Software Engineering 301:
Software Analysis & Design

Conclusions

Labs

- Last set of labs are due Friday at midnight

- I will turn the labs back on early next week so you can practice
  - No effect on grades!

Final exam

- Format just like midterm
  - Bring a pencil
- 1 page cheat sheet again
- 25 questions
- Entire course is fair game, but focus on design
  - HINT: You want to study my Assignment 4 solution and the lecture slides related to it

Agenda

- Modelling
- Testing
- Requirements
- Design
- Future directions

Models

- Every model is an idea or set of ideas about something
  - discards unimportant details
  - emphasizes important details
    - "important" details depend on purpose of model
- This is also abstraction
  - An object-oriented implementation is therefore a model of something!
- Models must be represented
  - notation, words, images, physical constructs

Abstraction versus detail

- Two key questions:
  - How much (and which) detail should be shown?
  - What should the form of the model be?
- Answers depend on:
  - What is your purpose?
  - Who is your target audience?
  - What level of maturity are you aiming for?

Are models worthwhile?

- IF …
  - they are cheaper to create/understand than the real thing
  - they permit analysis of complex points
  - they can be understood
- Common developer's reaction to models:
  - "Models are a waste of time; I'll just write source code"
  - The mistake is in the universal assumption here:
    - SOMETIMES models are a waste of time, not always
    - "Just-enough" modelling is the target to strive for, not more, not less

Structure versus behaviour

- Structural diagram
- Behaviour:
  - sequence diagram
  - (communication diagram)
  - state machine diagram
  - (activity diagram)

Three points of importance in your models

- Syntax
  - Using the appropriate symbols to convey your intent
- Semantics
  - Combinations of symbols imply certain properties
  - e.g., *A IS-A B* and *B IS-A C* means that *A IS-A C*
- Concepts
  - The ideas to be represented

Testing

- Testing is a process of comparing expected results against actual results
  - differences indicate the presence of bugs
  - lack of differences DO NOT indicate the absence of bugs
- Testing is performed to PROVE THAT BUGS EXIST

Test case selection

- Exhaustive testing is impossible for all but the most trivial situations
  - Bugs likely lie in cases where you have not tried
- We must choose which cases we worry about
- Risk: which cases are most likely to occur in practice, what potential bugs would be most serious
- Standard strategies: boundaries, equivalence classes

When to test?

- An implementation that compiles and runs is need to perform tests
- Tests can be planned before implementation starts
  - Allows us to better understand what we are trying to implement

Automated testing

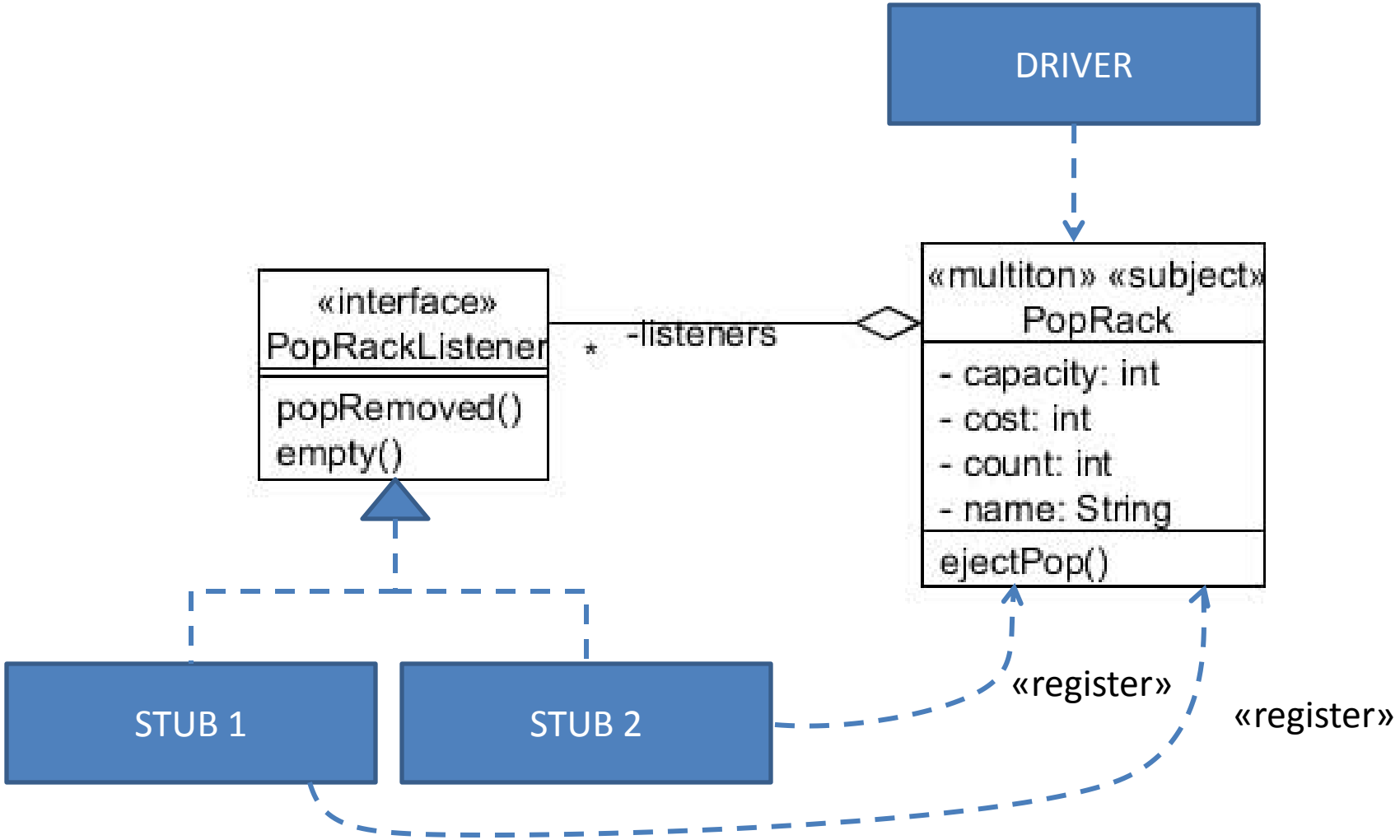
- Manual testing is tedious, error-prone, and not very systematic
- The same test cases will need to be performed multiple times over time
- Automated tests are a simple means of checking for regression, as well as serving during debugging

Testing at different granularities

- Individual units, a set of integrated units, or the entire system
- Why?

1. Problems may involve incompatible components that work fine in isolation
2. Test cases may not have detected problem

Where to put the stubs & drivers?



Requirements

- The requirements for the system define:
  - WHAT the system should do
  - Not HOW the system should do it (usually)
- Different stakeholders have different goals
- Requirements (and stakeholders' understanding) change over time
  - Iteration and design for change are key
- "Must" vs. "should" vs. "would be nice"

Requirements analysis

- Two kinds:
  - Looking for problems
    - correctness, completeness, consistency, etc.
  - Looking for design constraints
    - text analysis, object identification, etc.

Design

- A set of decisions about how to realize the requirements
  - Made before the implementation (specification)
  - Made after the implementation (descriptive model)

Good design

- Fulfills the requirements
- Selects amongst the remaining possibilities to achieve a set of non-functional properties
  - Depend on the specifics of the project
  - Worrying about changeability and security are typical choices nowadays
- Every design ought to aim for a set of goals

How to design?

- Iterate:
  - Break big things down into smaller ones
  - Put little things together into bigger ones
  - Reuse existing solutions (libraries, design patterns)
  - Look for weaknesses and strengths
  - Try out a bit and be ready to back out of it

High-level versus low-level

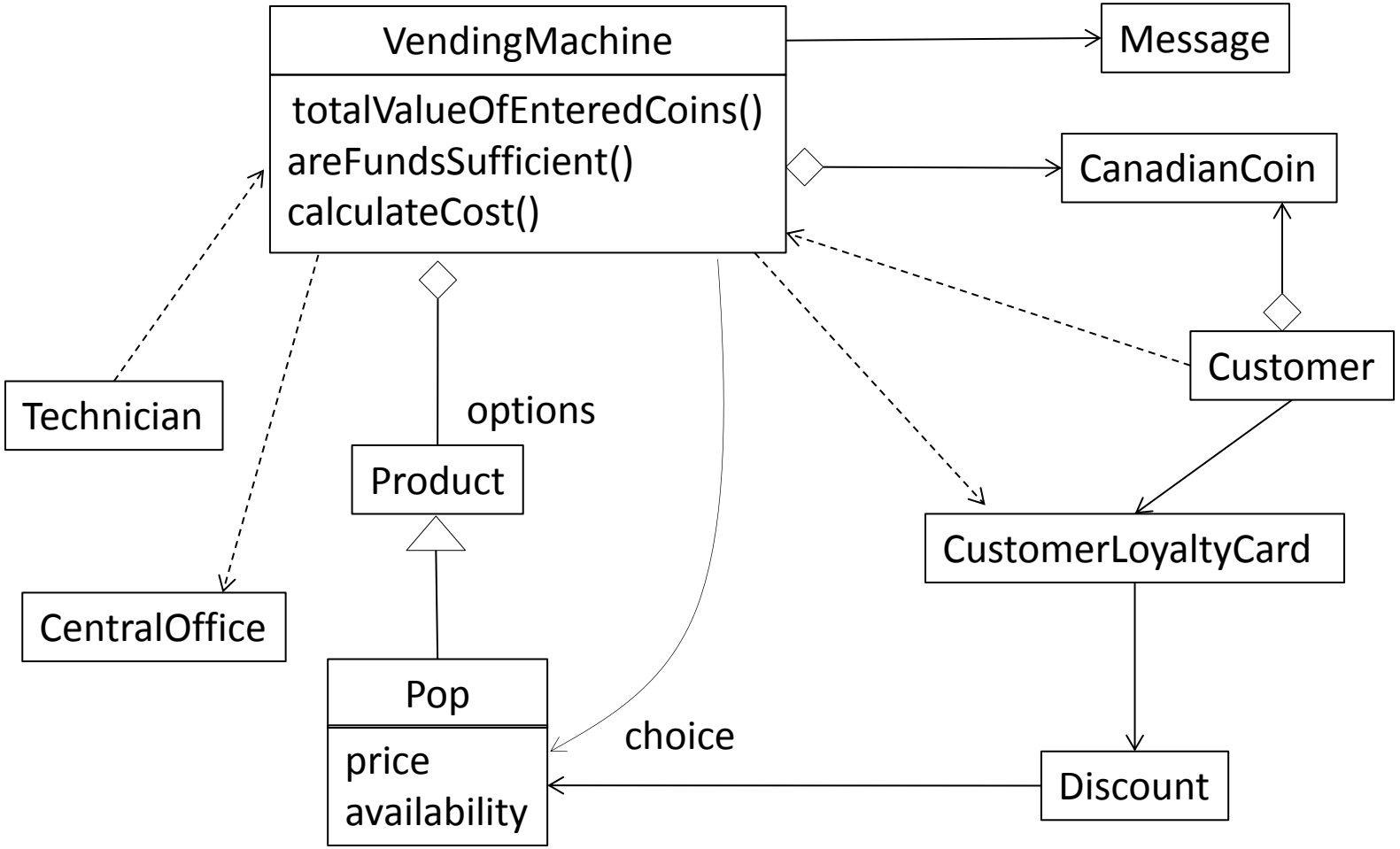
- Low level design consists of what goes inside of a given class
- High level design worries about sets of classes and sets of sets of classes
- High level design is necessarily more abstract; thus more complicated to deal with

Typical process

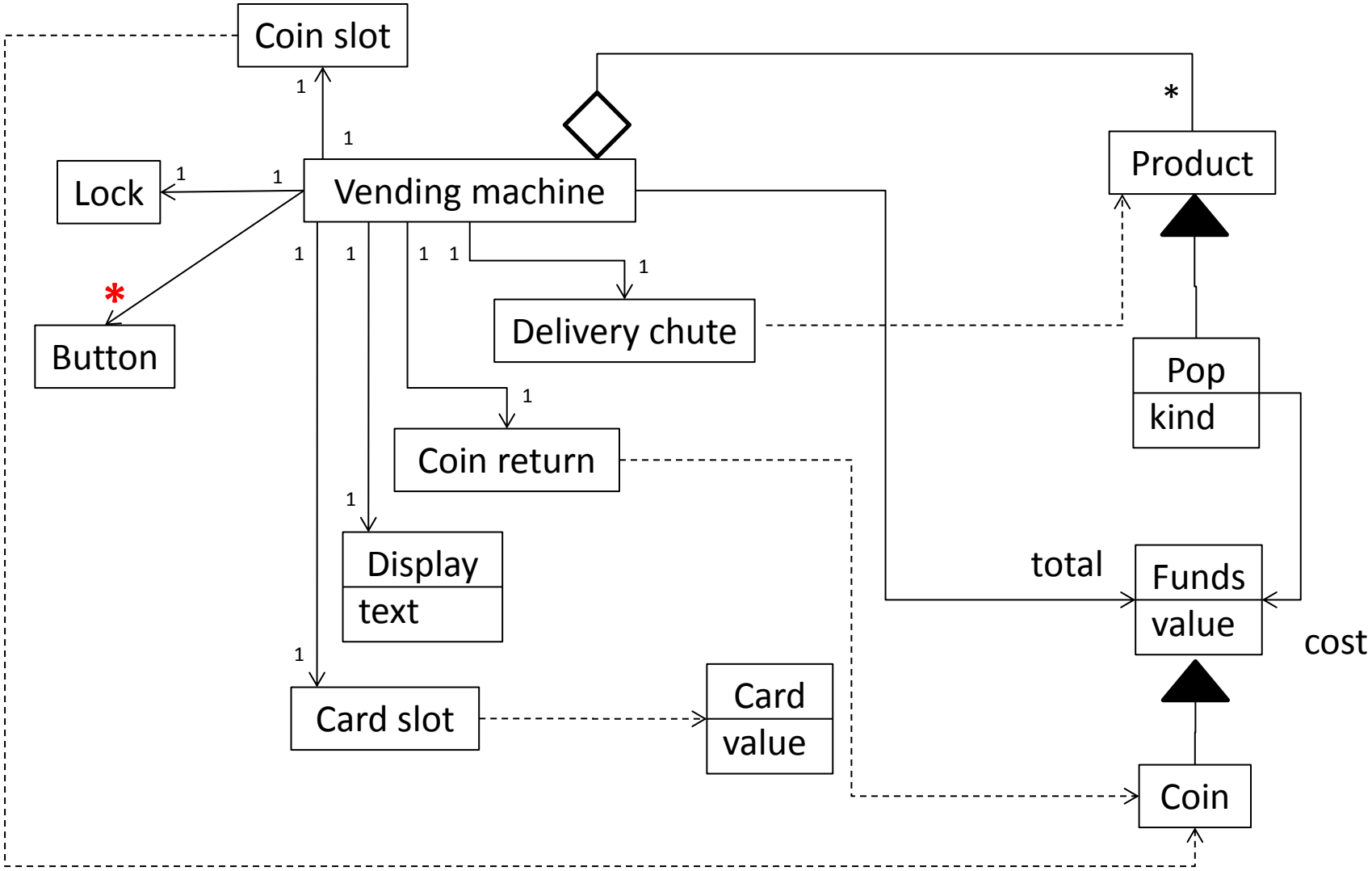
- Perform text analysis to identify objects and classes; refine
- Consider how the functionality will be supported; refine
- Partition; consider whether the partitioning has reasonable properties
- Look for problems; repeat

Analysis for design

- The words that stakeholders use are likely key concepts that the software will need to model
- Identify parts of speech in text that describes the requirements
  - nouns, verbs, adjectives, etc.
  - natural language representation, use cases, etc.
- Organize as classes, fields, etc.
  - Eliminate redundancies, ambiguities
  - Iterate



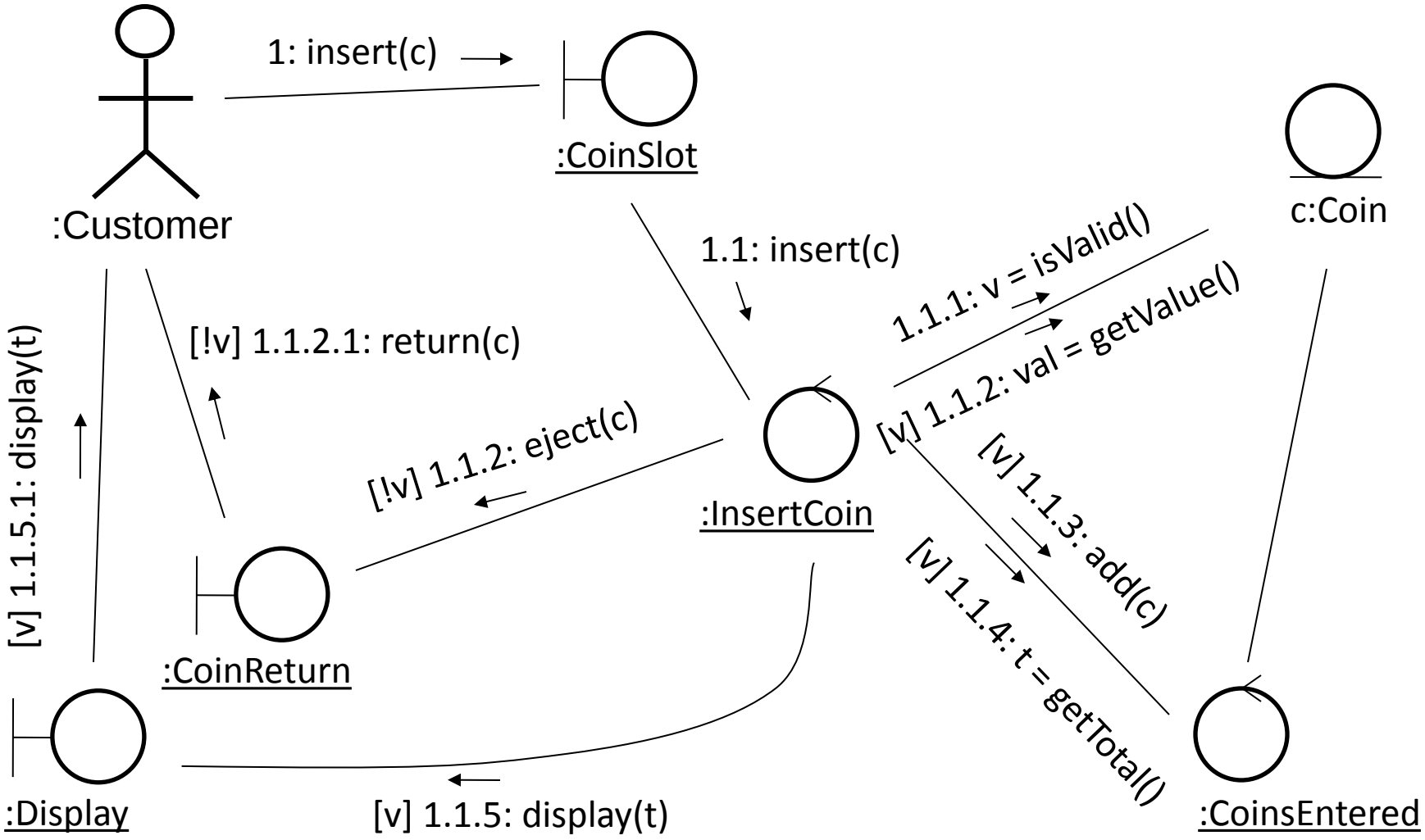
Structural model of the analysis



Analysis for design

- But the requirements describe externally visible properties
- Thus the analysis will only deal with external properties
- Care needs to be taken to ensure that needed functionality is not forgotten

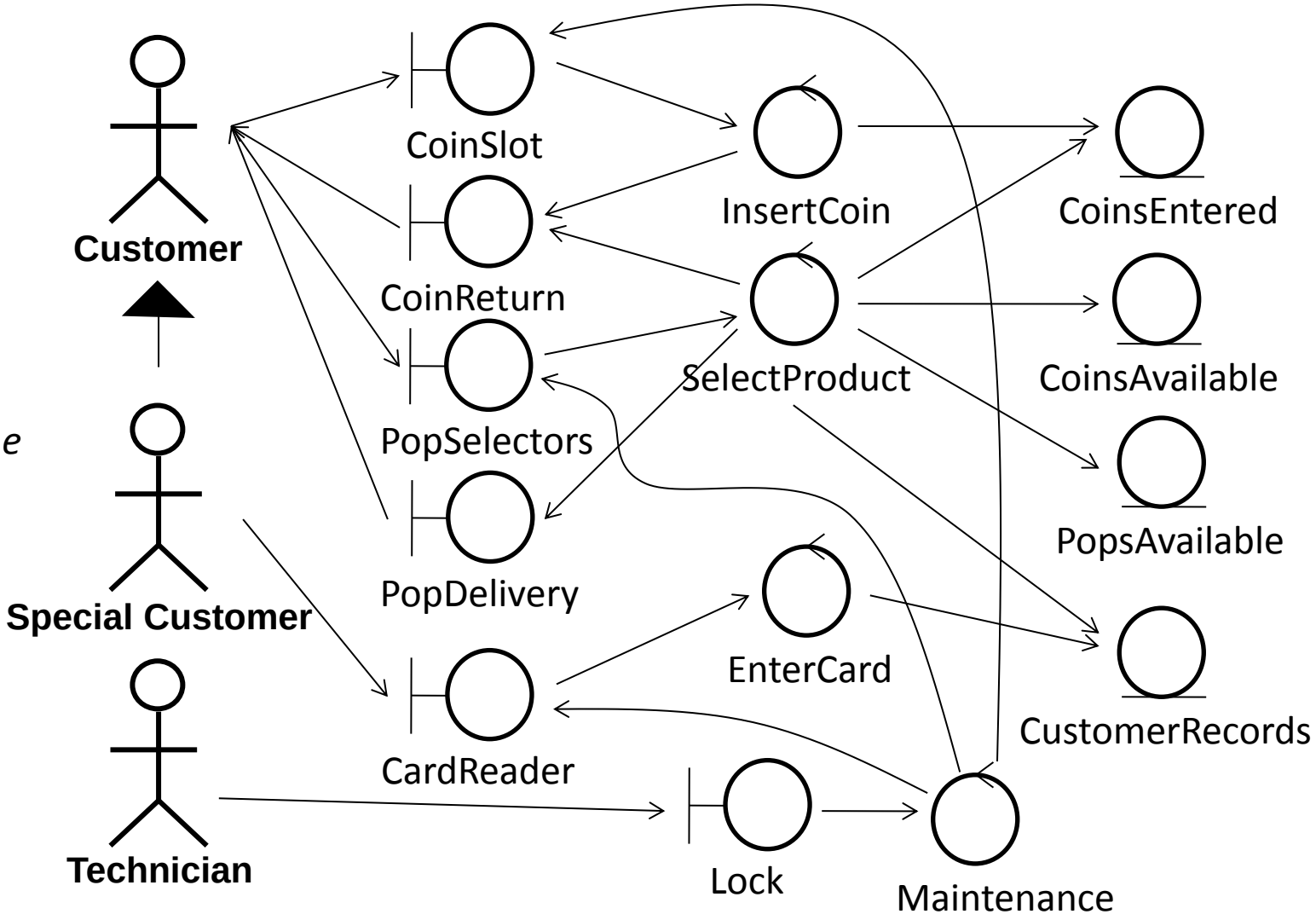
Realization for Insert Coin functionality



Simplified dynamic analysis model

Display should be here too, as a boundary object

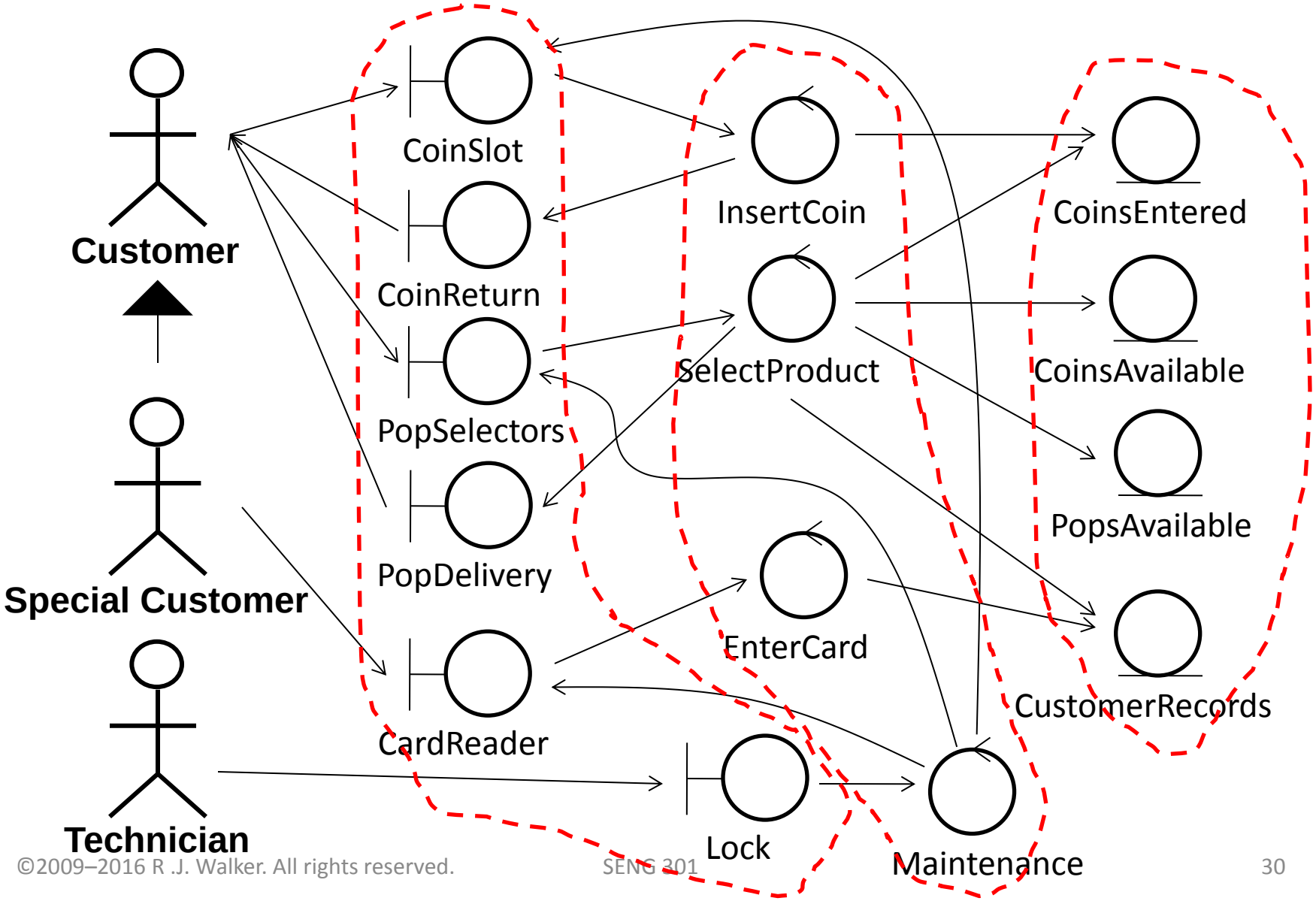
Messages being passed are suppressed in the diagram



Partitioning

- One route forward is to start figuring out how to divide the complex, overall system into smaller, more manageable pieces
  - This is partitioning
- But not all alternatives are equally attractive:
  - hardware – logic – data
  - hardware – funds – product – communication – business rules

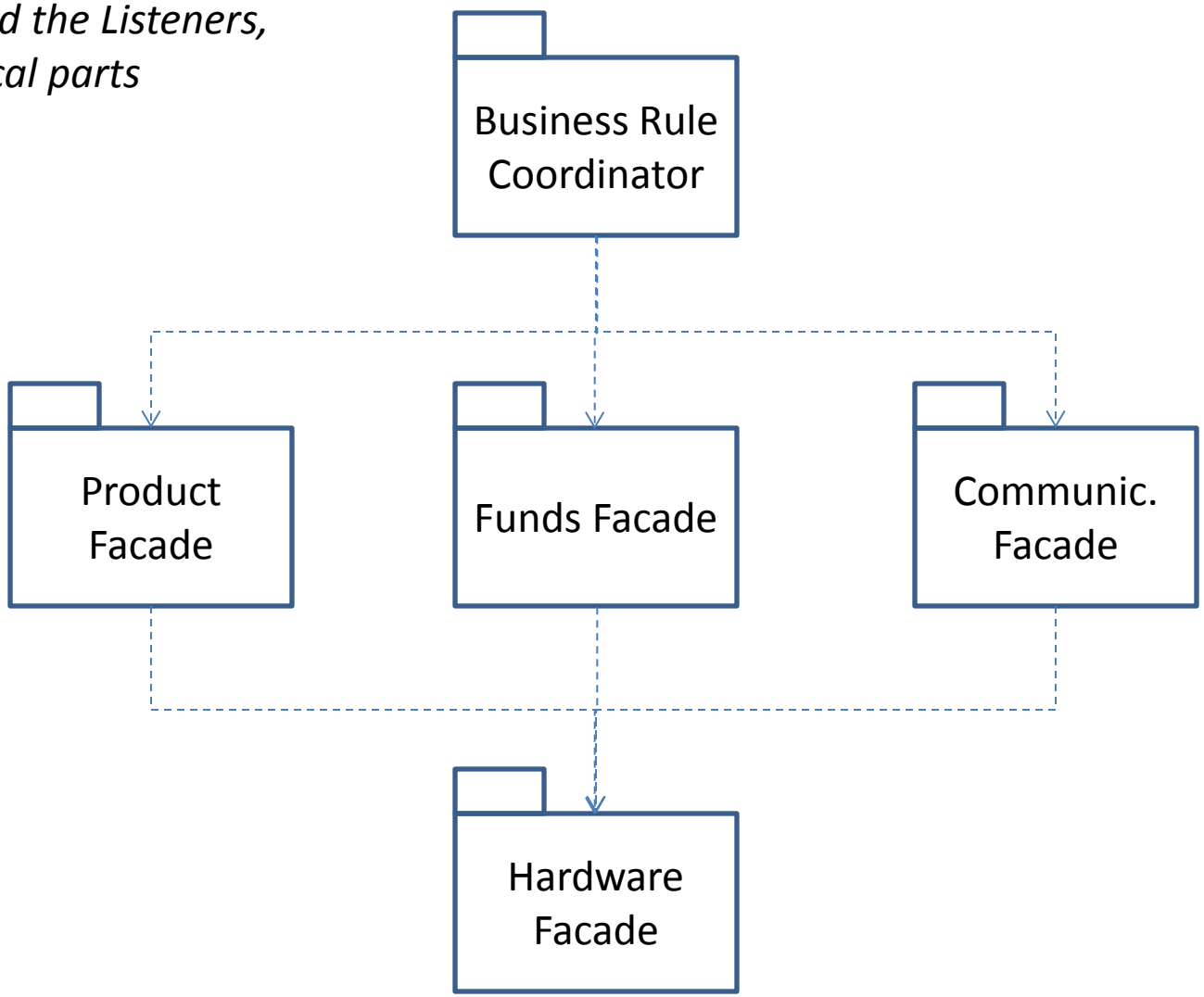
Simplified dynamic analysis model



Hardware – logic – data

- Fails to abstract away complex interactions
- Fails to simplify the individual pieces
- Fails to lead to more than superficially simple parts

I've suppressed the Listeners, and the physical parts



Other partitioning

- The functionality is well abstracted
- Each part provides an abstract machine for the parts depending on it
- Each part has a well-defined purpose, reduced from the purpose of the whole
- The parts are simpler than the whole

In general

- Partitioning needs to consider the physical needs of your system
  - Multiple, distributed computers
  - Pre-existing applications (e.g., database) to be used
- Communication overhead is important
- Security concerns are important

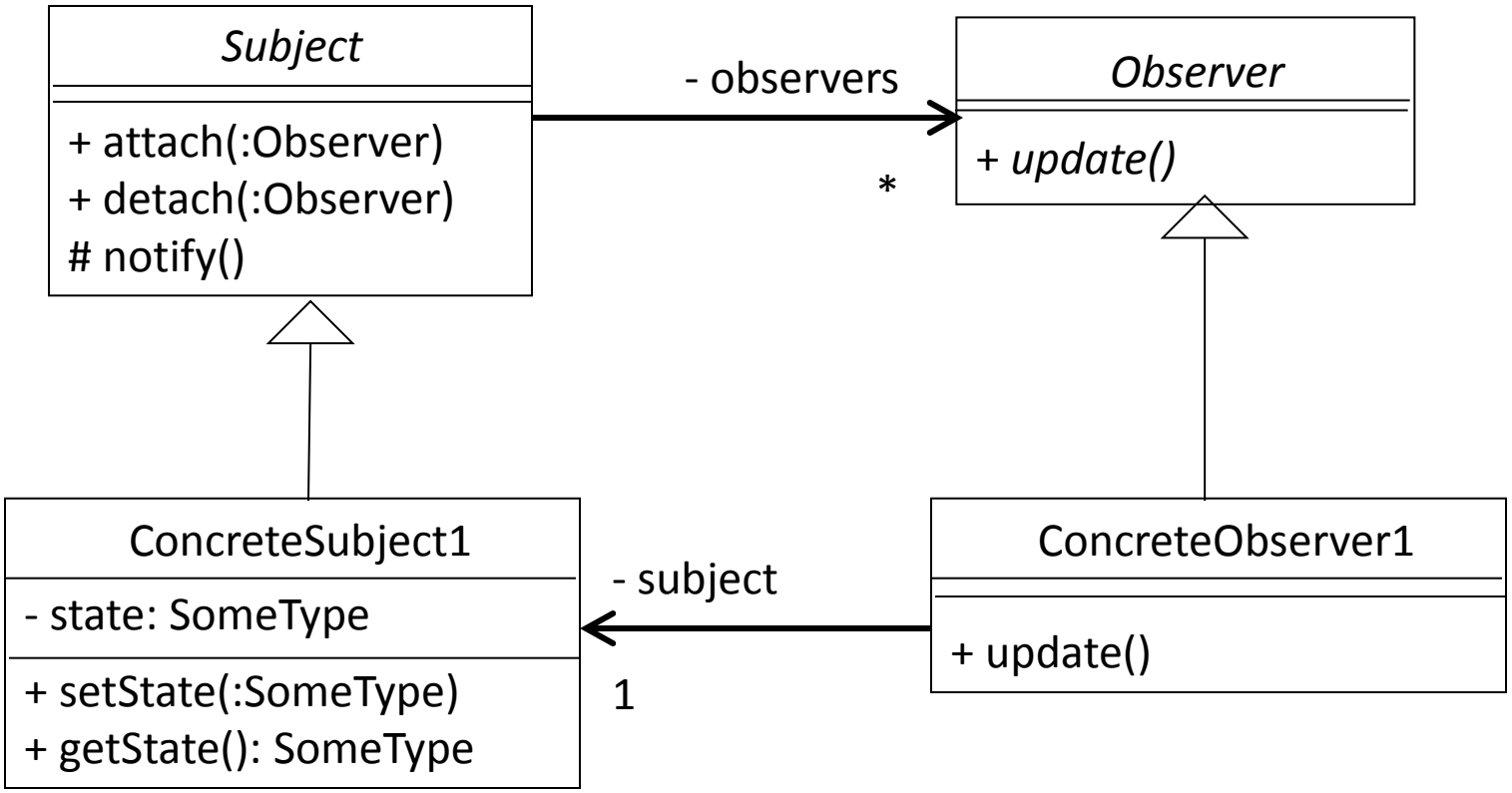
What's a design pattern?

- "describes a problem which occurs over and over again in our environment,
- "describes the core of the solution to that problem,
- "in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Design patterns

- We looked at three (observer, façade, singleton), but there are many more
- Confusions:
  - Abstract diagrams are suggestive of designs, they are not real designs
  - Every design has consequences (both positive and negative); design patterns are no different

Observer: structure



Design principles

- Rules of thumb about good and bad things to do in a design
- Derived from people's experience in software development
- These do not apply in all situations and all the time
- Do not disregard them unless you are sure that you need to

The principles

- Divide and conquer
- High cohesion
- Low coupling
- Hide information
- Keep it simple
- Anticipate problems

Abstraction is your friend

- If something is complicated, it is likely to change
- If something is complicated, it is hard to understand
- If something is complicated, it is easy to get it wrong
- Abstract it away until the complexity is eliminated or at least hidden

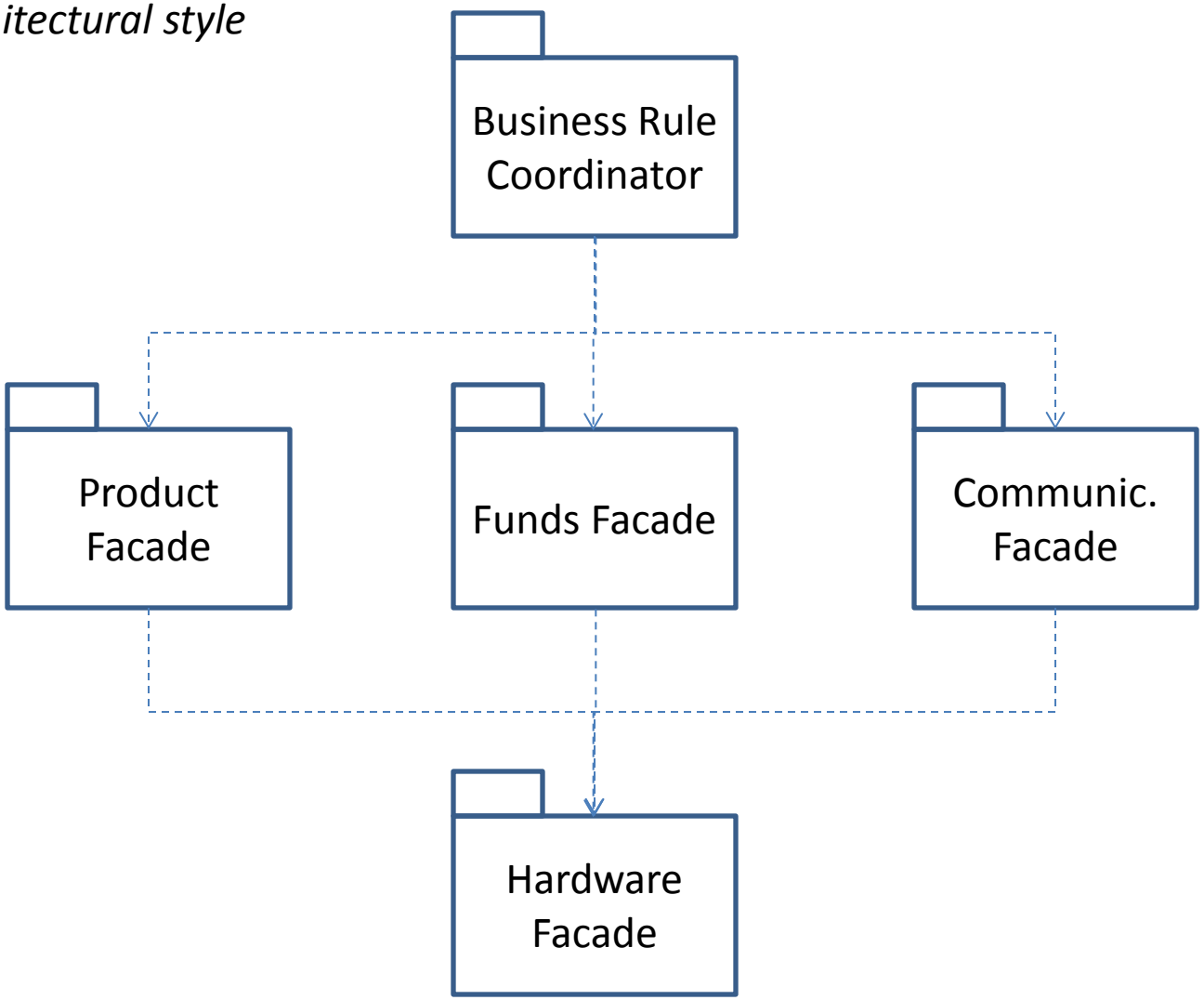
Coupling and cohesion

- The parts of your system have to work together, but too much detailed knowledge causes brittleness
- Lack of cohesion makes the ideas disappear, increasing complexity with all its problems

Software architecture

- Components and connectors
  - The big entities, where they live
  - The interfaces provided and used by the big entities
  - The protocols used to communicate between them
- A real software architecture is *concrete*
- Its abstract shape may conform to one or more *architectural styles*

A layered architectural style



Refactoring

- When the design is not right (or no longer right), you can change it to make it better
- When this does not alter the functionality, we call it *refactoring*
- Internal improvement
- Tool support eliminates much tedium and danger, but cannot be perfect
- Automated testing necessary afterwards

Evolvability

- Software changes for four reasons:
  - to improve internal design
  - to fix bugs
  - to enhance functionality
  - to move to a new environment
- The existing design can affect the ease with which these changes can be achieved
  - More changes and/or more severe changes needed => less evolvable

Evolvability

- Consider the entities that can change:
  - classes, methods, fields, etc.
- Consider how they can change:
  - add, delete, modify
- Consider any constraints needed by the design:
  - e.g., overriding method expects that superclass provides the same method

Evolvability

- Ownership boundaries
  - Would you be able to see all the code affected by a potential change?
  - Would you be able to modify all of it?

FUTURE DIRECTIONS

What have we not covered?

- Requirements
- Testing
- Advanced designs
- Evolution and reuse
- Databases and data mining
- User interfaces
- Software tool construction

- Program comprehension
- Security
- Performance
- Process improvement

- Evaluation
- Pattern discovery
- Recommendation systems
- Inference
- API and language usability

FINAL EXAM