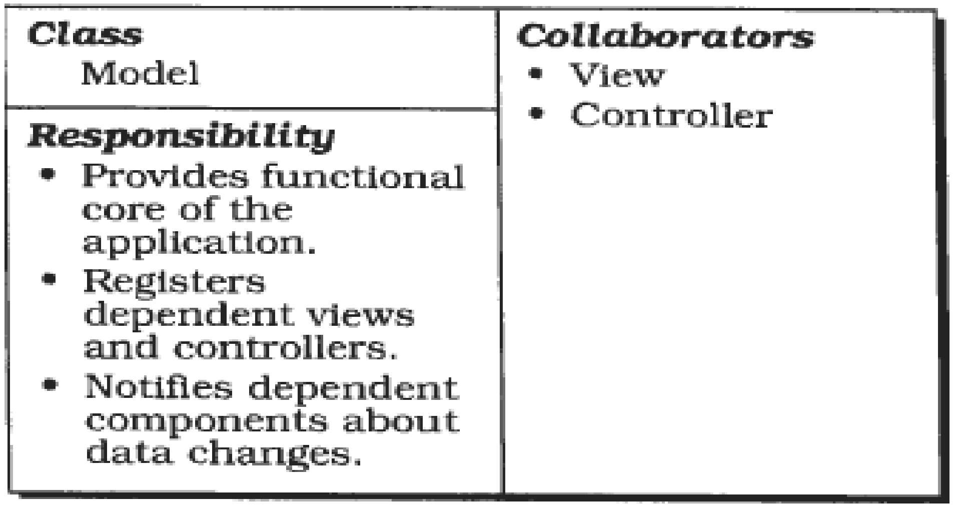
- 需要平衡下列条件：
  - 相同的信息在不同的窗口中有不同的表示，例如直方图和饼图
  - 应用程序的显示和行为必须立即反映出对数据的操作
  - 用户界面应易于改变，甚至在运行期间也应该可以修改
  - 支持不同视感标准或者移植用户界面不应影响应用程序内核的代码

4. 解决方案

- MVC divides an interactive application into the three areas: ***processing, output,*** and ***input.***
- *The **model** component encapsulates core data and functionality.* The model is independent of specific output representations or input behavior.
- ***View** components display information to the user.* A view obtains the data from the model. There can be multiple views of the model.
- *Each view has an associated controller component.* Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.

5. 结构

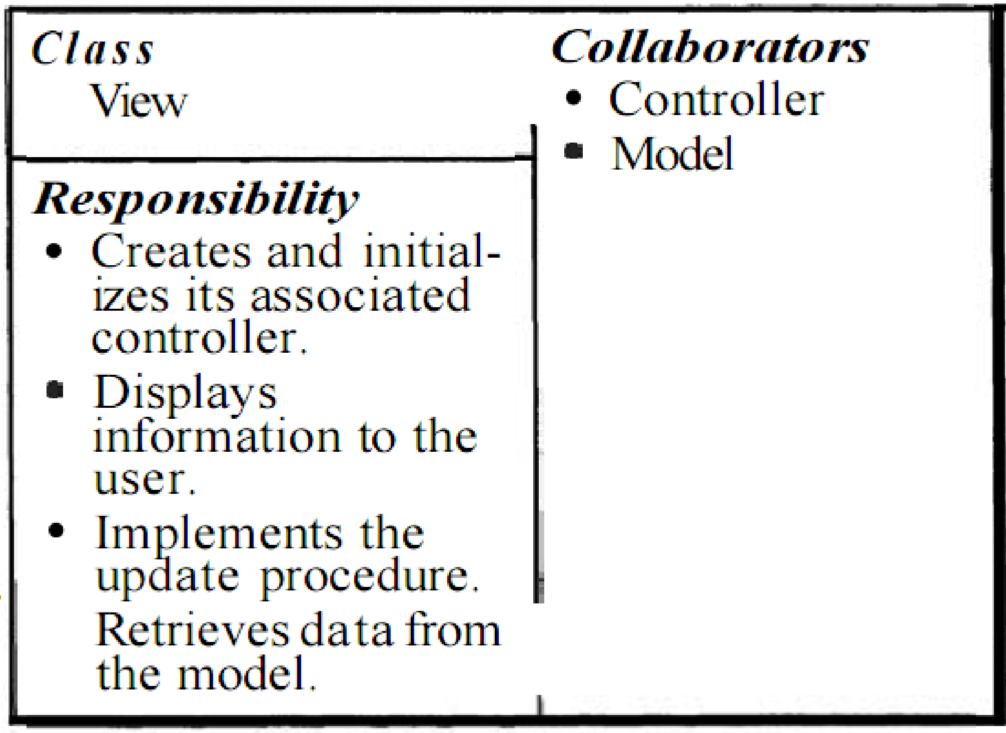
- 模型：应用程序的功能内核
  - 封装了对内部数据的处理过程
    - 这些过程由控制器组件调用
  - 提供访问内部数据的操作
    - 由视图组件调用
  - 变更-传播机制



Class Model	***Collaborators***
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	• View • Controller

5. 结构

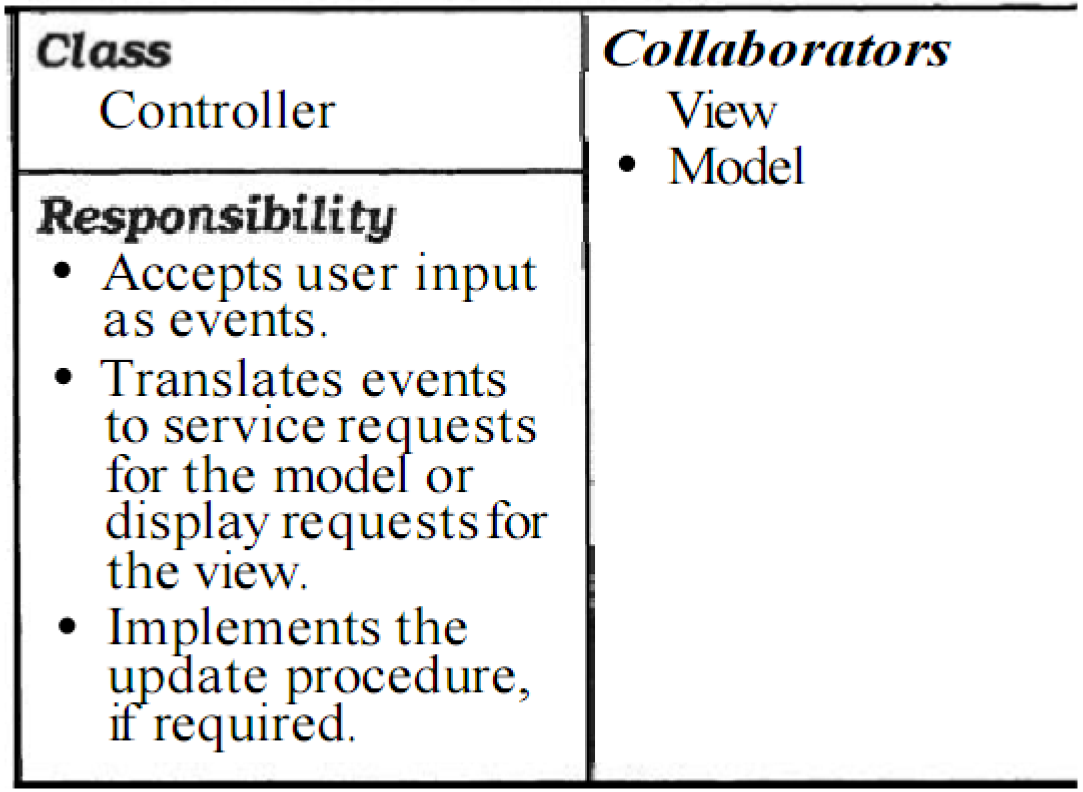
- 视图：向用户呈现信息
  - update操作
    - 从模型组件获取数据，并将其显示在屏幕上
    - 被变更-传播机制调用



Class View	***Collaborators***
Responsibility • Creates and initial-izes its associated controller. • Displays information to the user. • Implements the update procedure. Retrieves data from the model.	• Controller • Model

5. 结构

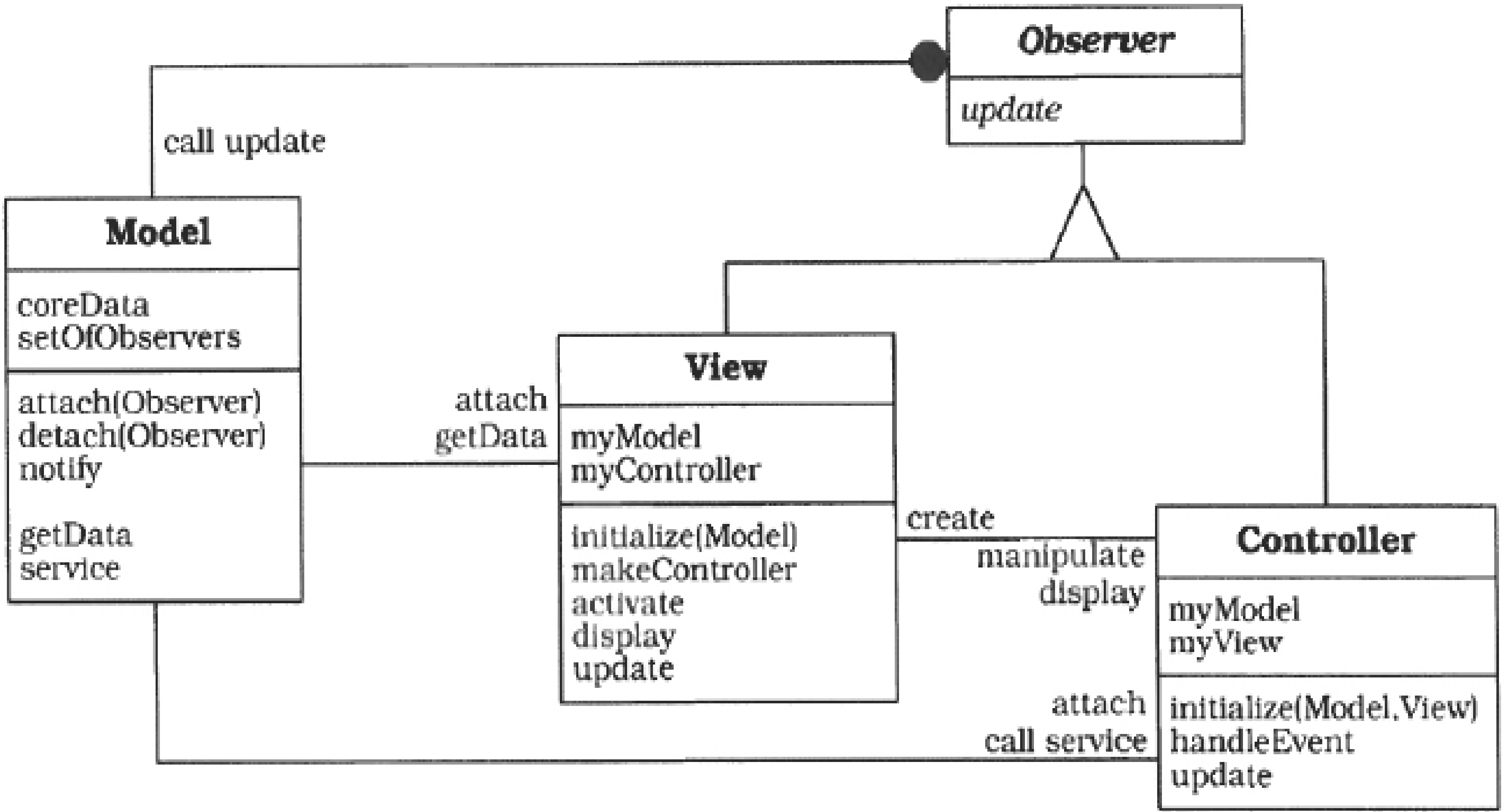
- 控制器：以事件的方式接受用户输入



Class Controller	***Collaborators***
Responsibility	View
• Accepts user input as events.	• Model
• Translates events to service requests for the model or display requests for the view.	
• Implements the update procedure, if required.	

5. 结构

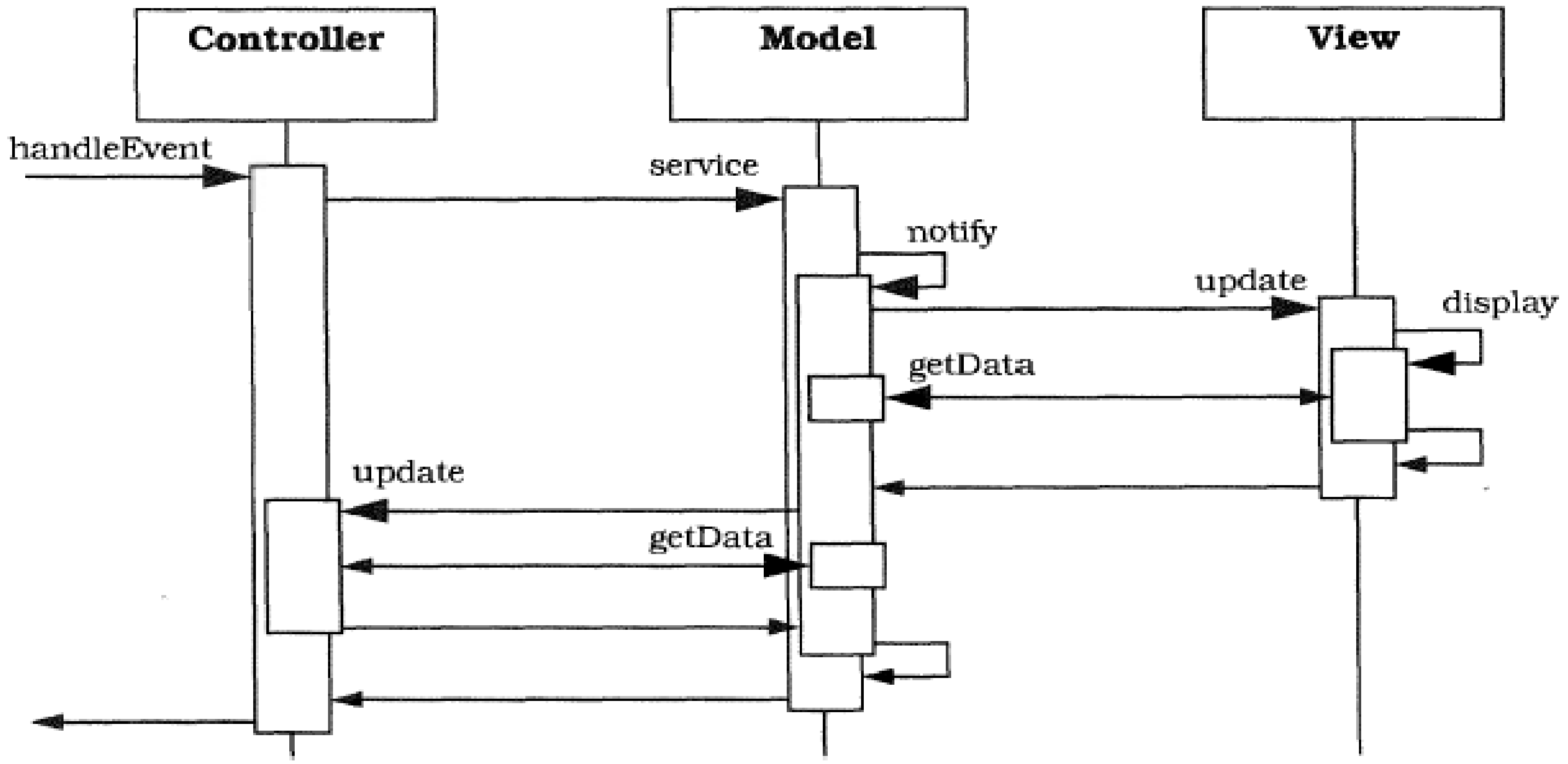
- 变更-传播机制示例



6. 动态特性——场景I 用户输入的处理

- The controller accepts user input in its event-handling procedure, interprets the event, and activates a service procedure of the model.
- The model performs the requested service. This results in a change to its internal data.
- The model notifies all views and controllers registered with the change-propagation mechanism of the change by calling their update procedures.
- Each view requests the changed data from the model and redisplays itself on the screen.
- Each registered controller retrieves data from the model to enable or disable certain user functions. For example, enabling the menu entry for saving data can be a consequence of modifications to the data of the model.
- The original controller regains control and returns from its event-handling procedure.

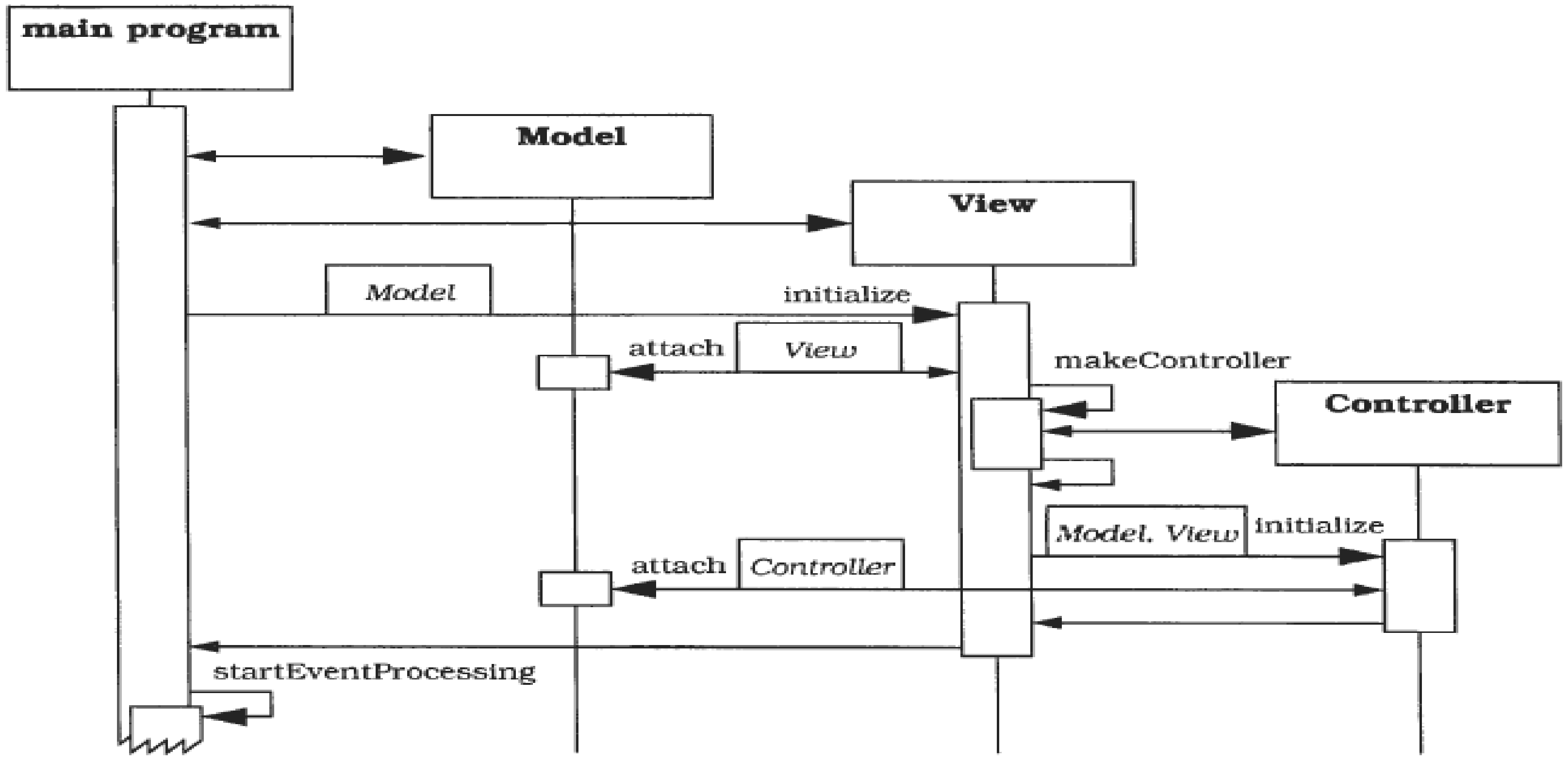
6. 动态特性——场景I


Controller
Model
View
handleEvent
service
notify
update
display
getData
update
getData

6. 动态特性——场景II 初始化过程

- The model instance is created, which then initializes its internal data structures.
- **A** view object is created. This takes a reference to the model as a parameter for its initialization.
- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.
- After initialization, the application begins to process events.

6. 动态特性——场景II


main program
Model
View
Model
initialize
attach
View
makeController
Controller
Model, View
initialize
attach
Controller
startEventProcessing

7. 实现

1）人机交互从核心功能中分离出来

Separate human-computer interaction from core functionality.

```
class Model{
     List<long>   votes;
     List<String> parties;
public:
     Model(List<String> partyNames);

// access interface for modification by controller
     void clearVotes(); // set voting values to 0
     void changeVote(String party, long vote);

// factory functions for view access to data
     Iterator<long>   makeVoteIterator(){
          return Iterator<long>(votes);
     }
     Iterator<String> makePartyIterator(){
          return Iterator<String>(parties);
     }
// ... to be continued
}
```

7. 实现

2） 实现变更-传播机制

```
class Observer{ // common ancestor for view and controller
public:
    virtual void update() { }
// default is no-op
};

class Model{
// ... continued
public:
    void attach(Observer *s) { registry.add(s); }
    void detach(Observer *s) { registry.remove(s); }
protected:
    virtual void notify();
private:
    Set<Observer*> registry;
};
```

```
void Model::notify(){
    // call update for all observers
    Iterator<Observer*> iter(registry);
    while (iter.next()){
        iter.curr()->update();
    }
}
```

7. 实现

3）设计并实现视图

- 设计每个视图的外观
- 实现一个画图过程
  - 从模型取得数据，然后以某种方式显示到屏幕上
- 实现更新过程
  - 最简单方式：直接调用画图过程
  - 频繁更新情况下的优化
    - 更新过程提供额外参数，据此判断是否需要重新绘制
    - 其他事件也要求重绘时，不执行重绘。事件都处理完毕了，再重绘
- 初始化过程
  - 支持模型的变更-传播机制（attach）
  - 建立与控制器将的连接

7. 实现

3）设计并实现视图

```
class View : public Observer {
public:
    View(Model *m) : myModel(m), myController(0)
        { myModel->attach(this); }
    virtual ~View() { myModel->detach(this); }
    virtual void update() { this->draw(); }
    // abstract interface to be redefined:
    virtual void initialize() ;// see below
    virtual void draw() ;      // (re-)display view
// ... to be continued below
   Model *getModel() { return myModel; }
   Controller *getController() { return myController; }
protected:
   Model        *myModel;
   Controller   *myController; // set by initialize
};

class BarChartView : public View {
public:
    BarChartView(Model *m) : View(m) { }
    virtual void draw();
};
```

```
void BarChartView::draw(){
    Iterator<String> ip = myModel->makePartyIterator();
    Iterator<long> iv = myModel->makeVoteIterator();
    List<long> dl; //for scaling values to fill screen
    long     max = 1;// maximum for adjustment

    // calculate maximum vote count
    while (iv.next()) {
        if (iv.curr() > max ) max = iv.curr();
    }
    iv.reset();
    // now calculate screen coordinates for bars
    while (iv.next()) {
        dl.append((MAXBARSIZE * iv.curr())/max);
    }

    // reuse iterator object for new collection:
    iv = dl; // assignment rebinds iterator to new list
    iv.reset();

    while (ip.next() && iv.next()) {
        // draw text: cout << ip.curr() << " : " ;
        // draw bar: ... drawbox(BARWIDTH, iv.curr());...
    }
}
```

7. 实现

4）设计并实现控制器

- 对应用程序的每个视图，指定回应用户动作的系统的行为
    - 如果基础平台将用户的每个动作作为一个事件来发送，控制器用一个还专门过程来接受并解释这些事件
- 初始化
    - 将控制器绑定到它的模型和视图，并使之能进行事件处理

7. 实现

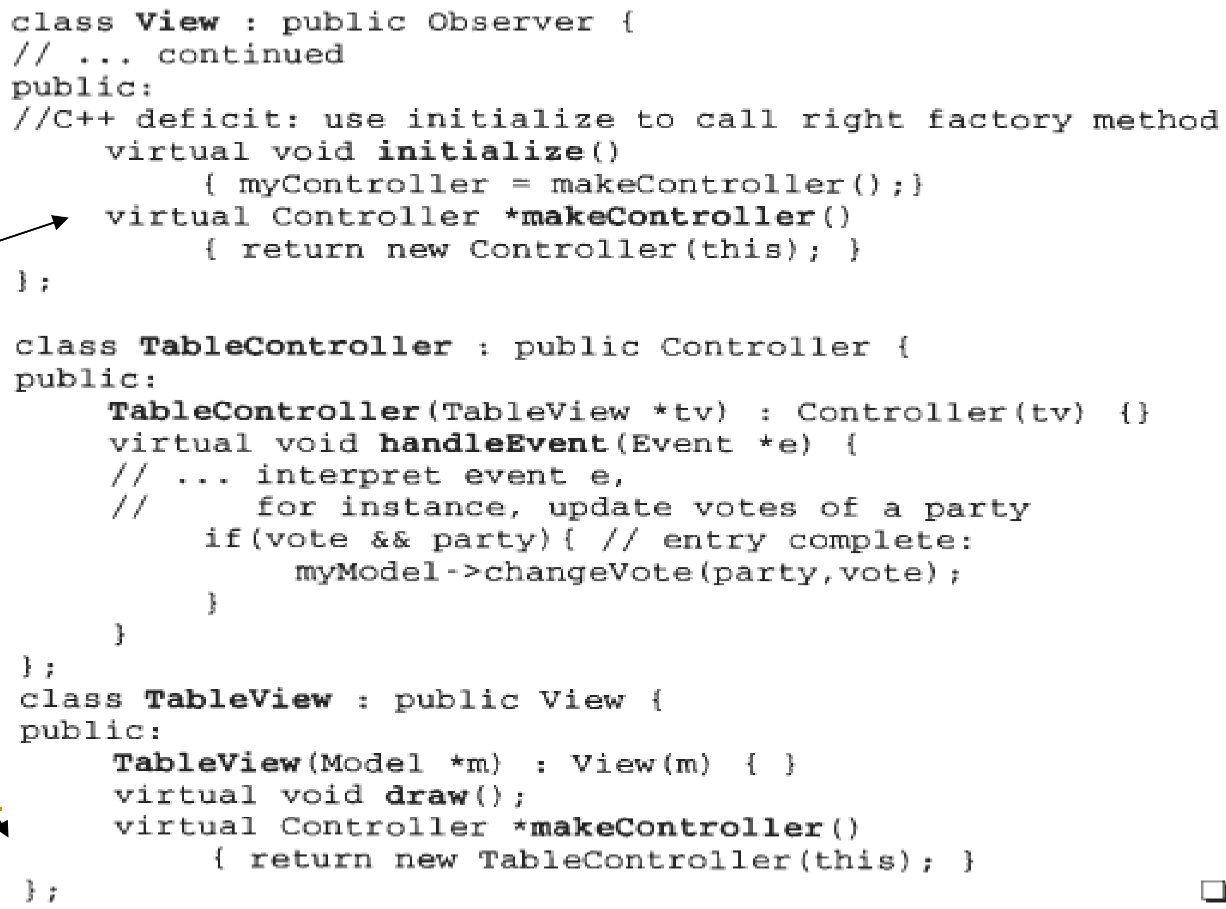
4）设计并实现控制器

```
class Controller : public Observer {
public:
     virtual void handleEvent(Event *) { }
          // default = no op

     Controller( View *v) : myView(v) {
          myModel = myView->getModel();
          myModel->attach(this);
     }

     virtual ~Controller() { myModel->detach(this); }
     virtual void update() { } // default = no op
protected:
     Model     *myModel;
     View      *myView;
};
```

7. 实现

5）设计并实现视图-控制器关系

工厂方法模式



```
class View : public Observer {
// ... continued
public:
//C++ deficit: use initialize to call right factory method
     virtual void initialize()
          { myController = makeController();}
     virtual Controller *makeController()
          { return new Controller(this); }
};

class TableController : public Controller {
public:
     TableController(TableView *tv) : Controller(tv) {}
     virtual void handleEvent(Event *e) {
     // ... interpret event e,
     //     for instance, update votes of a party
          if(vote && party){ // entry complete:
               myModel->changeVote(party,vote);
          }
     }
};
class TableView : public View {
public:
     TableView(Model *m) : View(m) { }
     virtual void draw();
     virtual Controller *makeController()
          { return new TableController(this); }
};
```

7. 实现

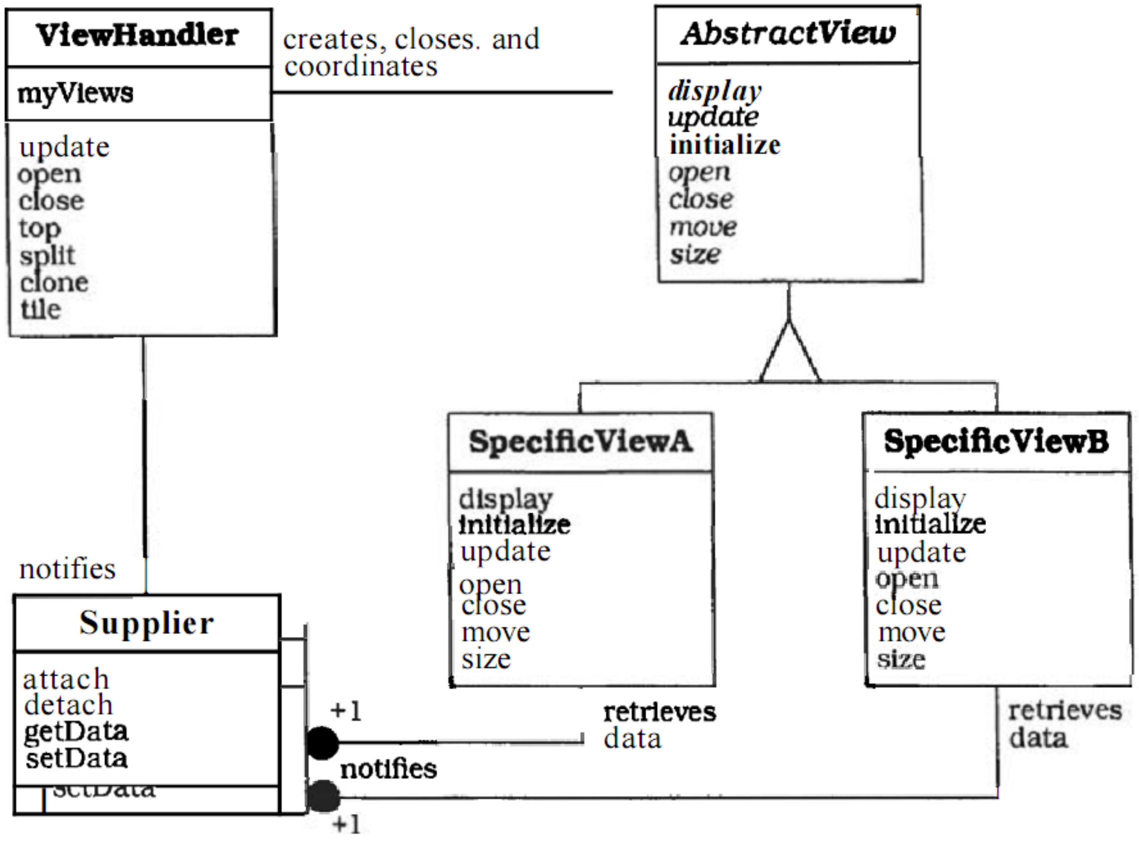
6） ***Implement the set-up of MVC***

```
main() {
     // initialize model
     List<String> parties;    parties.append("black");
     parties.append("blue "); parties.append("red  ");
     parties.append("green"); parties.append("oth. ");
     Model m(parties);

     // initialize views
     TableView *v1 = new TableView(&m);
     v1->initialize();
     BarChartView *v2 = new BarChartView(&m);
     v2->initialize();
     // now start event processing ...
```

7. 实现

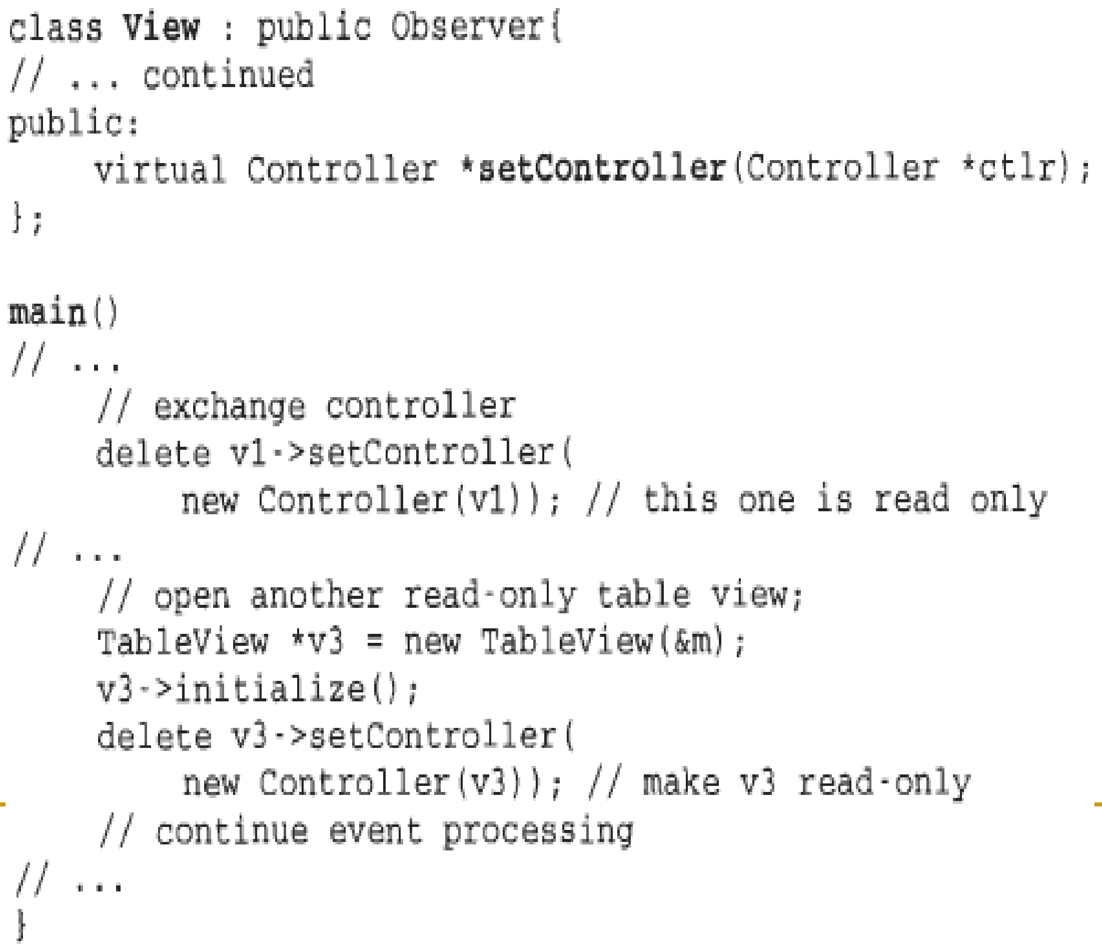
7） 创建动态视图

- 视图管理组件
  - 管理打开视图
  - 可以在最后一个视图被关闭时终止应用程序
  - 可用view handler模式



7. 实现

8） ‘***Pluggable’ controllers***



```
class View : public Observer{
// ... continued
public:
    virtual Controller *setController(Controller *ctlr);
};

main()
// ...
    // exchange controller
    delete v1->setController(
        new Controller(v1)); // this one is read only
// ...
    // open another read-only table view;
    TableView *v3 = new TableView(&m);
    v3->initialize();
    delete v3->setController(
        new Controller(v3)); // make v3 read-only
    // continue event processing
// ...
}
```

7. 实现

9）层次化视图和控制器的基础结构

- 组合模式 Composite
- 责任链模式 Chain of responsibility

10）进一步去除系统依赖性

- 桥接模式 Bridge

9. 变体

- 文档-视图体系结构
  - Document-View

10. 已知应用

- **Smalltalk**
- **MFC**
- **ET++**

11.效果

- 优点
  - ***Multiple views of the same model.***
  - ***Synchronized views.***
  - ***'Pluggable' views and controllers.***
  - ***Exchangeability of 'look and feel'.***
  - ***Framework potential.***

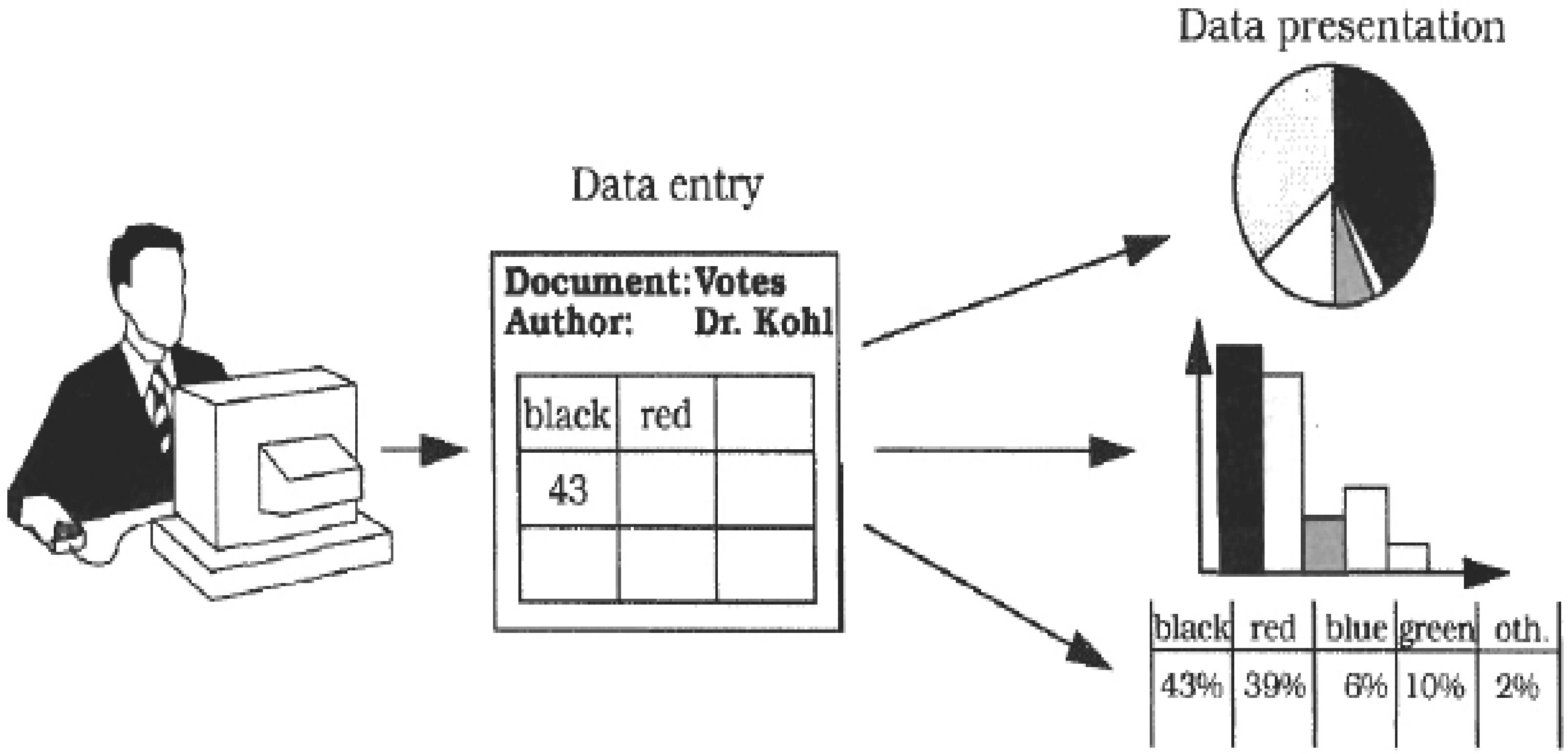
11.效果

- 不足
  - Increased complexity.
  - Potential for excessive number of updates.
  - Intimate connection between view and controller.
  - Close coupling of views and controllers to a model.
  - **Inefficiency *of data access in view.***
  - ***Inevitability of change to view and controller when porting.***
  - ***Difficulty of* using *MVC with modern user-interface took.***

2.4.2 表示-抽象-控制

- The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

1. 例子


Data entry
Document: Votes
Author: Dr. Kohl
black
red
43
Data presentation
black
red
blue
green
oth.
43%
39%
6%
10%
2%

2. 语境

- 在Agent的协助下开发一个交互式应用程序
- Development of an interactive application with the help of agents.

3. 问题

- 交互式系统经常被视为一个协作Agent的集合
- 需要平衡下列条件：
  - Agents often maintain their own state and data.
  - Interactive agents provide their own user interface, since their respective human-computer interactions often differ widely.
  - Systems evolve over time.

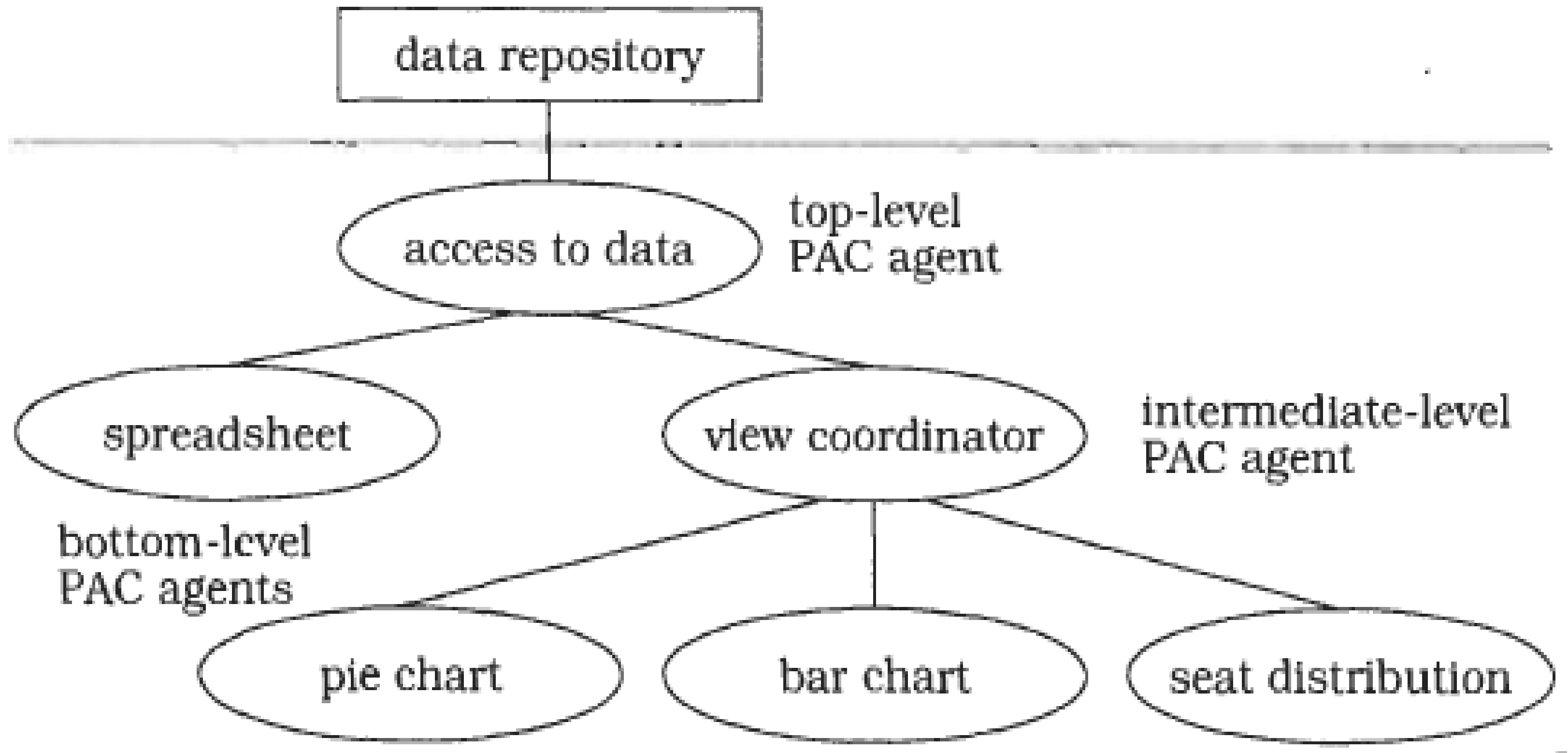
4. 解决方案

- 以PAC Agent树状层次结构构建交互式应用程序
  - 每个Agent由…构成
    - 表示组件：提供了PAC Agent的可视行为
    - 抽象组件：维护了构成Agent基础的数据模型，提供对这些数据进行操作的功能
    - 控制组件：连接表示组件和抽象组件，提供本Agent和其他Agent的通信

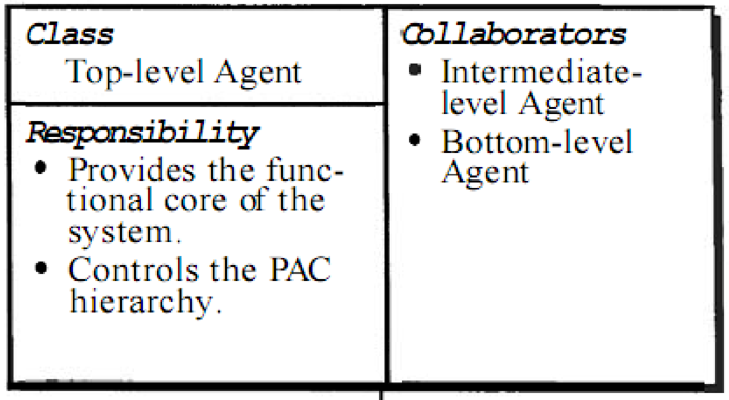
4. 解决方案

- 顶层Agent
  - 提供系统的核心功能
  - 顶层Agent包括不能分配给某一特定任务的用户界面
- 底层Agent
  - 表达了独立语义概念，系统的用户可以基于这些概念进行操作
  - E.g 电子数据表和图表
- 中间Agent
  - 表达低层Agent的组合，或者低层Agent之间的关系

4. 解决方案


data repository
access to data
top-level PAC agent
spreadsheet
view coordinator
intermediate-level PAC agent
bottom-level PAC agents
pie chart
bar chart
seat distribution

5. 结构



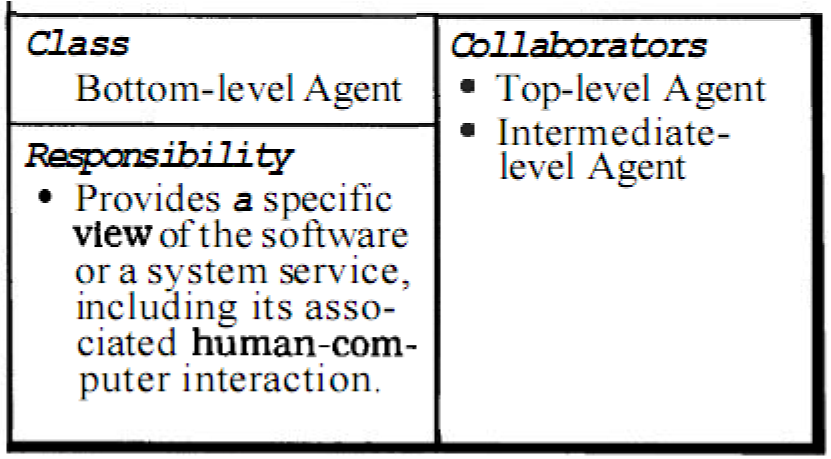
Class Top-level Agent	Collaborators
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	• Intermediate-level Agent • Bottom-level Agent

- 顶层Agent
  - 抽象组件的职责：提供软件的全局数据模型
  - 表示组件的职责：通常没有什么
  - 控制组件的职责
    - **It allows lower-level agents to make use of the services of the top-level agents, mostly to access and manipulate the global data model. Incoming service requests from lower-level agents are forwarded either to the abstraction component or the presentation component.**
    - **It coordinates the hierarchy of PAC agents. It maintains information about connections between the top-level agent and lower-level agents. The control component uses this information to ensure correct collaboration and data exchange between the top –level agent and lower-level agents.**
    - **It maintains information about the interaction of the user with the system. For example, it may check whether a particular operation can be performed on the data model when triggered by the user. It may also keep track of the functions called to provide history or undo/redo services for operations on the functional core**

5. 结构

- 政治选举信息系统的例子
  - 顶层Agent的抽象组件：
    - 提供了对底层数据仓库的特殊应用接口
  - 顶层Agent的控制组件：
    - 组织本Agent与低层Agent的通信和协作
  - 顶层Agent的表示组件：
    - 无

5. 结构



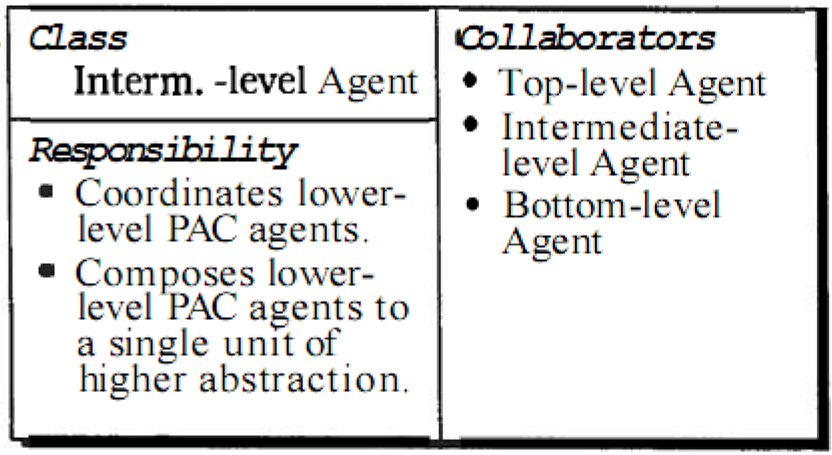
Class Bottom-level Agent	Collaborators • Top-level Agent • Intermediate-level Agent
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	

- 底层Agent：描绘了应用领域的一个具体语义概念
  - The presentation component of a bottom-level PAC agent presents a specific view of the corresponding semantic concept, and provides access to **all** the functions users can apply to it.
  - The abstraction component of a bottom-level PAC agent has a similar responsibility as the abstraction component of the top-level PAC agent, maintaining agent-specific data.
  - The control component of a bottom-level PAC agent maintains consistency between the abstraction and presentation components, thereby avoiding direct dependencies between them.
  - The control component of bottom-level PAC agents communicates with higher-level agents to exchange events and data.
-

5. 结构

- 政治选举信息系统的例子
  - 底层的直方图Agent
    - 抽象组件：
      - 存放图中的选举数据，维护特定图表信息，如数据的表示顺序
    - 表示组件：
      - 负责在窗口中显示直方图
      - 提供可以运行其上的所有功能，如放大、移动和打印等
    - 控制组件：
      - 表示组件和抽象组件之间的迂回层
      - 负责直方图Agent和视图协调程序Agent的通信

5. 结构

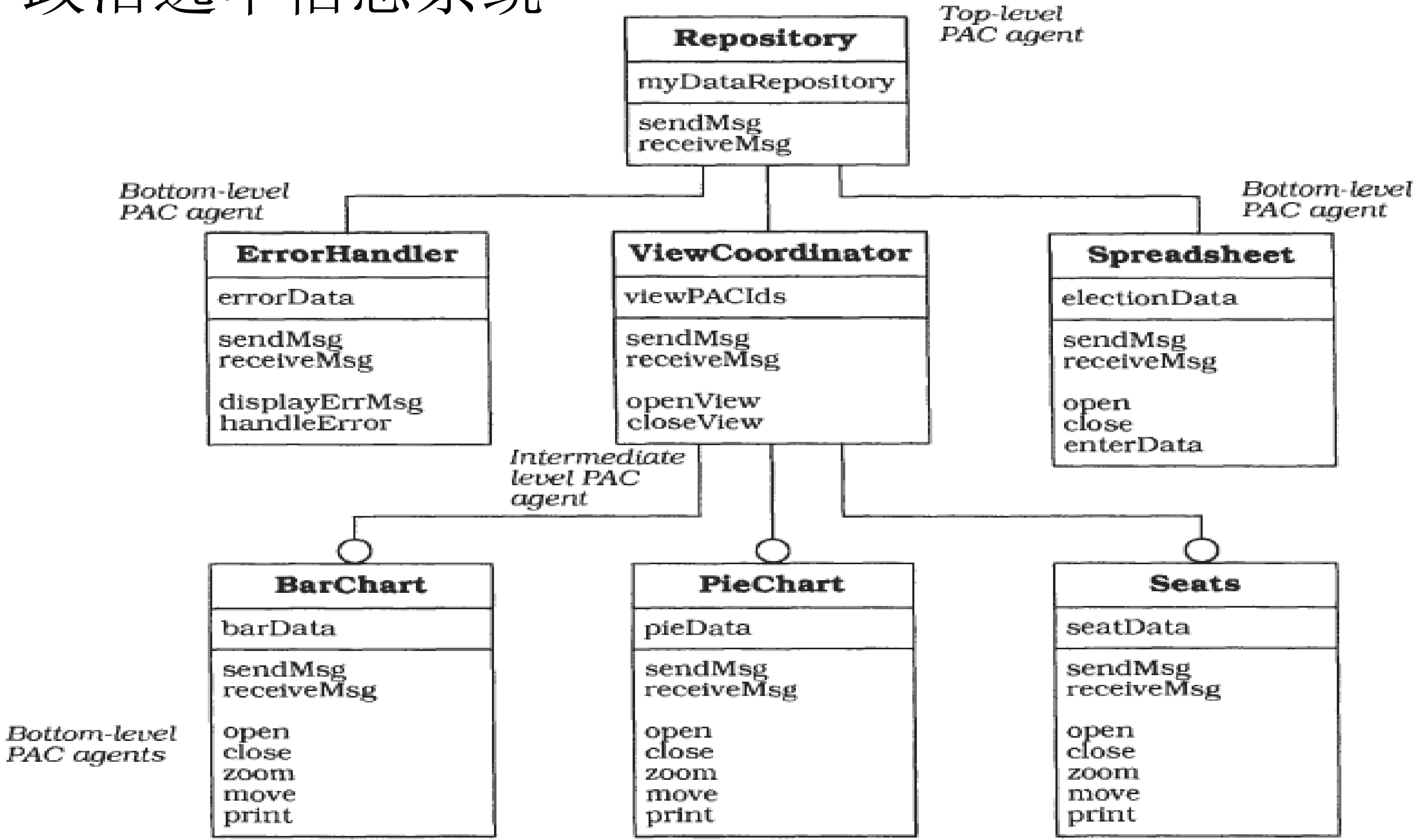


Class Interm. -level Agent	Collaborators
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent

- 中层Agent
  - 完成两种任务：合成与协作
  - 抽象组件：维护中层Agent中的特殊数据
  - 表示组件：实现了它的用户接口
  - 控制组件：具有和顶层Agent、底层Agent的控制组件相同的职责
- 政治选举信息系统的例子
  - 某个中层Agent
    - 提供一个允许用户产生选举数据视图的调色板

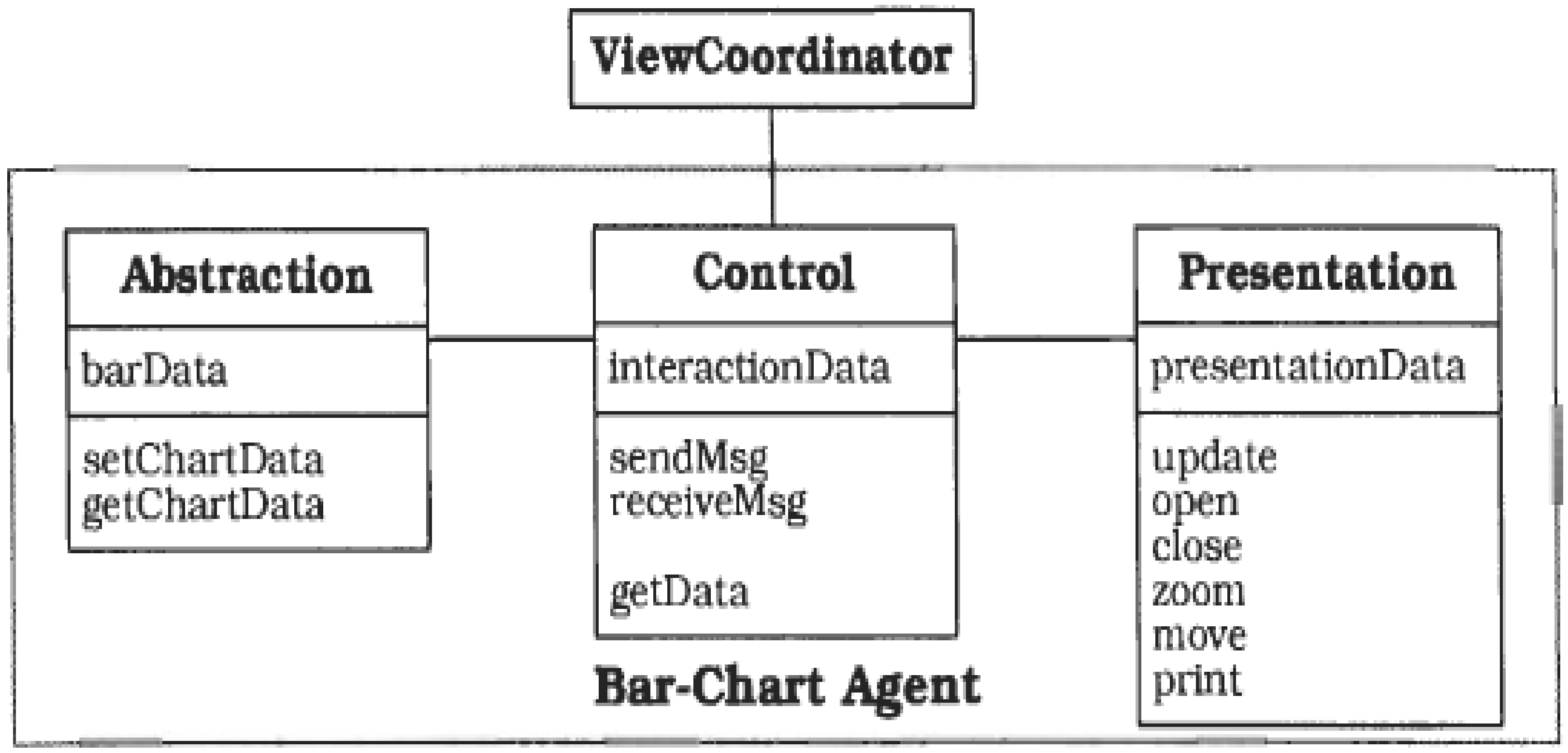
5. 结构

- 政治选举信息系统



5. 结构

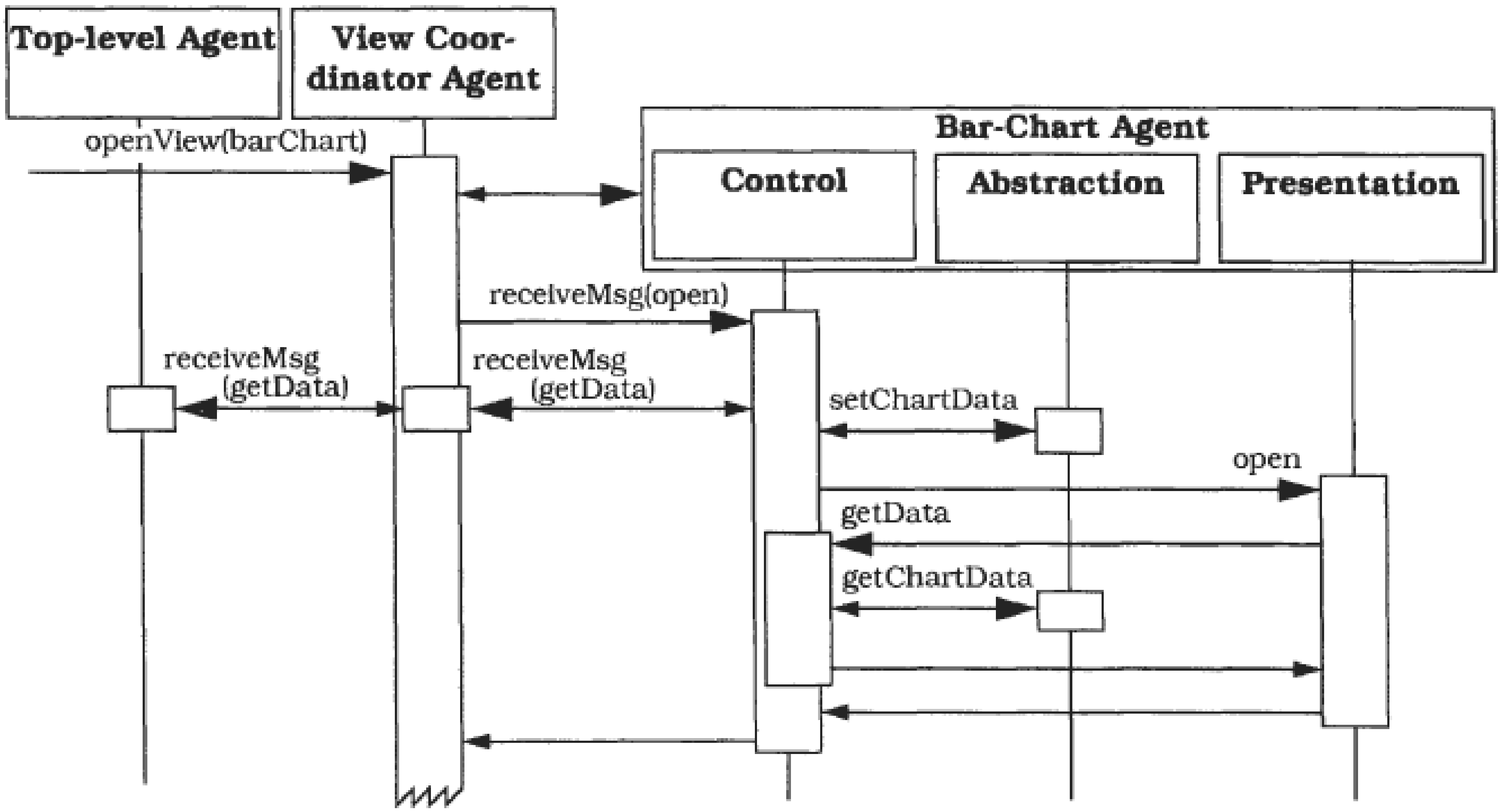
- 一个Agent的内部结构



6. 动态特性——场景I

- A user asks the presentation component of the view coordinator agent to open a new bar chart.
- The control of the view coordinator agent instantiates the desired bar-chart agent.
- The view coordinator agent sends an 'open' event to the control component of the new bar-chart agent.
- The control component of the bar-chart agent first retrieves data from the top-level PAC agent. The view coordinator agent mediates between bottom and top-level agents. The data returned to the bar-chart agent is saved in its abstraction component. Its control component then calls the presentation component to display the chart.
- The presentation component creates a new window on the screen, retrieves data from the abstraction component by requesting it from the control component, and finally displays it within the new window

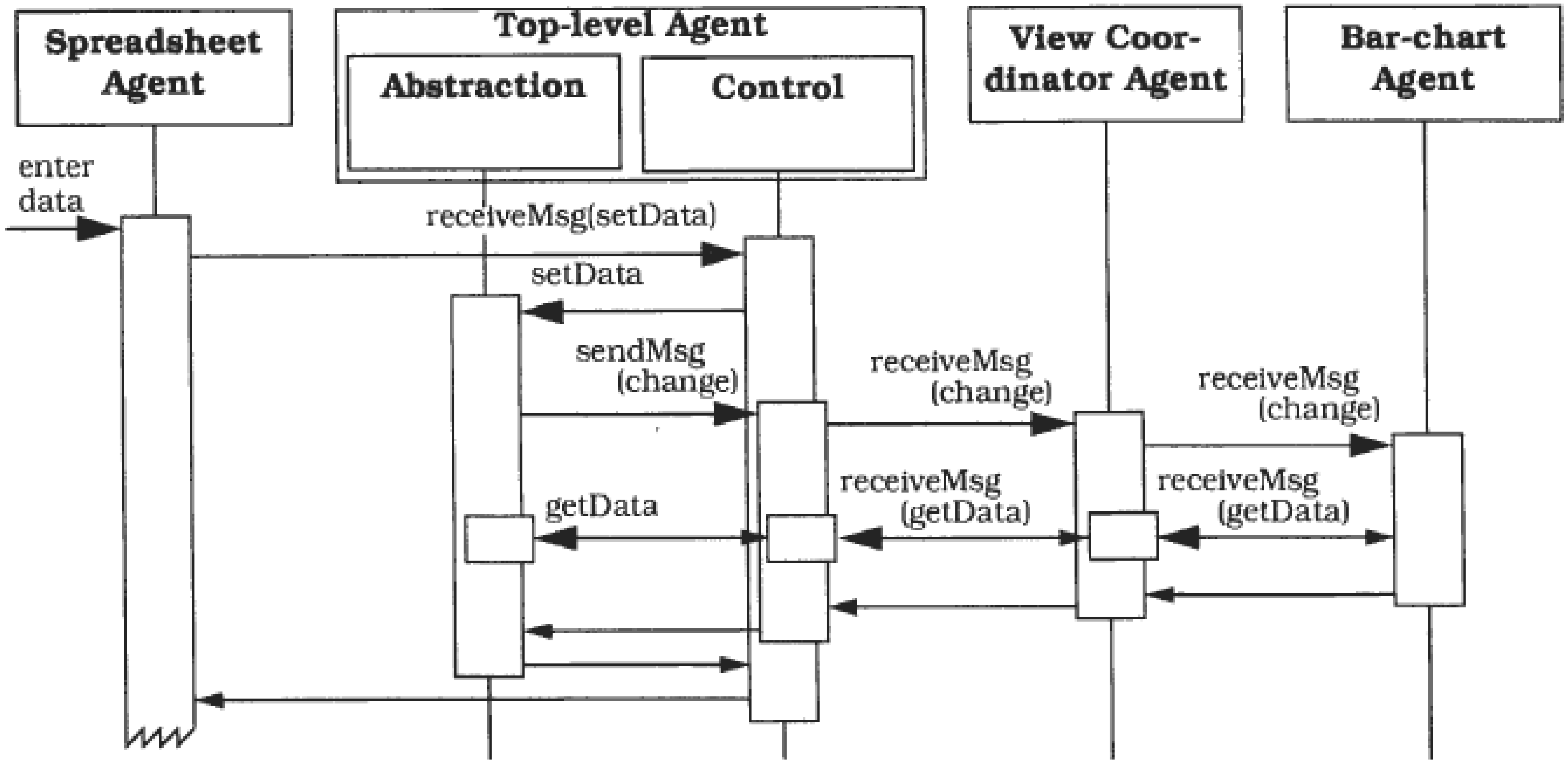
6. 动态特性——场景I


Top-level Agent
View Coordinator Agent
Bar-Chart Agent
Control
Abstraction
Presentation
openView(barChart)
receiveMsg(open)
receiveMsg (getData)
receiveMsg (getData)
setChartData
open
getData
getChartData

6. 动态特性——场景II

- The user enters new data into a spreadsheet. The control component of the spreadsheet agent forwards this data to the top-level PAC agent.
- The control component of the top-level PAC agent receives the data and tells the top-level abstraction to change the data repository accordingly. The abstraction component of the top-level agent asks its control component to update all agents that depend on the new data. The control component of the top-level PAC agent therefore notifies the view coordinator agent.
- The control component of the view coordinator agent forwards the change notification to all view PAC agents it is responsible for coordinating.
- **As** in the previous scenario, all view PAC agents then update their data and refresh the image they display.

6. 动态特性——场景II


Spreadsheet Agent
Top-level Agent
Abstraction
Control
View Coor-dinator Agent
Bar-chart Agent
enter data
receiveMsg(setData)
setData
sendMsg (change)
receiveMsg (change)
receiveMsg (change)
getData
receiveMsg (getData)
receiveMsg (getData)

7. 实现

1） *定义一个应用模型*

- 分析该问题领域，并将该问题映射到适当的软件结构
- 回答如下问题：
  - Which services should the system provide?
  - Which components can fulfill these services?
  - What are the relationships between components?
  - How do the components collaborate?
  - What data do the components operate on?
  - How will the user interact with the software?

7. 实现

2） *为组织PAC层次定义一般策略*

- 最低共同祖先规则
- 考虑层次的深度
  - 层次越深越能更好地反映一个应用程序到自包含概念的分解
  - 深的层次运行时往往效率较低且难以维护

7. 实现

3） *Specify A the top-level PAC agent***.** Identify those parts of the analysis model that represent the functional core of the system.

4） *Specify the bottom-level PAC agents.* Identify those components of the analysis model that represent the smallest self-contained units of the system on which the user can perform operations or view presentations.

5） *Specify bottom-level PAC agents for system services.* Often an application includes additional services that are not directly related to its primary subject.

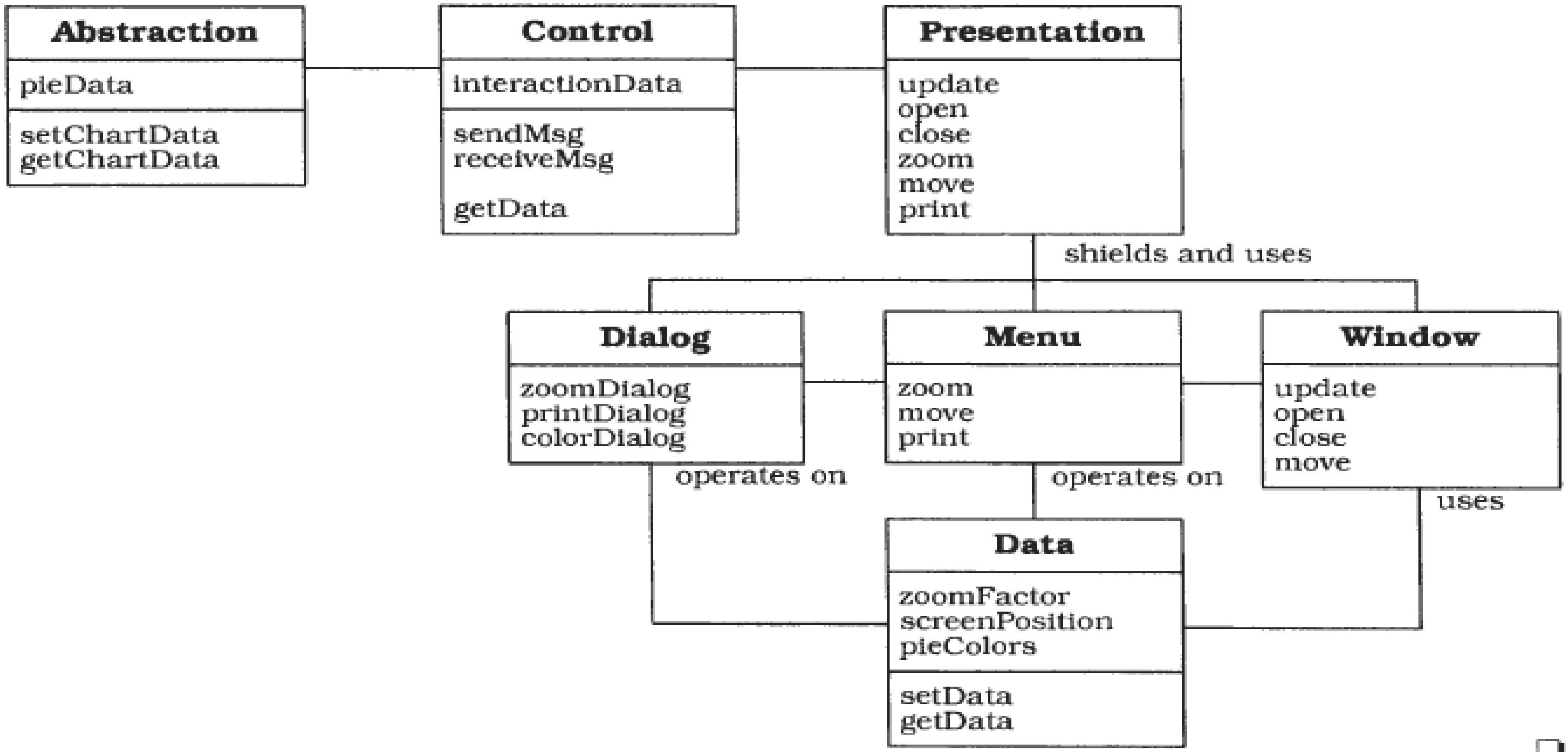
7. 实现

6）*Specify intermediate-level PAC agents to compose lower-level PAC agents.* Often, several lower-level agents together form a higher-level semantic concept on which users can operate.

7）*Specify intermediate-level PAC agents to coordinate lower-level PAC agents.* Many systems offer multiple views of the same semantic concept.

7. 实现

8） ***Separate core functionality from human-computer interaction.*** **For every PAC agent, introduce presentation and abstraction components**



7. 实现

9）*Provide the external interface*.

- One way of implementing communication with other agents is to apply the Composite Message pattern
- A second option is to provide a public interface that offers every service of an agent as a separate function.
- A PAC agent can be connected to other PAC agents in a flexible and dynamic way by using registration functionality, as introduced by the Publisher-Subscriber pattern.
- If a PAC agent depends on data or information maintained by other PAC agents, you should provide a change-propagation mechanism. Such a mechanism should involve all agents and all levels of the hierarchy and work in both directions.

7. 实现

```
enum ViewKind { barchart, pieChart, seats };
     // type of available views of election data
class DataSetInterface { /* ... */ };
     // Common interface for datasets, messages, and
     // events, according to the specifications of the
     // Composite Message pattern [SC95b]
class PACId { /* ... */ };
     // Provides a handle to a PAC agent
class VCControl {
     // Data member specifications
     PACId          parent:  // higher-level agent
     List<PACId>    children;// lower-level agents
     // More data member specifications ...
private:
     void attach(PACId agent, parentAgent = 0);
     void detach(PACId agent);
          // Registration functionality for connecting
          // dependent view agents and the top-level agent
          // with the view coordinator agent.
     DataSetInterface sendMsg(DataSetInterface data);
          // Sending events, messages, or data to other PAC
          // agents including change notifications
     void openView(ViewKind kind);
     void closeView(PACId agent);
          // Opening and closing views including
          // creation, registration,and deletion
          // of bottom-level agents displaying charts
public:
     DataSetInterface receiveMsg(DataSetInterface data);
          // Receiving events, messages, or data from other
          // PAC agents including change notifications
};
```

7. 实现

10）*Link the hierarchy together.*

- After implementing the individual PAC agents you can build the final PAC hierarchy. Connect every PAC agent with those lower-level PAC agents with which it directly cooperates.
- Provide the PAC agents that dynamically create and delete lower-level PAC agents with functionality to dynamically extend or reduce the PAC hierarchy. For example, the view coordinator agent in our information system creates a new view PAC agent if the user wants to open a particular view, and deletes this agent when the user closes the window in which the view is displayed.

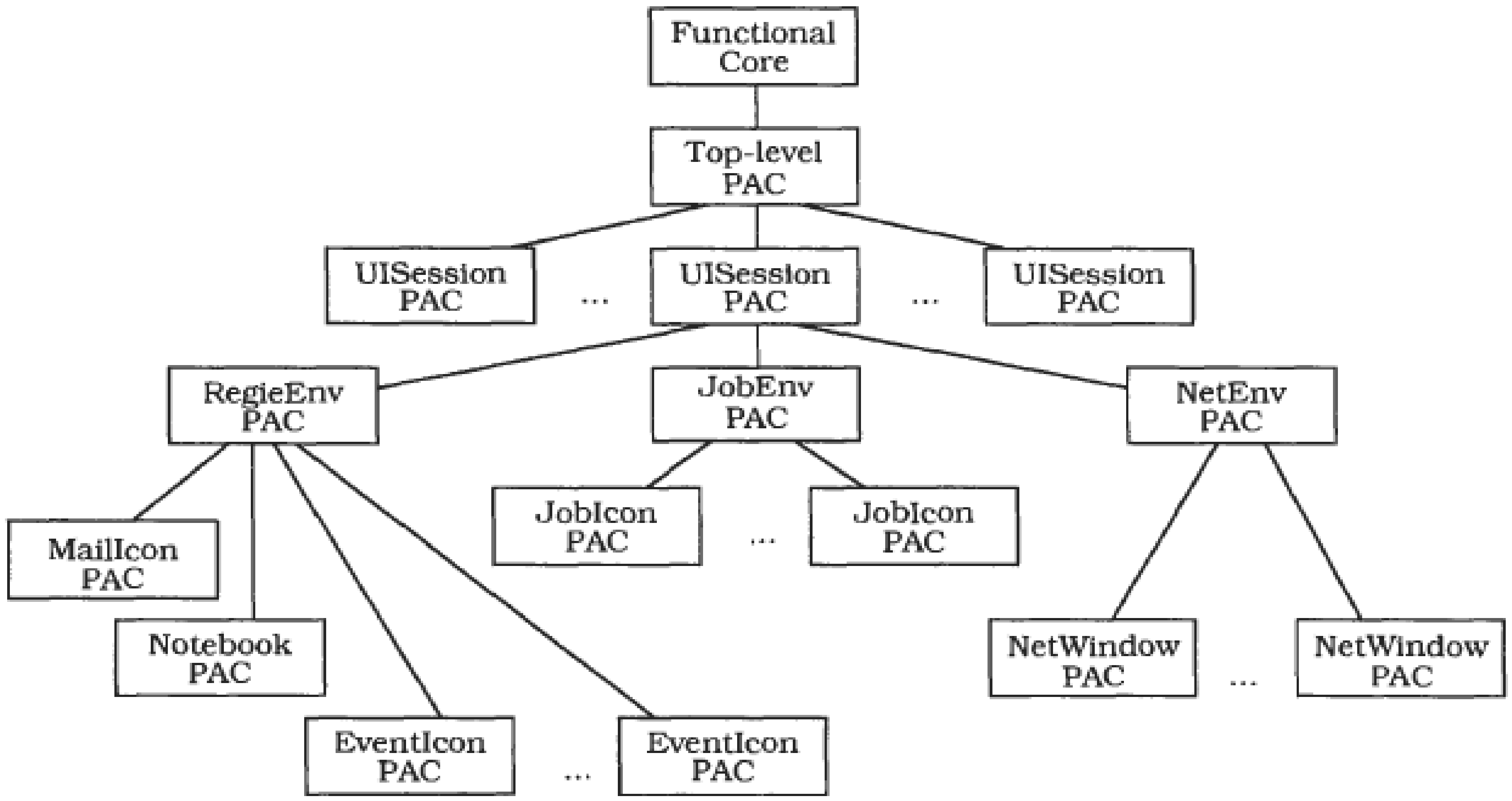
9. 变体

- *PAC* agents *as* active objects. Many applications, especially interactive ones, benefit from multi-threading. The mobile robot system is an example of a multi-threaded PAC architecture. Every PAC agent can be implemented as an active object that lives in its own thread of control. Design patterns like Active Object and Half-Sync/Half-Async can help you implement such an architecture.
- PAC agents as processes. To support PAC agents located in different processes or on remote machines, use proxies to locally represent these PAC agents and to avoid direct dependencies on their physical location. Use the Forwarder-Receiver pattern) or the Client-Dispatcher-Server pattern (323) to implement the inter-process communication (IPC) between PAC agents.

10. 已知应用

- **网络通信量管理**
  - 从交换单元收集通信量数据
    - Gathering traffic data from switching units
  - 阀值检查与溢出异常的产生
    - Threshold checking and generation of overflow exceptions.
  - 网络异常的日志与跟踪
    - Logging and routing of network exceptions.
  - 通信流量和网络异常的可视化
    - Visualization of traffic flow and network exceptions.
  - 显示整个网络的各种用户可配置的视图
    - Displaying various user-configurable views of the whole network.
  - 通信量数据的统计评估
    - Statistical evaluations of traffic data.
  - 存储历史通信量数据
    - Access to historic traffic data.
  - 系统管理与配置
    - System administration and configuration.

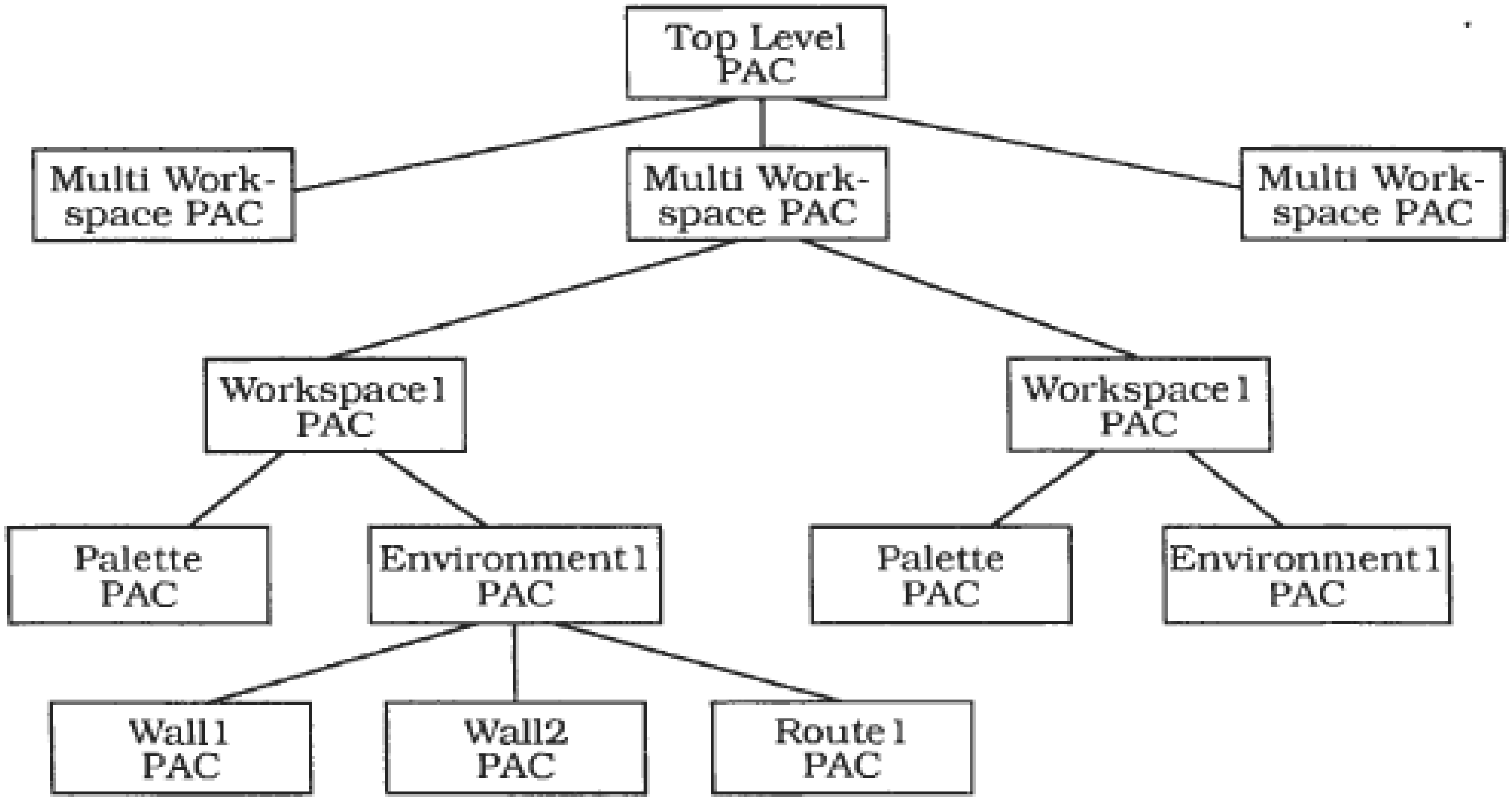
网络通信量管理


Functional Core
Top-level PAC
UISession PAC
…
UISession PAC
…
UISession PAC
RegieEnv PAC
JobEnv PAC
NetEnv PAC
MailIcon PAC
Notebook PAC
EventIcon PAC
…
EventIcon PAC
JobIcon PAC
…
JobIcon PAC
NetWindow PAC
…
NetWindow PAC

10. 已知应用

- 移动机器人 **Mobile Robot**
  - 允许操作员进行的活动
    - Provide the robot with a description of the environment it will workin, places in this environment, and routes between places.
    - Subsequently modify the environment.
    - Specify missions for the robot.
    - Control the execution of missions.
    - Observe the progress of missions.

Mobile Robot



11.效果

- 优点
  - *Separation of concerns.*
  - *Support for change and extension.*
  - *Support for multi-tasking.*

11.效果

- 缺点
  - *Increased system complexity.*
  - *Complex control component.*
  - ***Efficiency.*** The overhead in the communication between PAC agents may impact system efficiency.
  - ***Applicability.*** The smaller the atomic semantic concepts of an application are, and the greater the similarity of their user interfaces, the less applicable this pattern is.

参见

The *Model-View-Controller* pattern (125)also separates the functional core of a software system **from** information display and user input handling. MVC, however, defines its controller as the entity responsible for accepting user input and translating it into internal semantics. This means that MVC effectively divides the **user-**accessible part—the presentation in PAC—into view and control. It lacks mediating control components. Furthermore, MVC does not separate self-reliant **subtasks** of a system into cooperating but loosely- coupled agents.

2.5 适应性系统

- 系统随时间演化——添加新的功能和更改现有的服务.
  - *微核(Microkernel)模式*应用于必须能够适应变更系统需求的软件系统。
    - **It separates a minimal functional core from extended functionality and customer-specific parts.**
    - **The microkernel also serves as a socket for plugging in such extensions and coordinating their collaboration.**
    - 支持小的、有效的和可移植的操作系统的设计，并用新服务来支持其扩展
      - **Chorus、Mach、Windows NT**
    - 适用于：要求对不同平台具有高度适应性并能满足客户定制需求的系统

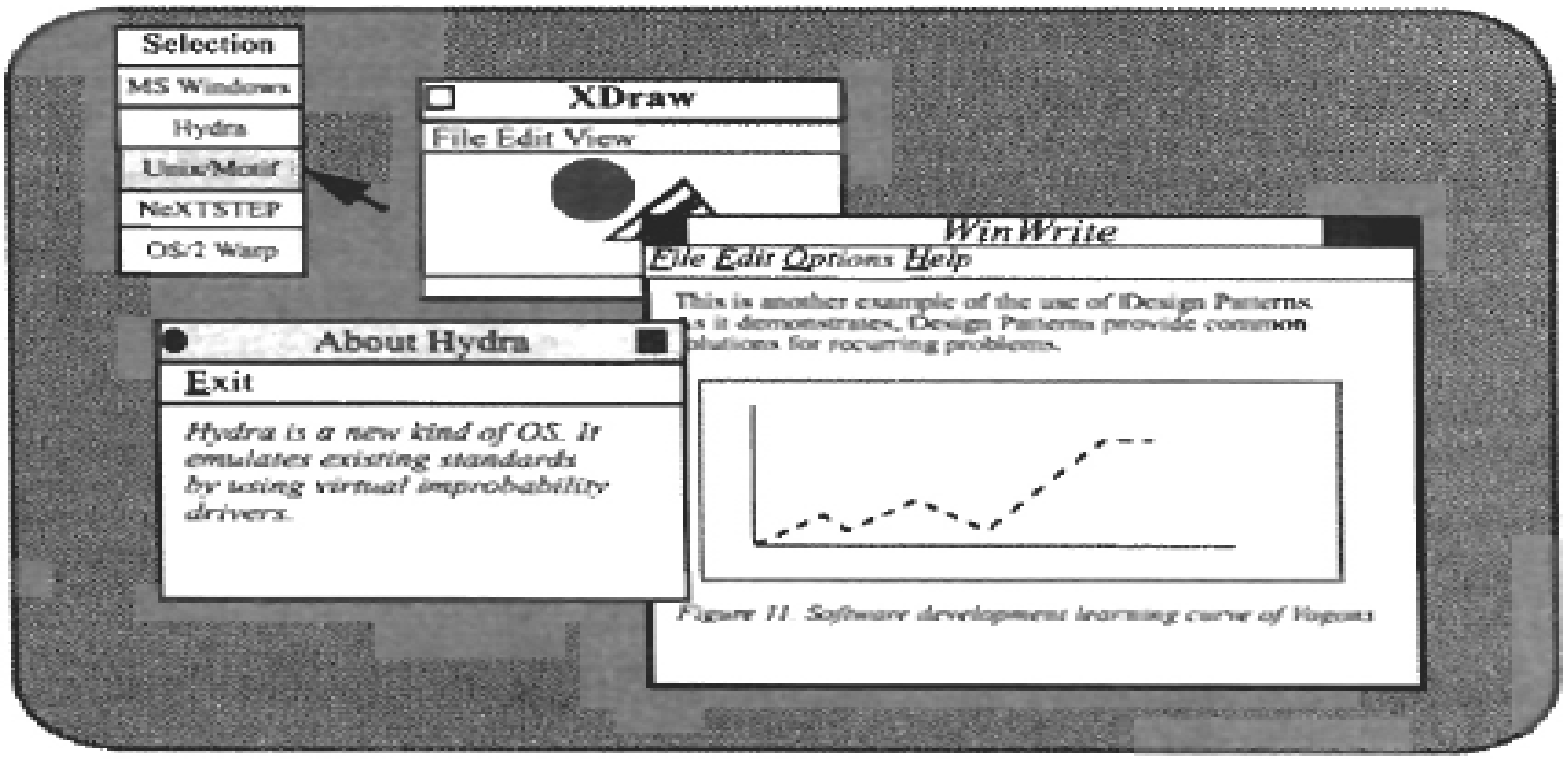
2.5 适应性系统

- *映像(Reflection)模式*为动态地改变软件系统的结构和行为提供了一种机制，它支持诸如类型结构和函数调用机制等基本方面的修改
  - **In this pattern, an application is split into two parts.**
    - **A meta level provides information about selected system properties and makes the software self-aware.**
    - **A base level includes the application logic. Its implementation builds on the meta level.**
    - **Changes to information kept in the meta level affect subsequent base-level behavior.**
    - 设计一个系统，使这个系统维护了自身的信息，并使用这些信息来保持可变性和可扩展性
      - 设计语言：**CLOS**、**Smalltalk**
      - 操作系统：**Apertos**
      - 工业应用程序：**CORBA**、**OLE**

2.5.1 微核

- The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

1. 例子



Hydra is a new kind of OS. It emulates existing standards by using virtual improbability drivers.

This is another example of the use of Design Patterns.
s it demonstrates, Design Patterns provide common
lutions for recurring problems.

Figure 11. Software development learning curve of Vogons

2. 语境

- 使用类似的编程接口的几个应用程序的开发建立在相同核心功能上
- ***The development of several applications that use similar programming interfaces that build on the same core functionality.***

3. 问题

- 特别需要考虑
  - 应用程序平台必须适应连续的硬件和软件进化.
  - 应用程序平台应该是可移植的、可扩展的、可适应的，使之易于集成不断出现的技术.
- 这又导致以下的强制条件:
  - 在你的领域中的应用程序需要支持不同的但又相似的应用程序平台

    The applications in your domain need to support different, but similar, application platforms.
  - 应用程序可以按类分组，每组以不同方式使用相同的功能核心，需要基于应用程序平台模拟现有标准

    The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.
- 应用程序平台的功能核心应该分离进一个组件，该组件占用最小的存储空间，其服务消耗处理能力尽可能小

  The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

4. 解决方法

- 将应用程序平台上的基本服务封装到一个微核组件中.
  - Core functionality that cannot be implemented within the microkernel without unnecessarily increasing its size or complexity should be separated in internal servers.
  - External servers implement their own view of the underlying microkernel.
    - To construct this view, they use the mechanisms available through the interfaces of the microkernel. Every external server is a separate process that itself represents an application platform. Hence, a Microkernel system may be viewed as an application platform that integrates other application platforms.
  - 客户机通过微核提供的通信能力与外部服务器通信

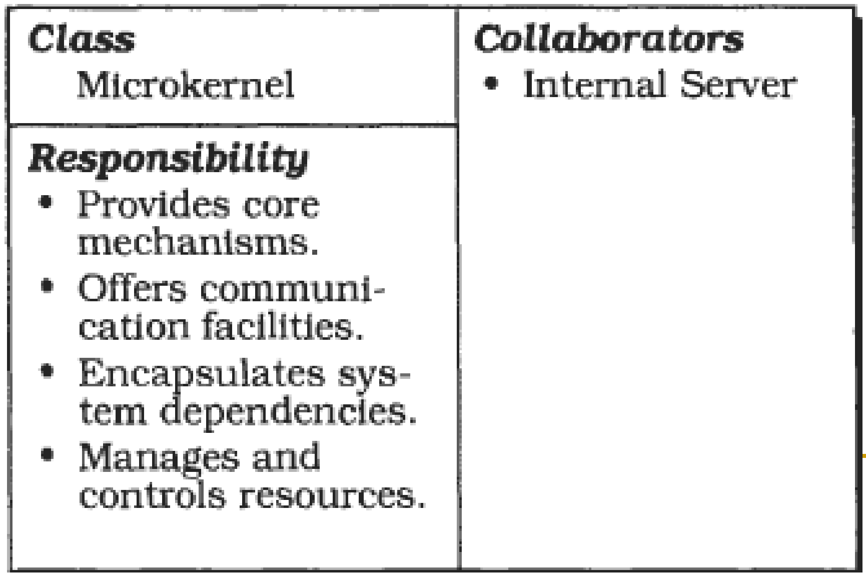
    Clients communicate with external servers by using the communication facilities provided by the microkernel.

5. 结构

- ***内部服务器 Internal servers***
- ***外部服务器 External servers***
- ***适配器 Adapters***
- ***客户机 Clients***
- ***微核 Microkernel***

5. 结构

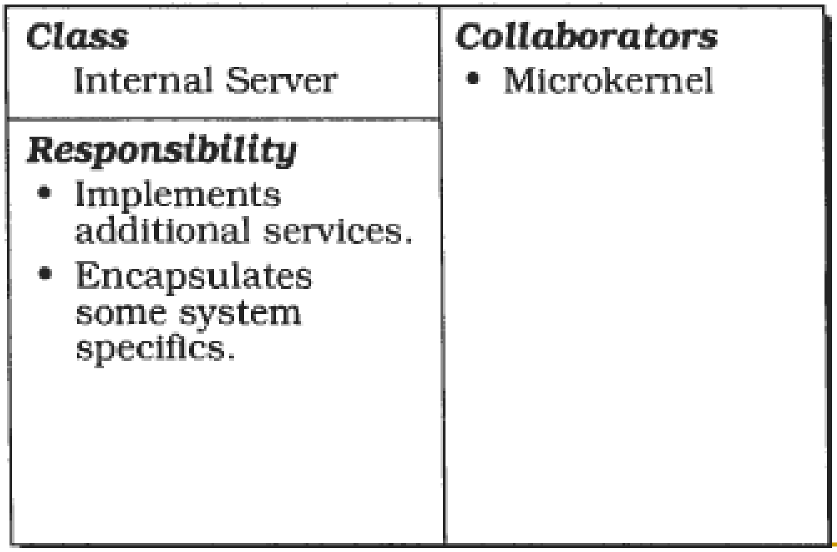
- 微核
  - 实现了诸如通信手段或资源处理那样的主要服务
  - 许多系统特定的附属物被封装在微核中
  - 微核也负责维护像进程或文件那样的系统资源。控制和协调对这些资源的访问



Class Microkernel	**Collaborators**
Responsibility • Provides core mechanisms. • Offers communi-cation facilities. • Encapsulates sys-tem dependencies. • Manages and controls resources.	• Internal Server

5. 结构

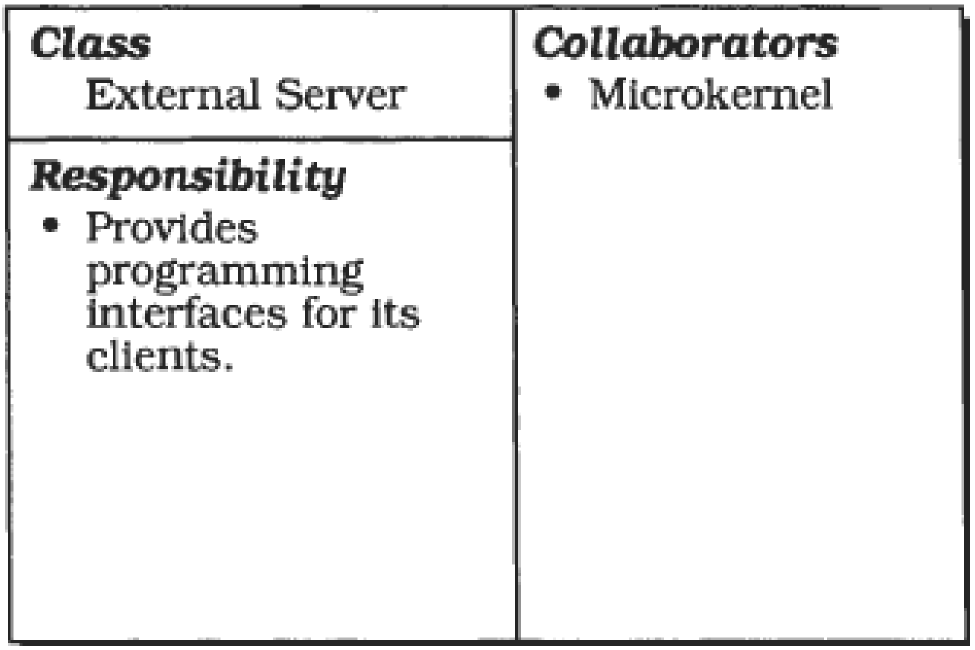
- 内部服务器
  - 内核服务器只能由微核组件访问
  - 额外和更复杂的服务由内部服务器实现，必要时微核会激活或加载内部服务器



Class Internal Server	***Collaborators***
Responsibility	• Microkernel
• Implements additional services. • Encapsulates some system specifics.	

5. 结构

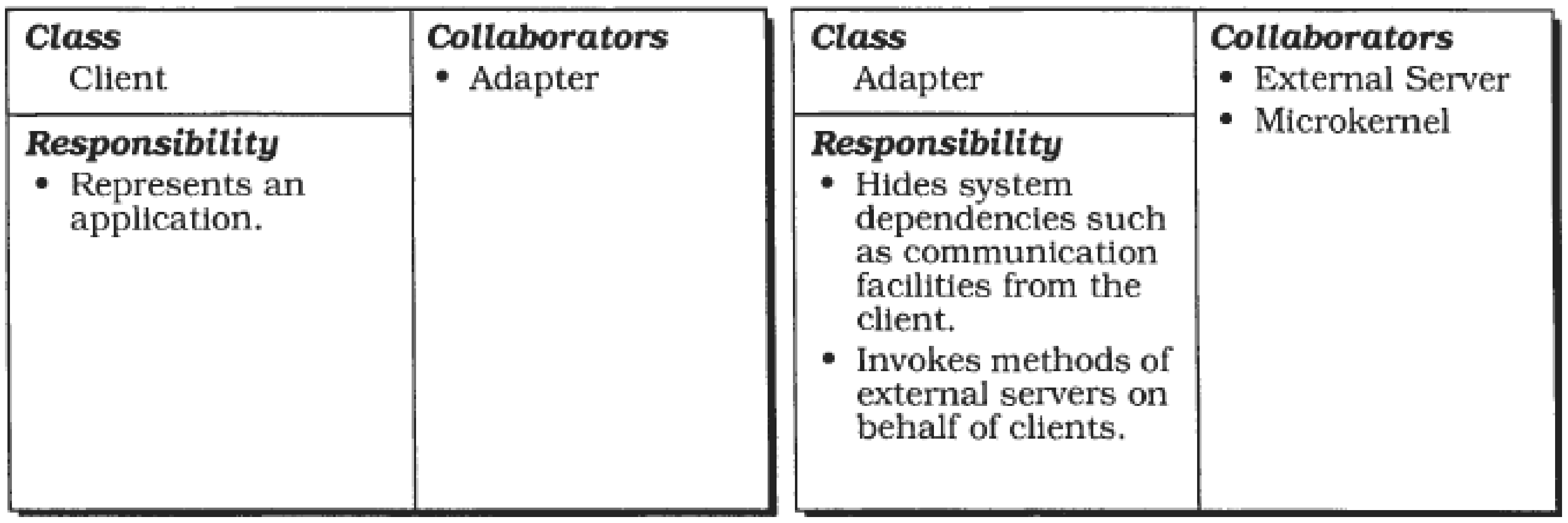
- 外部服务器



Class External Server	**Collaborators**
Responsibility	• Microkernel
• Provides programming interfaces for its clients.	

5. 结构

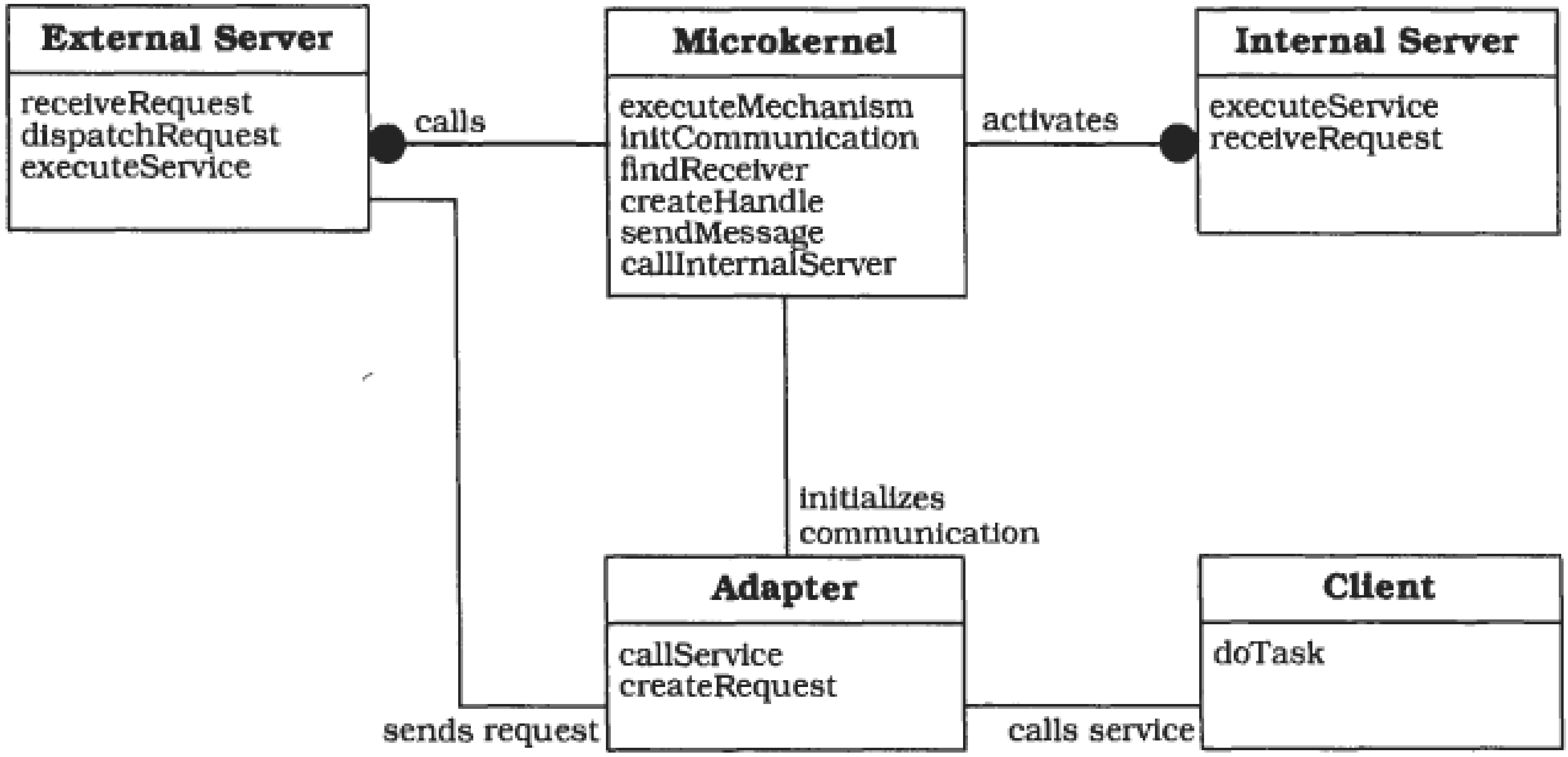
- 客户机
- 适配器



Class Client	***Collaborators***
Responsibility	• Adapter
• Represents an application.	

Class Adapter	***Collaborators***
Responsibility	• External Server • Microkernel
• Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

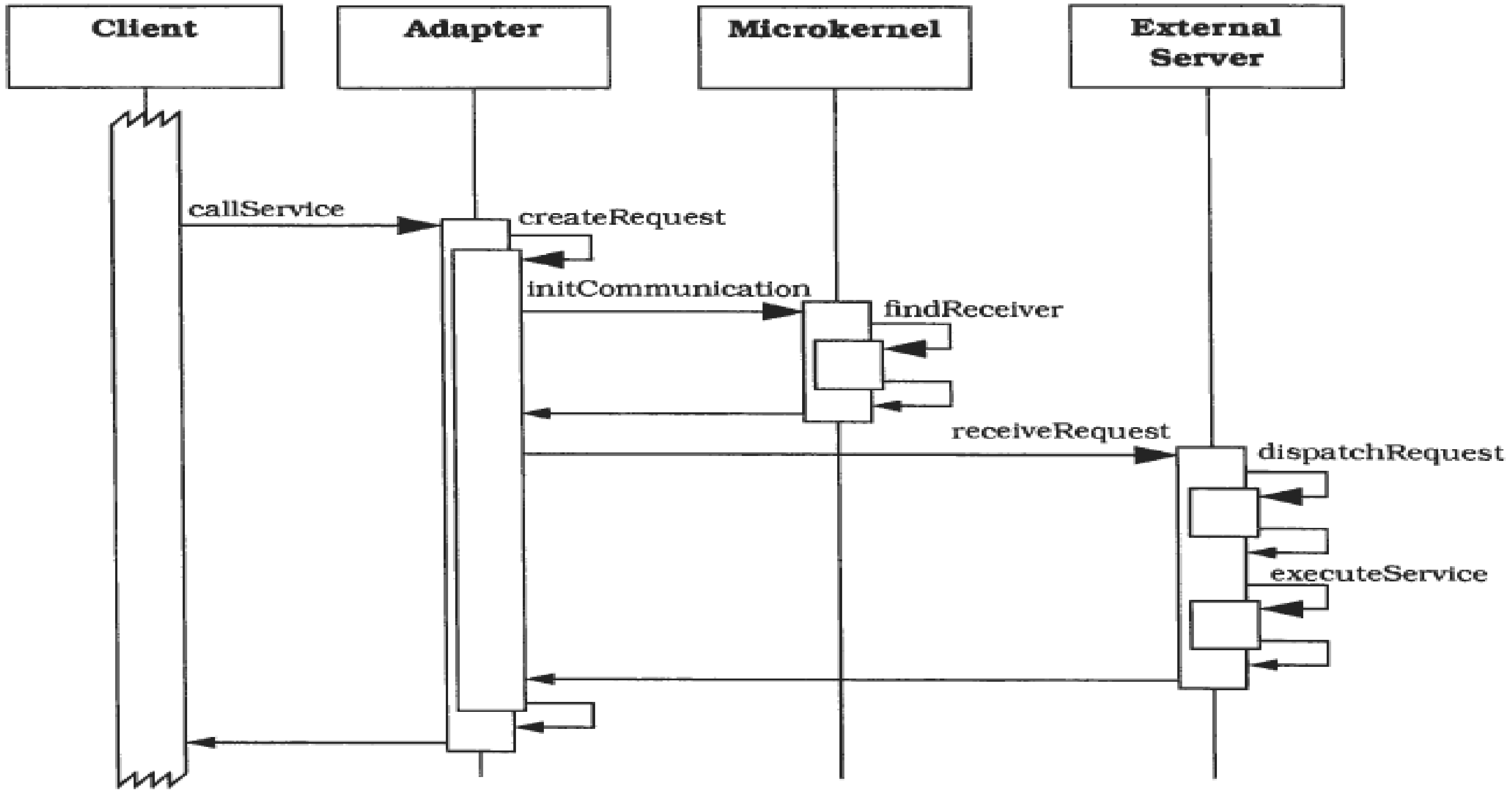
5. 结构


External Server
receiveRequest
dispatchRequest
executeService
calls
Microkernel
executeMechanism
initCommunication
findReceiver
createHandle
sendMessage
callInternalServer
activates
Internal Server
executeService
receiveRequest
initializes communication
Adapter
callService
createRequest
sends request
calls service
Client
doTask

6. 动态特性——场景I

- At a certain point in its control flow the client requests a service from an external server by calling the adapter.
- The adapter constructs a request and asks the microkernel for a communication link with the external server.
- The microkernel determines the physical address of the external server and returns it to the adapter.
- After retrieving this information, the adapter establishes a direct communication link to the external server.
- The adapter sends the request to the external server using a remote procedure call.
- The external server receives the request, unpacks the message and delegates the task to one of its own methods. After completing the requested service, the external server sends all results and status information back to the adapter.
- The adapter returns to the client, which in turn continues with its control flow.

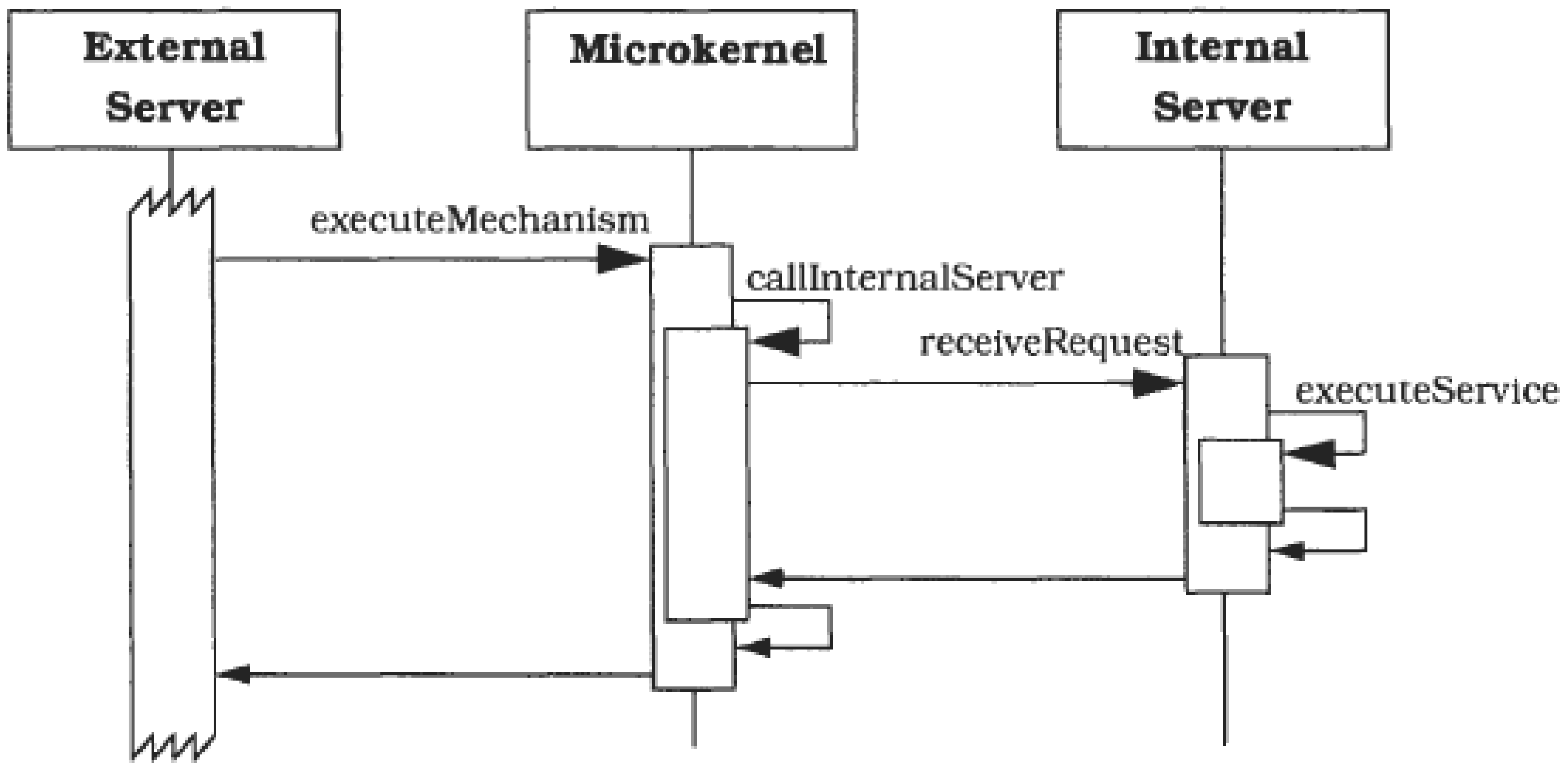
6. 动态特性——场景I


Client
Adapter
Microkernel
External Server
callService
createRequest
initCommunication
findReceiver
receiveRequest
dispatchRequest
executeService

6. 动态特性——场景II

- The external server sends a service request to the microkernel.
- A procedure of the programming interface of the microkernel is called to handle the service request. During method execution the microkernel sends a request to an internal server.
- After receiving the request, the internal server executes the requested service and sends all results back to the microkernel.
- The microkernel returns the results back to the external server.
- Finally, the external server retrieves the results and continues with its control flow.

6. 动态特性——场景II


External Server
Microkernel
Internal Server
executeMechanism
callInternalServer
receiveRequest
executeService

7. 实现

1. ***分析应用领域。Analyze the application domain.*** If you already know the policies your external servers need to offer, or if you have a detailed knowledge about the external servers you are going to implement, continue with step 2. If not, perform a domain analysis and identify the core functionality necessary for implementing external servers, then continue with step 3.
2. ***分析外部服务器 。Analyze external servers.*** Analyze the policies external servers are going to provide. You should then be able to identify the functionality you require within your application domain.
3. ***分类服务。Categorize the services.*** Whenever possible, group all the functionality into semantically-independent categories.

7. 实现

4. ***划分类别。Partition the categories.*** Separate the categories into services that should be part of the microkernel, and those that should be available as internal servers. You need to establish criteria for this separation.
5. ***Find a consistent and complete set of operations and abstractions for every category you identified in step 1.*** Remember that the microkernel provides mechanisms, not policies. Each policy an external server provides must be implemented through use of the services the microkernel offers through its interfaces.
   - Creating and terminating processes and threads.
   - Stopping and restarting them.
   - Reading from or writing to process address spaces.
   - Catching and handling exceptions.
   - Managing relationships between processes or threads.
   - Synchronizing and coordinating threads.

7. 实现

6. Determine strategies for request transmission and retrieval. Specify the facilities the microkernel should provide for communication between components.

- ***Synchronous Remote Procedure Calls (RPCs).*** RPCs enable a client to invoke the services of a remote server as if they were implemented by local procedure calls. The mechanisms necessary for supporting RPCs, for example the packing and unpacking of requests or the transmission of messages across process boundaries, are hidden from the caller and the sewer called.
- ***Asynchronous Mailboxes.*** A *mailbox* is a type of message buffer. A set of components is allowed to read messages from the mailbox, another set of components has permission to write messages to it. A component may be allowed to perform both activities.

7. 实现

7. Structure the microkernel component. If possible, design the microkernel using the Layers pattern to separate system-specific parts from system-independent parts of the microkernel.

- The lowermost layer consists of low-level objects that hide hardware-specific details such as the bus architecture from other parts of the microkernel.
- In the intermediate layers the primary services are provided by system objects, such as objects responsible for memory management and objects used for managing processes.
- The uppermost layer comprises all the functionality that the microkernel exposes publicly, and represents the gateway to the microkernel services for any process.

7. 实现

8. ***To specify the programming interfaces of the microkernel, you need to decide how these interfaces should be accessible externally.*** You must obviously take into account whether the microkernel is implemented as a separate process or as a module that is physically shared by other components.
9. ***The microkernel is responsible for managing all system resources such as memory blocks, devices or device contexts-a handle to an output area in a graphical user interface implementation.*** The microkernel maintains information about resources and allows access to them in a coordinated and systematic way.

7. 实现

10. Design and implement the internal servers as separate processes or shared libraries. Perform this step in parallel with steps 7-9, because some of the microkernel services need to access internal servers. It is helpful to distinguish between active and passive servers:

- Active servers are implemented as processes
- Passive servers as shared libraries

7. 实现

11. Implement the external servers. All the policies the external servers include are based on the services available in the programming interfaces of the microkernel. **An** external server receives requests, analyzes them, executes the appropriate services and sends the results back to the caller. When executing services, the external server may call operations in the microkernel.

- 要为**Hydra**开发的外部服务器
  - **A** full implementation of Microsoft’s Win32 and Win16 APIs, to allow users to run Windows **NT,** Windows 3.11 and Windows 95 applications.
  - The complete functionality provided by IBM OS/2 Warp 2.0.
  - ***An*** implementation of Openstep.
  - All relevant UNIX System V interfaces specified by X/Open.

7. 实现

12. Implement the adapters. The primary task of an adapter is to provide operations to its clients that are forwarded to an external server.

- Whenever the client calls a function of the external server, the adapter packages all relevant information into a request and forwards the request to the appropriate external server. The adapter then waits for the server's response and finally returns control to the client, using the facilities for inter-component communication.

7. 实现

13. Develop client Applications or use existing ones for the ready-to-run Microkernel system. When creating a new client for a specific external server, its architecture is only limited by the constraints imposed by the external server. That is, clients depend on the policies implemented by their external server.

- In Hydra we can develop Microsoft Windows applications by accessing the services of the Microsoft Windows external server via the Microsoft Windows adapter.

8. 已解决的例子

- ***Building an external server*** on top of the Hydra microkernel that implements all the programming interfaces provided by MacOS, including the policies of the Macintosh user interface.
- ***Providing an adapter*** that is designed as a library, dynamically linked to clients. For every API function available in a native MacOS system, a syntactically-identical procedure must be provided by the library.
- ***Implementing the internal servers*** required for MacOS. For example, one internal server provides the network protocol AppleTalk. The microkernel must be modified to invoke these additional internal servers on behalf of the MacOS server.

9. 变体

- ***Microkernel System with indirect Client-Server connections.***
- ***Distributed Microkernel System***

10. 已知应用

- The **Mach** operating system
- The operating system **Amoeba**
- **Chorus**
- Windows NT
- Microkernel Datenbank Engine

11. 效果——优点

- ***Portability***.
  - In most cases you do not need to port external servers or client applications if you port the Microkernel system to a new software or hardware environment.
  - Migrating the microkernel to a new hardware environment only requires modifications to the hardware-dependent parts.
- ***Flexibility and Extensibility.***
- ***Separation of policy and mechanism.***
- Distributed Microkernel variant
  - ***Scalability.***
  - ***Reliability.***
  - ***Transparency.***

11. 效果——不足

- Performance.
- Complexity of design and implementation

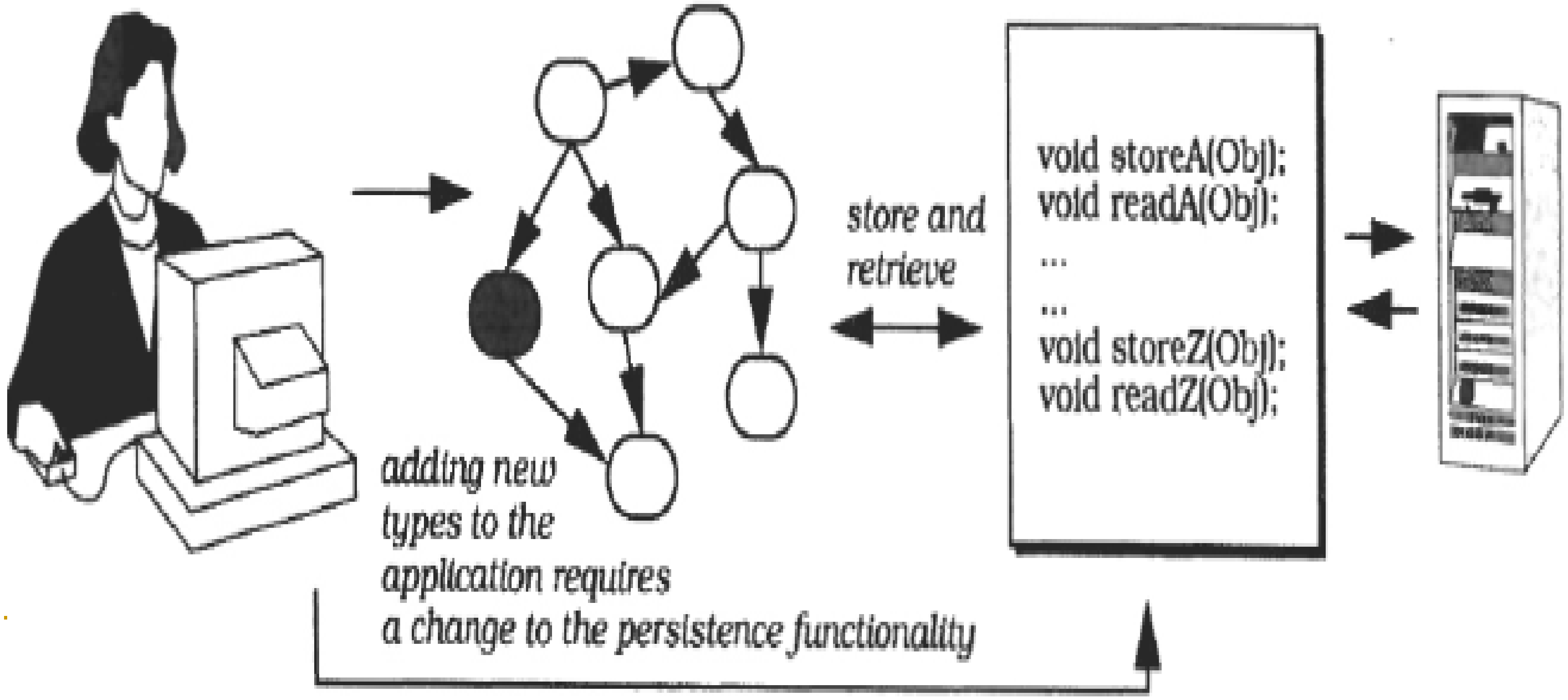
2.5.2 映像 Reflection

- The **Reflection** architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects. such as type structures and function call mechanisms. In this pattern, an application is split into two parts.
  - **A** meta level provides information about selected system properties and makes the software self-aware.
  - A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

Also Known As Open Implementation, Meta-Level Architecture

1.例子

- 需要向磁盘写入对象再从磁盘读入对象的C++应用程序
  - 应用程序的类结构发生改变时…
  - 独立于特定类型结构的持久性组件(persistence component)



2. 语境

- *构建一个支持它们自己对先验知识修改的系统*
- ***Building systems that support their own modification a priori.***

3. 问题

- 设计一个在很大程度上满足不同需求先验知识的系统
  - 描述一个可以随时修改和扩展的体系结构
- 与该问题相关的要求
  - Changing software is tedious, error prone, and often expensive.
  - Adaptable software systems usually have a complex inner structure.
  - The more techniques that are necessary for keeping a system changeable, such as parameterization, subclassing, mix-ins, or even copy and paste, the more awkward and complex its modification becomes. **A** uniform mechanism that applies to all kinds of changes is easier to use and understand.
  - Changes can be of any scale, from providing shortcuts for commonly-used commands to adapting an application framework for a specific customer.
  - Even fundamental aspects of software systems can change, for example the communication mechanisms between components

4. 解决方案

- *元层次(meta level)* 提供了一个软件的自表示，来给出软件自身结构和行为的知识，元层次由由元对象**(metaobjects)**构成。

  The meta level provides a self-representation of the software to give it knowledge of its own structure and behavior, and consists of so-called ***rnetaobjects.***

- *基本层次(base level)*定义了应用程序逻辑。其实现使用元对象来保持这些可能要改变方面的独立性。

  The base level defines the application logic. Its implementation uses the metaobjects to remain independent of those aspects that are likely to change.

- 被指定的操作元对象的*接口(Interface)*，被称为元对象协议**(metaobject protocol ,MOP)**，它允许客户机描述特殊的变化。

  An interface is specified for manipulating the metaobjects. It is called the ***metaobject protocol (MOP***), and allows clients to specify particular changes, such as modification of the function call mechanism metaobject mentioned above.

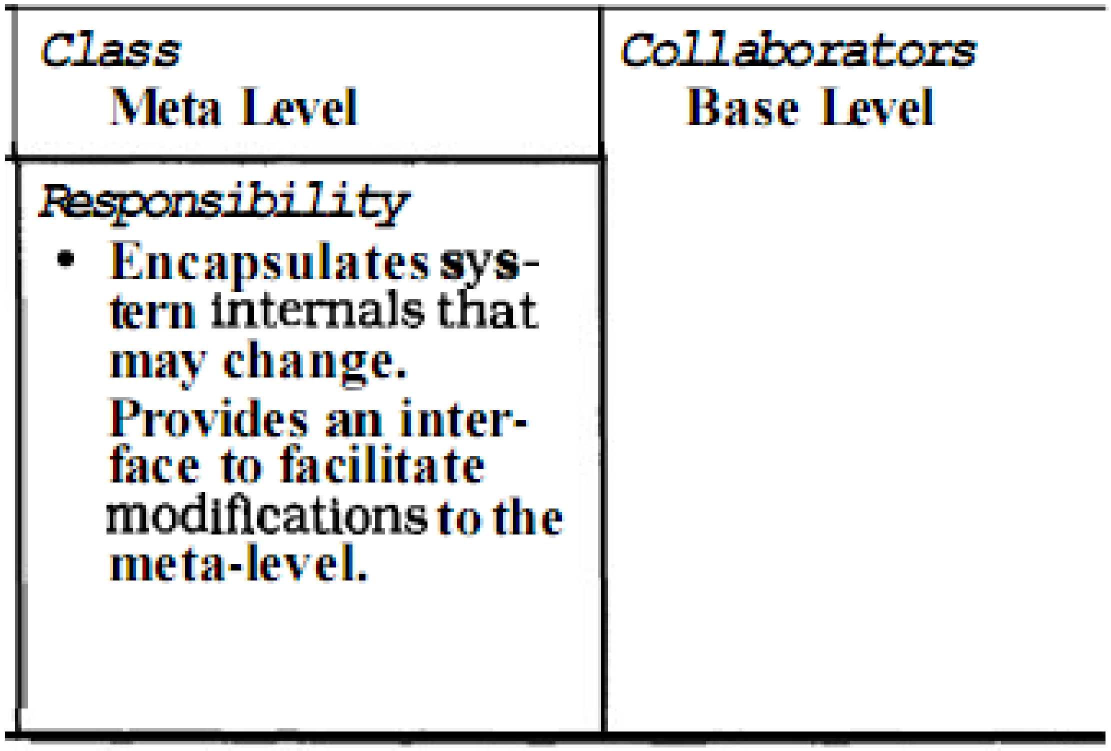
4. 解决方案

- 例子中的持久性组件
  - 属于应用程序的base level
  - 元对象：提供运行时类型信息

5. 结构

- 元层次(meta level)有一组元对象构成
  - Each metaobject encapsulates selected information about a single aspect of structure, behavior, or state of the base level
  - 这些信息的来源
    - 由系统运行期间的环境提供
    - 用户定义的
    - 从运行期间的base level获取
- 所有的元对象一起提供一个应用程序的自表示
  - E.g. 分布式系统中，提供基本层次组件的物理位置信息的元对象
- 元对象的接口允许基本层次存取它维护的信息或它提供的服务
  - 元对象不允许基本层次修改其内部状态

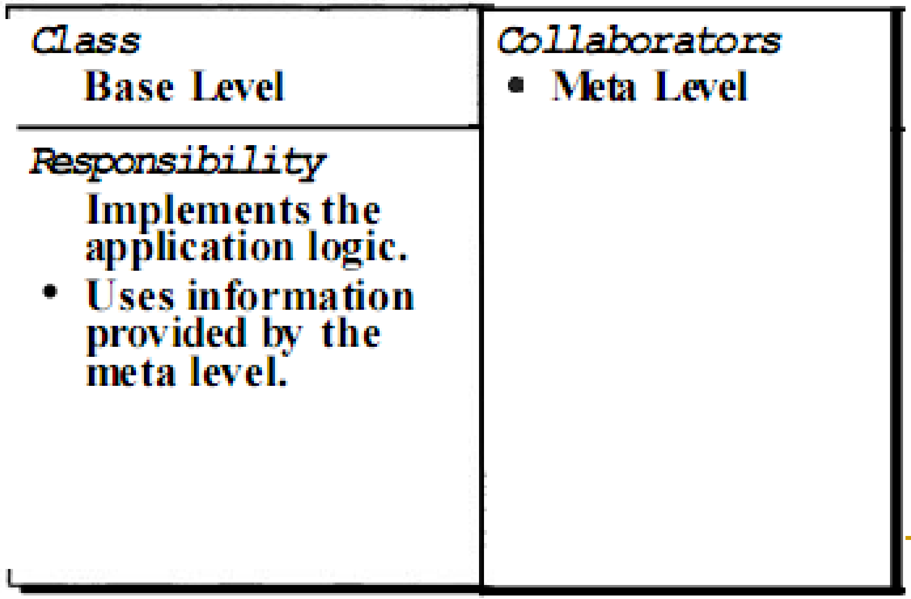
5. 结构



Class Meta Level	*Collaborators* Base Level
Responsibility • Encapsulates sys-tem internals that may change. Provides an inter-face to facilitate modifications to the meta-level.	

5. 结构

- 基本层次(base level)建模并实现了软件的应用程序逻辑
  - 基本层次使用元对象提供的信息和服务
    - E.g. 组件的位置信息

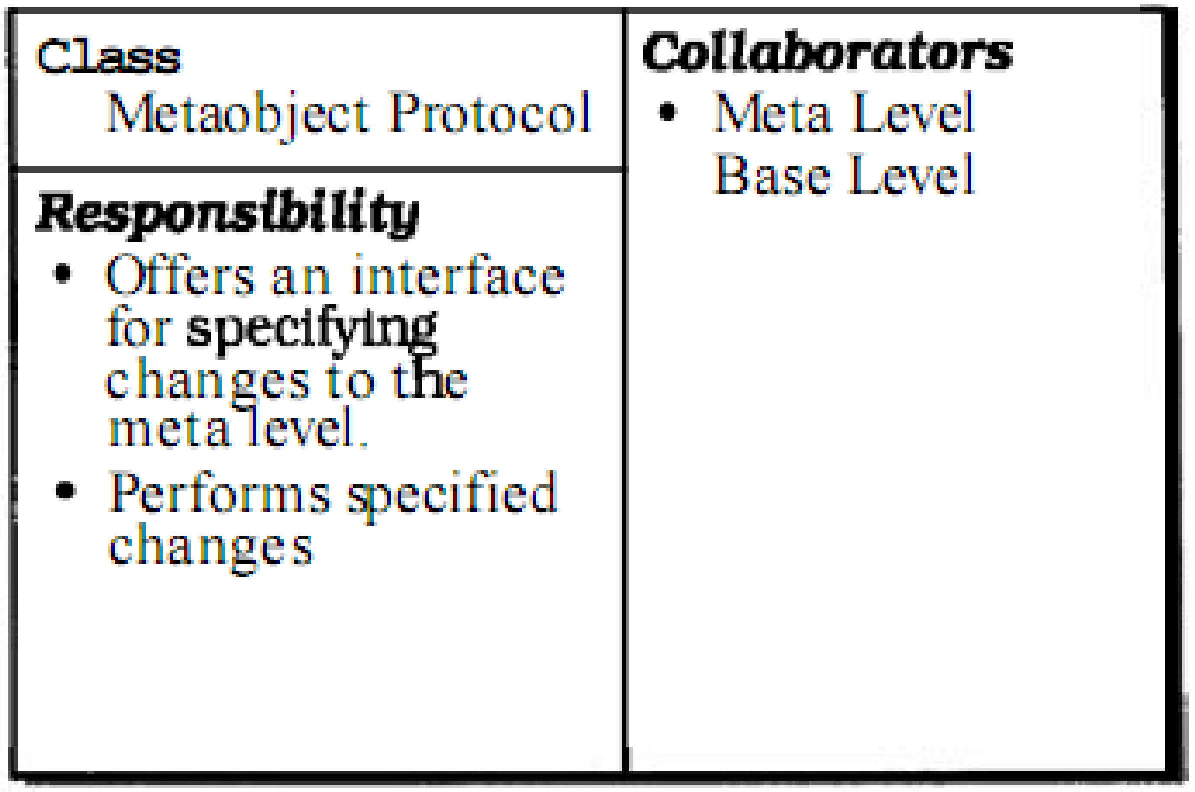


Class Base Level	*Collaborators*
Responsibility Implements the application logic. • Uses information provided by the meta level.	• Meta Level

5. 结构

- 元对象协议(MOP)：元层次的外部接口
  - MOP的客户程序可以是：
    - 基本层次的组件
    - 其他应用程序
    - 授权的用户
  - MOP的客户程序可以指定…的修改，由MOP负责执行这些修改
    - Metaobjects
    - Their relationships using the base level
- MOP通常被设计为一个独立的组件

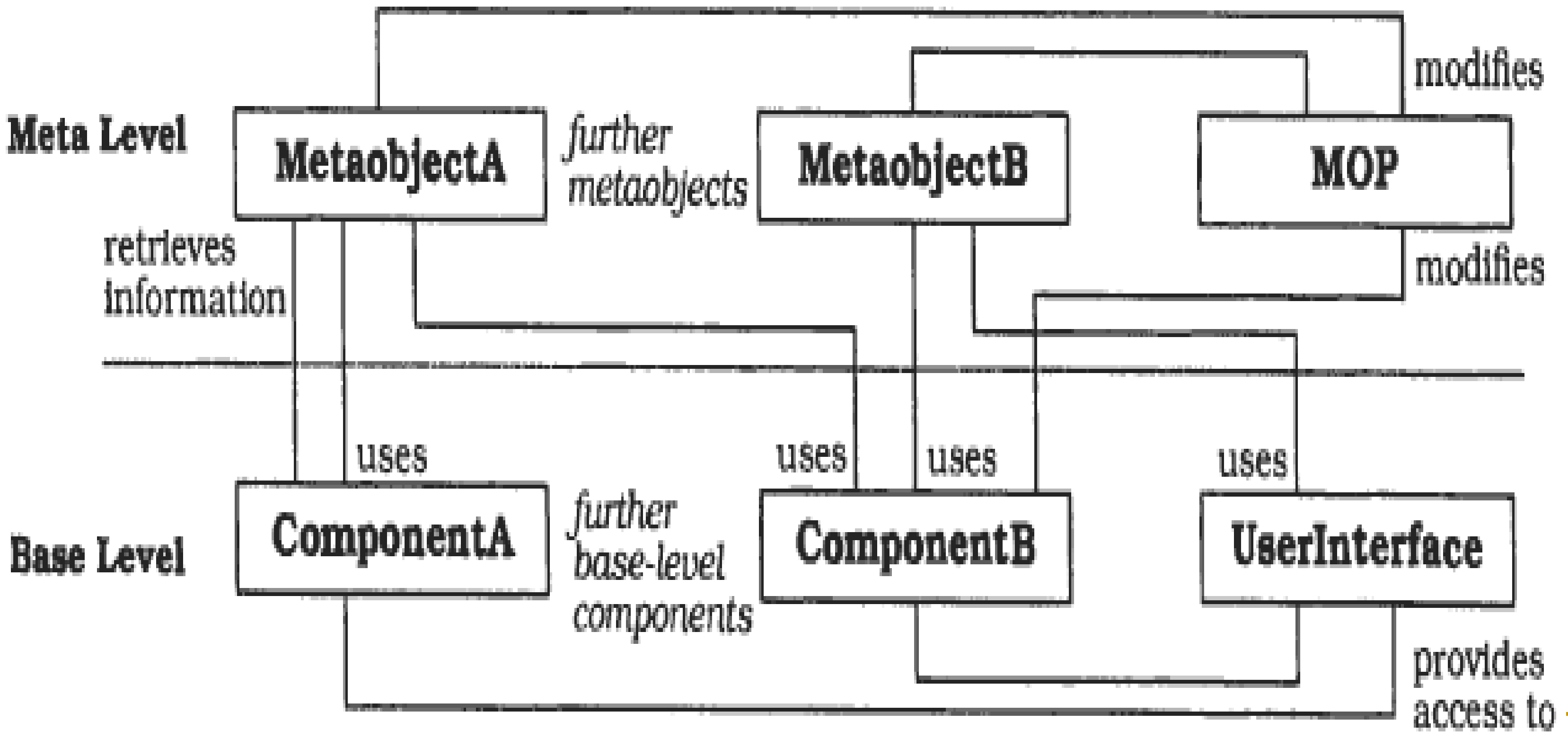
5. 结构



Class Metaobject Protocol	***Collaborators***
Responsibility • Offers an interface for specifying changes to the meta level. • Performs specified changes	• Meta Level Base Level

5. 结构

- 两层之间存在相互依赖性



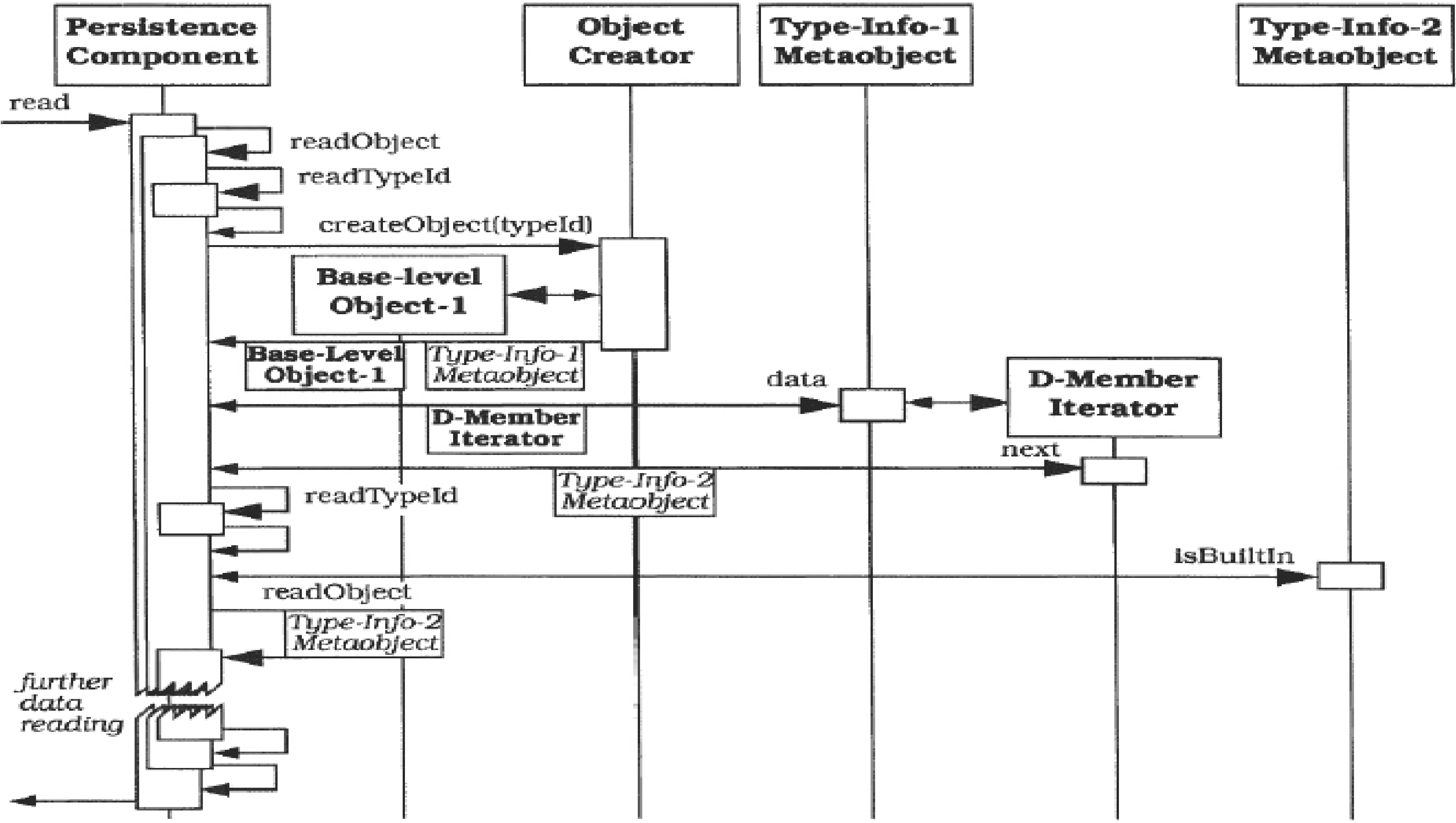
6. 动态特性——场景I

- The user wants to read stored objects. The request is forwarded to the read() procedure of the persistence component, together with the name of the data file in which the objects are stored.
- Procedure read() opens the data file and calls an internal readobject() procedure which reads the first type identifier.
- Procedure readObject() calls the metaobject that is responsible for the creation of objects. The 'object creator' metaobject instantiates an 'empty' object of the previously-determined type. It returns a handle to this object and a handle to the corresponding run-time type information (RTTI) metaobject.

6. 动态特性——场景I

- Procedure **readobject**() requests an iterator over the data members of the object to be read from its corresponding metaobject. The procedure iterates over the data members of the object.
- Procedure **readobject()** reads the type identifier for the next data member. If the type identifier denotes a built-in type--a case we do not illustrate--the **readobject()** procedure directly assigns the next data item from the file to the data member, based on the data member's size and offset within the object. Othenvise **readobject()** is called recursively. This recursion starts with the creation of an 'empty' object if the data member is a pointer. If not, the recursively called **readobject()** operates on the existing layout of the object that contains the data member.
- After reading the data, the **read**() procedure closes the data file and returns the new objects to the client that requested them.

6. 动态特性——场景I


Persistence Component
Object Creator
Type-Info-1 Metaobject
Type-Info-2 Metaobject
read
readObject
readTypeId
createObject(typeId)
Base-level Object-1
Base-Level Object-1
Type-Info-1 Metaobject
data
D-Member Iterator
D-Member Iterator
next
Type-Info-2 Metaobject
readTypeId
isBuiltIn
readObject
Type-Info-2 Metaobject
further data reading

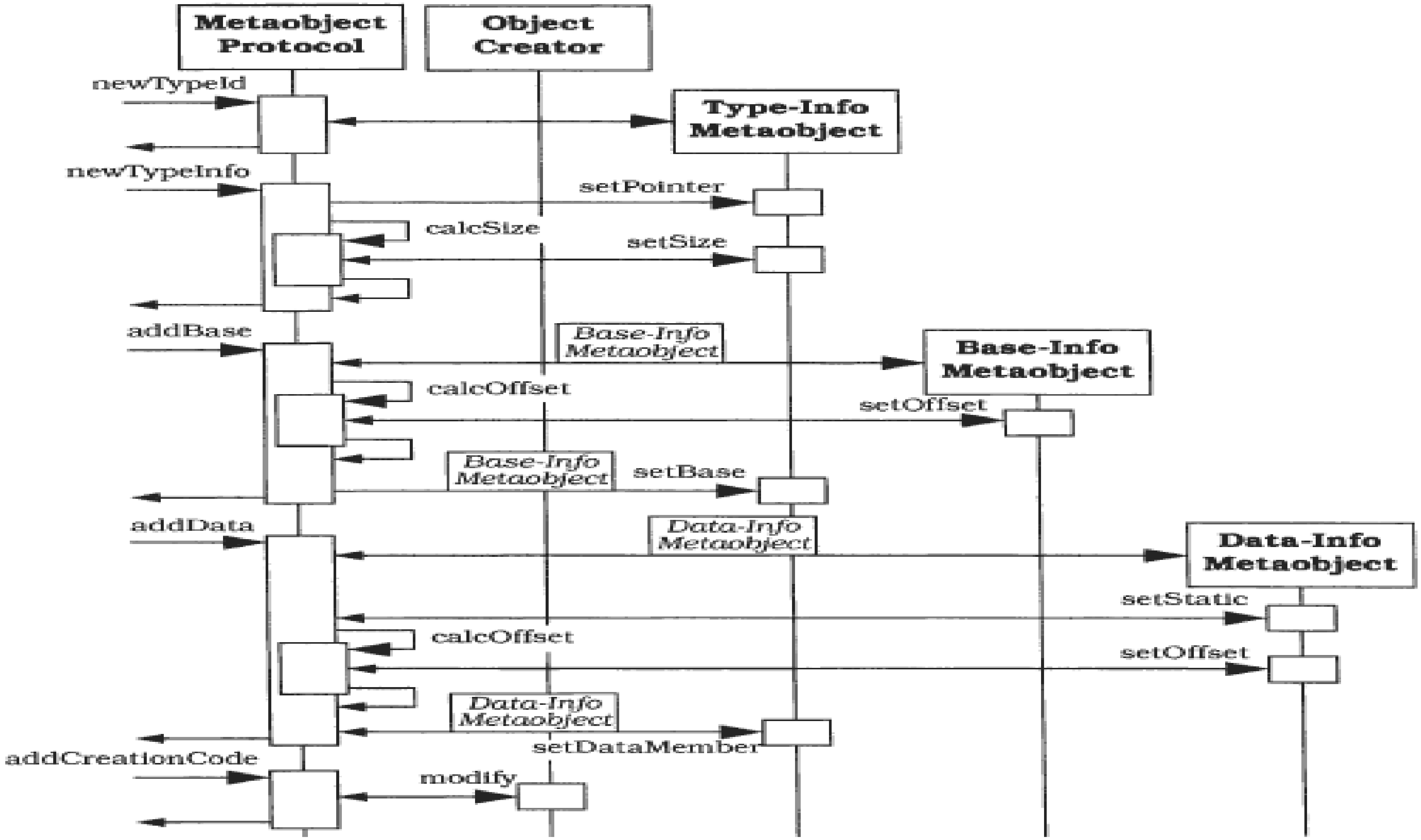
6. 动态特性——场景II

- A client invokes the metaobject protocol to specify run-time type information for a new type in the application. The name of the type is passed as an argument.
- The metaobject protocol creates a metaobject of class type-info for this type. This metaobject also serves as a type identifier.
- The client calls the metaobject protocol to add extended type information. This includes setting the size of the type, whether or not it is a pointer, and its inheritance relationships to other types. To handle the inheritance relationship, the metaobject protocol creates metaobjects of class baseInfo. These maintain a handle to the type-info object for a particular base class and its offset within the new type.
- In the next step, the client specifies the inner structure for the new type. The metaobject protocol is provided with the name and type of every data member. For every data member the metaobject protocol creates an object of class dataInfo. It maintains a handle to the type-info object for the type of the member, its name, and whether or not it is a static data member. The dataInfo object also maintains the absolute address of the data member if it is static, otherwise its offset within the new type.

6. 动态特性——场景II

- The client invokes the metaobject protocol to modify existing types that include the new type as a data member. Appropriate data member information is added for every type. Since this step is very similar to the previous one, we do not illustrate it in the object message sequence chart that follows.
- Finally, the client calls the metaobject protocol to adapt the 'object creator' metaobject. The persistence component must be able to instantiate an object of the new type when reading persistent data.
- The metaobject protocol automatically generates code for creating objects of the new type, based on the prevlously-added type information. It further integrates the new code with the existing implementation of the 'object creator' metaobject, compiles the modified implementation, and links it with the application.

6. 动态特性——场景II


Metaobject Protocol
Object Creator
Type-Info Metaobject
newTypeId
newTypeInfo
setPointer
calcSize
setSize
addBase
Base-Info Metaobject
Base-Info Metaobject
calcOffset
setOffset
Base-Info Metaobject
setBase
addData
Data-Info Metaobject
Data-Info Metaobject
setStatic
calcOffset
setOffset
Data-Info Metaobject
setDataMember
addCreationCode
modify

7. 实现

1) 定义应用程序的模型

- Analyze the problem domain and decompose it into an appropriate software structure. Answer the following questions:
  - Which services should the software provide?
  - Which components can fulfil these services?
  - What are the relationships between the components?
  - How do the components cooperate or collaborate?
  - What data do the components operate on?
  - How will the user interact with the software?

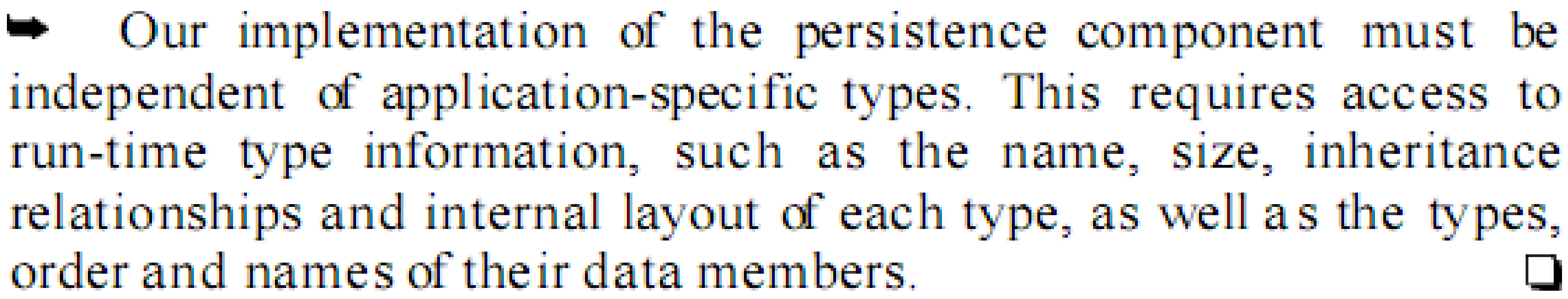
7. 实现

2) 确定变化的行为

- Analyze the model developed in the previous step and determine which of the application services may vary and which remain stable.
  - Real-time constraints, such as deadlines, time-fence protocols and algorithms for detecting deadline misses.
  - Transaction protocols, for example optimistic and pessimistic transaction control in accounting systems.
  - Inter-process communication mechanisms, such as remote procedure calls and shared memory.
  - Behavior in case of exceptions, for example the handling of deadline misses in real-time systems.
  - Algorithms for application services, such as country specific VAT calculation.

7. 实现

3)***Identify structural aspects*** of the system, which, when changed, should not affect the implementation of the base level.

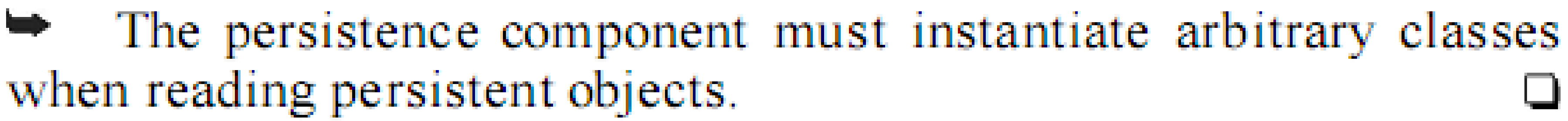


➡ Our implementation of the persistence component must be independent of application-specific types. This requires access to run-time type information, such as the name, size, inheritance relationships and internal layout of each type, as well as the types, order and names of their data members. ❑

7. 实现

4) Identify system services that support both the variation of application services identified in step 2 and the independence of structural details identified in step 3.

- Resource allocation
- Garbage collection
- Page swapping
- Object creation



➡ The persistence component must instantiate arbitrary classes when reading persistent objects. ❑

7. 实现

5) 定义元对象

- 对前三步中明确的每一方面，定义适当的元对象
- Encapsulating behavior is supported by several domain-independent design patterns, such as Objectifier. Strategy. Bridge, Visitor, and Abstract Factory
- E.g.
  - 函数调用机制的元对象：策略对象
  - 组件的多重实现：桥接模式
  - 状态信息：状态模式

7. 实现

5) 定义元对象

```
class type—info {
    //...
private:
    type—info(const type—info& rhs);
    type—info& operator=(const type—info& rhs);
public:
    virtual       -type—info0 ;
    int           operator-=(const type—info& rhs) const;
    int           operator!=(const type—info& rhs) const;
    int           before(const type—info& rhs) const;
    const char*   name() const;
};
```

```
class extTypeInfo {
    // ...
public:
    const bool     isBuiltIn() const;
    const bool     isPointer() const;
    const size—t   size() const;
    baseIter*      bases(int direct = 0) const;
    dataIter*      data(int direct = 0) const;
};
```

```
class BaseInfo {
    // ...
public:
    const type_info*  type() const;
    const long        offset() const;
};
```

```
class DataInfo {
    // ...
public:
    const char*       name() const;
    const type_info*  type() const;
    const bool        isStatic() const;
    const long        offset0 const;
    const long        address() const;
};
```

7. 实现

6) 定义元对象协议

- Support a defined and controlled modification and extension of the meta level, and also a modification of relationships between base-level components and metaobjects.
  - Integrate it with the metaobjects. Every metaobject provides those functions of the metaobject protocol that operate on it.
  - Implement the metaobject protocol as a separate component.

7. 实现

6) 定义元对象协议

- 例子中的情况
  - 提供了一个类MOP（单件对象）

```
const type_info* getInfo(char* typeName) const;
const extTypeInfo* getExtInfo(char* typeName) const;
```

```
void newTypeId(char* typeName);
void newTypeInfo(char* typeName,
             bool builtIn, bool pointer);

void deleteInfo(char* typeName);

void addBase(char* typeName, char* baseName);
void addData(char* typeName,
             char* memberType,char* memberName);
void deleteBase(char* typeName, char* baseName);
void deleteData(char* typeName, char* memberName);

void addCreationCode(char* typeName);
void deleteCreationCode(char* typeName);
```

7. 实现

6) 定义元对象协议

- 例子中的情况

```
void MOP::addBase(char* typeName, char* baseName) {
     BaseInfo* base;
     // Is extended type information for type typeName
     // and type information for type baseName available?
     if ((!eMap.element(typeName)) ||
                             (!tMap.element(baseName)))
          // error handling ...

     // Instantiate the baseInfo object for type baseName
     base = new BaseInfo(tMap[baseName]);
     // Calculate the offset of the base class.
     base->baseoffset = calcOffset(typeName, baseName);
     // Add the new baseInfo object to the list of
     // bases within the extTypeInfo object for
     // type typeName
     eMap[typeName]->baseList.add(base);
}
```

7. 实现

7) 定义基本层次

- Implement the functional core and user interface of the system according to the analysis model developed in step 1.

7. 实现

```
void Persistence::storeObject
                     (void* object, char* typeName) {
    type—info*           objectId;
    extTypeInfo*         objectInfo;
    baseIter*            iterator;

    // Get type information about the object to be stored
    objectId        = mop->getInfo(typeName);
    objectInfo      = mop->getExtInfo(typeName);
    iterator        = objectInfo->data();

    // Mark the object to avoid storing duplicates
    markObject(object);

    // Object is of type char*?
    else if (!strcmp("char*", objectId->name()))
        storeString(object);

    // Object is a pointer != NULL?
    // *(char**)object means that we interpret the
    // generic pointer object as a pointer to an address
    else if ((objectInfo->isPointer()) &&
                       (!(*(char**)object)))
        // Dereference the pointer
        storeObject(*(char**)object,
            iterator->curr()->type()->name());

    // Object is a user-defined type with data members
    else while (!iterator->atEnd()) {
        // If not marked, store the data member,
        // else store the marker
        if (!marked( (char*)object +
                iterator->curr()->offset()))
            storeObject((char*)object +
                iterator->curr()->offset(),
                iterator->curr()->type()->name());
        else
            storeMarker((char*)object +
                iterator->curr()->offset());

        iterator->next();
    };
    delete iterator;
};
```

8. 已解决的例子

- 例子中的情况
  - C++并不能很好地支持映像
  - 考虑Java和C#

9.变体

- ***带几个元层次的映像***
- ***Refection with several meta levels.***
  - Sometimes metaobjects depend on each other. For example, consider the persistence component. Changes to the run-time type information of a particular type requires that you update the 'object creator' metaobject. To coordinate such changes you may introduce separate metaobjects, and-conceptually-a meta level for the meta level, or in other words, a meta meta level. In theory this leads to an infinite tower of reflection

10.已知应用

- **CLOS**
  - The system first determines the methods that are applicable to a given invocation.
  - It then sorts the applicable methods in decreasing order of precedence.
  - The system finally sequences the execution of the list of applicable methods. Note that in CLOS more than one method can be executed in response to a given invocation.

10.已知应用

- MIP
  - The first layer includes information and functionality that allows software to identify and compare types. This layer corresponds to the standard run-time type identification facilities for C++.
  - The second layer provides more detailed information about the type system of an application. For example, clients can obtain information about inheritance relationships for classes, or about their data and function members. This information can be used to browse type structures.
  - The third layer provides information about relative addresses of data members, and offers functions for creating 'empty' objects of user-defined types. In combination with the second layer, this layer supports object I/O.
  - The fourth layer provides full type information, such as that about friends of a class, protection of data members, or argument and return types of function members. This layer supports the development of flexible inter-process communication mechanisms, or of tools such as inspectors, that need very detailed information about the type structure of an application. The metaobject protocol of MIP allows you to specify

10.已知应用

- **PGen** is a persistence component for **C++** that is based on MIP. It allows an application to store and read arbitrary **C++** object structures.

10.已知应用

- **NEDIS.**
  - Properties for certain attributes of classes, such as their allowed value ranges.
  - Functions for checking attribute values against their required properties. NEDIS uses these functions to evaluate user input, for example to validate a date.
  - Default values for attributes of classes, used to initialize new objects.
  - Functions specifying the behavior of the system in the event of errors, such as invalid input or unexpected 'null' values of attributes.
  - Country-specific functionality, for example for tax calculation.
  - Information about the 'look and feel' of the software, such as the layout of input masks or the language to be used in the user interface.

10.已知应用

- **OLE** 2.0 provides functionality for exposing and accessing type information about OLE objects and their interfaces.

11. 效果——优点

- ***No explicit modification of source code*.** You do not need to touch existing code when modifying a reflective system. Instead, you specify a change by calling a function of the metaobject protocol.
- ***Changing a software system is easy*.** The metaobject protocol provides a safe and uniform mechanism for changing software.
- ***Support for many kinds of change*.** Metaobjects can encapsulate every aspect of system behavior, state and structure.

11. 效果——不足

- ***Modifications at the meta level may cause damage***. Even the safest metaobject protocol does not prevent users from specifying incorrect modifications.
- ***Increased number of components***. It may happen that a reflective software system includes more metaobjects than base-level components.
- ***Lower efficiency***. Reflective software systems are usually slower than non-reflective systems.
- ***Not all potential changes to the software are supported.***
- ***Not all languages support refection***