

Re-engineering Software Architecture of Home Service Robots: A Case Study

Moonzoo Kim, Jaejoon Lee, Kyo C. Kang
Software Engineering Laboratory, POSTECH

Youngjin Hong, Seokwon Bang
Samsung Advance Institute of Technology

May, 2005

Pohang University of
Science and Technology
(POSTECH)



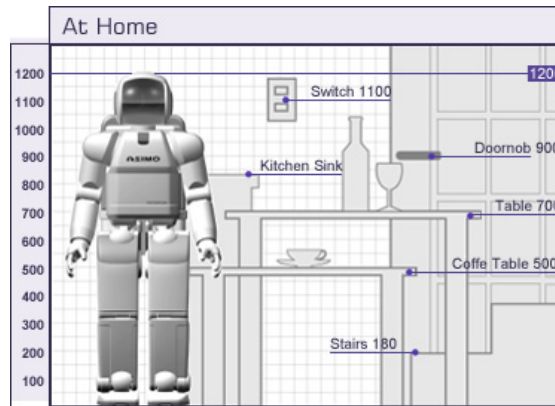
Copyright © 2005
SE Lab., Dept. of CSE
POSTECH, R.O.Korea



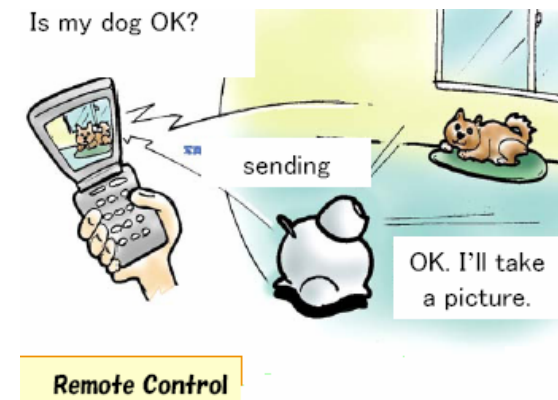
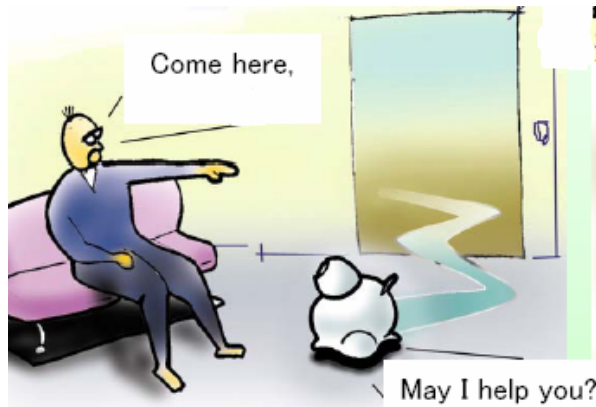
- Introduction
- Re-engineering Software Architecture
- Experimental Results
- Lessons Learned



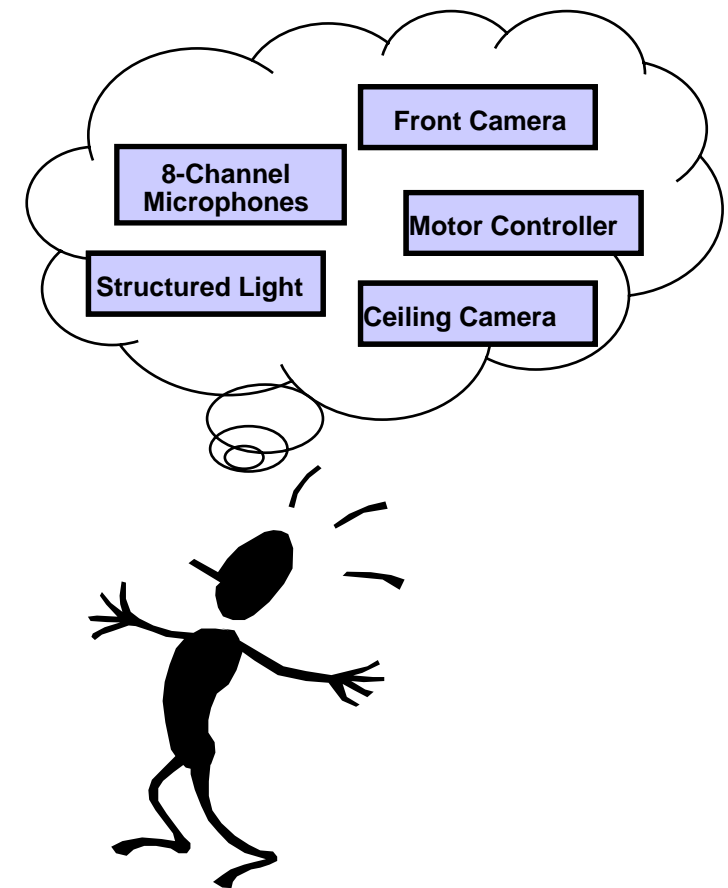
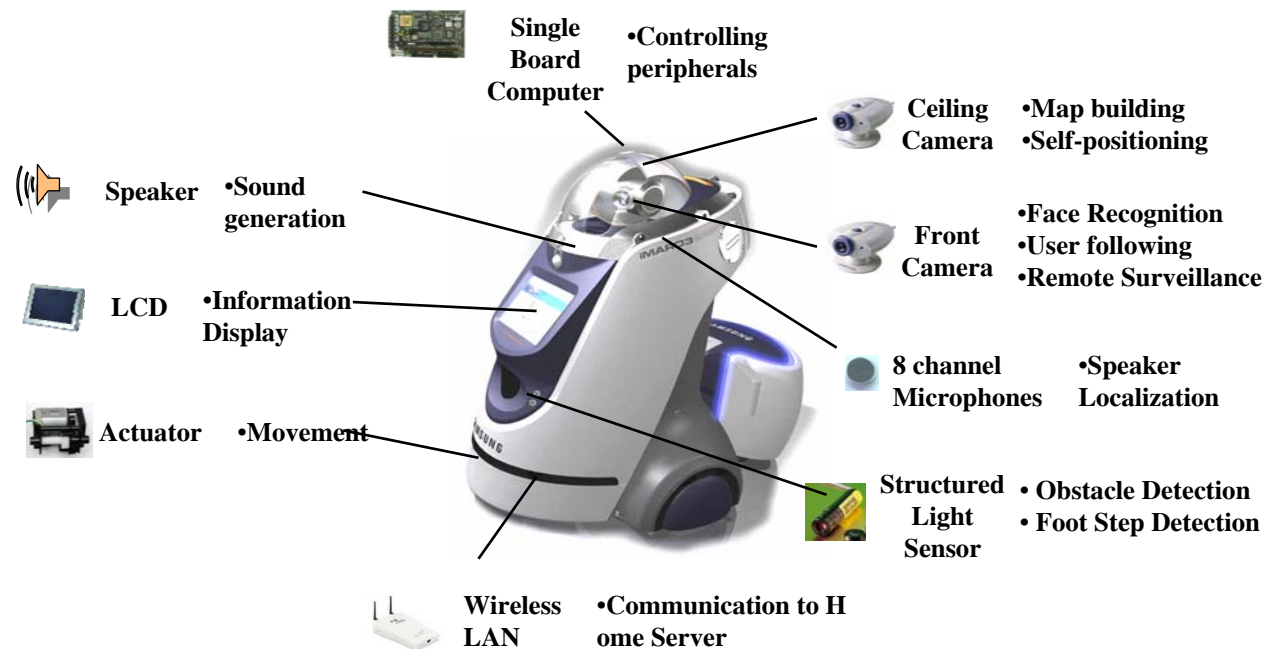
- Designed for providing various services to human user
 - Service areas : home security, patient caring, cleaning, etc
 - Markets for home service robots are still being formed



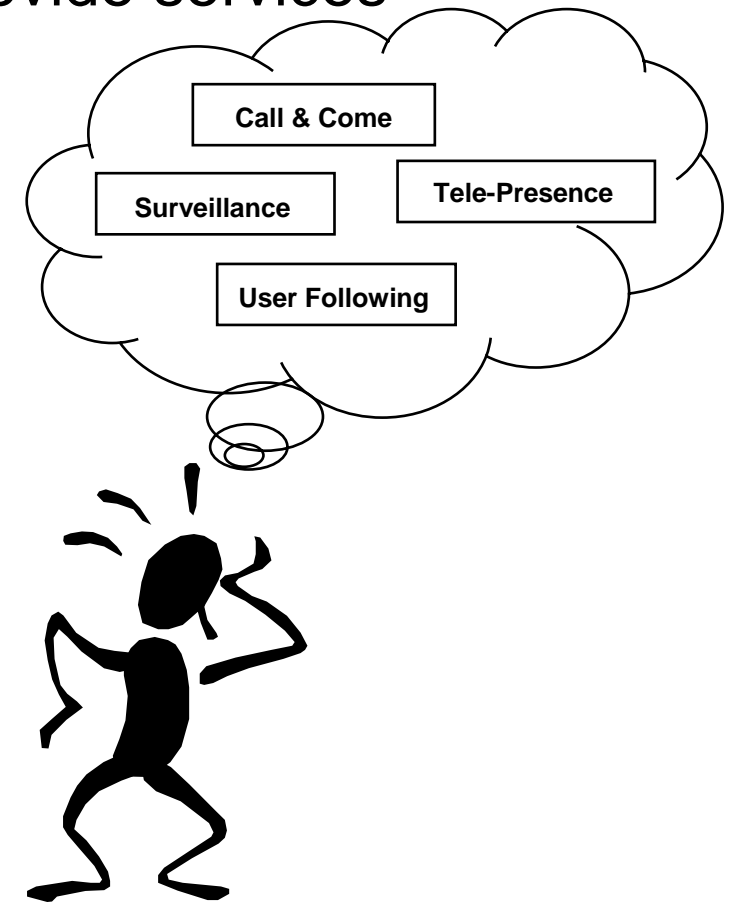
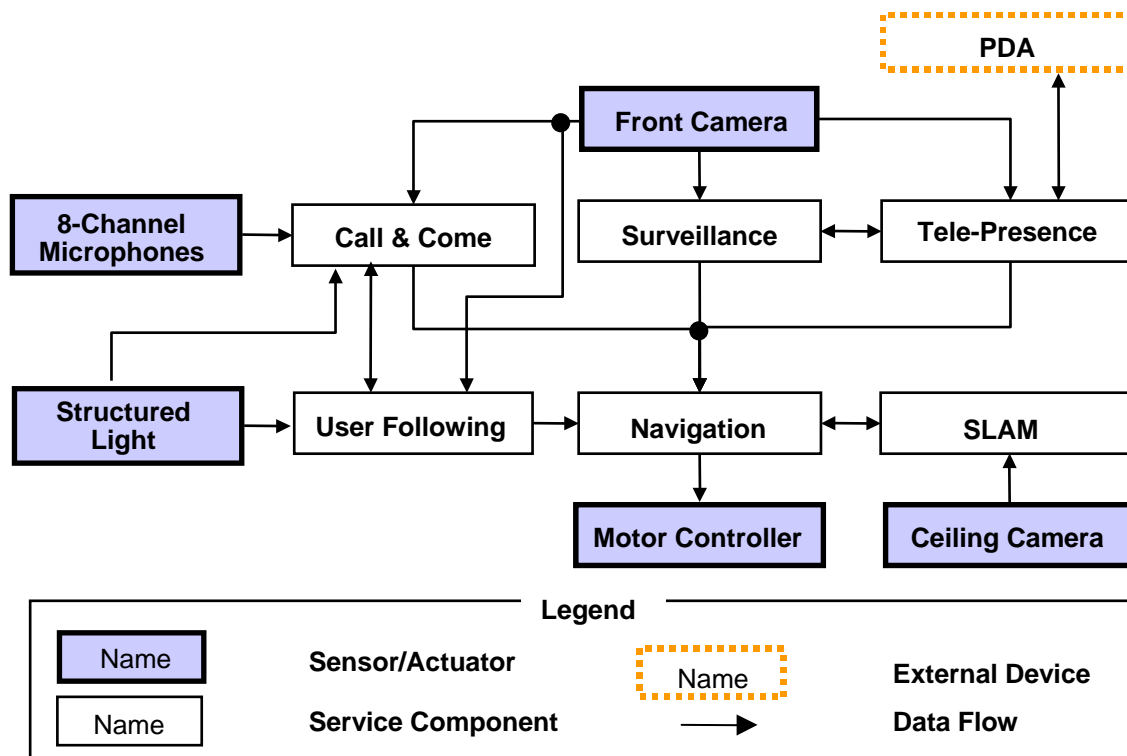
* The above heights are examples to serve as a reference(mm).



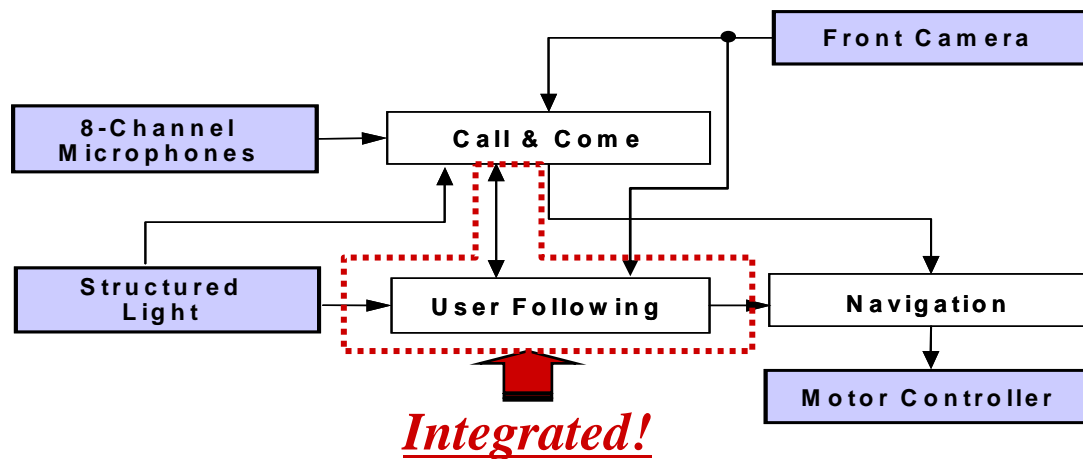
- Robots are created based on various **technical components**
 - Speech recognizer, vision recognizer, actuator, etc



- Robot developers concentrate on technical components only, resulting in **integration in an ad-hoc and bottom-up way**
 - Difficult to coordinate components to provide services

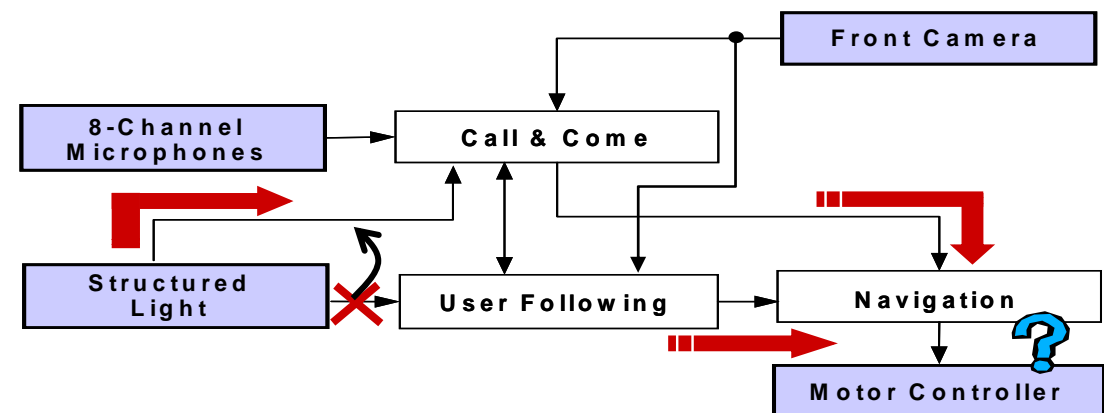


- This development practice easily results in **feature interaction problem**.



- CC had higher priority for SL
- UF had higher priority for Navigation

- A feature interaction problem had occurred when CC was activated during the UF service.



■ Problems due to bottom-up integration

- Lack of global view
- Difficulty in analyzing the behavior of integrated systems
- Integration often requires modifications of other components

■ Feature interaction problems

- Invisible interactions between the components
- Difficulty to trace the cause of problems (debugging difficulty)

Cannot develop products in reasonable project time



Cannot evolve according to quickly changed market demands

Cannot satisfy required quality attributes (e.g. safety and temporal properties)



- To provide **hierarchical and modular SA**
 - Top-down global views
 - Visualization of component interactions
 - High adaptability for evolving features/ technologies

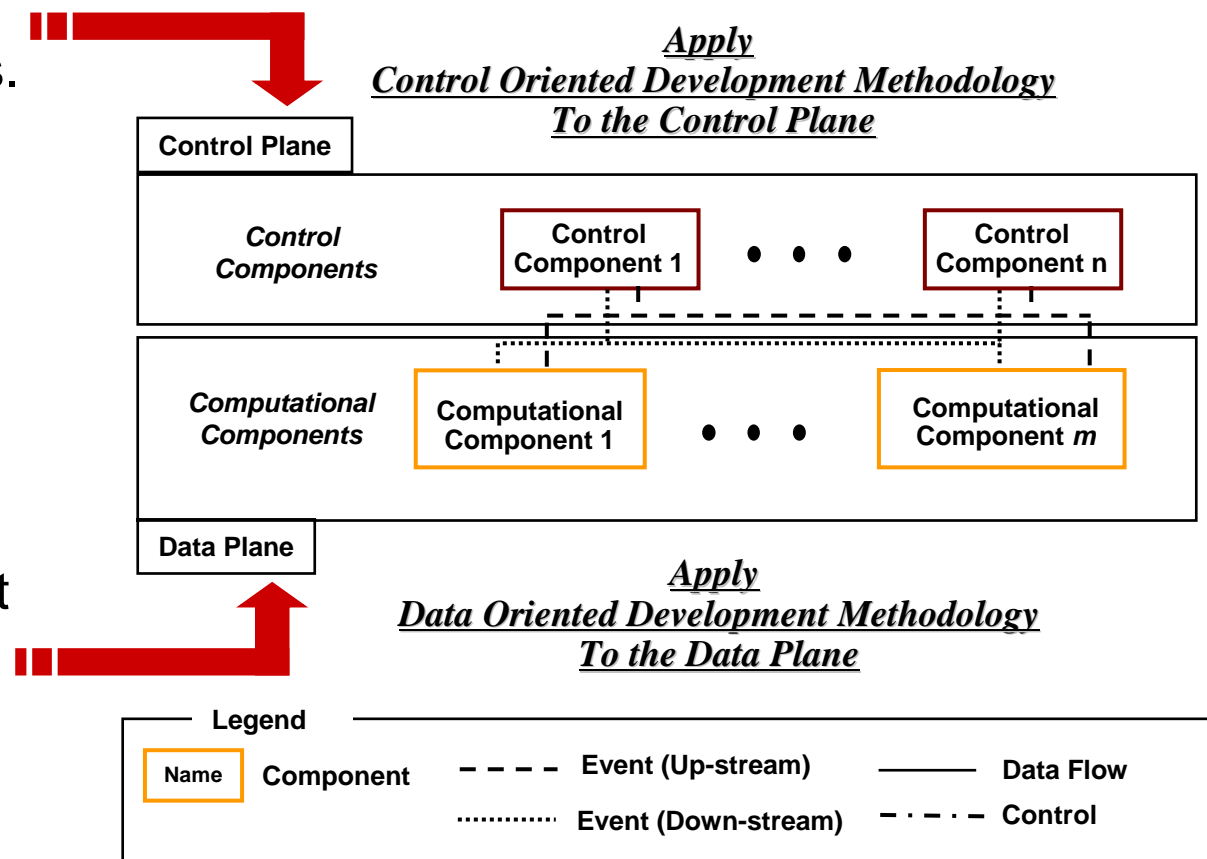
- Re-engineering based on the following **principles**
 1. Separation of control plane from computational plane
 2. Distinction between global behavior and local behavior
 3. Layering in accordance with data refinement hierarchy



■ Principle 1: Separation of Control Components from Computational Components.

The first class of data is **control data** for handling robot behaviors.
 : correctness is the foremost concern due to complexity of reactive system.

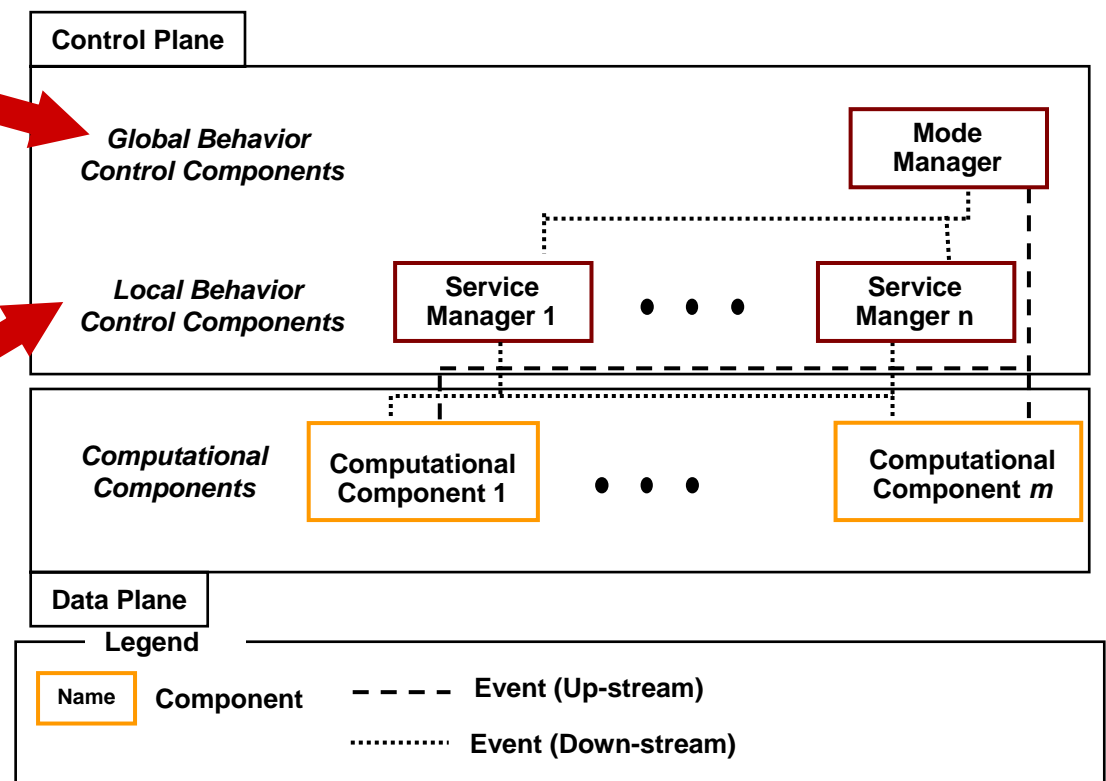
The second class of data is **computational data** for handling robot function.
 : efficient computation is the most important goal.



■ Principle2: Separation of Local Behaviors from Global Behaviors

Mode manager components defines the system modes and the interaction policy between service components.

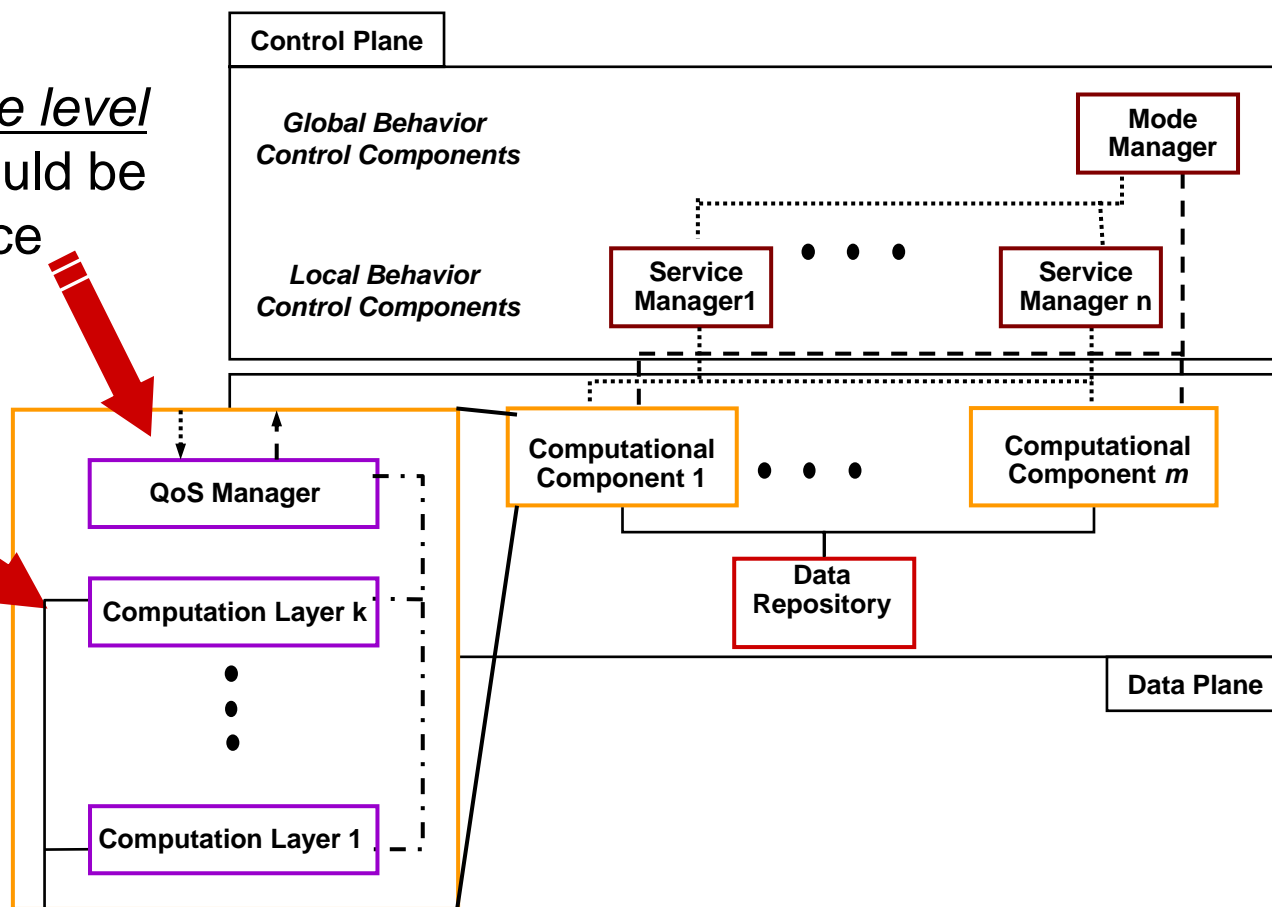
Service manager components defines the behavior of service feature by controlling the computational components.



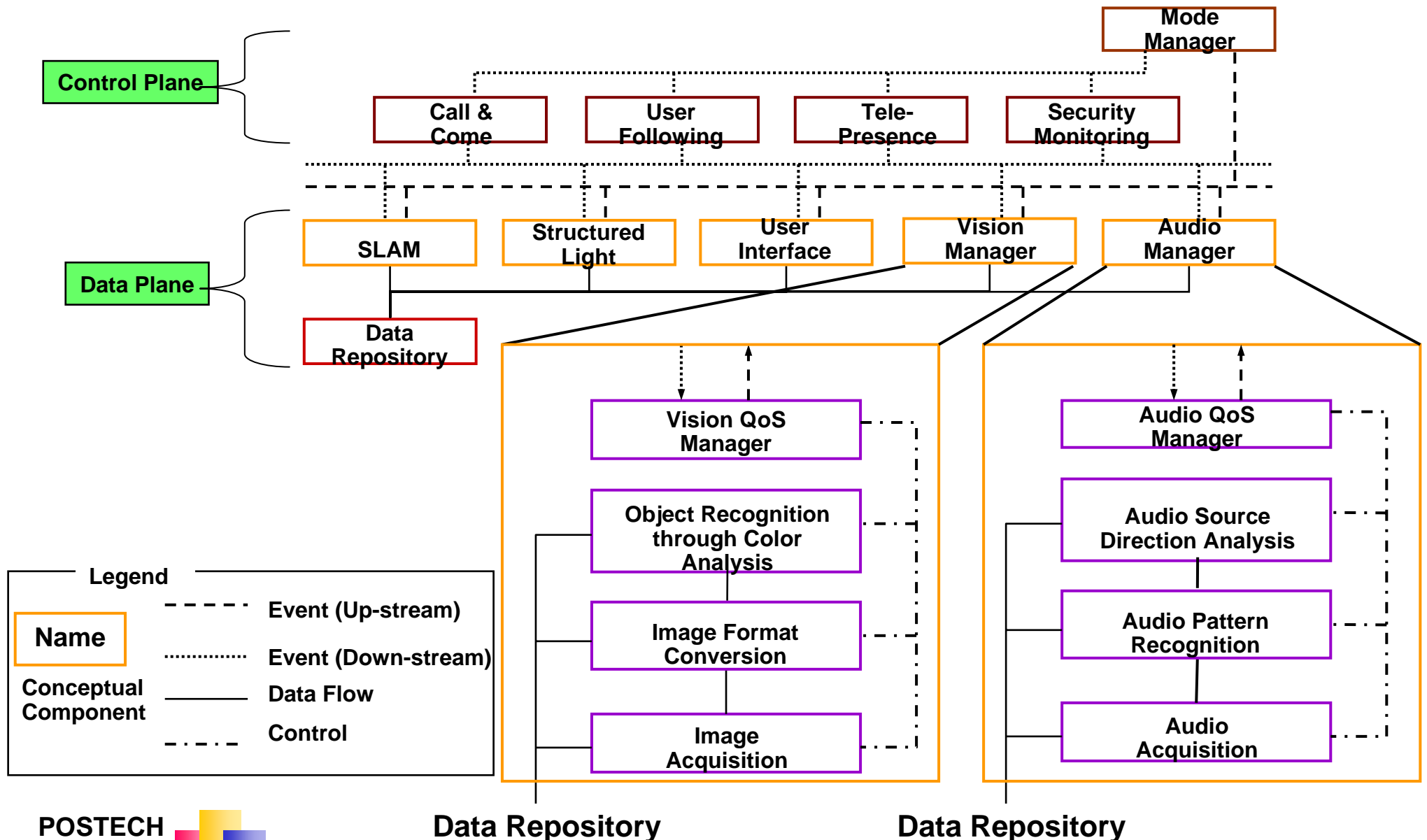
■ Principle3: Layering in Accordance with Data Refinement Hierarchy

QoS Manager determines *the level at which the computation* should be performed according to service

There exist **data refinement hierarchy** for data computation and different service features may use *different computational component layers*.



New Software Architecture *Re-engineering Software Architecture*



■ Previous Implementation of the CC service

- A main function ProcessState() is called every 100 milliseconds.
- CC executes through sequential steps to allow preemption

```
01: class CCallComeDlg {
02:     int m_order;
03:     ...
04:     void processState() {
05:     ...
06:         switch(m_order) {
07:             case 0: STOP();
08:                 m_order++;
09:                 break;
10:             case 1: ROTATE();
11:                 m_order++;
12:                 break;
13:             case 2: static int nCount = 0;
14:                 if (abs(m_betaO-curO) == 0) nCount++;
15:                 else nCount = 0;
16:                 if (nCount > 2) m_order++;
17:                 break;
18:             ...
19:             case 9: CALL_N_COME_FINISHED();
20:                 m_order = -1;
21:                 break;
22:         } /* End of processState()
23:     }
```

This pattern of reactive programming is a straight forward way to allow preemption in C++, but it is, **at least, error prone.**

“static int nCount = 0”
Assume that nCount is declared as a static local variable at line 13. What happen?



■ Skeleton Esterel Code for Control Plane

```
01:module control_plane: % Control Plane
02:input EVENT: integer;
03:output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
04:signal CALL_COME, CALL_STOP in
05:run mode_man||run cnc||run uf||run tp||run sm;
06:end signal
07:end module
08:
09:module cnc: % Call and Come service
10:function human_in_range() : boolean;
11:input CALL_COME,CALL_STOP; %come,stop commands
12:output STOP,ROT,GO,CC_DONE,CS_DONE,DET,N_DET;
13:var mv:=false:boolean,n:integer in
14:  every immediate [CALL_COME or CALL_STOP ] do
15:    present
16:      case CALL_COME do % come command
17:        mv := true;
18:        emit STOP; pause;
19:        run rot_det;
20:        ...
21:        emit CC_DONE;pause;
22:      case CALL_STOP do % stop command
23:        emit STOP;
24:        if mv=true then emit CS_DONE;
25:        else mv:=true;pause;run rot_det end if;
26:      end present;
27:      mv := false;
28:    end every
29:end var
30:end module
31:...
```

Esterel implementation

- Reactive operators
- Explicit communication between component through signals
- Generates C code with seamless integration with existing C code

Esterel **handles a preemptive event e** by a primitive operator
EVERY e DO statements
END EVERY (line 14 to line 28).



Layered Implementation of Vision Manager

- The **layered architectural pattern** is organized based on the data refinement hierarchy.

Interface

```
class Layer3 {
protected :
    Layer2 *lowerLayer;

public :
    virtual bool L3Service()= 0;
    void setLowerLayer(Layer2 *l){
        lowerLayer = l; }
}
```

```
class Layer2 {
protected :
    Layer1 *lowerLayer;

public :
    virtual bool L2Service()= 0;
    void setLowerLayer(Layer1 *l){
        lowerLayer = l; }
}
```

Implementation

```
class Vision_L3_FaceRecognition
: public Layer3 {
public :
    virtual bool L3Service()
    {
    {...
        if(lowerLayer->L2Service()){
            ...
            if(m_faceRec.Rec()){
                DR::setData(m_facePattern);
            }
        }
    }
}
```

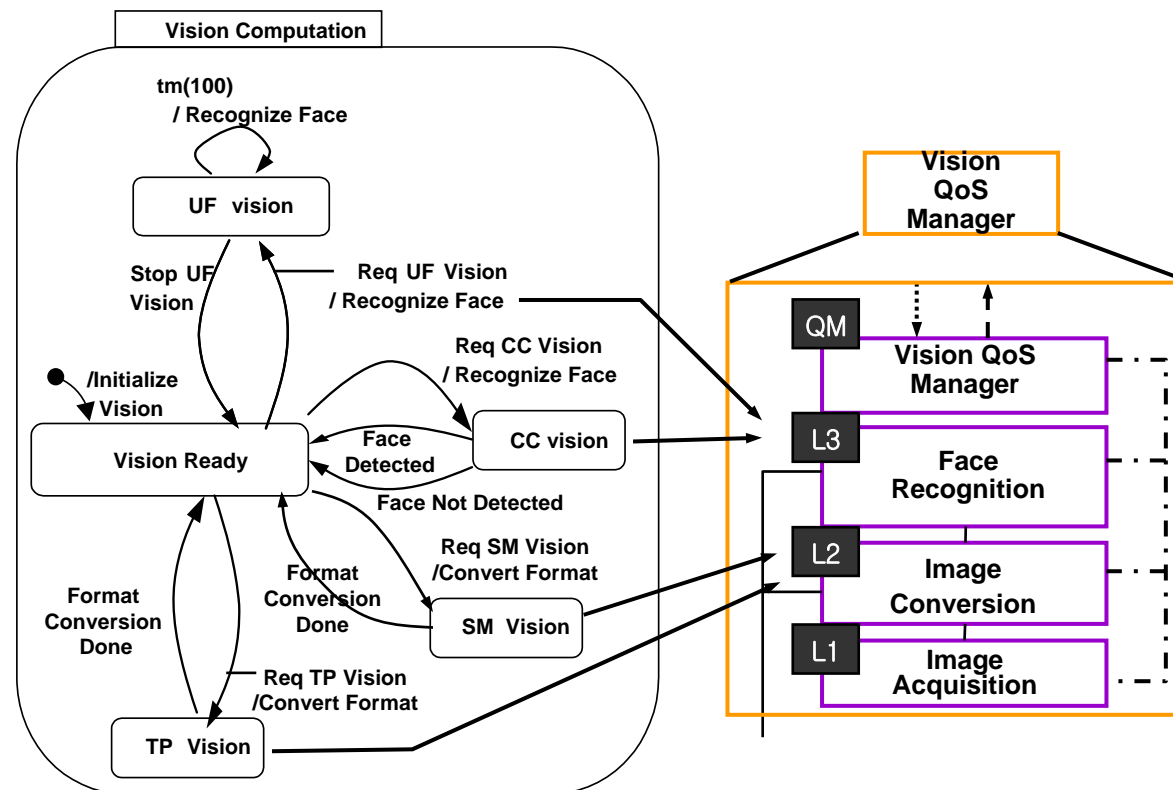
```
class Vision_L2_FormatConversion
: public Layer2 {
public :
    virtual bool L2Service()
    {
    {...
        if(lowerLayer->L1Service()){
            ...
            if(m_frmtConversion.Conv()){
                DR::setData(m_imgFormat);
            }
        }
    }
}
```

1. Image data from the front camera are captured (**Layer 1**),
2. then converted into a file format (**Layer 2**)
3. finally a human face is identified by analyzing colors in the file (**Layer 3**).

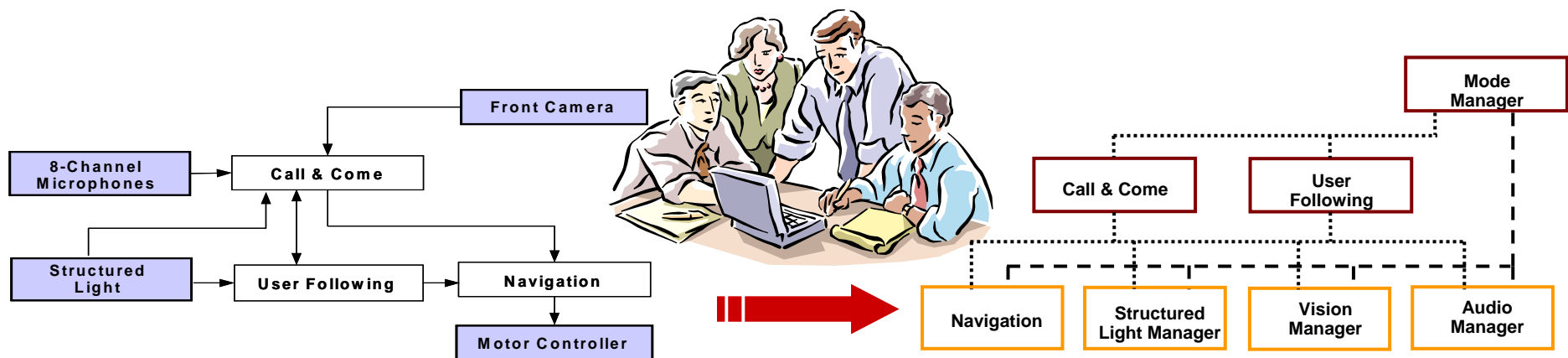


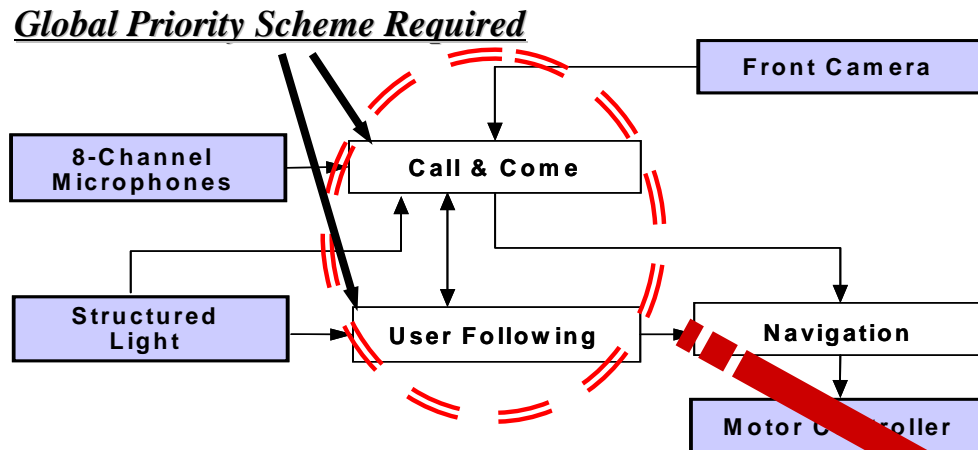
Vision QoS Manager

- The QoS manager layer selects the 'right' level of data refinements.



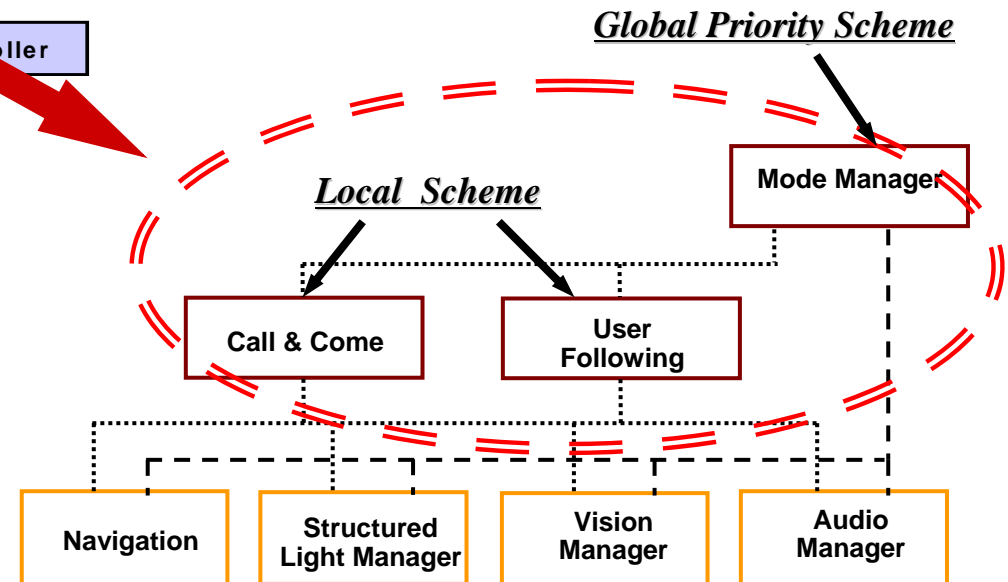
- From the experience of re-engineering SHR100, we are convinced that **re-engineering is essential**
 - Due to the limited development time, developers tend to concentrate only on **technical components at the early state** without considering how they will be integrated.
 - Once feasibility of the project is confirmed through an early prototype, **re-engineering the product at later stage** should be enforced for increased quality of the product.





- We found that unclear global priority scheme was one of the primary causes of feature interaction problems.

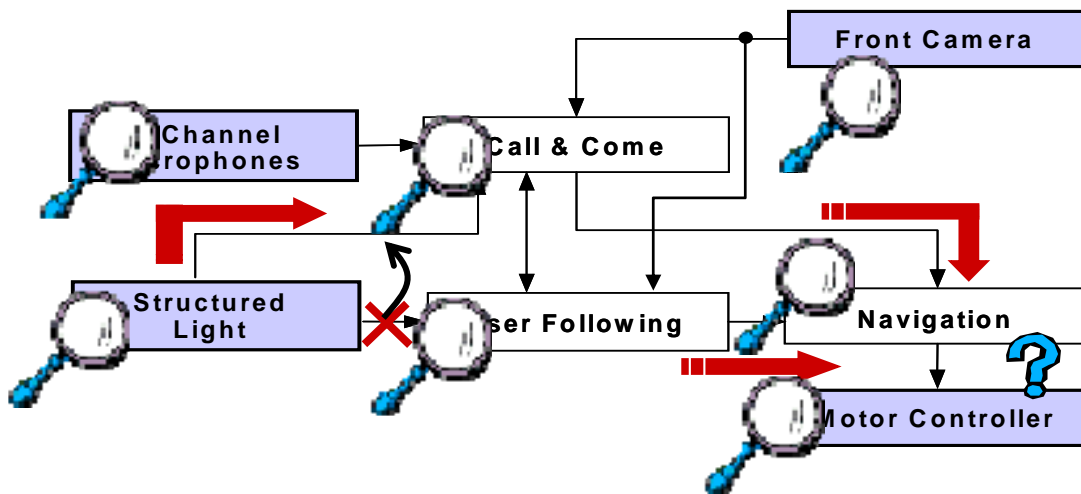
-With the new architecture, the **global priority scheme is separated** from the service components and manageability of priority was increased drastically.



Needs of Monitoring Capability

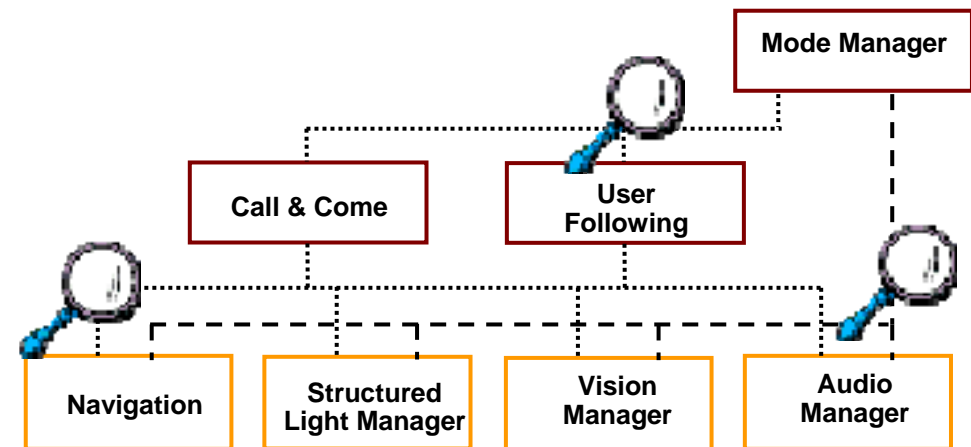
Lessons Learned

- A monitoring capability is an important aid for tracking down possible sources of a problem.



-Determining where to put probes is difficult, if the role of each component and the ways they interact each other are not clear

-The new SA that we proposed could alleviate this difficulty with **clear interaction strategy between components**

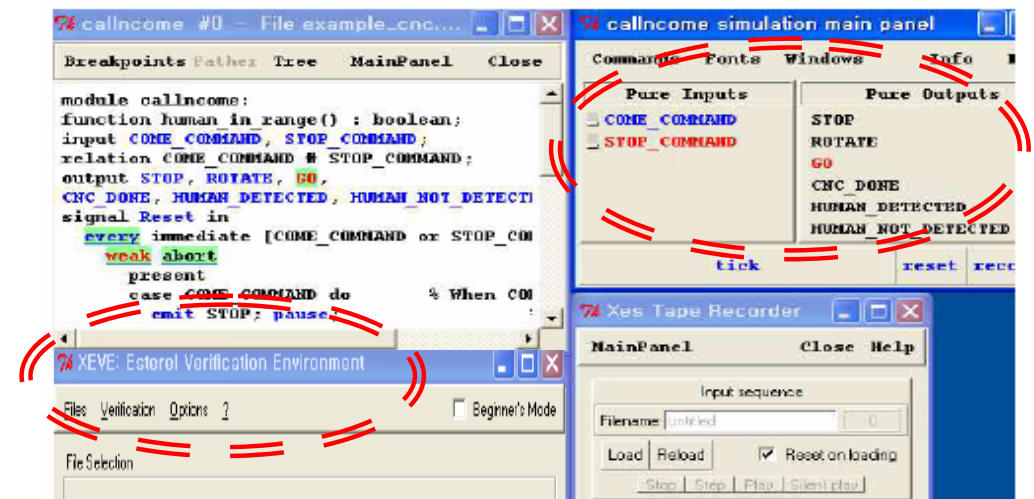


```

m_bHuman = FALSE;
MY_TRACE("[CALL AND COME] Mn %d",
switch (param.order)
{
case 0:
    if (m_pSLMain)
    {
        MY_TRACE("[SSL Proce:
        m_pSLMain->SetProces:
    }
    /**/
    m_tracker.cmdTurn(body.d
    param.order++;
    MY_TRACE("[pInCallCome:T
}
break;
case 1:
    static CPose befPose;
    static int nCnt = 0;
    pose curPose = m_tracker
    
```

- We uncovered subtle bugs which decrease the accuracy of detecting a user
- Implementing preemption in C++ is error prone.

- Esterel enables **clear interactions among the components**
- Esterel has **formal semantics** as Mealy machine, which allows rigorous analysis such as model checking



■ A Case Study of Re-engineering Home Service Robot

- Based on the three engineering principles, we designed a new SA and re-engineered existing source code.
- We used Esterel for re-engineering the control plane of SHR100.
- By this re-engineering, interactions among the components became visible and the responsibility of behaviors could be assigned to components clearly.

■ Future work

- Resource management problem
- Guideline for reverse-engineering

