Component composition through architectural patterns for problem frames

Christine Choppy
Christine Choppy(AT)lipn.univ-paris13.fr
Université Paris 13, LIPN

joint work with Maritta Heisel and Denis Hatebur

University Duisburg-Essen – Faculty of Engineering
Department of Computer Science
Workgroup Software Engineering

Overall Goal / Context

- Define pattern-based software development process
- Use different kinds of patterns in the different phases
- Provide well-defined transition between phases

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure
- introduced on the level of detailed object oriented design, now defined for different activities.

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure
- introduced on the level of detailed object oriented design, now defined for different activities.
- *Problem Frames* (Jackson) classify software development problems,

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure
- introduced on the level of detailed object oriented design, now defined for different activities.
- *Problem Frames* (Jackson) classify software development problems,

  Architectural styles/ "architectural patterns" characterise software architectures

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure
- introduced on the level of detailed object oriented design, now defined for different activities.
- *Problem Frames* (Jackson) classify software development problems,
  Architectural styles/ "architectural patterns" characterise software architectures
  Design Patterns for finer-grained software design,
  frameworks less abstract, more specialised. *idioms/"code patterns"*: low-level patterns related to specific programming languages

Patterns

- a promising approach to software development, a means to reuse software development knowledge on different levels of abstraction, classify sets of software development problems or solutions that share the same structure
- introduced on the level of detailed object oriented design, now defined for different activities.
- *Problem Frames* (Jackson) classify software development problems,

  Architectural styles/ "architectural patterns" characterise software architectures

  Design Patterns for finer-grained software design, *frameworks* less abstract, more specialised. *idioms/"code patterns"*: low-level patterns related to specific programming languages
- construct software in a systematic way, body of accumulated knowledge, not starting from scratch

Problem Frames (M. Jackson)

- *problem frames* concept: present, classify, understand software development problems

Problem Frames (M. Jackson)

- *problem frames* concept: present, classify, understand software development problems
- characterisation of a class of problems in terms of their main components and the connections between these components

Problem Frames (M. Jackson)

- *problem frames* concept: present, classify, understand software development problems
- characterisation of a class of problems in terms of their main components and the connections between these components
- Once a problem is successfully fitted to a problem frame, its most important characteristics are known

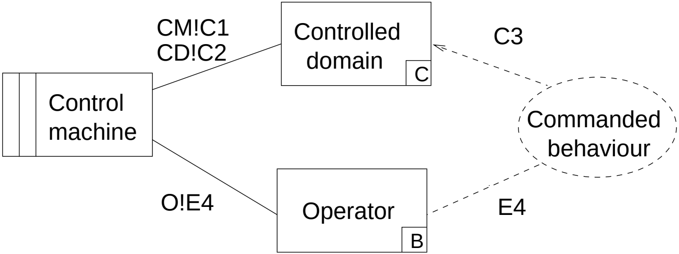
Problem Frames (M. Jackson)

- *problem frames* concept: present, classify, understand software development problems
- characterisation of a class of problems in terms of their main components and the connections between these components
- Once a problem is successfully fitted to a problem frame, its most important characteristics are known
- diagram: involved domains, requirements, design, interfaces

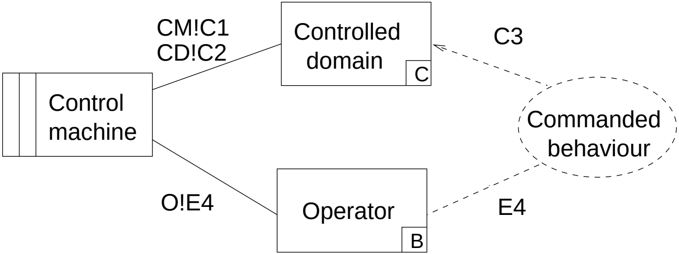
Problem Frames (M. Jackson)

- *problem frames* concept: present, classify, understand software development problems
- characterisation of a class of problems in terms of their main components and the connections between these components
- Once a problem is successfully fitted to a problem frame, its most important characteristics are known
- diagram: involved domains, requirements, design, interfaces
- five basic problem frames, variants

The Commanded Behaviour frame



The Commanded Behaviour frame



There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.

Basic Approach

- Decompose problems into simple subproblems, fitting to *problem frames*.
- Problem frames are only useful if we know how to *solve* the problems fitting to them.
- Hence, we defined *architectural patterns* for each problem frame.
- These are used to set up a solution structure for each subproblem.
- Important: compose solutions to subproblems on the *architectural level*, not on the code level.
- Use *relations between subproblems* to construct the composed solution structure.

Layered Software Architectures

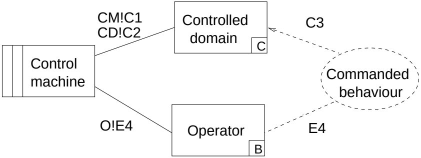
HAL Hardware Abstraction Layer: consists of drivers for external components; needed for portability

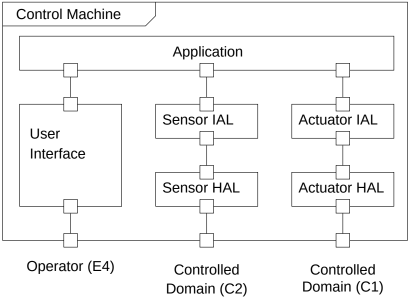
IAL Interface Abstraction Layer: provides input data or accepts output data, respectively

Application Layer: computes output data from input data

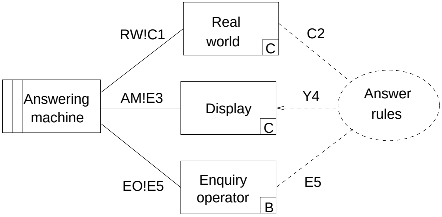
We represent the architectural patterns as UML composite structure diagrams.

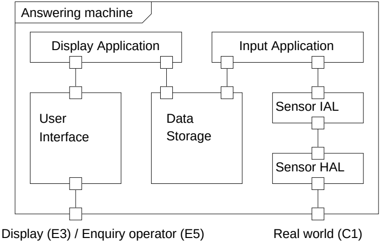
Commanded Behaviour
Frame Diagram and Architectural Pattern



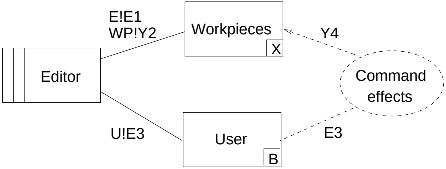


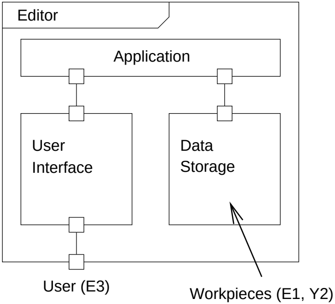
Commanded Information
Frame Diagram and Architectural Pattern





Workpieces
Frame Diagram and Architectural Pattern





Note that there is only one interface with the environment.

Procedure

- Instantiate Problem Frames
- Express dependencies / relationships between problem diagrams
- Instantiate corresponding architectural pattern for each Problem Frame
- Merge architectures

Express subproblem dependencies / relationships

Relationships between subproblems:

- parallel
- sequential
- alternative
- . . .

Possible notation: grammars

Instantiate Architectural Patterns

- If a subproblem fits to a known problem frame, then a simple instantiation of the pattern will suffice.
- If a subproblem is not an exact instance of a problem frame but a variant, then modifications of our architectural patterns will be necessary.
- If a subproblem is unrelated to any problem frame, then an appropriate architecture has to be developed from scratch.
- For each interface contained in a subproblem architecture, the corresponding operations or signals, respectively, have to be defined, and provided and required interfaces must be distinguished

Merge Architectures I

Decide if two components contained in different subproblem architectures should occur only once in the global architecture, i.e., they should be merged. Distinguish the following cases:

1. The components are hardware (HAL) or interface abstraction layers (IAL), establishing the connection to some hardware device.
   Such components should be merged if and only if they are associated to the same hardware device.
2. Two application components belong to subproblems being related sequentially or by alternative.
   Such components should be merged into one application component.

Merge Architectures II

3. Two application components belong to parallel subproblems and share some output phenomena.
   Such components should be merged, because the output must be generated in a way satisfying both subproblems.
4. Two application components belong to parallel subproblems and share some input phenomena.
   If the components do not share any output phenomena, both alternatives (merging the components, or keeping them separate) are possible. If the components are not merged, then the common input must be duplicated.
5. Two application components belong to parallel subproblems and do not share any interface phenomena.
   Such components should be kept separately.

Requirements for ATM

R1 To use the ATM a valid pin and a bank card is required. (*Authenticate*)

R2 The withdrawal should be refused when the request is bigger than the balance. (*Request*)

R3 The card should be retracted if the customer does not take the ejected card. (*TakeCard*)

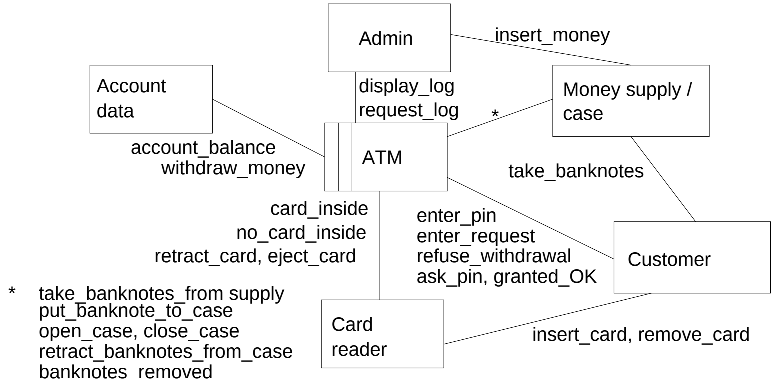
R4 The account is updated when the customer takes the money. (*UpdateAccount*)

R5 After the withdrawal was granted and the card ejected, the money should be taken from the supply, put to the money case, and the case should be opened. After the customer took the money, the money case should close, otherwise the money should be retracted. (*TakeMoney*)

R6 All input phenomena should be logged. (*Log*)

R7 The logged input phenomena can be queried by the administrator. (*DisplayLog*)

Context Diagram for ATM

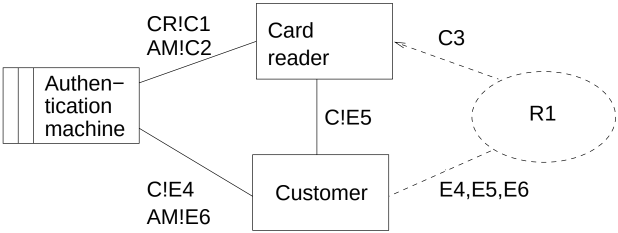


Subproblem dependencies / relationships

```
<start> ::= (<idle> || Log || DisplayLog)
<idle> ::= (Authenticate <authenticated> | Authenticate <idle>)
<authenticated> ::= (Request <granted> | Request <refused>)
<granted> ::= (TakeCard <granted_no_card> | TakeCard <idle>)
<refused> ::= TakeCard <idle>
<granted_no_card> ::= (UpdateAccount || TakeMoney)[1] <idle>
```

[1] Both react to the signal "takeBanknotes"

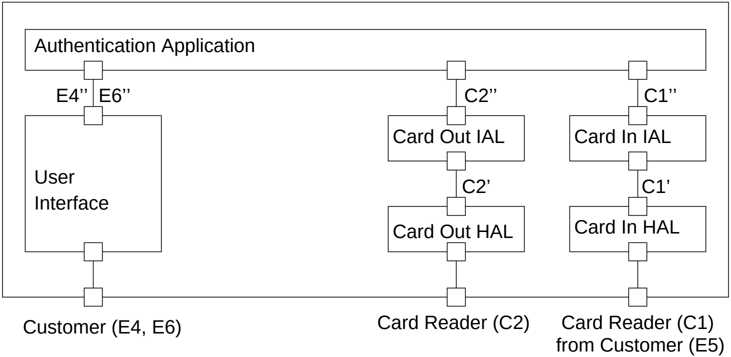
Problem Diagram for Authenticate (commanded behavior variant)



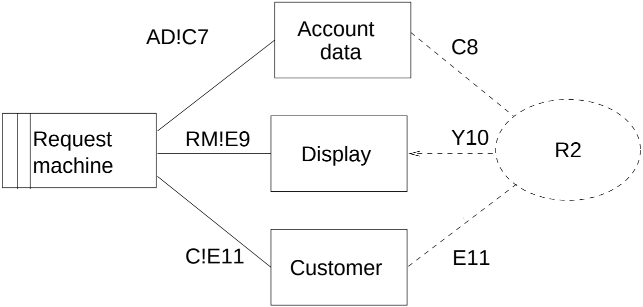
C1: $\{card\_inside\}$
C2: $\{retract\_card\}$
C3: $\{card\ control\}$

E4: $\{enter\_pin\}$
E5: $\{insert\_card\}$
E6: $\{ask\_pin\}$

Architecture for Authenticate



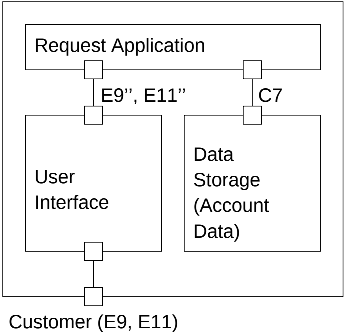
Problem Diagram for Request (commanded information)



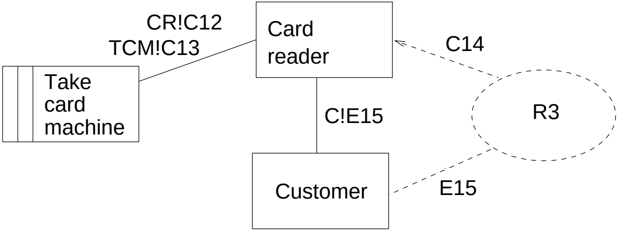
C7: $\{account\_balance\}$
C8: $\{account\_data\}$
E9: $\{granted\_OK,$
$refuse\_withdrawal\}$

Y10: $\{withdrawal\_possible\}$
E11: $\{enter\_request\}$

Architecture for Request



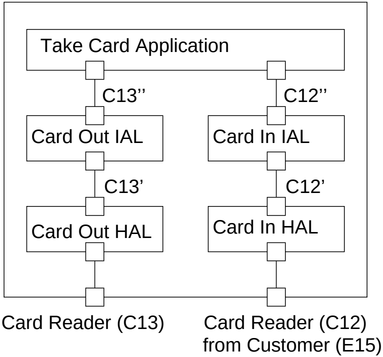
Problem Diagram for Take Card (required behavior variant)



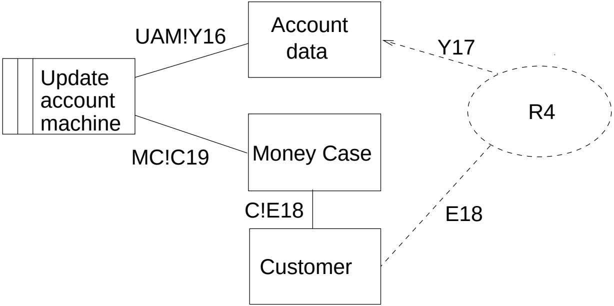
C12: $\{card\_inside\}$
C13: $\{eject\_card, retract\_card\}$

C14: $\{eject, \ retract\}$
E15: $\{take\_card\}$

Architecture for Take Card

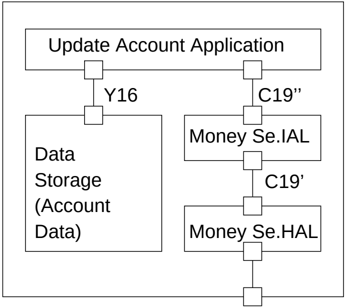


Problem Diagram for Update Account (workpieces variant)

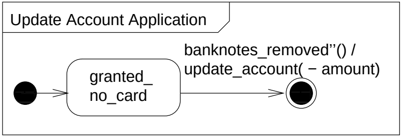


Y16: {*update_account*} E18: {*take_banknotes*}
Y17: {*account_data*} C19: {*banknotes_removed*}

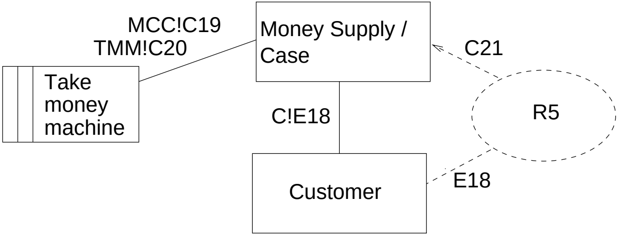
Architecture for Update Account



Money Supply/
Case (C19)
Sensor from
Customer (E18)

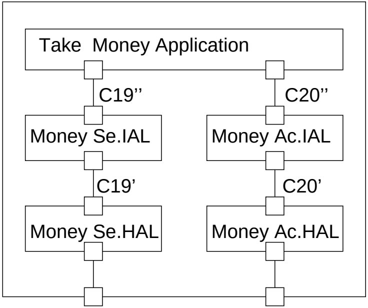


Problem Diagram for Take Money (required behavior variant)



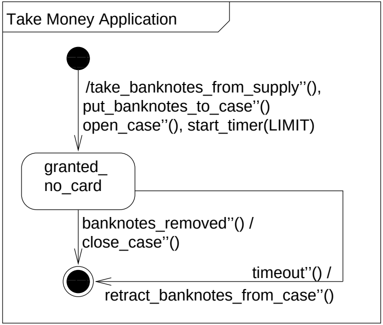
E18: $\{take\_banknotes\}$
C19: $\{banknotes\_removed\}$
C20: $\{take\_banknotes\_from\_supply,\ put\_banknote\_to\_case,\ open\_case,\ close\_case,\ retract\_banknotes\_from\_case\}$
C21: $\{control\_money\_supply,\ control\_money\_case\}$

Architecture for Take Money

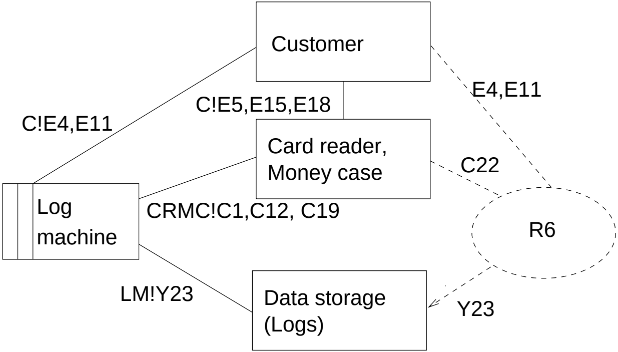


Money Supply/
Case (C19)
Sensor from
Customer (E18)

Money Supply/
Case (C20)
Actuator


Take Money Application
/take_banknotes_from_supply''(),
put_banknotes_to_case''()
open_case''(), start_timer(LIMIT)
granted_
no_card
banknotes_removed''() /
close_case''()
timeout''() /
retract_banknotes_from_case''()

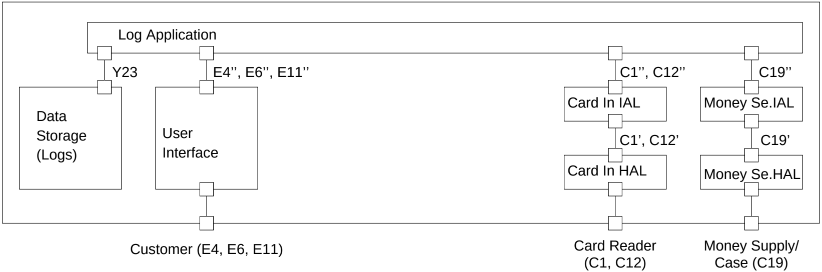
Problem Diagram for Log (Workpieces variant)



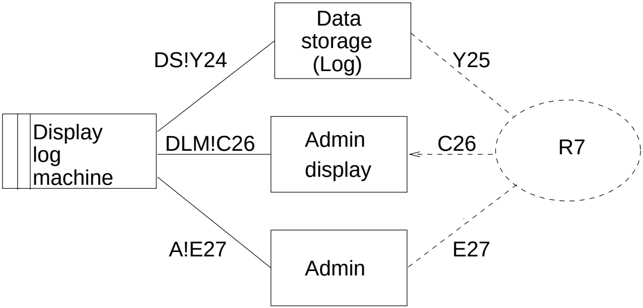
Y22: $\{card\_reader\_money\_case\_input\_phenomena\}$
E23: $\{log\_data\}$

C1, C12 . . . as given in the other figures

Architecture for Log

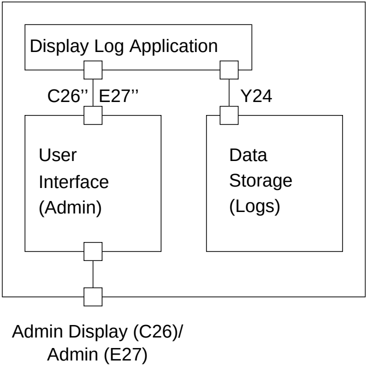


Problem Diagram for Display Log (commanded information)

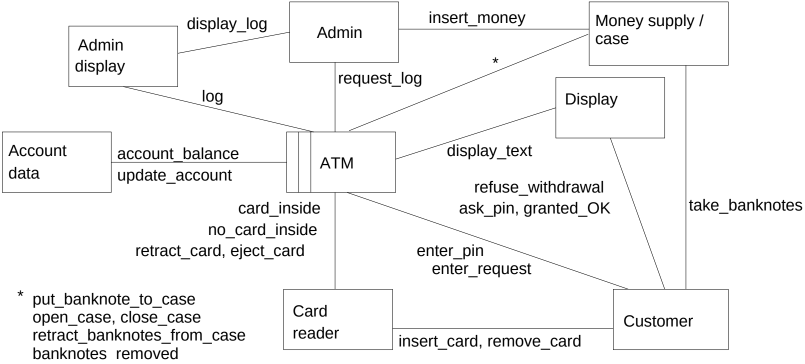


Y24: $\{log\_data\}$
Y25: $\{logged\_input\_phenomena\}$
C26: $\{log\}$
E27: $\{request\_log\}$

Architecture for Display Log



Refined Context Diagram



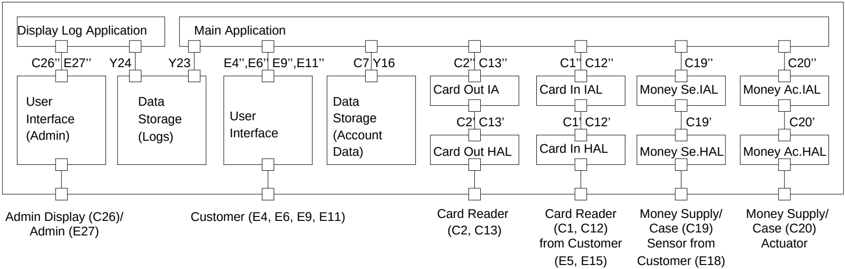
* put_banknote_to_case
open_case, close_case
retract_banknotes_from_case
banknotes removed

Develop Global Architecture

- *Take Money*, *Update Account*: parallel and share input phenomena $\Longrightarrow$ merge application components (decided)
- *Log*, all other subproblems: parallel and share input phenomena $\Longrightarrow$ merge application components (decided)
- *Authenticate*, *Request*, *Take Card*, merged problem (consisting of *Take Money*/*Update Account*/*Log*): sequentially or by alternative $\Longrightarrow$ merge to *Main Application*.
- *Display Log*, all other subproblems: parallel, do not share any interface $\Longrightarrow$ no merge

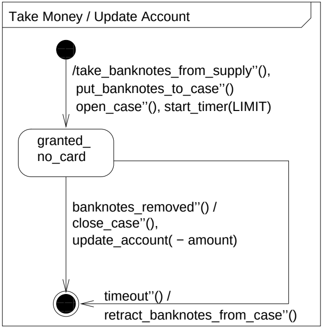
All components that are IALs or HALs are merged with the components of the same name in the other subproblem architectures.

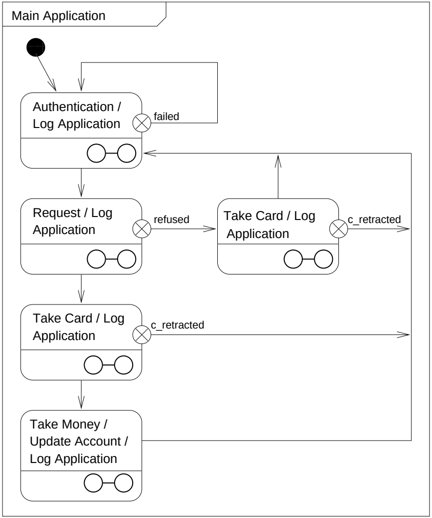
Global Architecture



Next, all of these components have to be specified and implemented.

Merging State Machines


Take Money / Update Account
/take_banknotes_from_supply''(),
put_banknotes_to_case''()
open_case''(), start_timer(LIMIT)
granted_
no_card
banknotes_removed''() /
close_case''(),
update_account(− amount)
timeout''() /
retract_banknotes_from_case''()



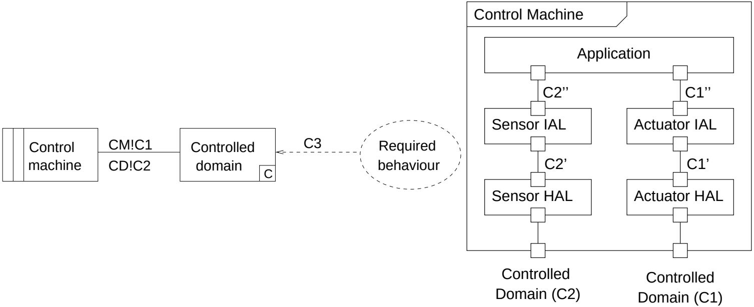
Conclusions

- We have developed *one (heuristic)* way of tackling the composition problem.
- Our process can be refined and enhanced.
- It will not always yield *optimal* solutions.
- This situation is analogous to tailor-made clothes vs. ready-to-wear clothes.
- However, this is what we want, because we are dealing with *normal* design.

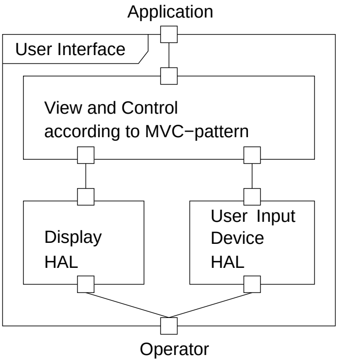
Literature

- Christine Choppy, Denis Hatebur and Maritta Heisel: Architectural Patterns for Problem Frames. In *IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures*, Vol. 152, No. 4, pp. 198–208, 2005.

Required Behaviour
Frame Diagram and Architectural Pattern

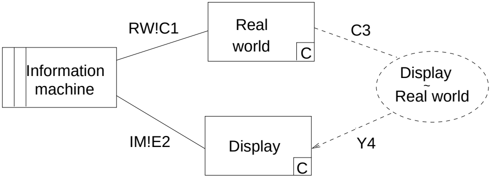


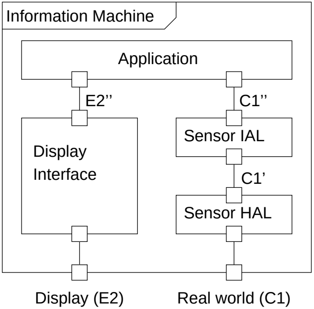
Detailed Architectural Pattern for User Interface



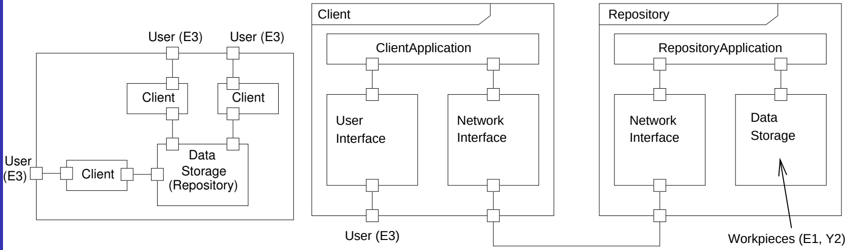
The interface of this component to the Application component should be the interface to the model, i.e., the User Interface comprises the View and Controller parts of the MVC (Model-View-Control) pattern.

Information Display
Frame Diagram and Architectural Pattern



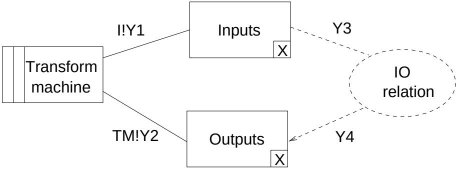


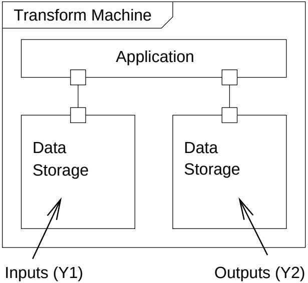
Architectural Pattern for Remote Access to Data Storage



Non-functional requirements might state that distributed access to the workpieces must be provided. The mapping from the *Repository* architectural style to the *Layered* architectural style is shown here.

Transformation
Frame Diagram and Architectural Pattern


Transform machine
I!Y1
Inputs
X
Y3
IO relation
TM!Y2
Outputs
X
Y4


Transform Machine
Application
Data Storage
Data Storage
Inputs (Y1)
Outputs (Y2)