Software Engineering 301:
Software Analysis and Design

# Moving from requirements to design

# Agenda

- Finding objects via text analysis
- Analysis models
- Partitioning
- Example: Judging two alternatives

# Finding objects

- What words are involved when talking about the vending machine and its software?
  - Emphasis on nouns and verbs
- This is sometimes called **domain analysis**
  - Usually starts from written descriptions
  - The point is to identify objects and their interactions
  - Note that this is NOT yet design
  - Not all these potential objects will end up in the software!!!
- Identify the candidate objects for the vending machine software…

# Remember …

- "Ming inserts $2 in Canadian coins in the vending machine. He decides which of the available kinds of pop he would like, and selects it. The pop costs less than $2, so the pop is dropped into the delivery chute, and his change is returned through the coin return."
- Ming changes his mind altogether and wants his money back
- Ming realizes that he doesn't have enough change, and wants his money back
- The machine doesn't have enough of the pop that he chose
- The machine doesn't have enough change (when does he find out?)
- Ming decides that he would like to return his pop purchase

# Finding objects

- Now we have a whole bunch of words and phrases that might matter
- 2 things to do:
  - A quality analysis to eliminate words that aren't really appropriate
  - A transformational analysis to put these into a different form that's easier to deal with

# Text analysis

- Nouns:
  - Ming, $2, coins, vending machine, kinds, pop, delivery chute, change, coin return, mind, money, machine, purchase
  - coin, customer, value, total
  - product, card, funds, technician
- Verbs
  - inserts, selects, decides, costs, dropped, returned
  - displayed, added
  - enter, service

# Text analysis

*specific objects are eliminated (usually)*

- Nouns:
  - ~~Ming~~, ~~$2~~, coins, vending machine, kinds, pop, delivery chute, change, coin return, mind, money, machine, purchase
  - coin, customer, value, total
  - product, card, funds, technician
- Verbs
  - inserts, selects, decides, costs, dropped, returned
  - displayed, added
  - enter, service

# Text analysis

*synonyms are eliminated*

- Nouns:
  - ~~Ming~~, ~~$2~~, ~~coins~~, vending machine, kinds, pop, delivery chute, change, coin return, mind, money, ~~machine~~, ~~purchase~~
  - coin, customer, value, total
  - product, card, ~~funds~~, technician
- Verbs
  - inserts, selects, decides, costs, dropped, returned
  - displayed, added
  - enter, service

# Text analysis

*some objects refer to hardware, so we'll keep these for now*

- Nouns:
  - ~~Ming~~, ~~$2~~, ~~coins~~, vending machine, kinds, pop, delivery chute, change, coin return, ~~mind~~, money, ~~machine~~, ~~purchase~~
  - coin, customer, value, total
  - product, card, ~~funds~~, technician
- Verbs
  - inserts, selects, decides, costs, dropped, returned
  - displayed, added
  - enter, service

# Text analysis

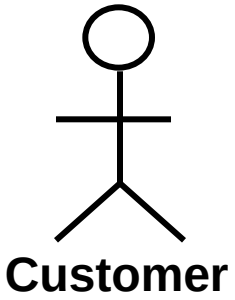
*other objects are interrelated, but not identical: keep these for now*

- Nouns:
  - ~~Ming~~, ~~$2~~, ~~coins~~, vending machine, kinds, pop, delivery chute, change, coin return, ~~mind~~, money, ~~machine~~, ~~purchase~~
  - coin, customer, value, total
  - product, card, ~~funds~~, technician
- Verbs
  - inserts, selects, decides, costs, dropped, returned
  - displayed, added
  - enter, service

# ANALYSIS MODELS

# Actors

- Requirements often make mention of **actors**
  - A person playing a role (e.g., instructor)
  - A physical device
  - External software
  - External organization

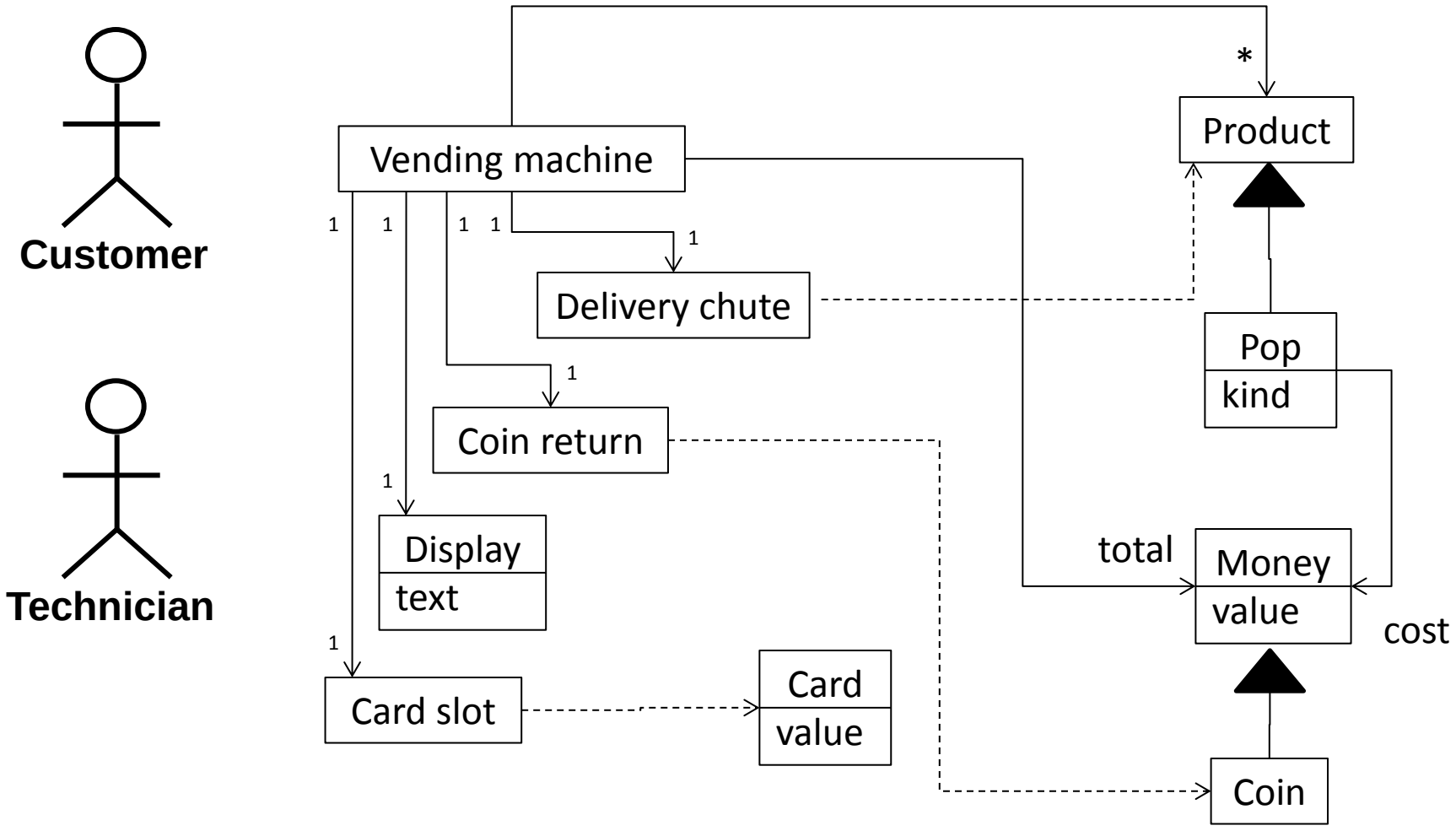



- An actor is NOT part of the system
  - An actor uses the system or an actor is used by the system (or both)

# Actors

- In UML actors are really just special kinds of class
- Actors can have all the relationships that classes can
- Generalization, dependency, association most useful

# Analysis structural model



# Remaining problems

- How do the actors interact with these classes?
- How do the classes get instantiated?
- Is there anything else missing?

- To answer these points, we start to look at the dynamic properties of the system
  - Use cases are again important

# Use case realizations

- We want to ensure that each use case can be made “real”
  - Ensure that each can be modelled as a set of interacting objects
  - Only some of these objects will come from the analysis class model
- Equivalent idea could come from user stories, scenarios, etc.

# Boundary, entity, controller

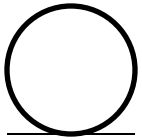
- Actors still play an important role
- During analysis, it is useful to think of objects as falling into three kinds:
  - A *boundary* object interacts with actors
    - Allows us to abstract away details of UI, hardware
  - An *entity* object generally represents a business concept to be found inside the system
    - It will usually be found in our analysis structural model
  - A *controller* object represents some complex behaviour, the business logic, embedded in the system

# Boundary, entity, controller

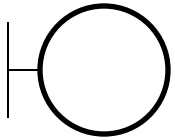
- Some people use the concept of a *home*, as a special-purpose controller object
  - Used to manage the creation, storage, and retrieval of entity objects
  - This term is non-standard, but the idea is common

# Boundary, entity, controller

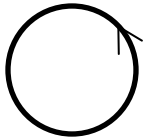
- Notation
  - Strictly speaking, these kinds of objects are still just objects, so standard UML notation would work
  - Boundary, entity, and controller serve as stereotypes
  - For simplicity, special icons are often used:



entity



boundary



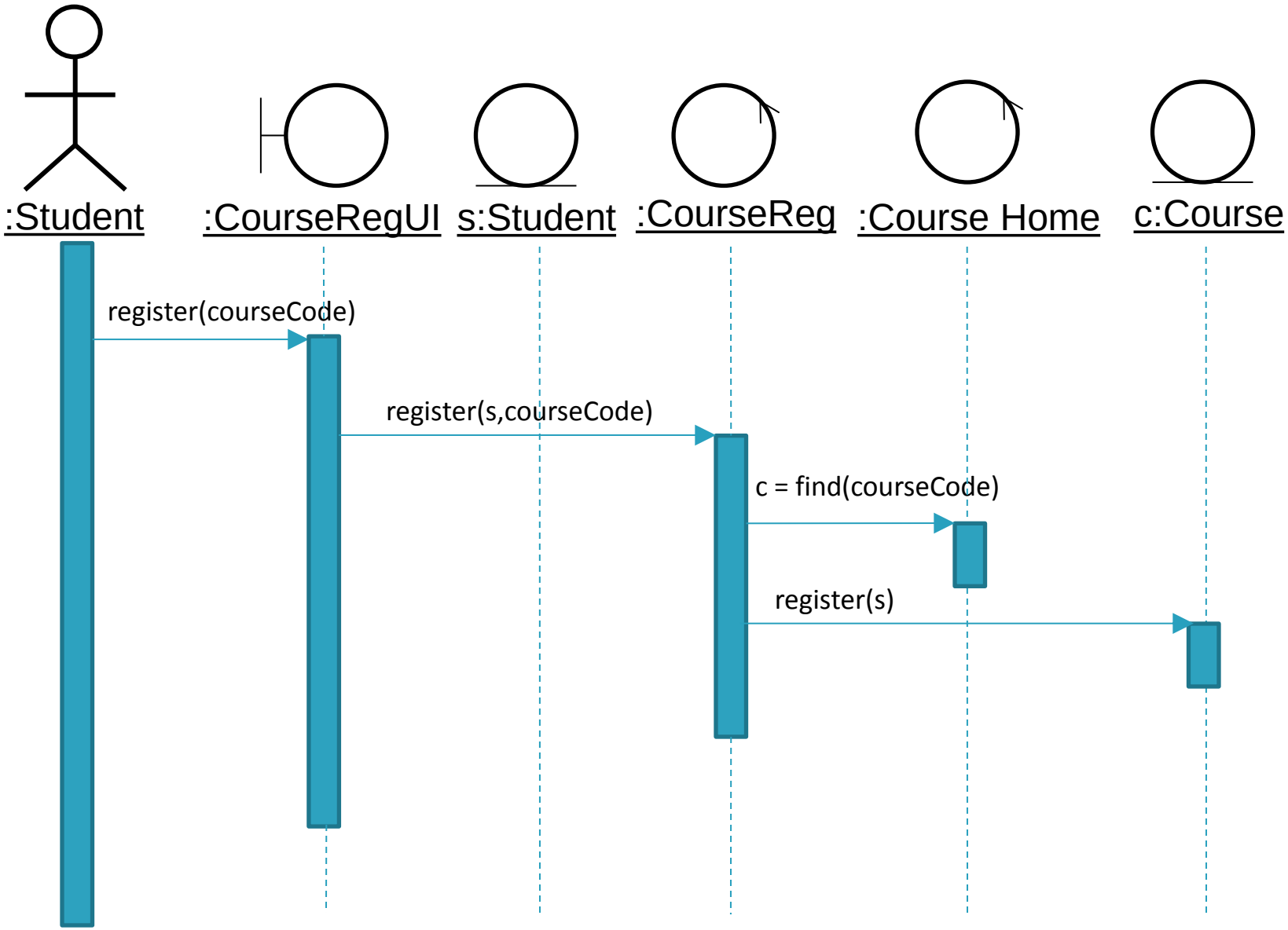
controller

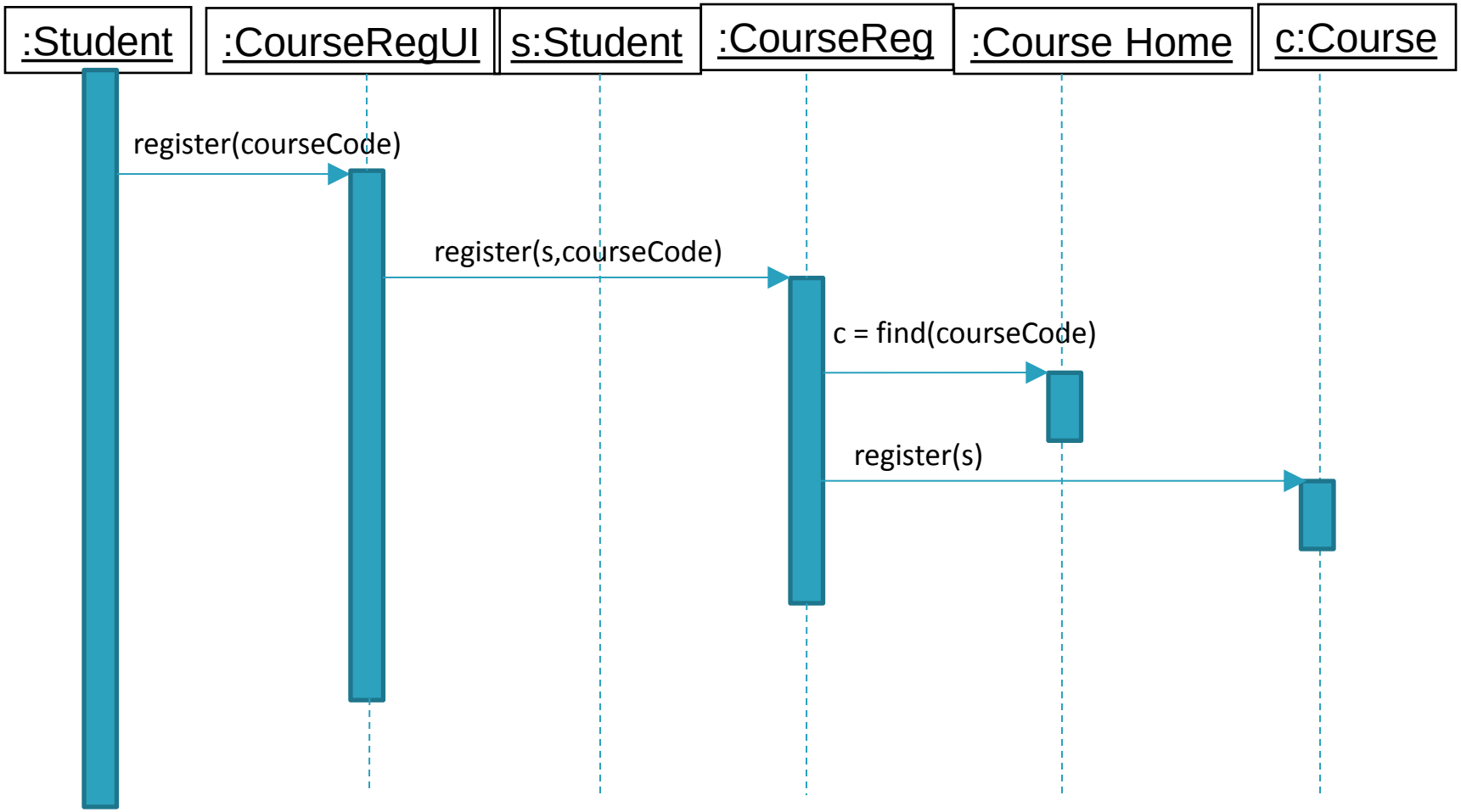
# Rules of thumb

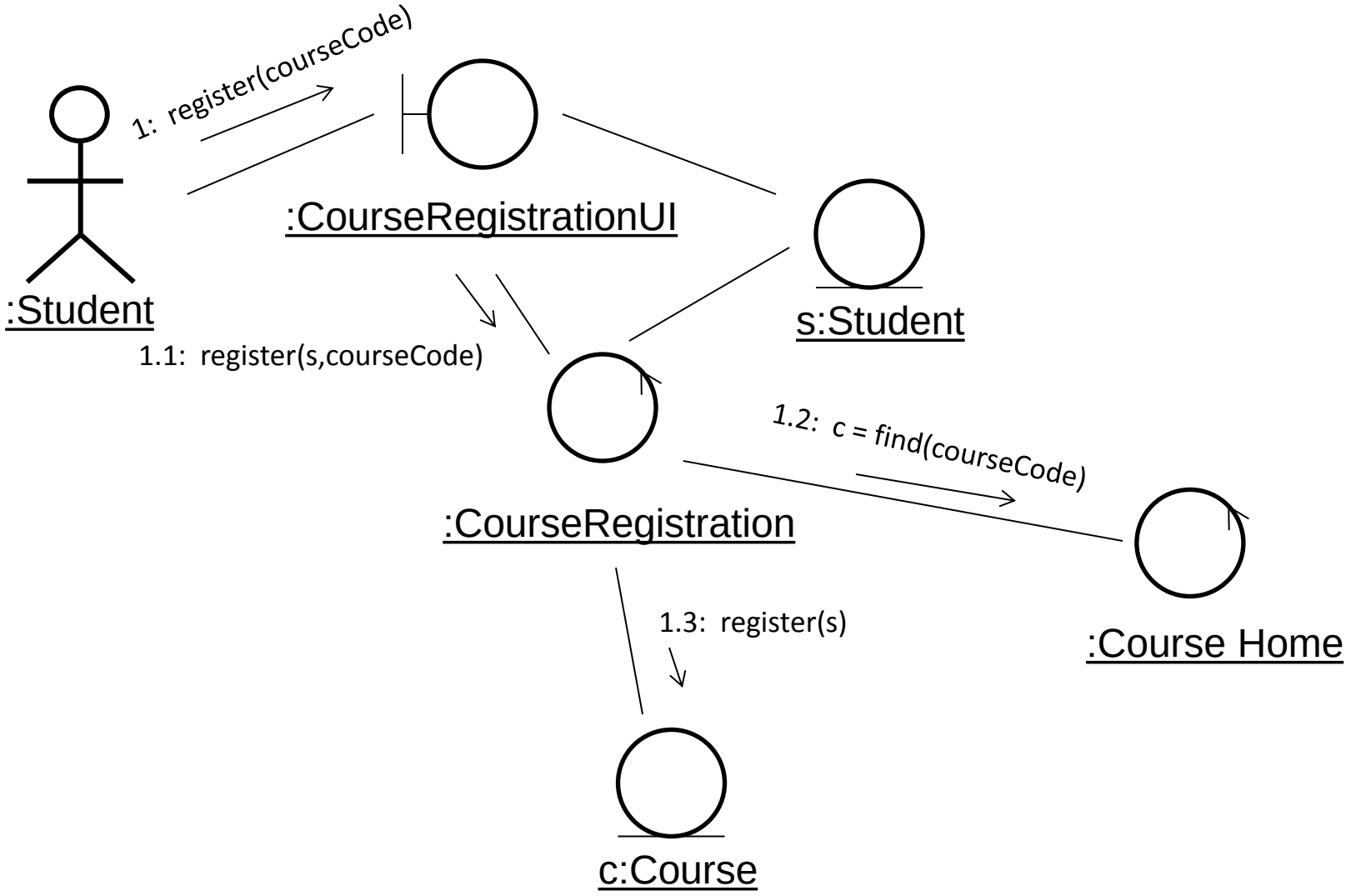
- Actors must only interact with boundary objects
- Boundary objects should interact only with controller objects
  - Controller object will sometimes pass entity objects to the boundary for presentation
- Entity objects are simple: data plus setters/getters

# Rules of thumb

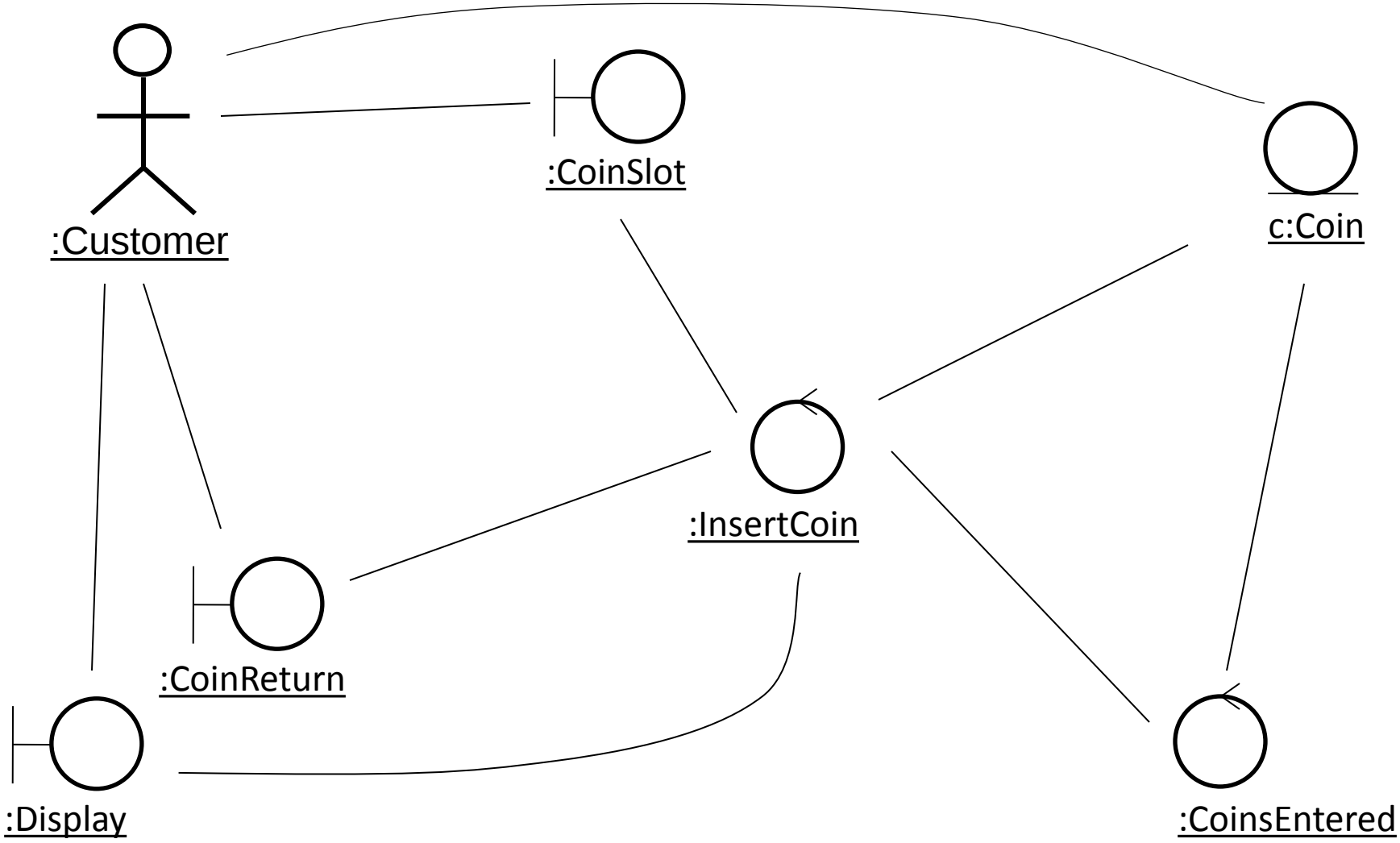
- These analysis objects are very unlikely to be retained into the design
- In particular, controller objects tend to get split up and combined with entities



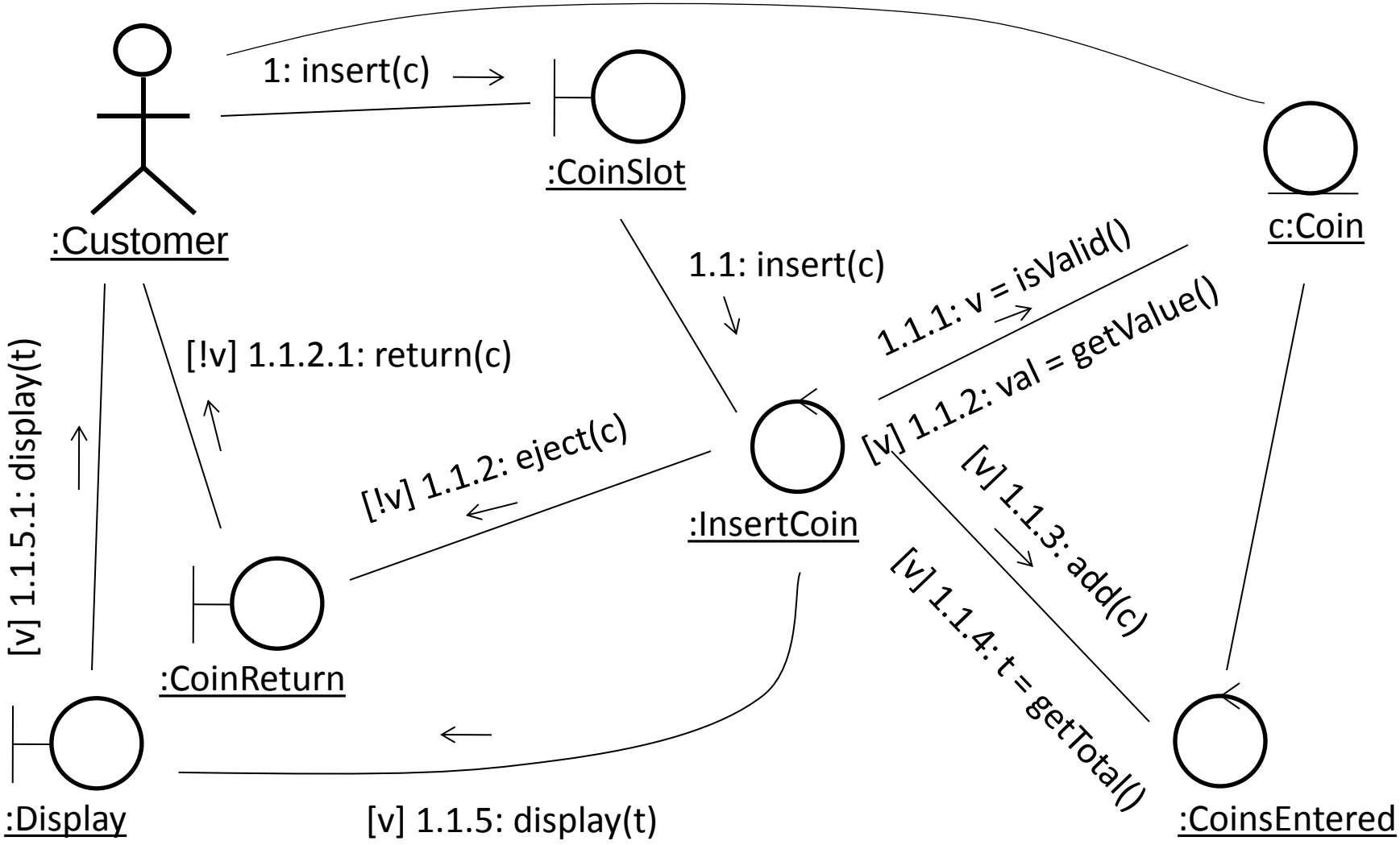




# Use case realization for Insert Coin



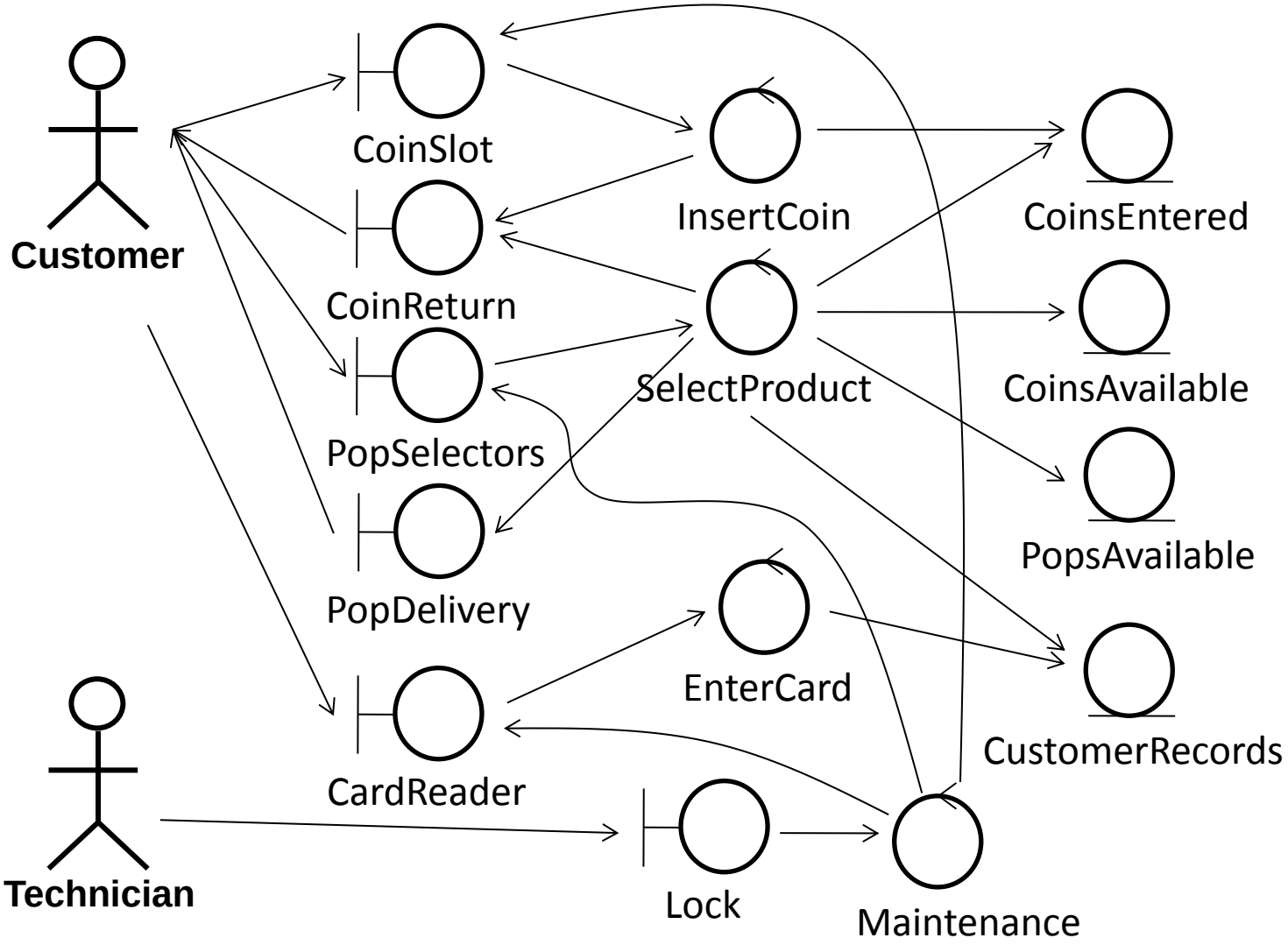
# Use case realization for Insert Coin



# Simplified dynamic analysis model

*Display should be here too, as a boundary object*

*These should be objects. And messages should be passed.*



# Issues

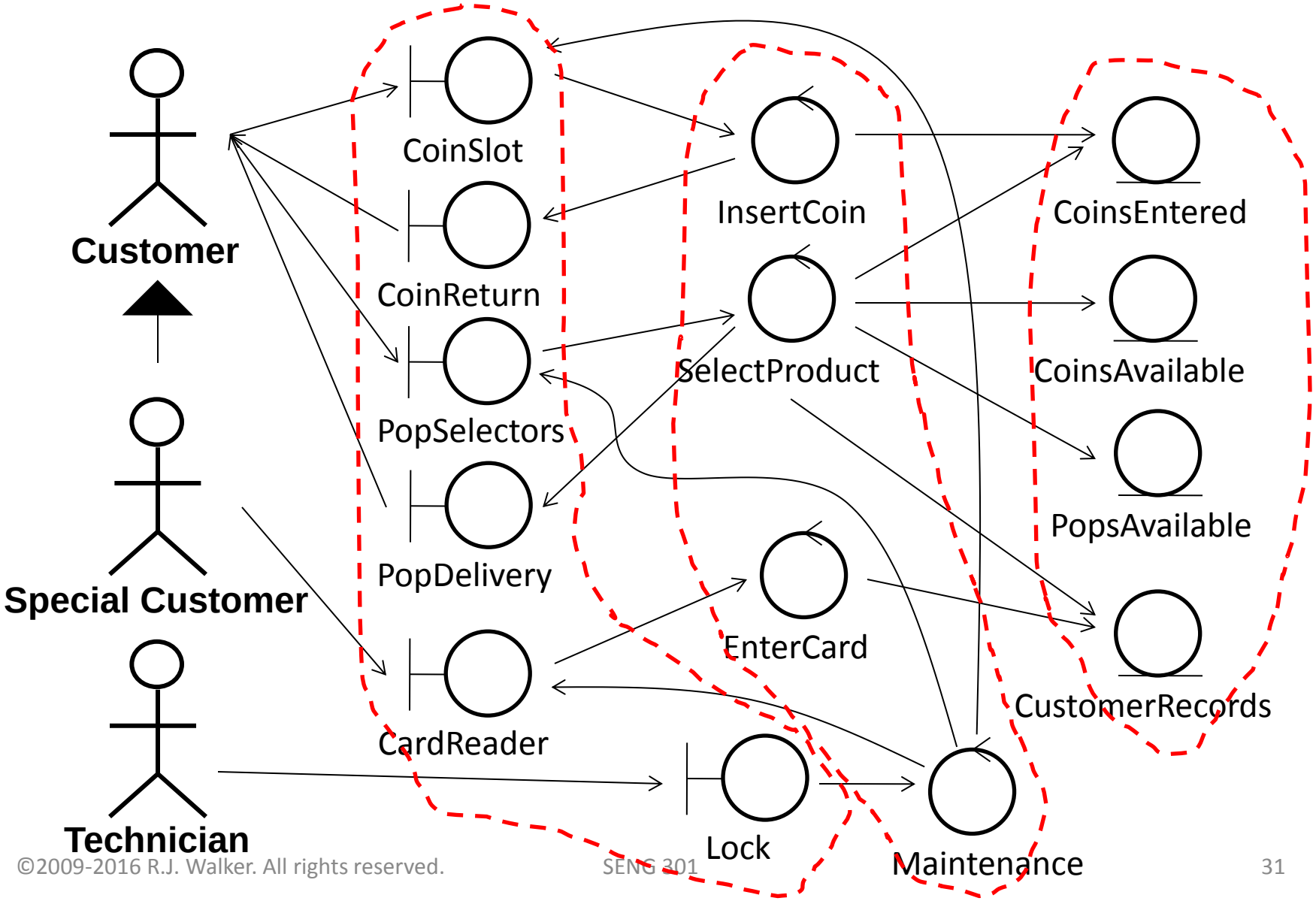
- The use case realizations (UCRs) interact with each other
  - Potential for high coupling in the design
  - Difficult to design/implement each UCR independently
  - Changes to one UCR are likely to impact other UCRs

# PARTITIONING

# Partitioning

- Maybe there is a reasonable way to split up the system into “big chunks” that are:
  - internally consistent & cohesive
  - provide simple (programmatic) interfaces for use by the rest of the system
  - allow one big chunk to know very little about the rest of the system

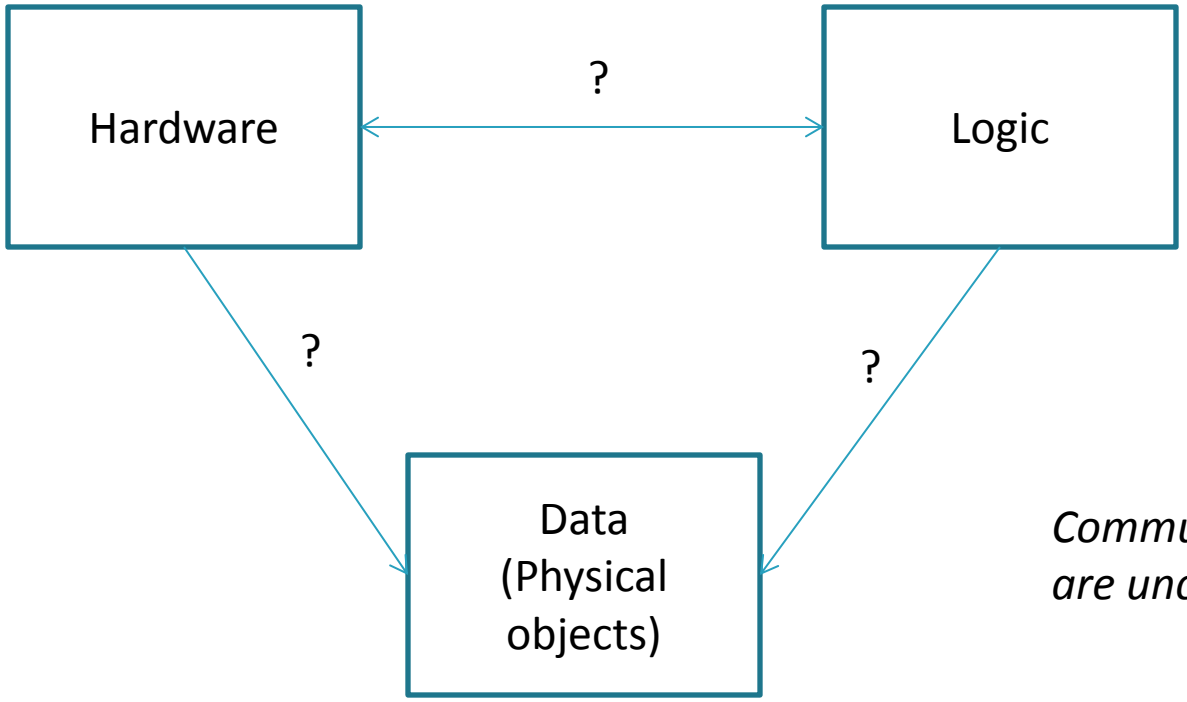
# Simplified dynamic analysis model



# Partitioning

- The analysis classes seem to fall into three kinds of categories:
  - Hardware
  - Logic
  - Data/products/supplies
- On the other hand, there is also:
  - Money handling
  - Pop dispensing
  - Maintenance
- Both could be useful ways of abstracting the ideas there, perhaps both could be used in some way
- But let's start with the Hardware-Logic-Data perspective

# Partitioning: Hardware-Logic-Data




*Communication details are unclear at this point*

*HEY! Is "Hardware" something in the software or something physical?*

*It needs to be both*

# Physical considerations

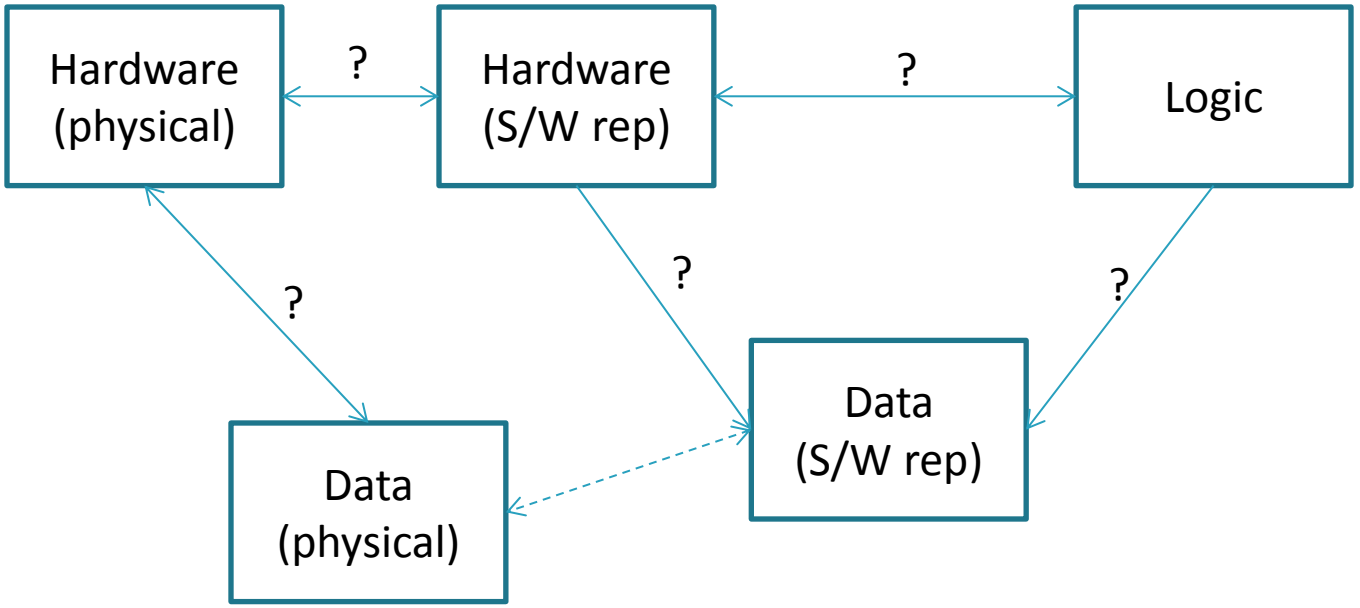
- Coins:
  - Enter a slot
  - Must be collected in some container
    - Return THESE coins if the customer wants
    - Return THESE coins if they are invalid
  - If used for a payment, they must be sorted and stored, available to act as change
- Pop:
  - Stored in racks, one kind per rack
  - Cost is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

# Physical considerations

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

# 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*

```
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)

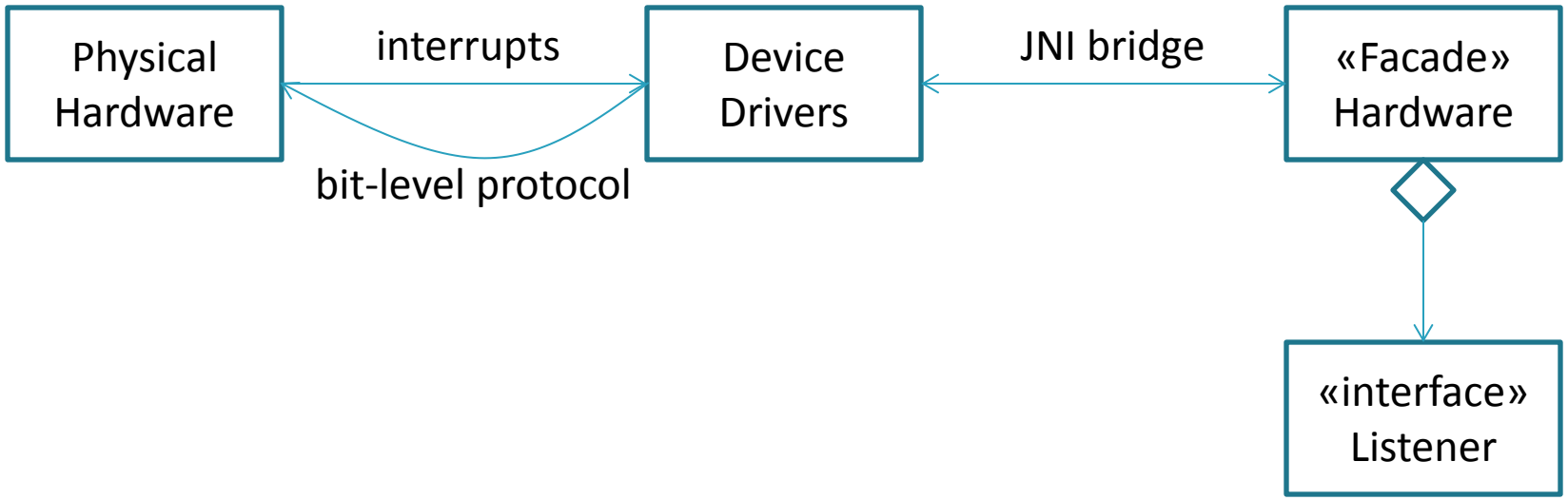
```
interface Listener {
    void coinInserted(int value);
    void popSelected(int rack);
    void coinReturnRequested();
    void deliveryDoorOpened();
    void deliveryDoorClosed();
    void serviceDoorUnlocked();
    void serviceDoorLocked();
    void popRackEmptied(int rack);
    void coinRackEmptied(int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

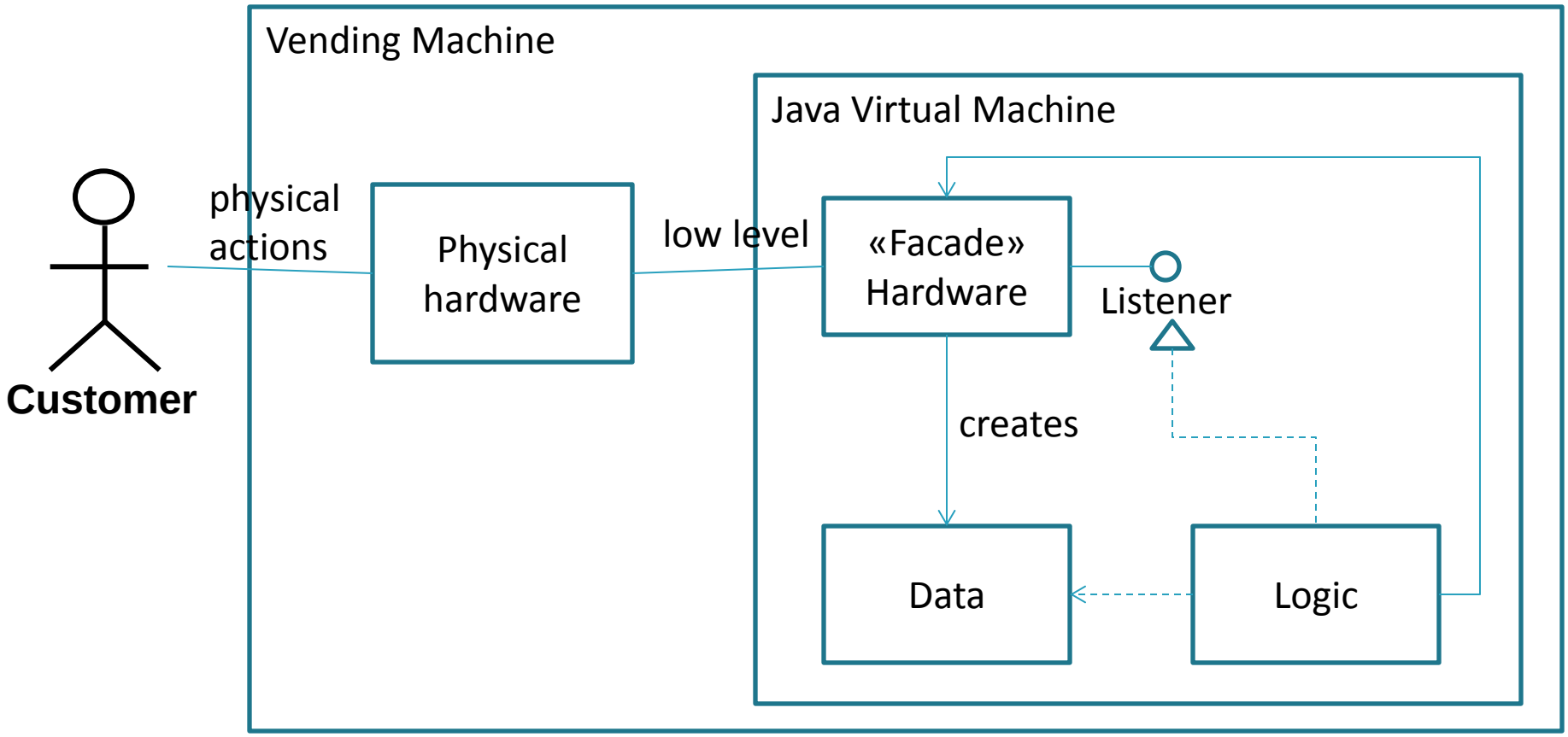
# Back to the big picture




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

# Back to the big picture

*Boxes should be "3D" to indicate nodes*




*But "Data" and "Logic" don't get me any closer to understanding what goes inside*

# Better (?) 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 is 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
  - Use Facade again 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

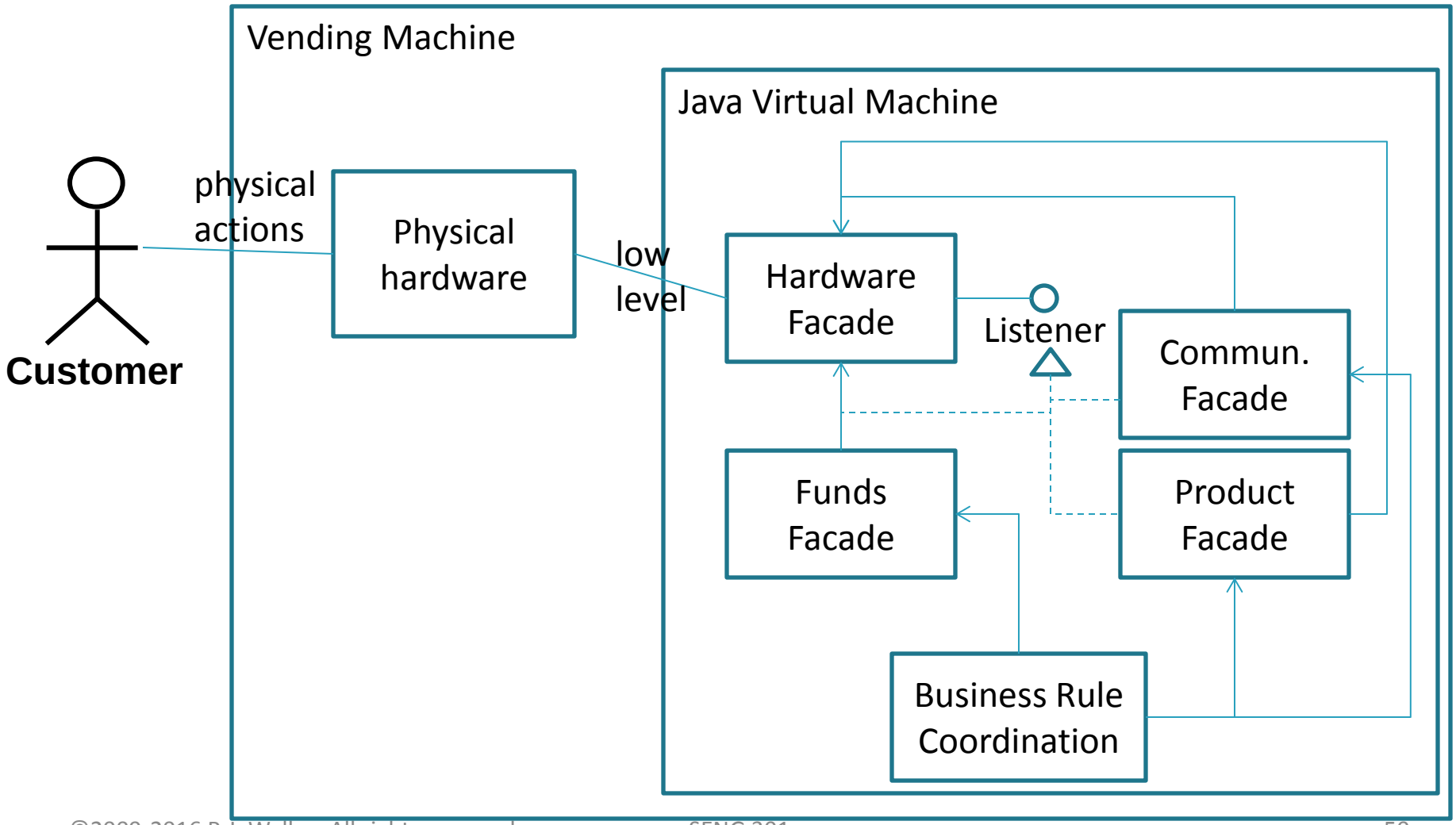
- Perhaps we should be more flexible about the display too?
- Maybe some machines will have very small displays, others will have big displays
- Maybe we will want audio output or accessible output for the visually impaired
- Yep, there should be a Facade for Communication, too

# 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

*Boxes should be "3D" to indicate nodes*

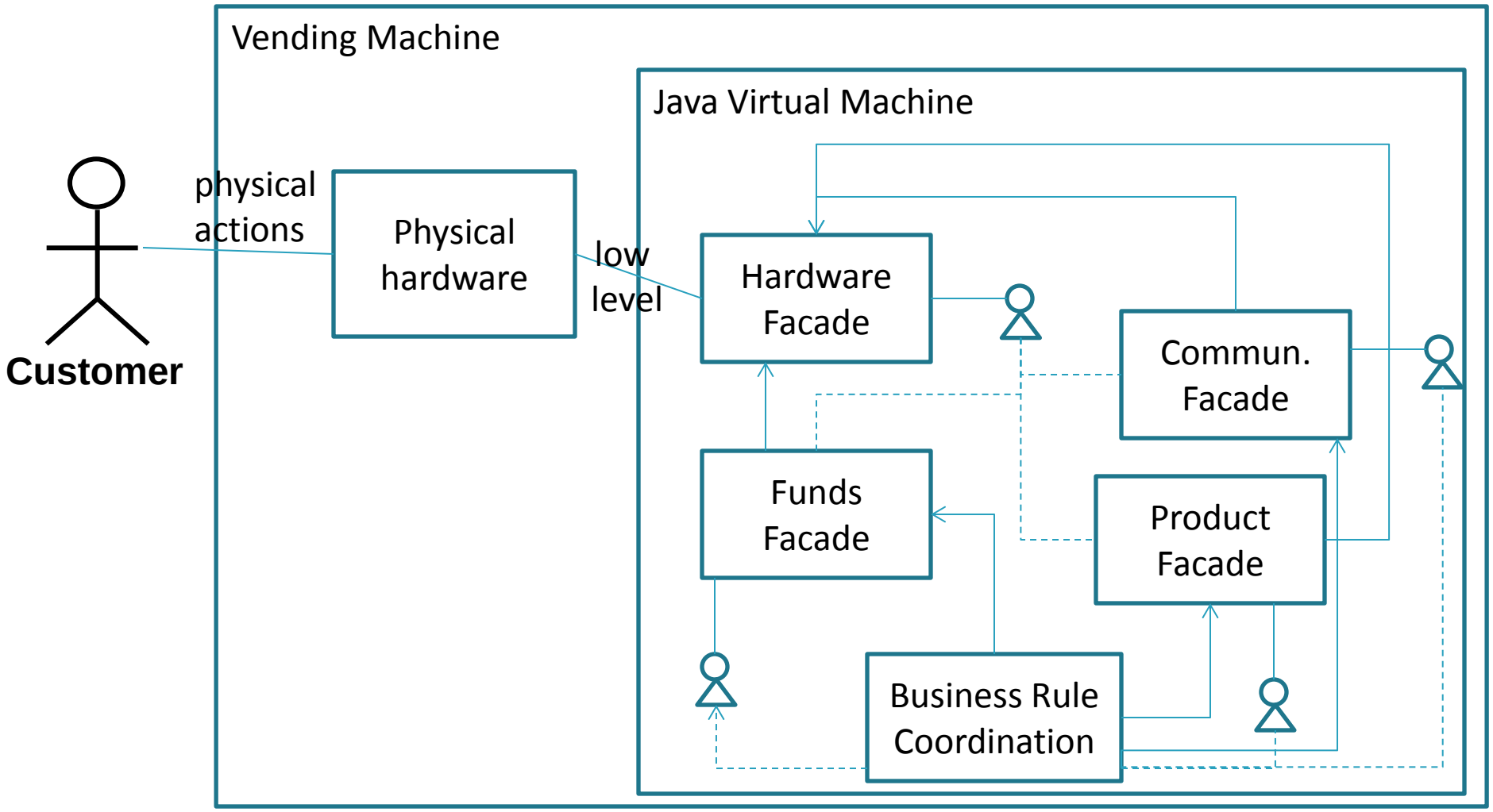


# But wait, there's more!

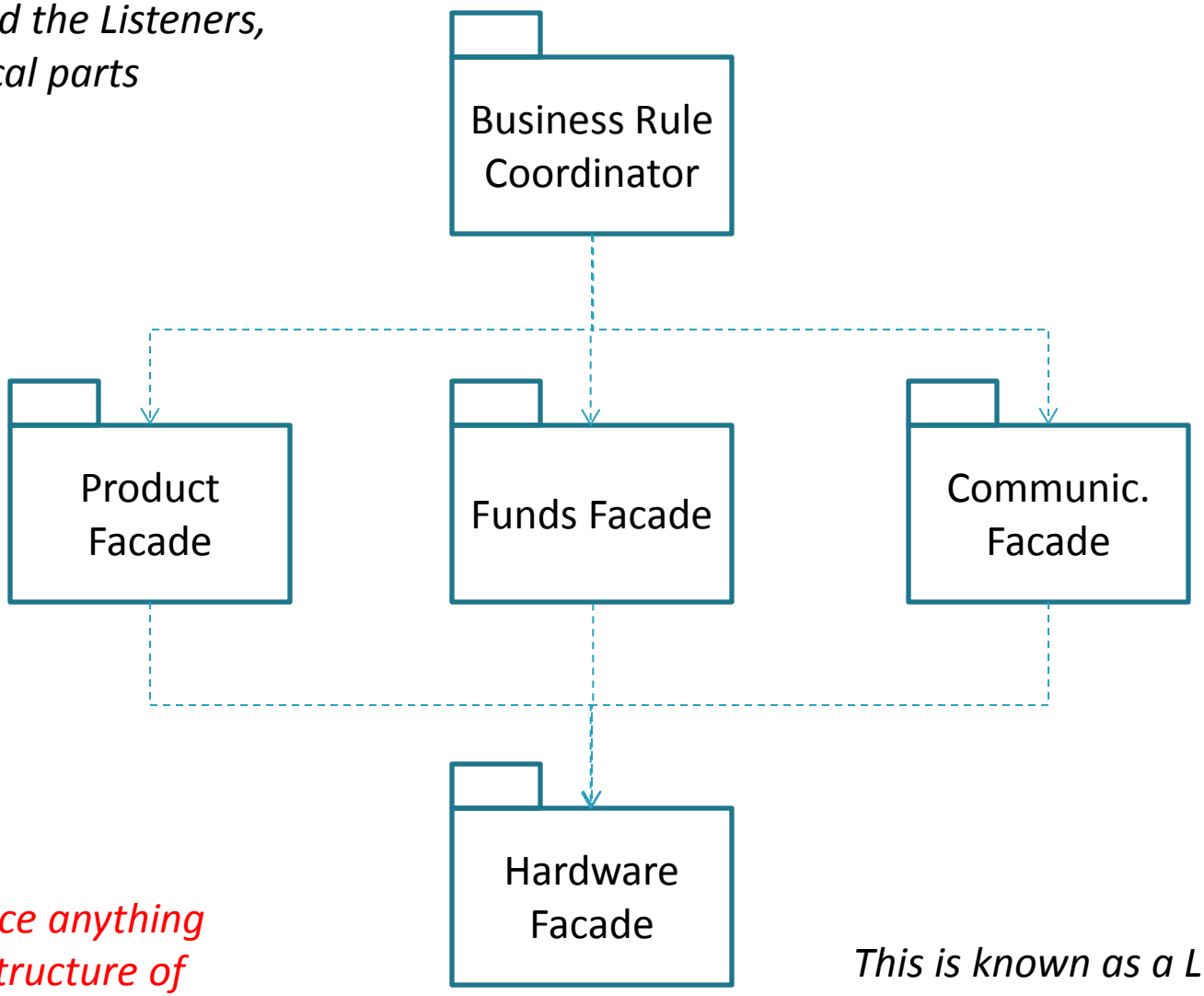
- How can the business rule coordinator know when it needs to do its job?
  - More Listeners

# Back to the big picture

*Boxes should be "3D" to indicate nodes; each interface "lollipop" should be labelled*



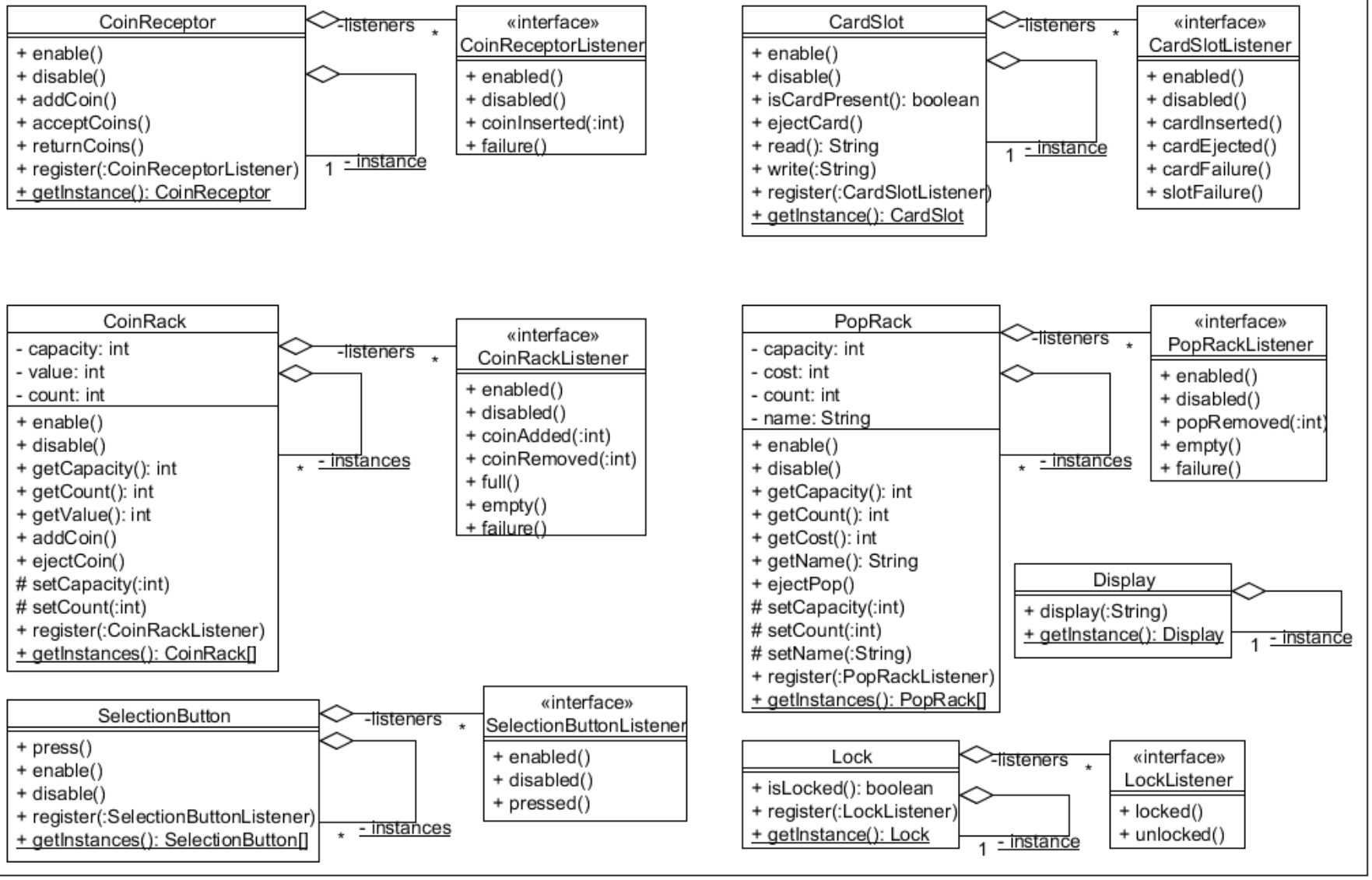
*I've suppressed the Listeners, and the physical parts*

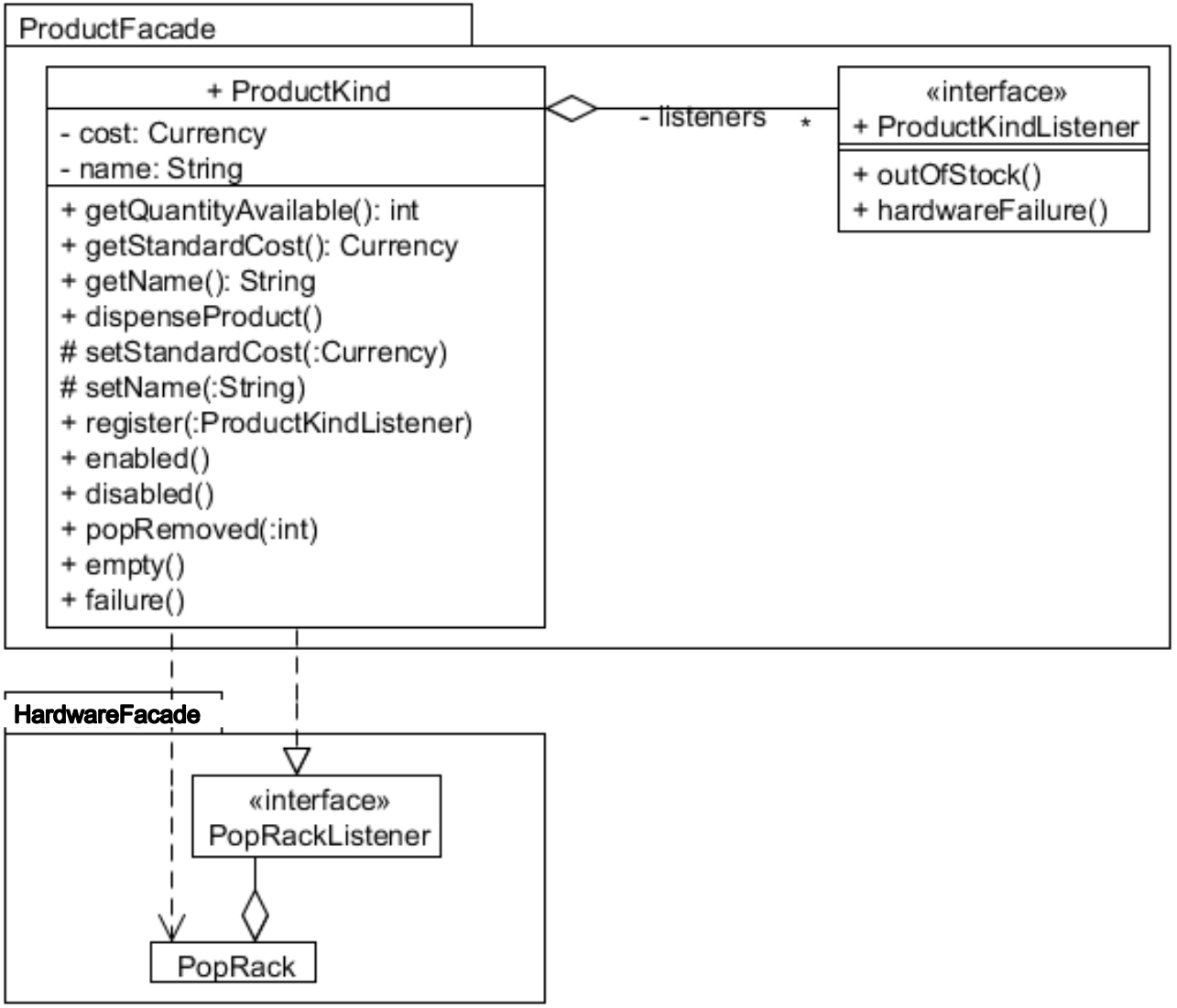


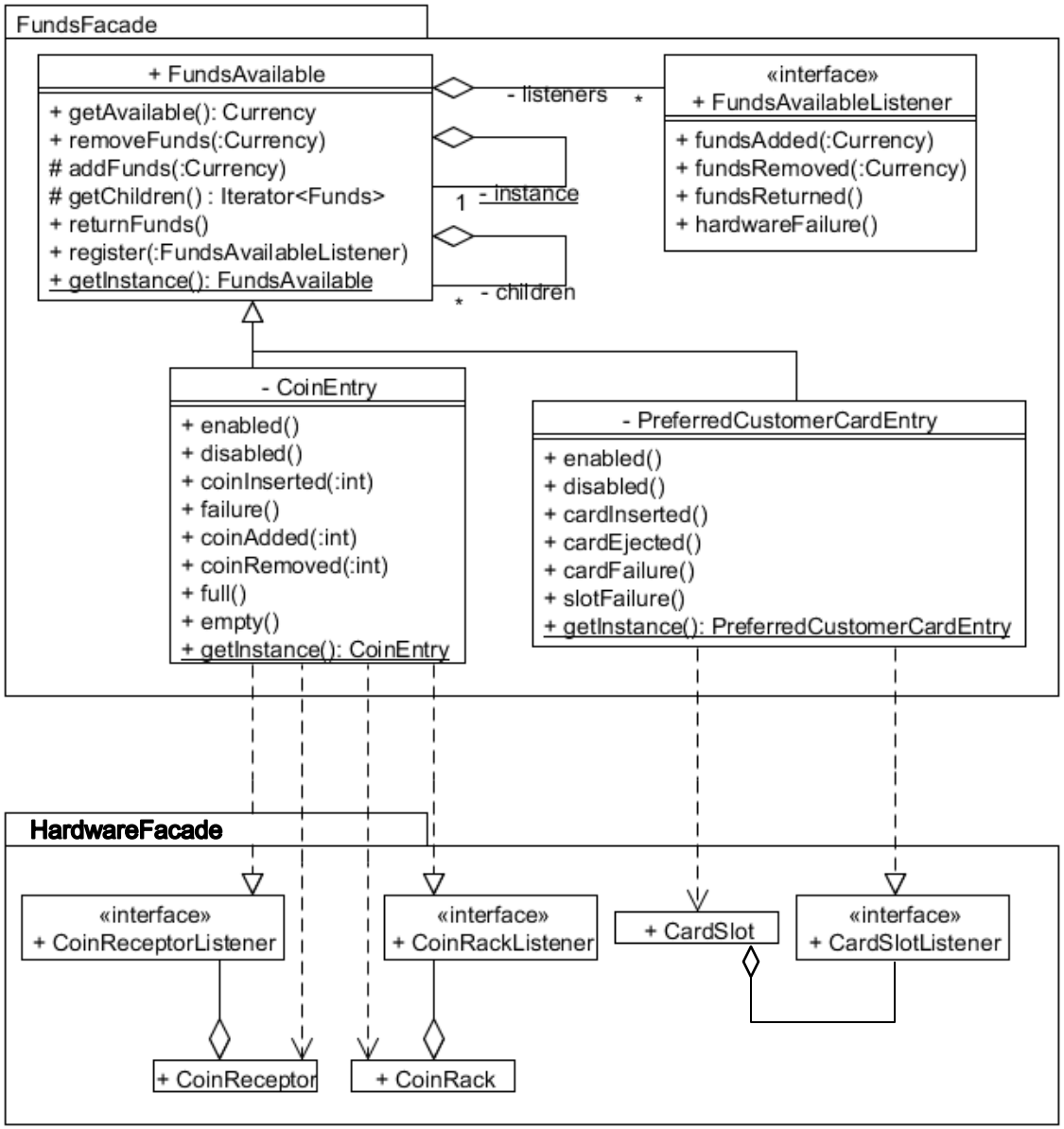

*Do you notice anything about the structure of this arrangement?*

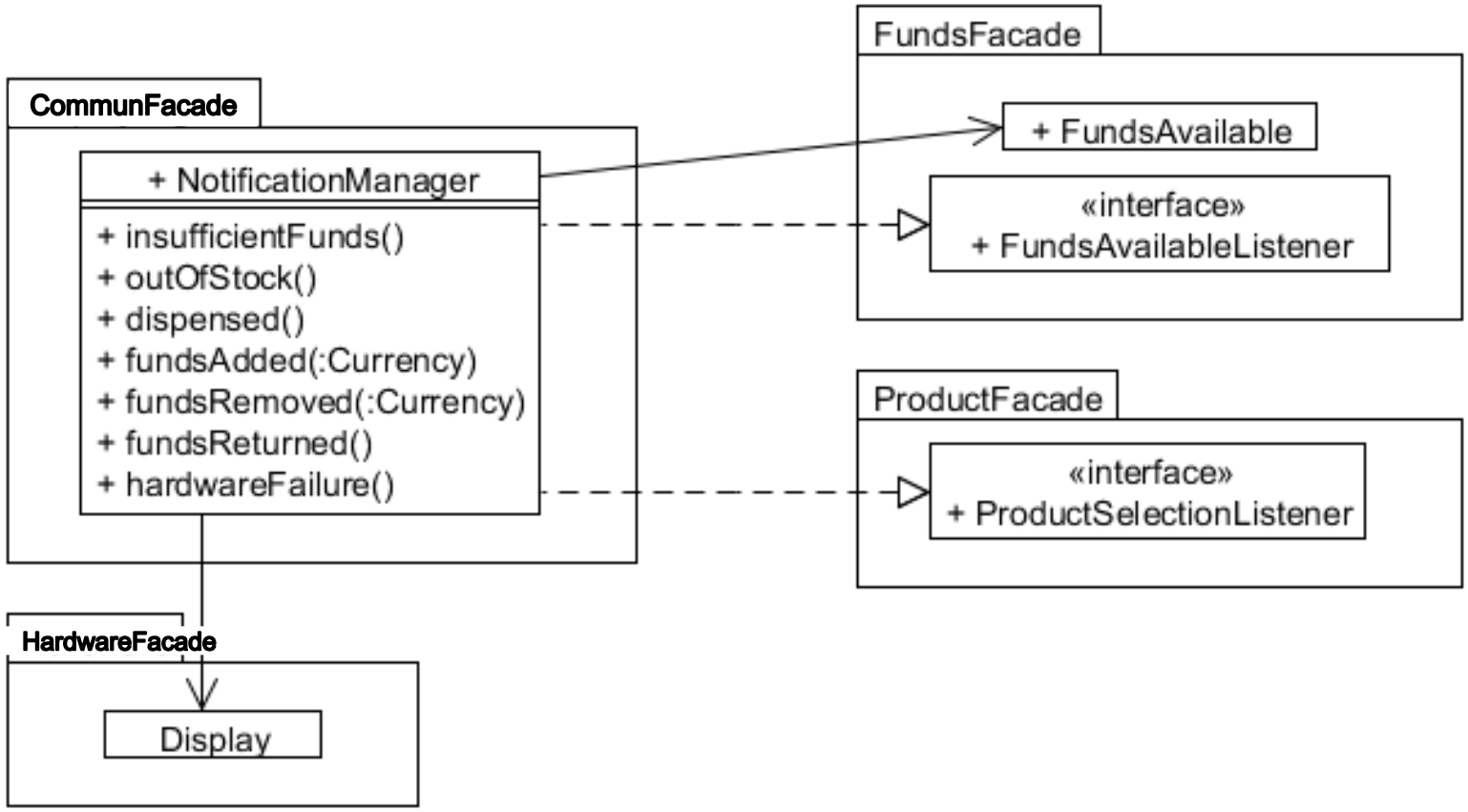
*This is known as a Layered architecture*

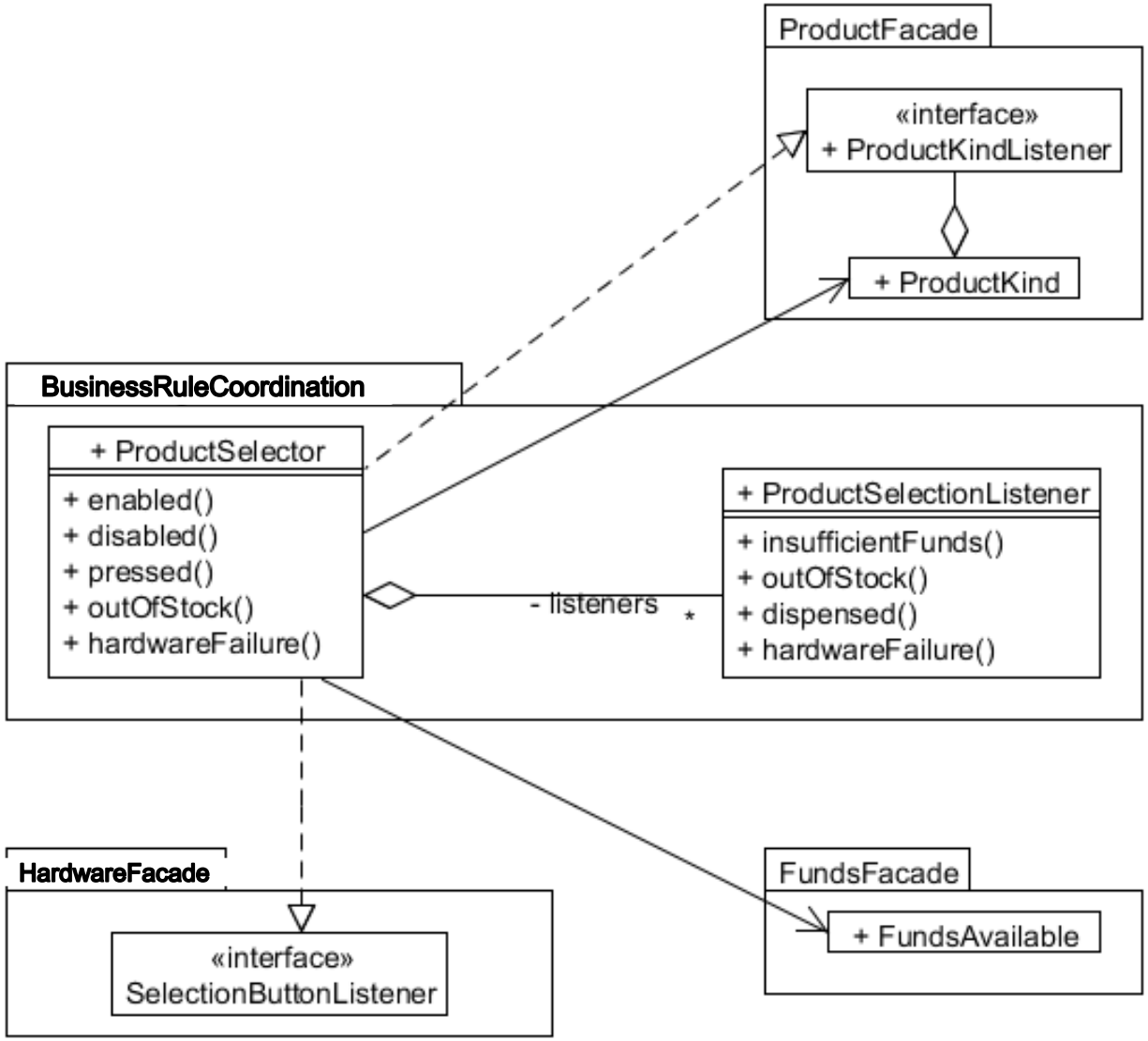
**HardwareFacade**











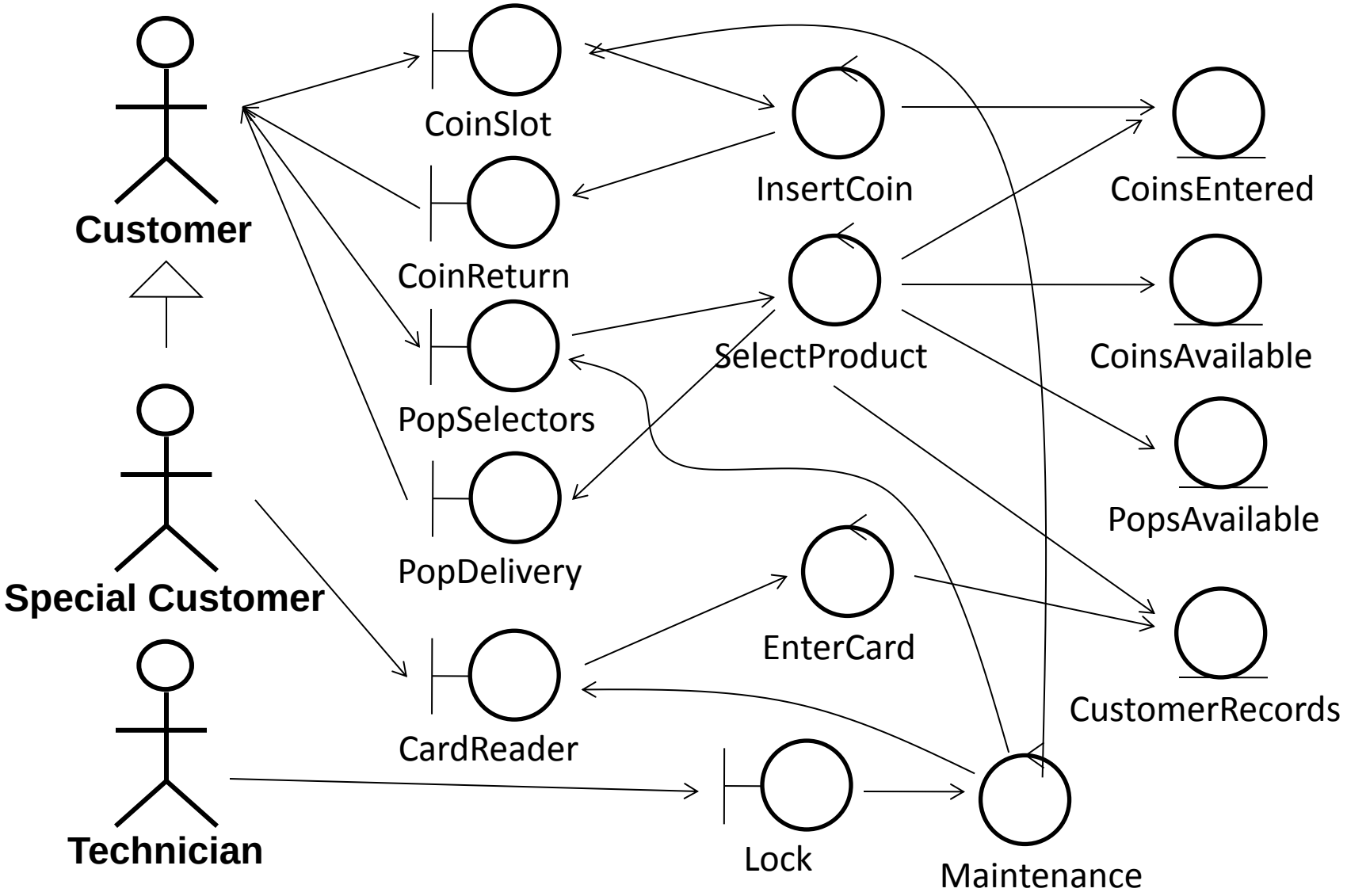
# An issue

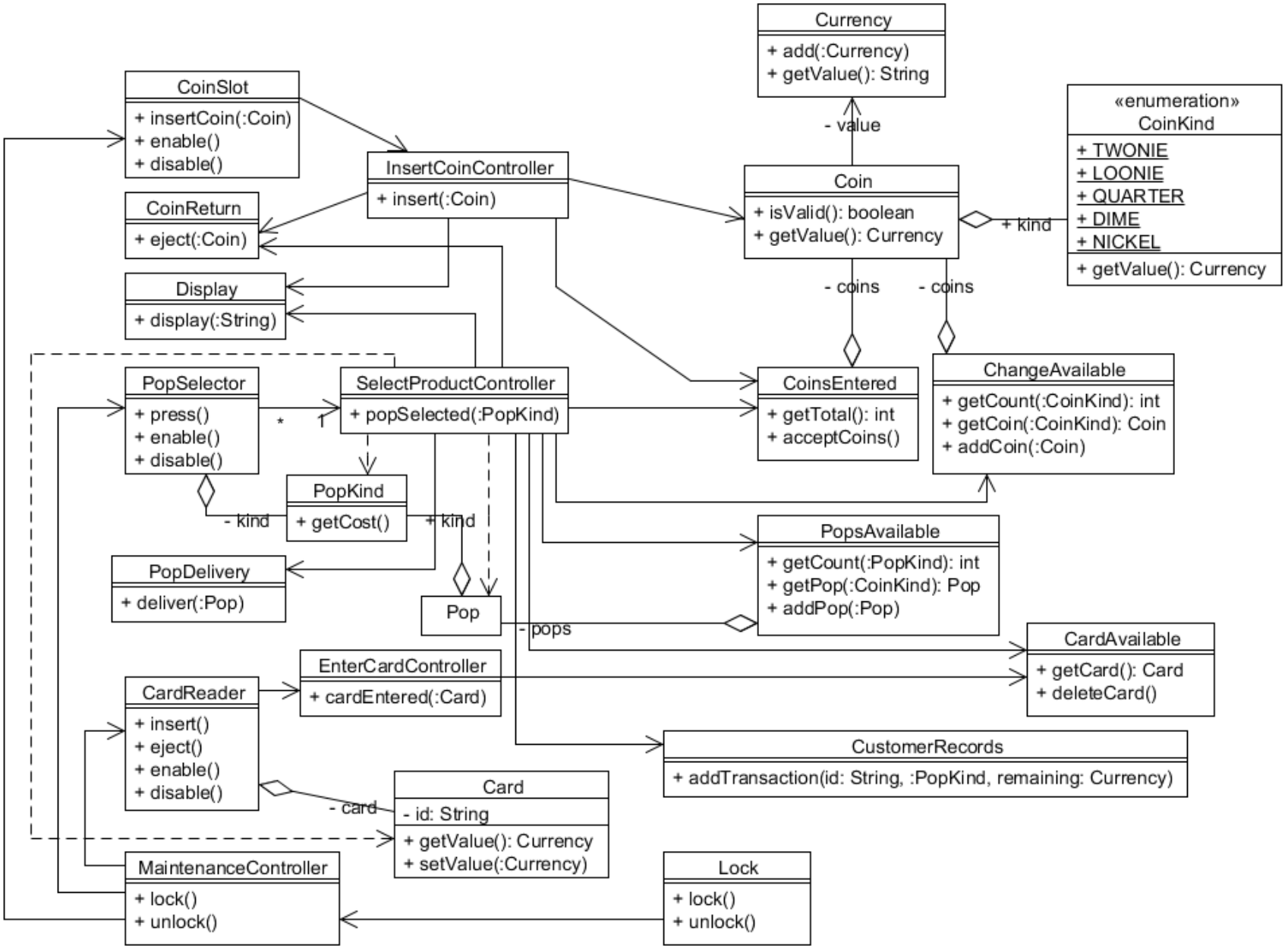
- There is a significant inconsistency between my high-level design and my low-level design
- (There is also a trivial one:
  - I forgot to have ProductSelector interact with the CommunicationsFacade)
- Exercise:
  - See if you can find it
  - Figure out how it could be repaired
- It is not uncommon for these discrepancies to arise: watch out for them

# Alternative approach

- Instead of thinking deeply to come up with a software architecture, and then apply lots of abstraction and design patterns …
- Why not take a more direct approach?
- Let's work directly from the analysis model

# Simplified dynamic analysis model





# Properties of alternative approach

- Much closer to the analysis model
  - Just a few more details added in
- The set of “hardware input classes” still initiate all the action
  - They directly call the controller classes
  - Each controller class calls just the parts of the system it needs to call
- Seems way simpler, right?

# Analyzing the solutions

- Which one is better?
- Dangerous means of analysis:
  - use common sense
  - smaller must mean better
  - let's respect authority: which one does Rob say is better?
- (Not that I don't appreciate your faith in me, but I make mistakes too)

# A principled approach

- Design principles
  - Divide and conquer
  - Aim for high cohesion
  - Aim for low coupling
  - Keep it simple
  - Anticipate problems

# Divide and conquer

- Both approaches can give the impression of supporting this principle
  - Approach #1 would appear to divide things up into smaller pieces ultimately
  - Approach #2 seems to still have some rather large pieces
    - Each controller class has a single method in it, and some of these would seem to need to do a LOT of things: NOT a good sign
    - COMMON MISTAKE: One method => simple
    - REALITY: A method can be very long and very complicated

# Keep it simple

- Who thinks Approach #2 is simpler?
  - Approach #1 uses a bunch of design patterns, lots of abstraction
    - that must be complicated
  - Approach #2: lots of direct calls
    - that must be simpler
- PROBLEM: Models inherently abstract
  - Simplicity in the model does not guarantee simplicity in the real thing
- Approach #1 does nothing "tricky" (design patterns may be unfamiliar to you, but these are common ones)
- I wouldn't want to be the one responsible for getting popSelected(:PopKind) working!

# Aim for low coupling

- Approach #2:
  - Most obviously, SelectPopController depends on 11 other types explicitly (plus Card, which I forgot/chose not to show)
    - Should we just split it into smaller steps?
    - The issue is that this would not be an OO design, but a procedural decomposition
- Approach #1
  - I found a couple of parts that depend on 5 other types (ProductSelector, CoinEntry)

# Aim for high cohesion

- Approach #2:
  - What does PopSelectionController deal with?
    - Every aspect of selecting pop: checking cost, checking coins entered, checking card entered, displaying whether the amount is sufficient or not, retrieving the appropriate pop, determining any change, ejecting coins, ejecting pop, ejecting card, …
  - Similar but less severe problems for other controllers
- Approach #1
  - ProductSelector explicitly delegates tasks to FundsAvailable and ProductKind, coordinates results but that is all (should interact with Display, too; I forgot that)

# Anticipate problems

- Problem 1: Maybe we forgot some functionality
  - Due to Approach #1's details, easier to modify it reliably
  - Approach #1 shows high cohesion in many of its pieces, and each piece can cope better with error handling
    - In Approach #2, it is very likely that you have forgotten to do certain things; centralized control makes this more likely

# Anticipate problems

- Problem 2: We want to add support for other payment types
  - Lack of abstraction in Approach #2 will result in PopSelectionController becoming ever more complicated internally, and its high coupling becoming even higher

# Anticipate problems

- Problem 3: When can you check to make sure that the pieces are working properly?
  - For Approach #2, only the controllers do much of anything, and they require the use of lots of "data classes"; hence, basically everything has to work before anything works
  - For Approach #1, the layered architecture implies that we can worry about individual packages and test these somewhat independently

# Counter-arguments

- "But Approach #2 is likely to be more efficient"
  - Only if you manage to get it to work, which will be difficult since modifications (including bug fixes) will be difficult to localize
  - Efficiency only matters relative to the context

# Counter-arguments

- "But Approach #2 will be cheaper and faster to develop"
  - Only if you don't understand anything about an OO approach
  - You will likely make mistakes and need to change details because you overlooked them or the requirements need to change: Approach #2 makes this harder
  - "Mega-methods" take a long time to develop (Approach #2)
  - Coordinating "mini-methods" is typically a lot faster

# Selecting the design goals

- Early on I mentioned that it is important to decide on the design goals for a given system

- From above discussion:
  - maintainability
  - comprehensibility
  - safety
  - security

*It appears that these are the most important properties for consideration*

- (Normally, you would decide this first, but I wanted to keep you thinking about whether Approach #2 was a VIABLE alternative)

# The takeaway message

- If you have understood:
  - “Approach #1 is object-oriented
  - “Approach #2 is procedural
  - “Therefore, Approach #1 is better”
  - … you missed the point
- Sometimes, a procedural approach is more appropriate, but I would say: not this time