Software Engineering 301:
Software Analysis and Design

Design: Introduction

Agenda

- What is design?
- Complexity
- How to do design?
- Some preliminary design issues for the vending machine

Design

- Free Online Dictionary:
  - *“To conceive or fashion in the mind; invent”*
  - *“To create or contrive for a particular purpose or effect”*
  - *“To have as a goal or purpose; intend”*
  - *“To create or execute in an artistic or highly skilled manner”*

Requirements vs. design

- Requirements
  - **WHAT** the system has to do, should do, or hopefully does
- Design (noun)
  - **HOW** the system provides its functionality
- Design (verb)
  - **DECIDING HOW** the system provides its functionality

Purpose of design

- Allows planning of how to meet the set of requirements for a project
  - *Design as specification*
  - Probably done ahead of implementation
  - May or may not be written down
- Communicates the abstract concepts underlying an existing implementation
  - *Design as abstract representation of an implementation*
  - Can be done after implementation

Two approaches to design-as-specification

- Design is complete “blueprint” for implementation, includes everything implementer needs to know
  - Very hard to achieve without iterating between design and implementation
  - If design diagrams need to be updated to agree with implementation, they are not serving as blueprints! (Why do this, then?)
  - In the end, this is a Waterfall model idea (unrealistic and wasteful)
- Design is a partial picture of the implementation, covering most important points
  - Minimizes the amount of time planning and writing documents
  - Significant points may be overlooked that require major restructuring of the system

What is a good design?

- One opinion: “Any design that meets the requirements of the system is a good design”
  - If no design can meet the requirements, there’s a problem with the requirements
  - This is a necessary but not sufficient condition

What is a good design?

- **If more than one design can fulfill the requirements**: how to choose amongst these?
  - Which one provides the non-functional properties that are generally recognized as most valuable:
    - modifiability, reliability, security, efficiency …
- The hard part:
  - not all non-functional properties can be simultaneously optimized
  - **decisions** about tradeoffs will need to be made

The traditional view

- “Efficiency is paramount”
  - hence, optimize, optimize, optimize
  - but, many optimizations lead to krufty code and are of minimal value: “premature optimization”

The current view

- “Modifiability is key”
  - software-that-is-used is always software-that-is-changed
  - (note that manual optimizations tend to be very difficult to modify)
  - this idea is less definite though: sometimes, it is not ***the*** most important factor
- The **design goals** for the particular system must be decided
  - specific non-functional properties, not just (for example) modifiability

Design is hard

- A “wicked” problem
  - Can only be defined by solving it (at least partially)
  - So much for blueprints!
- Consequence:
  - You need to solve it once to understand it
  - Then solve it again to do it well

Tradeoffs

- Design involves making decisions
  - Each decision leaves some possibilities open, and closes off others
  - You need to constantly ask: “do the positive consequences outweigh the negative ones?”
- Factors to consider:
  - Business context, training costs, technological maturity, growth, change, …

Agenda

- ~~What is design?~~
- Complexity
- How to do design?
- Some preliminary design issues for the vending machine

Software and humans

- Software was once conceived as being about machines
  - Instructions to machines, phrased for easier processing by machines
- However, people have to:
  - Write software
  - Read and modify (other people's) software
  - Talk to each other about software
- Modern software has evolved to make people's tasks easier for people

Human limitations

- Memory
- Error-proneness
- Fatigue
- Need for sleep
- Distraction

Managing complexity

- Why not have a large program consist of a single file?
  - Human understanding
    - Too hard to find functionality
    - Too much to remember
    - Too hard to understand how it works
  - Modifiability
    - Changes can easily break other parts
  - Efficiency
    - Oddly enough, such programs tend to work worse because people can't understand them well enough

Managing complexity

- Instead:
  - Abstraction
    - Make a complex idea simpler by hiding part of it
  - Structure
    - Define how parts should interact, and ensure that these rules are followed always
  - Layers
    - Repeat this process upwards (by abstracting more) or downwards (by inserting concrete details)
  - Consistency
    - Create and follow simple rules: people can see them and know what to expect

Agenda

- ~~What is design?~~
- ~~Complexity~~
- How to do design?
- Some preliminary design issues for the vending machine

How to do design?

- Iterate
  - Your first design attempt is likely to be flawed in big ways
    - Analyze it for strengths and weaknesses
    - Attempt to retain the former and eliminate the latter
  - Don’t get depressed if you need to keep trying
    - This is a learning process
    - You learn more from mistakes

How to do design?

- Divide and conquer
  - Any non-trivial system is too big and complex to take on in one big piece
  - Break it down into more manageable parts
  - Tackle each part individually
  - Reconstruct the whole, looking for problems
  - Iterate

How to do design?

- Top-down vs. bottom-up
  - You can work at an abstract level, and gradually refine it into more and more concrete details (top-down)
  - You can start with concrete details, and try to abstract them into common generalizations (bottom-up)
  - Which is better? …

How to do design?

- Top-down
  - Pros:
    - Conceptually easy
    - Can delay making many decisions
  - Cons:
    - Failure to identify important complexity (affecting decisions) until late

How to do design?

- Bottom-up:
  - Pros:
    - Early identification of needed low-level functionality, leading to better factoring
    - Ability to detect pieces that have been previously built
  - Cons:
    - Hard to abstract away details to arrive at useful generalizations
    - Can be hard to build a complete system from poorly chosen low-level parts

How to do design?

- … Which is better?
  - Neither!
- In practice, you need to do both, bouncing between high-level abstractions and low-level details

Design Levels

- High level
  - Major subsystems and packages
  - Identification of how these will be deployed onto hardware, along with communications protocols
    - Software architecture
  - Restrictions here help to ensure that the system maintains understandability and changeability

  - We often refer to the division of the system into subsystems/packages as *partitioning*

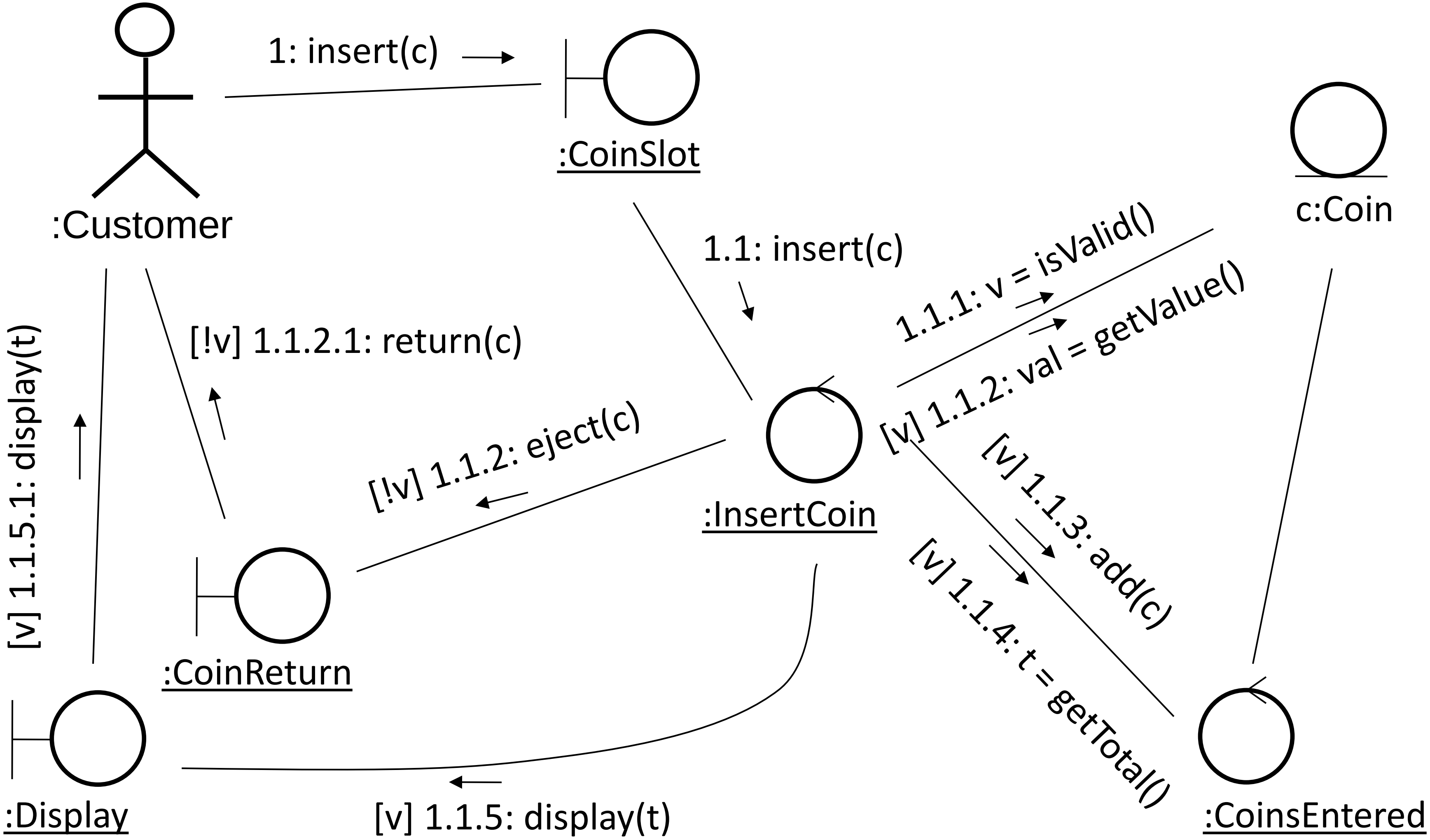
Design Levels

- Low level
  - Individual classes
  - Data and functions

Agenda

- ~~What is design?~~
- ~~Complexity~~
- ~~How to do design?~~
- Some preliminary design issues for the vending machine

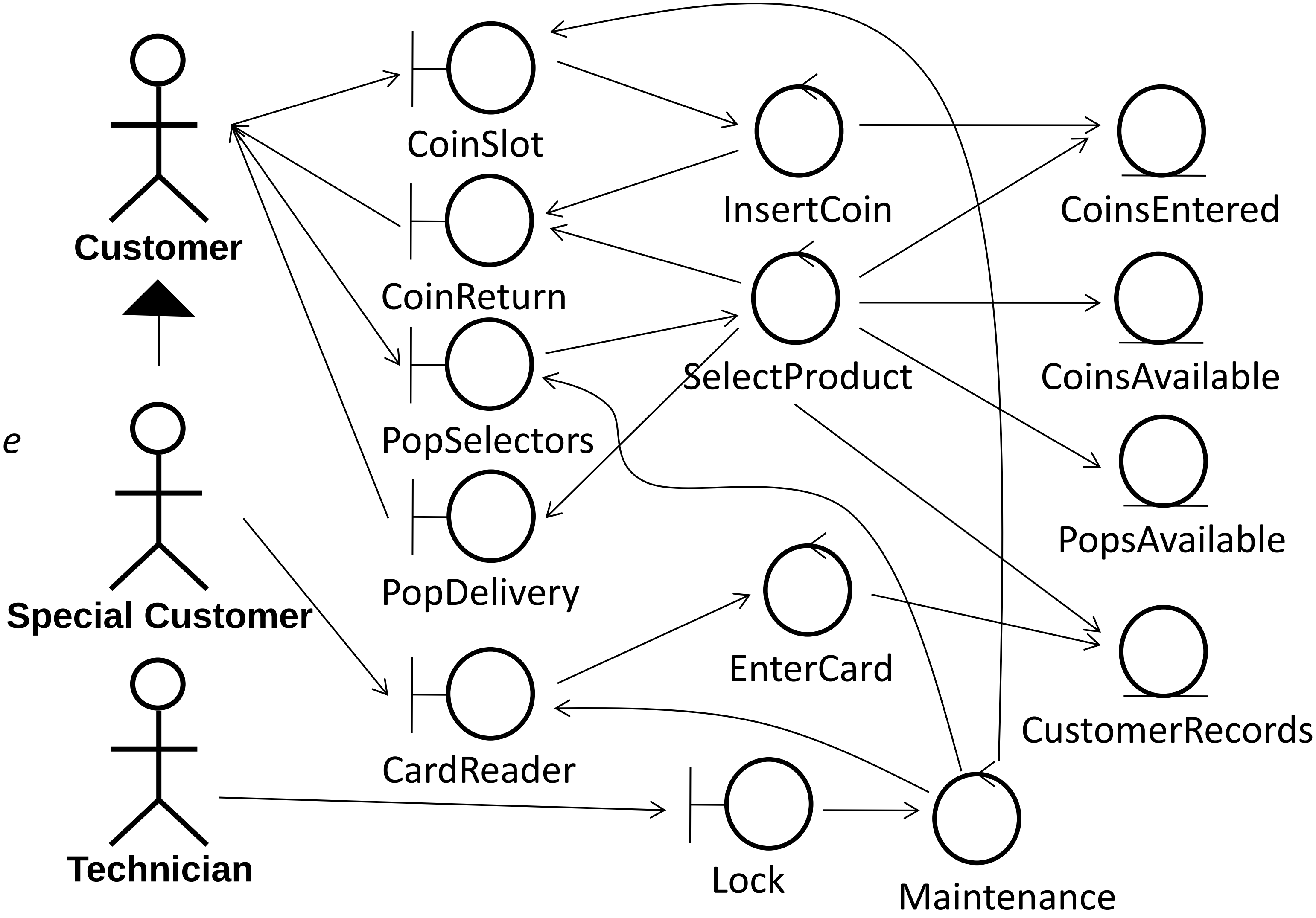
Realization for Insert Coin functionality



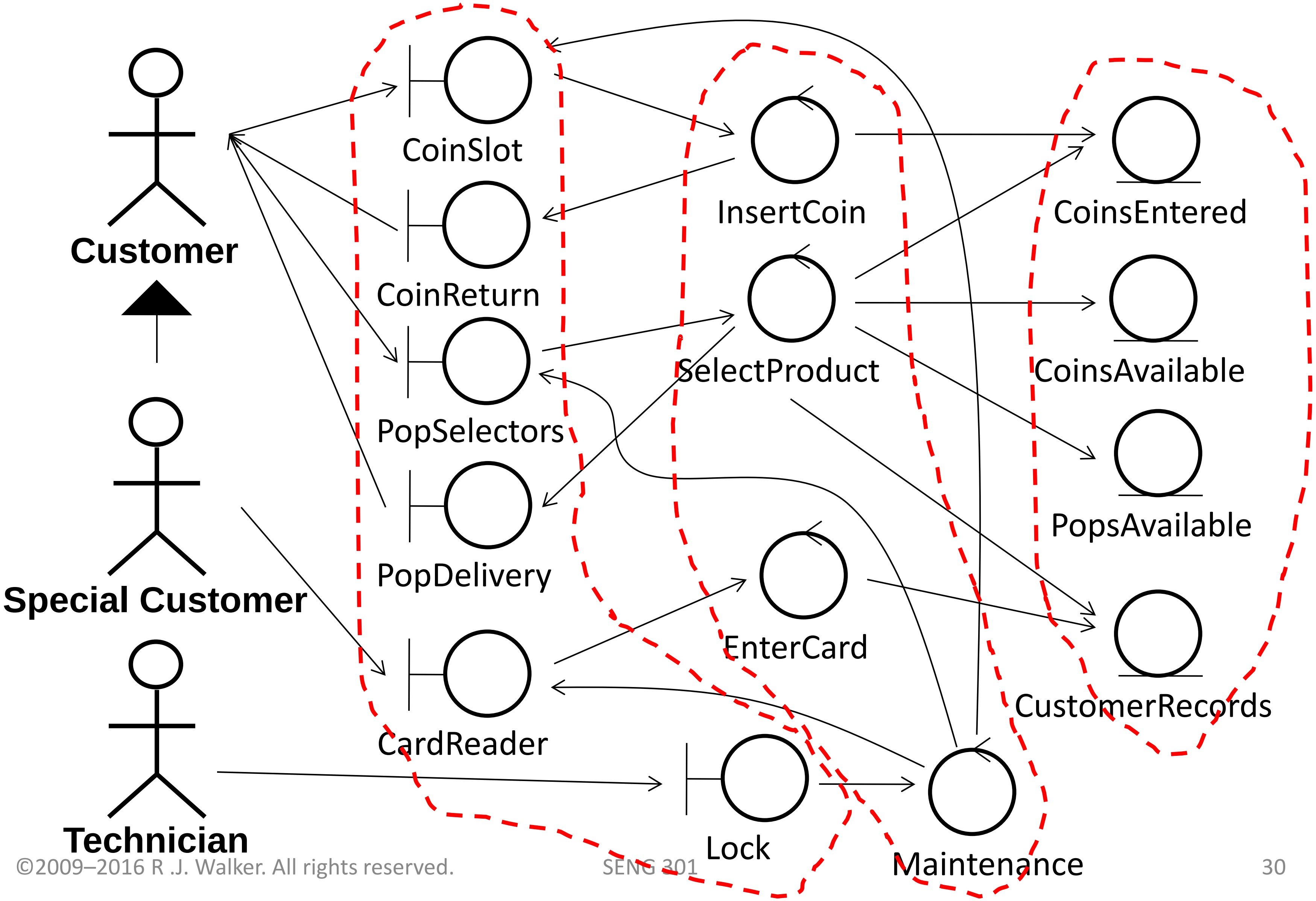
Simplified dynamic analysis model

Display should be here too, as a boundary object

Messages being passed are suppressed in the diagram



Partitioning the system, v1



Physical considerations for the vending machine

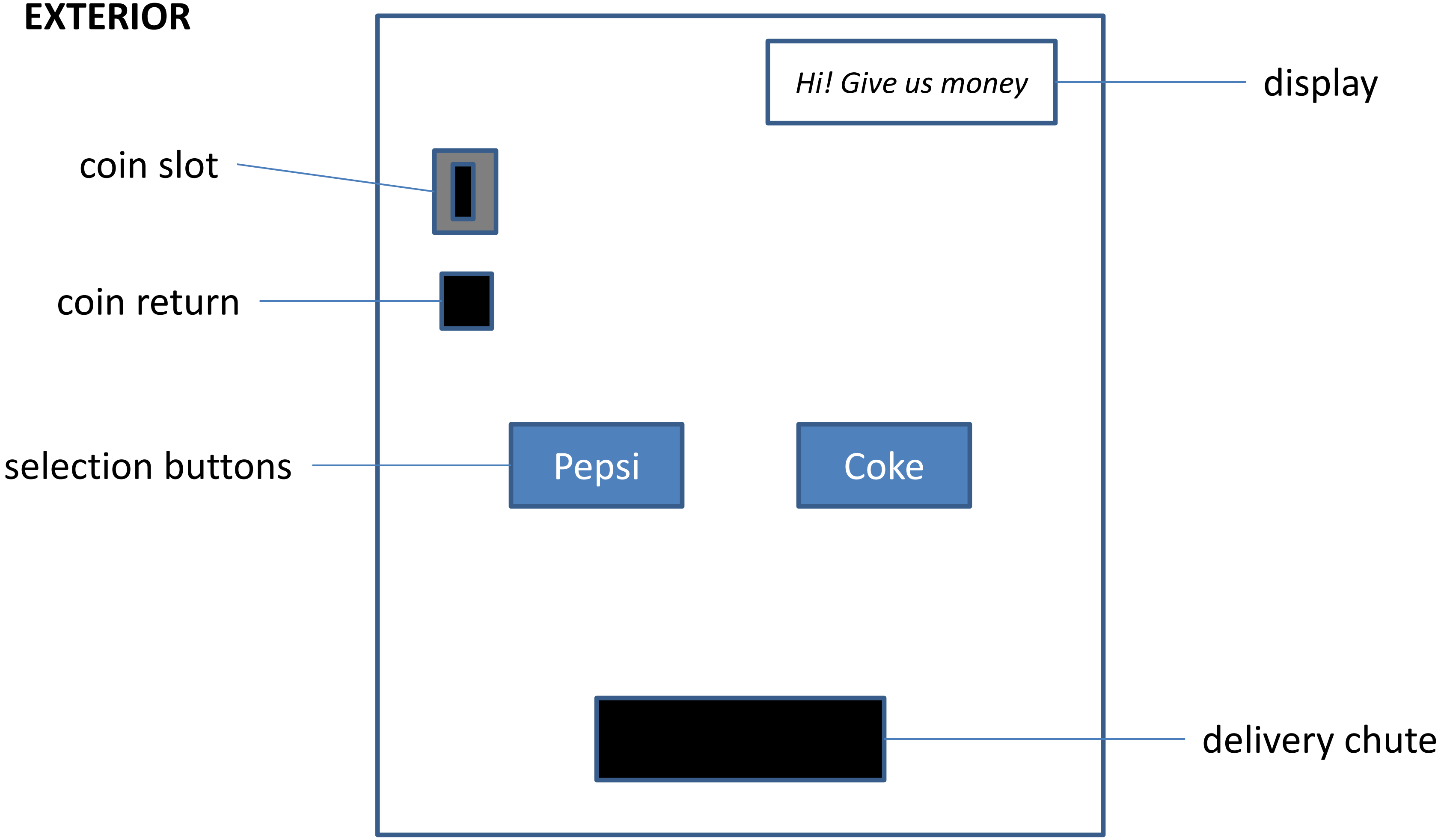
- Coins:
  - Enter a slot
  - Must be collected in some container
    - Return THESE coins if the customer wants
    - Return THESE coins if they are invalid
  - If used for a payment, they must be sorted and stored, available to act as change
- Pop:
  - Stored in racks, one kind per rack
  - Cost is set per rack

Physical considerations for the vending machine

- Maintenance:
  - Unlocking the machine must disable most or all of the functionality

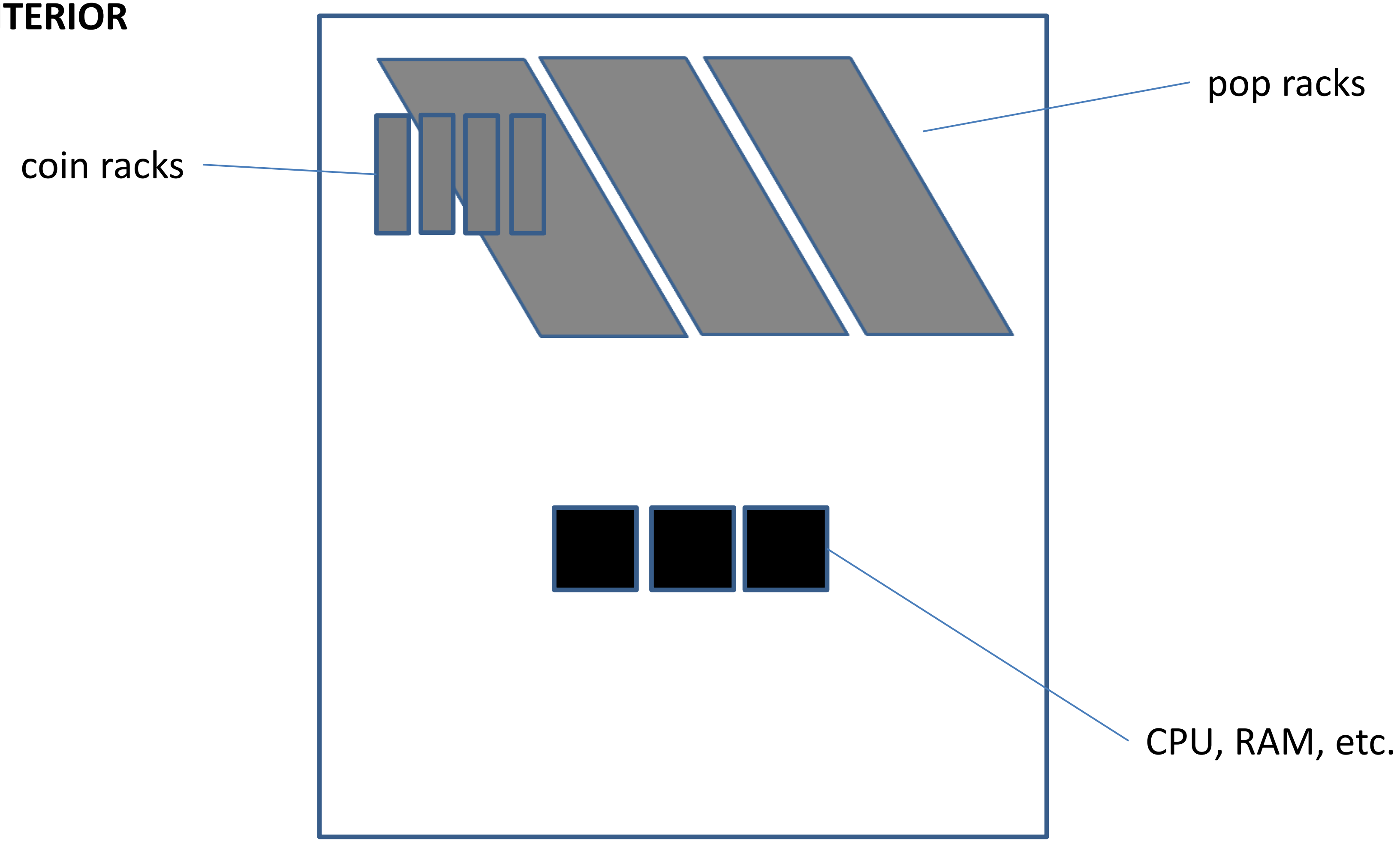
Physical considerations for the vending machine

EXTERIOR



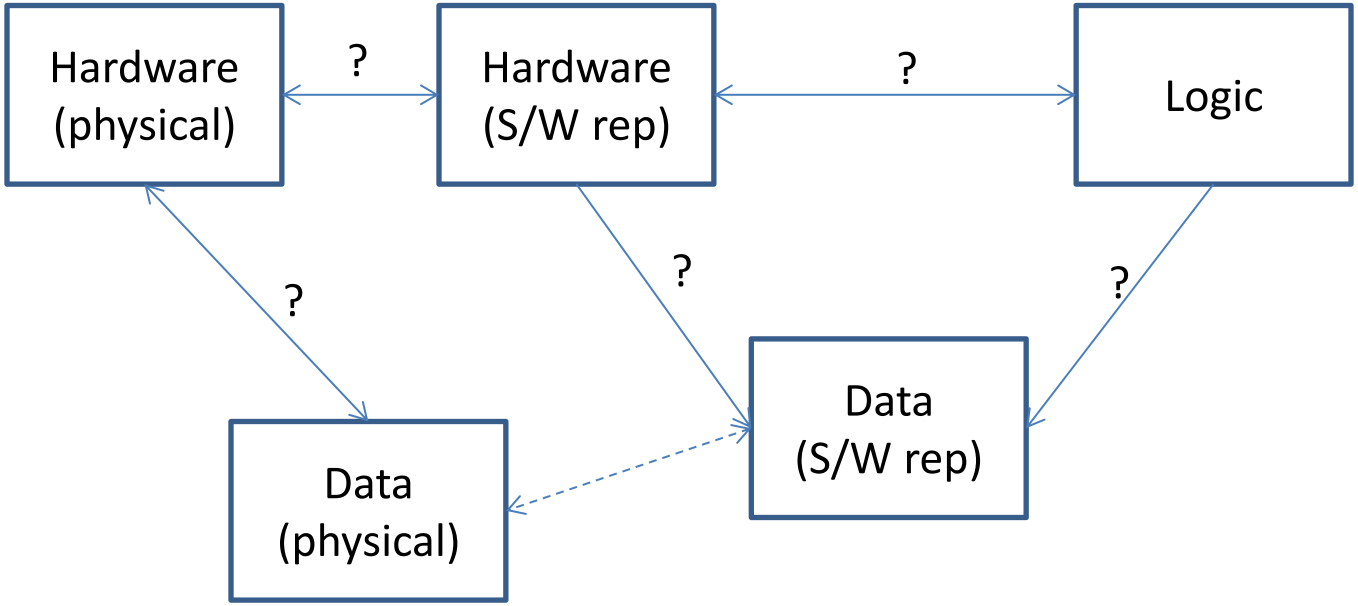
Physical considerations for the vending machine

INTERIOR



Hardware-software interface

- But we need some means for our software to interact with the hardware



It should look something like this

Hardware (software representation)

```
class Hardware {
    void enable();
    void disable();
    void acceptCoins();
    void returnCoins();
    int coinsEntered();
    CoinRack[] coinRacks();
    PopRack[] popRacks();
    void display(String s);

}
```

In essence, this is a “facade” for the underlying physical mechanisms

[The assignments’ code is a bit different.]

```
class CoinRack {
    enum CoinKind {...};
    CoinKind kind;
    int maxCapacity;
    int currentTotal;
    void release(int count);
}

class PopRack {
    int costInCents;
    int maxCapacity;
    int currentTotal;
    void release();
}
```

Hardware events

- How does the software know when hardware based events happen?
  - Polling: keep checking some memory location to see if a “flag” is turned on
  - Interrupt: a hardware-based signal causes the OS to activate a handler
  - Callback: software registers a method to call in the case that an event happens

Callbacks

- Imagine that the hardware is hidden behind a single class called Hardware
- Each kind of event to be supported via **callbacks**
  - Must allow registration of a **callback method**
- When the event happens, the registered callback method is called

Hardware (s/w rep, v2)

```
class Hardware {
    void enable();
    void disable();
    void acceptCoins();
    void returnCoins();
    int coinsEntered();
    CoinRack[] coinRacks();
    PopRack[] popRacks();
    void display(String s);
    void register(Listener l);
}
```

```
class CoinRack {
    enum CoinKind {...};
    CoinKind kind;
    int maxCapacity;
    int currentTotal;
    void release(int count);
}

class PopRack {
    int costInCents;
    int maxCapacity;
    int currentTotal;
    void release();
}
```

Hardware (s/w rep, v2)

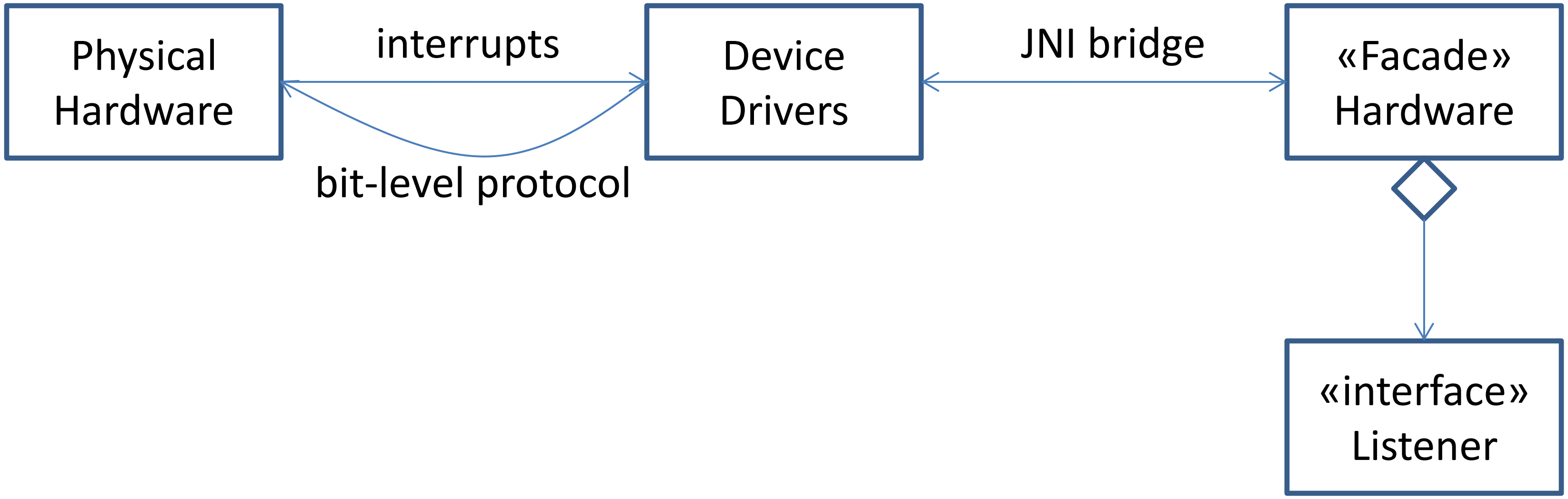
```
interface Listener {
    void coinInserted(int value);
    void popSelected(int rack);
    void coinReturnRequested();
    void deliveryDoorOpened();
    void deliveryDoorClosed();
    void serviceDoorUnlocked();
    void serviceDoorLocked();
    void popRackEmptied(int rack);
    void coinRackEmptied(int rack);
}
```

These are the kinds of event that can happen with the hardware (for the vending machine). A listener (also known as an observer) can be informed of hardware event occurrences by:

1. implementing this interface
2. being registered with the hardware

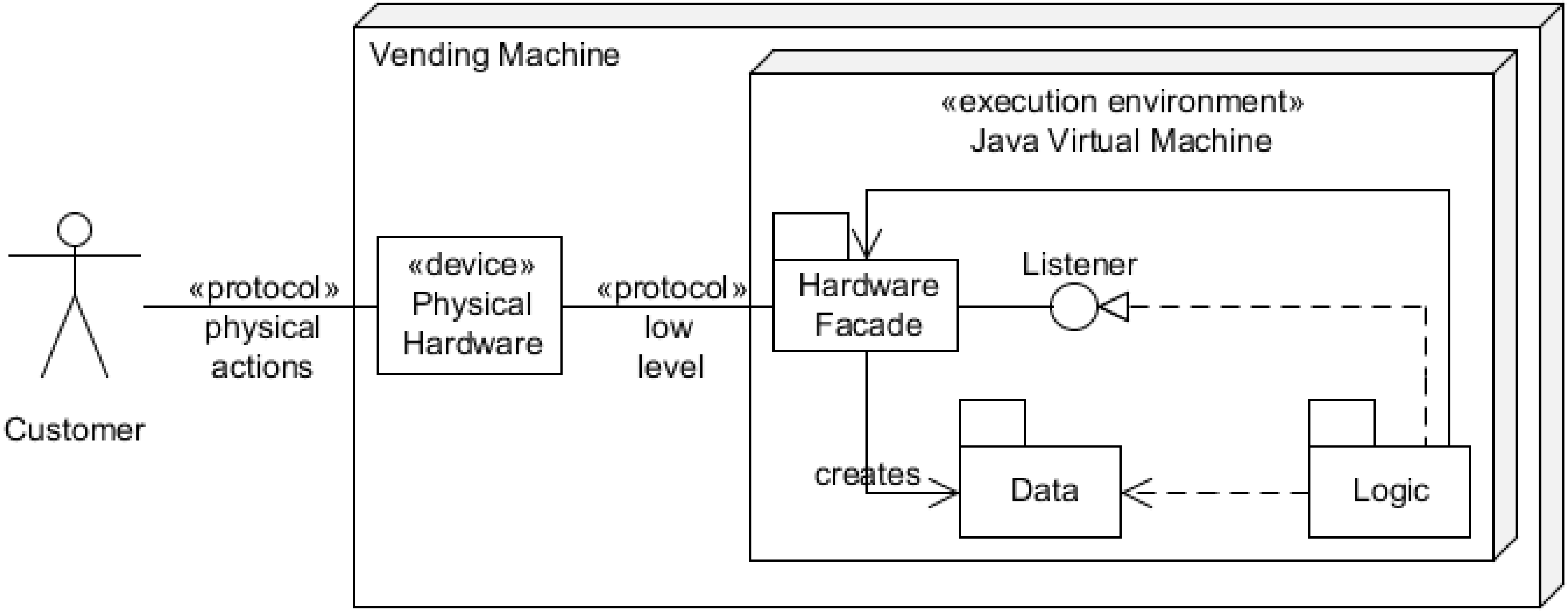
[This essentially uses the Observer design pattern, which we will talk about later.]

Back to the big picture



Because of THIS, we (meaning the developers of the application software) can ignore all THAT

Back to the big picture



[Notation: “Lollipops” are interfaces provided by packages.]

“Data” and “Logic” don’t get us any closer to understanding what goes inside.
In other words, this <u>partitioning</u> is poor because it does not simplify our problem.

An alternative partitioning

- What are the big functionalities?
  - Handling coins
  - Handling pop
  - Displaying information
  - Coordinating user actions with the above in order to follow business rules

Handling coins

- But, we also had those frequent buyer cards
  - Paper money?
  - Debit cards?
  - Credit cards?
  - Different kinds of currency?
  - Paypal?!?
  - Mixed modes of payment
- Yikes, sounds like this could get out of hand

Handling coins

- Should we decide now what kinds of payment are accepted?
- Since there seem to be lots of options, the answer is likely “NO”
  - We could start by assuming the simple case (just coins) but if we design our software well, adding new payment kinds will be fairly easy without having to rewrite the whole thing
  - A “facade” allows us to hide details of payment method from rest of system

Handling pop

- Similarly, a flexible design will allow us to ignore the kind of product (i.e., pop) that is being sold and details of its dispensing and refilling
- Another Facade for Products

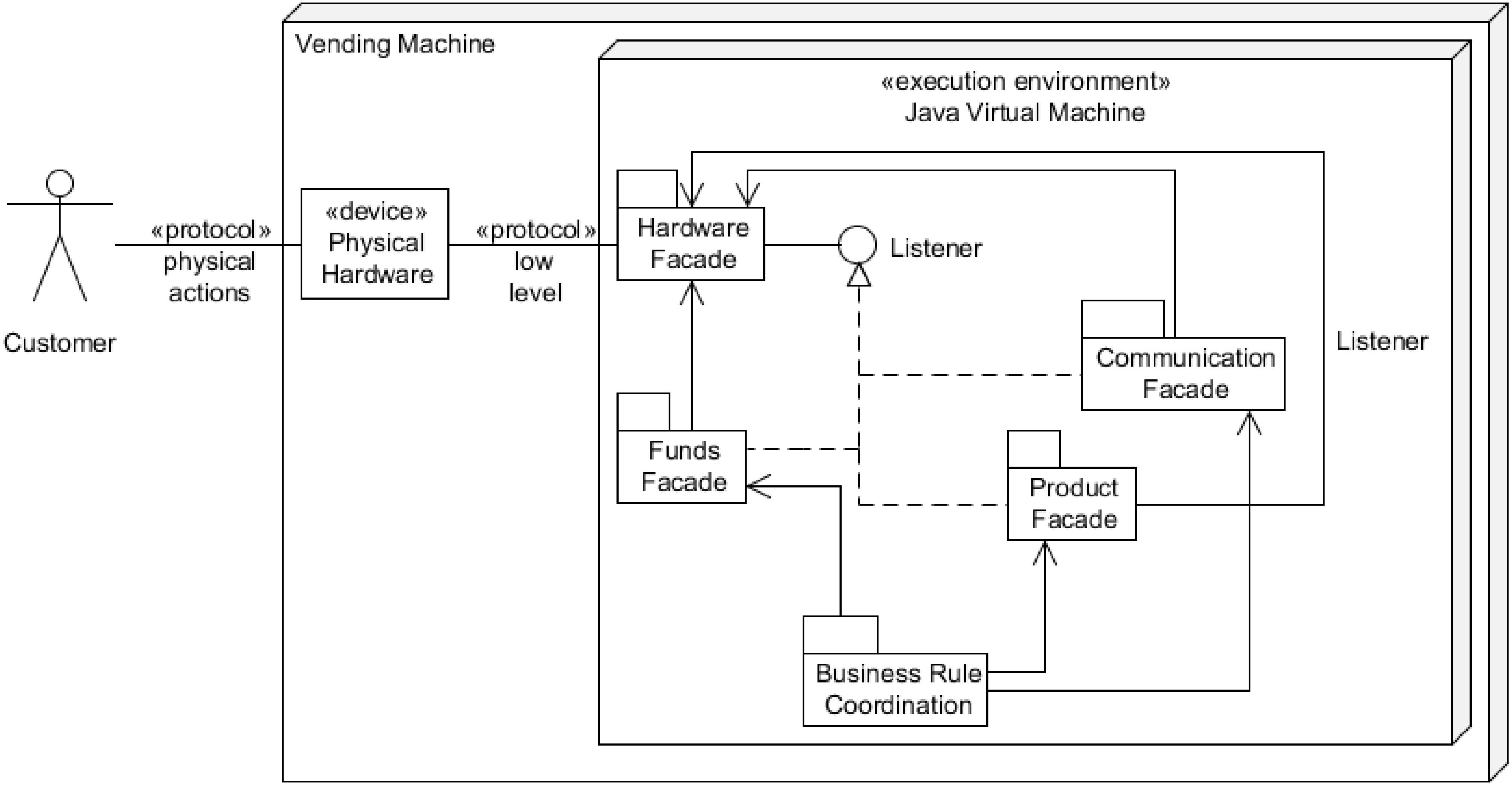
Displaying information

- Perhaps we should be more flexible about the display too?
- Maybe some machines will have very small displays, others will have big displays
- Maybe we will want audio output or accessible output for the visually impaired
- Yep, there should be a Facade for Communication, too

Coordination

- Finally, the business logic will consist of rules like:
  - Each product has a cost for a given customer
  - The cost must be less than or equal to the value of the offered payment
  - Excess payment is returned
  - The product is dispensed when proper payment is received
  - Communication is made to the customer at appropriate times
- These points can just make use of the Facades

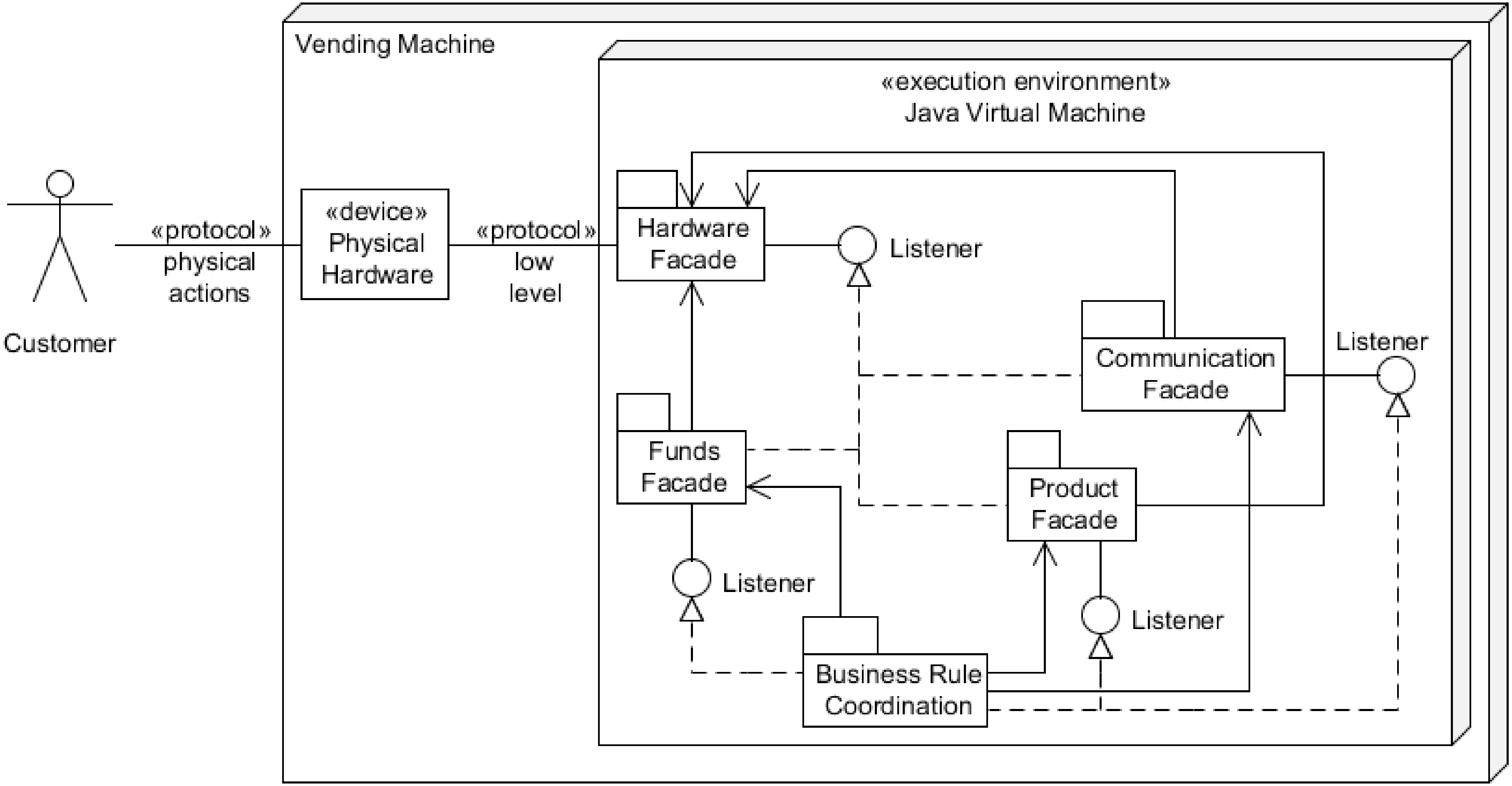
Back to the big picture



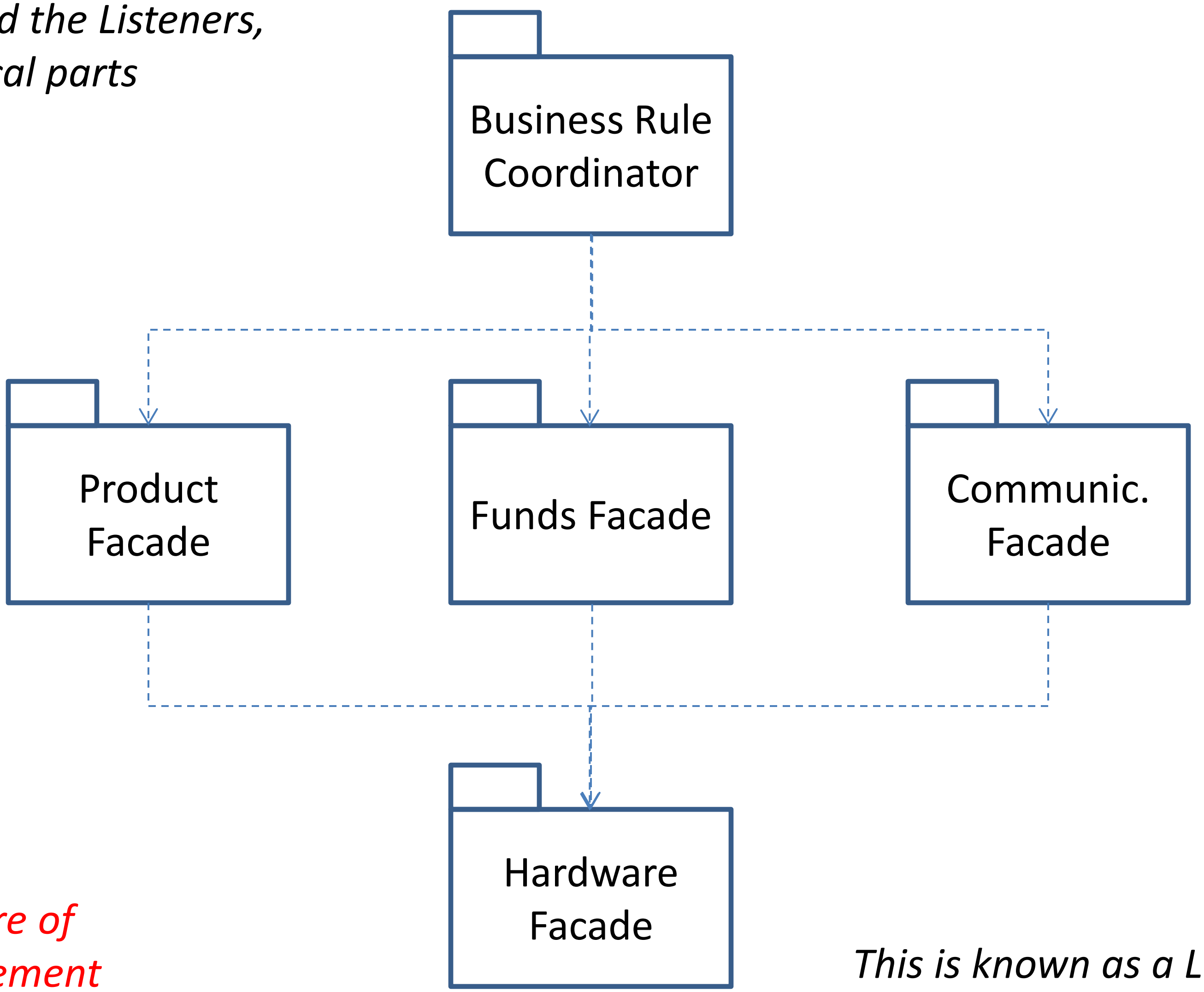
But wait, there’s more!

- How can the business rule coordinator know when it needs to do its job?
  - More Listeners

Back to the big picture



I've suppressed the Listeners, and the physical parts



The structure of this arrangement has two key properties: Find them

This is known as a Layered architectural style

Next time

- Design patterns