

1 -Overview

Definitions of SW Engineering

- Software Engineering [IEEE-93]:
 - The application of a **systematic**, disciplined, **quantifiable** approach to the **development, operation, and maintenance** of software; that is, the application of engineering to software.
 - Highlights the difference between programming and software engineering
- Canadian Standards Association:
 - “The systematic activities involved in the design, implementation and testing of software to optimize its production and support”

Examples of SE Failures

- *Soyuz spacecraft's descent from the ISS on May 3rd 2003*
 - Halfway back to Earth, for no apparent reason, the computer had suddenly begun searching for the ISS as if to dock with it.
- *Ariane 5 Flight 501:*
 - The space rocket was destroyed. Cause: poor specifications, usage testing, and exception handling.
- *Therac-25:*
 - Radiation therapy and X-ray machine killed several patients. Cause: unanticipated, non-standard user inputs.
- *NASA mission to Mars (Mars Climate Orbiter Spacecraft, 1999):*
 - Incorrect conversion from imperial→metric leads to loss of Mars satellite
- *Louvois project, French ministry of defense, 2001-2013:*
 - 10 years of development on Initial project (adapted COTS) abandoned unfinished in 2003
 - next version developed in-house 2004-2006, modified 2006, outsourced 2008, deployed 2011, total cost ~80M euros, to be abandoned (announced in late 2013).
 - Errors cost ~ 465M euros for the year 2012.

What are the problems?

- Highly Complex Software, distributed teams
- Increased quality demands on software products
- High cost and time pressure
- Shorter time to market
- Coordination problems within the projects
- Limited resources (e.g., qualified personnel)

Software Quality

- External Characteristics (of interest to stakeholders)

- Usability
- Efficiency
- Reliability
- Maintainability
- Reusability



Engineering is tradeoffs

Short term



Long term

- Internal Characteristics (impact maintainability and reliability)

- Comments
- Code Complexity: Nesting depth, branches, complex programming
- Modularity

Software Engineering Principles

- There are a number of general principles underlying and driving all software engineering techniques
 - They aim at dealing with the inherent complexity of software and help achieve quality goals, e.g., reliability, evolvability
 - I will refer to these principles throughout the course.
-
- Rigor and formality
 - Separation of concerns
 - Modularity
 - Abstraction
 - Anticipation of change
 - Generality
 - Incrementality

Survey of Some Process Models

- Waterfall Model
 - Phased-Release Model
 - Spiral Model
 - Unified Process
 - Agile Process
-
- All include the activities of the original waterfall model, but better process models are:
 - Iterative: repeat the steps and improve over previous attempts
 - Incremental: start with a small system and add more functionality bit by bit.

2 - Requirements Elicitation

What is a Requirement ?

- [Dutoit] A requirement is a feature that the system must have or a constraint that it must satisfy to be accepted by the customer.
- [Lethbridge] A requirement is a **statement** about what the proposed system will **do** that **all stakeholders** agree must be made true in order for the customer's **problem** to be adequately solved.

Statement: brief, concise, fact-based.

Do: what, not how

All stakeholders agree: contract

Problem: the customer's need

Functional vs. Non-Functional

- Functional requirement:
 - interaction between a system and its environment
 - defines a problem
- Non-Functional requirement:
 - **Constraint** on the problem
 - e.g., memory, platform, real-time constraints

Summary

- What?
 - Functional Requirements: what the system should do
 - Non-functional requirements: constraints
- Why?
 - Primarily for communication and agreement
 - => should be verifiable
- Who?
 - Customer
 - Managers
 - Developers
 - Test Engineers
 - Maintenance Engineers

1. Identify Actors

- Can be human or external system, device
- Define system boundaries
- Find all stakeholders
- May correspond to roles in an organization
- Need to differentiate roles only when they access different functionality
- Classes of functionality

2. Identify scenarios

Bridging the gap between the user and the developer

- *Scenarios*: Example of the use of the system in terms of a series of interactions between an actor and the system
- *Use cases*: Abstraction that describes a class of scenarios (e.g., end user functionalities)

3. Identify Use Cases

- A scenario is an instance of a use case
- A use case specifies all possible scenarios for a given piece of functionality
 - Find hints for a use case in the scenario descriptions, e.g., “Report Emergency “ in the first paragraph of the scenario is a candidate for a use case
 - Report Emergency accounts for all possible scenarios, i.e., Warehouse on fire, FenderBender, Earthquake etc.
- A use case is always initiated by an actor but may interact with other actors as well
- A use case is a complete flow of events through the system

Specifying a Use Case (template)

Use case are described by following template descriptions, which typically include the following sections (many different templates exist):

- Use Case Name
- Brief Description
- Precondition
- Primary Actor
- Secondary Actors
- Dependencies to other use cases
- Basic Flow
- Alternative Flows: Specific, Bounded, Global Alternative Flows
- Special requirements
- Technology and data variations
- Open issues

Advantages of using Use-Cases

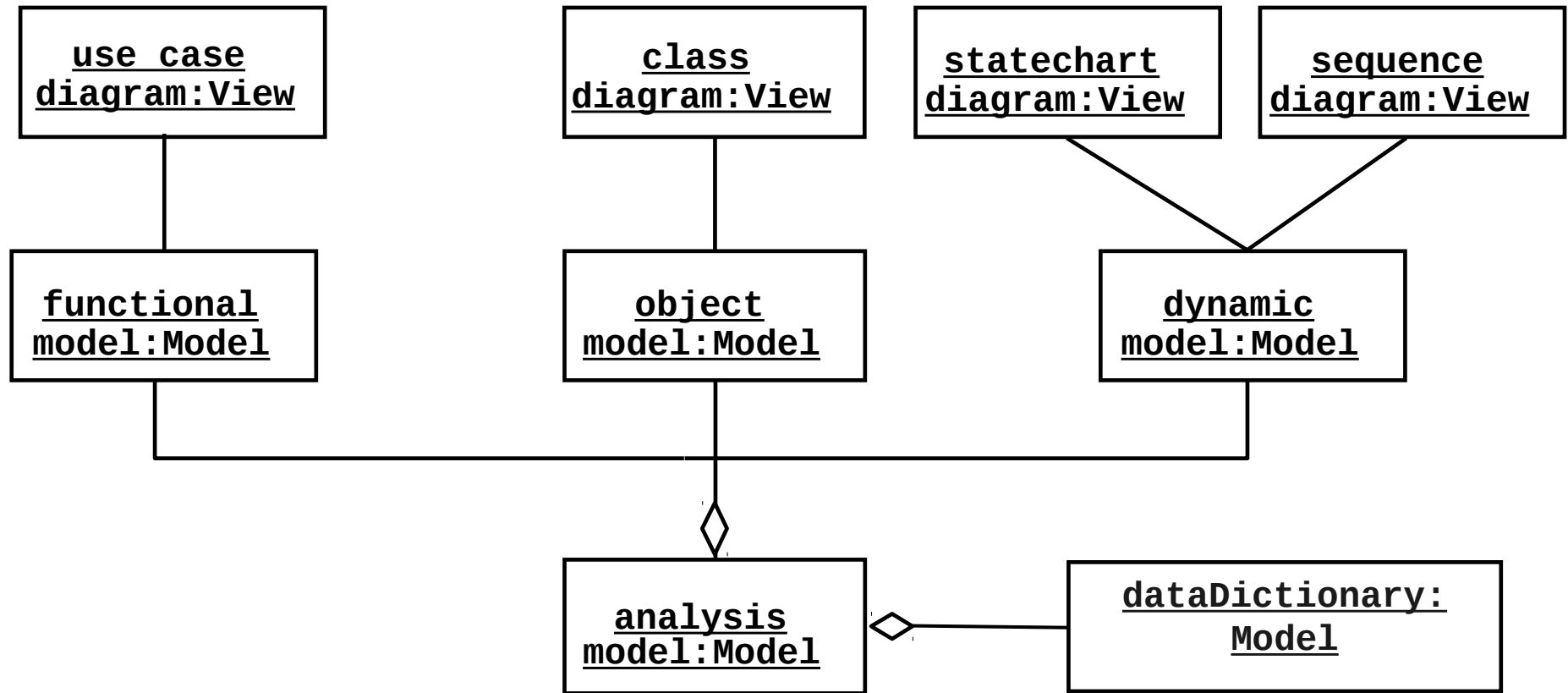
- Helps define the scope of the system (what it does and does not do)
- Can be used as part of the development plan
 - # use cases is an indicator of project size
 - progress can be measured by % use cases completed
- Form the basis of definition of test cases
- Can be used to structure user manuals

Disadvantages

- The use cases themselves must be validated with the customer
- Some aspects of functionality may not be covered by a use case analysis (only what's triggered by an actor) [e.g. Mars Rover]

3 – Requirements Analysis

Analysis Model (Bruegge and Dutoit, 2000)



Object Modeling

- Identify the **classes** of the **Problem Domain**
- Identify **relationships** between classes:
 - Association
 - Composition/Aggregation
 - Generalization
- [secondary]
 - Find the attributes
 - Find the methods
- Iteration is important
- Static model will be refined when devising dynamic models

Finding Classes

- Objects involved in the Use Cases (or problem description)
- Text analysis: nouns
- The objects you need to store/manipulate in the system to implement a requirement
- Actors? Only if you need to store information about them (e.g. customer)
- Start with too many, then narrow down

Identify Associations

- Identify classes that need to know about another class instances,
 - e.g., they have, create, access, destroy instances of that class
 - *EmergencyReport* can be created by *FieldOfficer*
 - *A Hotel* has *Rooms*
- Beware: “A has B” can be modeled as:
 - An attribute (B is simple type) **OR**
 - An association (B is a class):
- Association properties: Name, Roles, Multiplicities
 - May also be aggregation/composition
- Generalization: organize concepts, remove redundancy

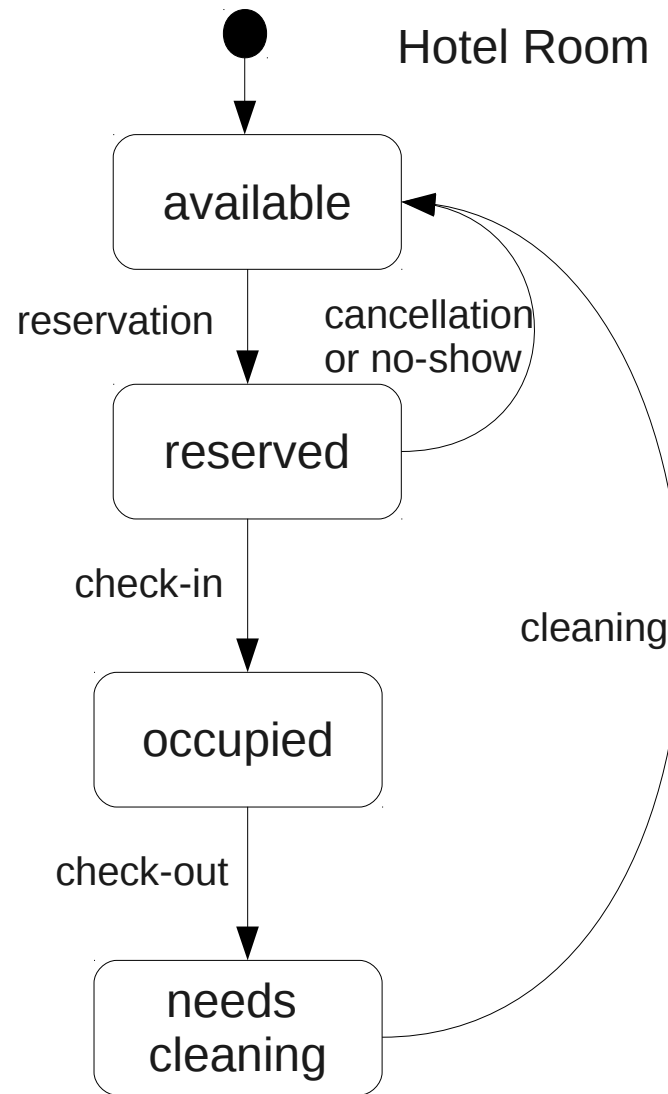
Identify Associations

- An iterative process :
 - Initial identification
 - Then, refinement (analyzing and verifying the associations)
- For every association, ask yourself : Is it relevant to the application ?
 - Is it needed to implement some requirement ?
 - If not, you can eliminate it from the model

Modeling Object Behavior

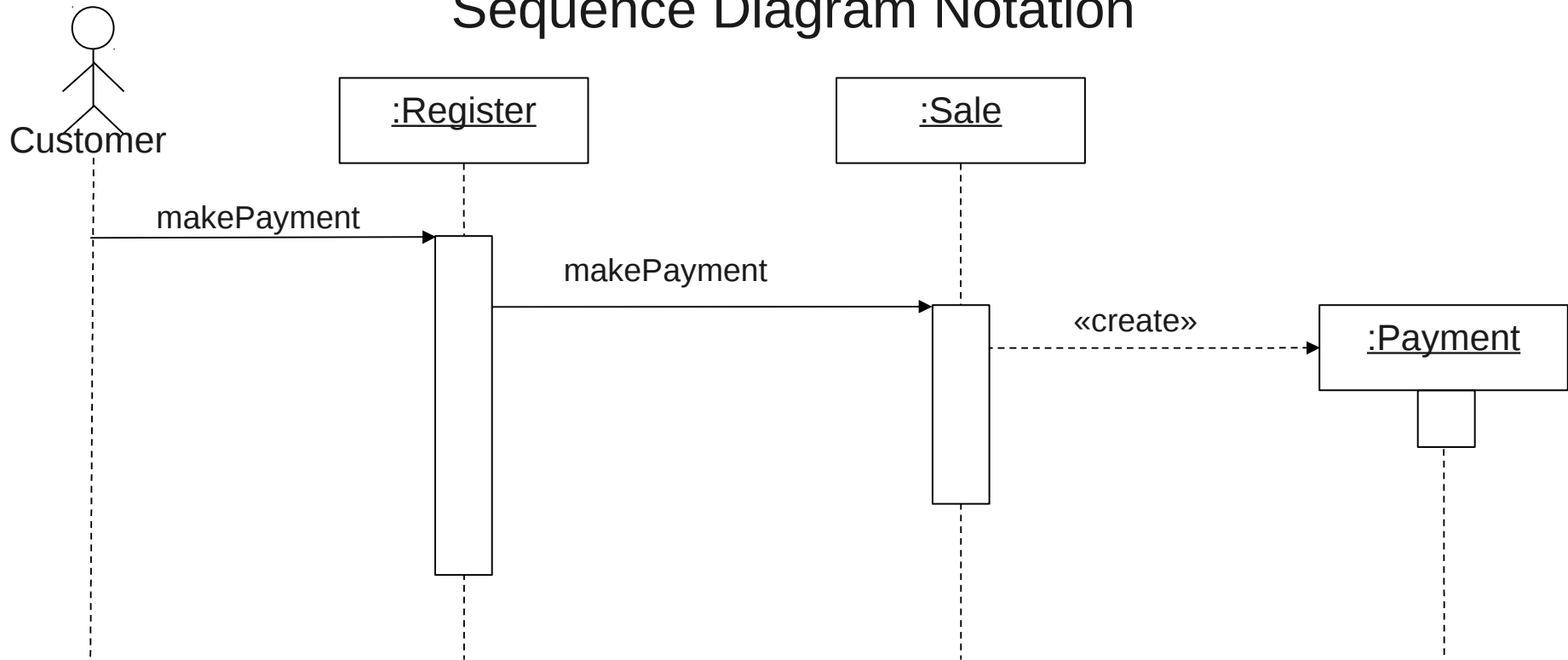
- Sequence diagrams represent complex behavior of several objects interacting
 - Shows how the behaviour of a use case is distributed among participating objects.
- Statechart diagrams capture behavior from the perspective of single object
 - Focus only on objects with non-trivial behavior (multi-modal, state-dependent)
- Help identify missing Use Cases
- Help identify missing objects and/or operations
- Build more formal description of the object behavior

State Diagram



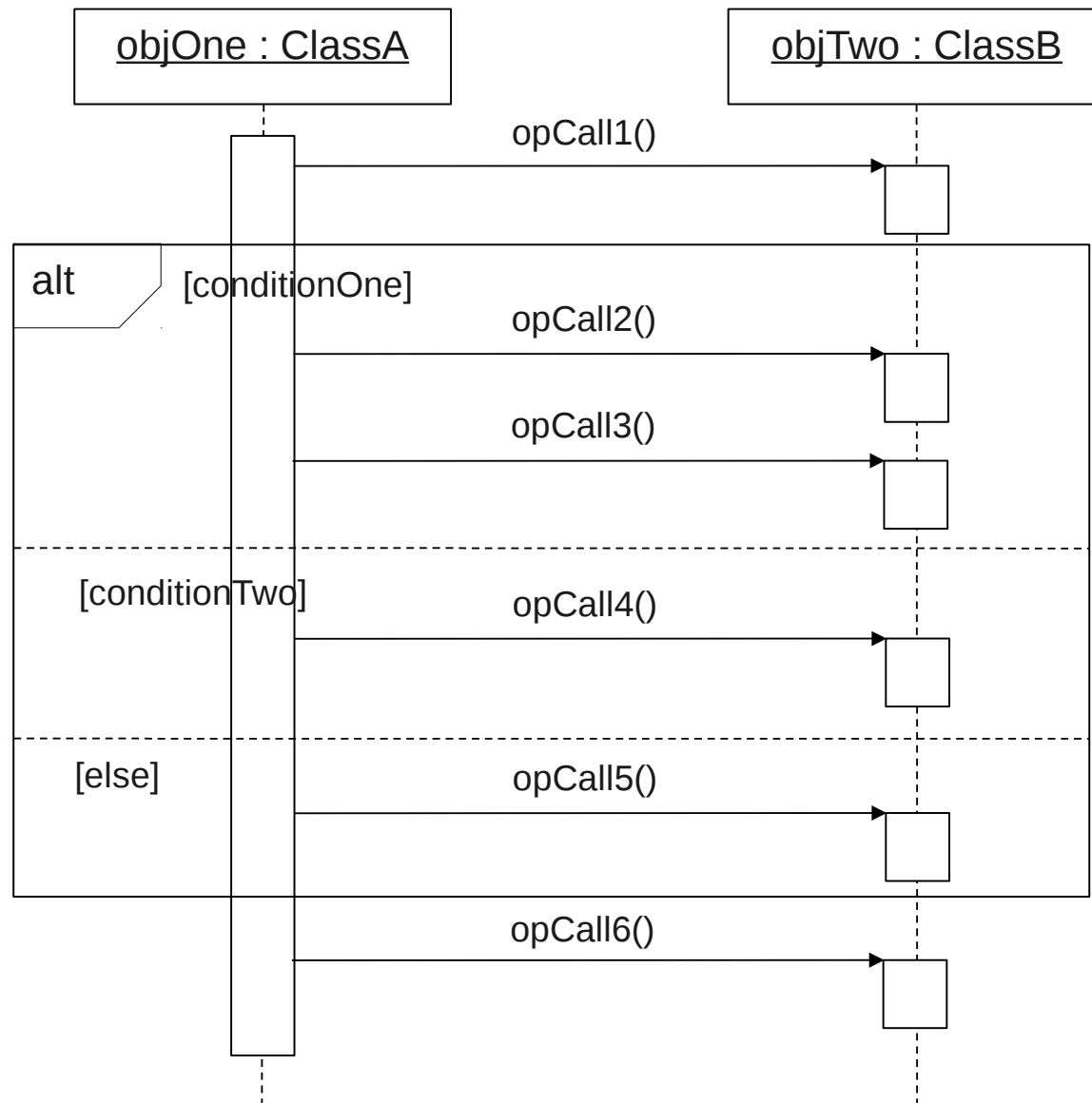
Entity Class

Sequence Diagram Notation



- The sequence diagram is read as follows:
 - The message *makePayment* is sent to an instance of a *Register*. The sender is an actor.
 - The *Register* instance sends the *makePayment* message to a *Sale* instance.
 - The *Sale* instance creates an instance of a *Payment*.

Frames—Control Flow



- **alt**: if/else
- **opt**: if/elseif/else
- **loop**: executes as long as condition is true
- **par**: do in parallel
- **break**: if[cond] then break
- **ref**: to another sequence diagram

Entity/Boundary/Control Classes

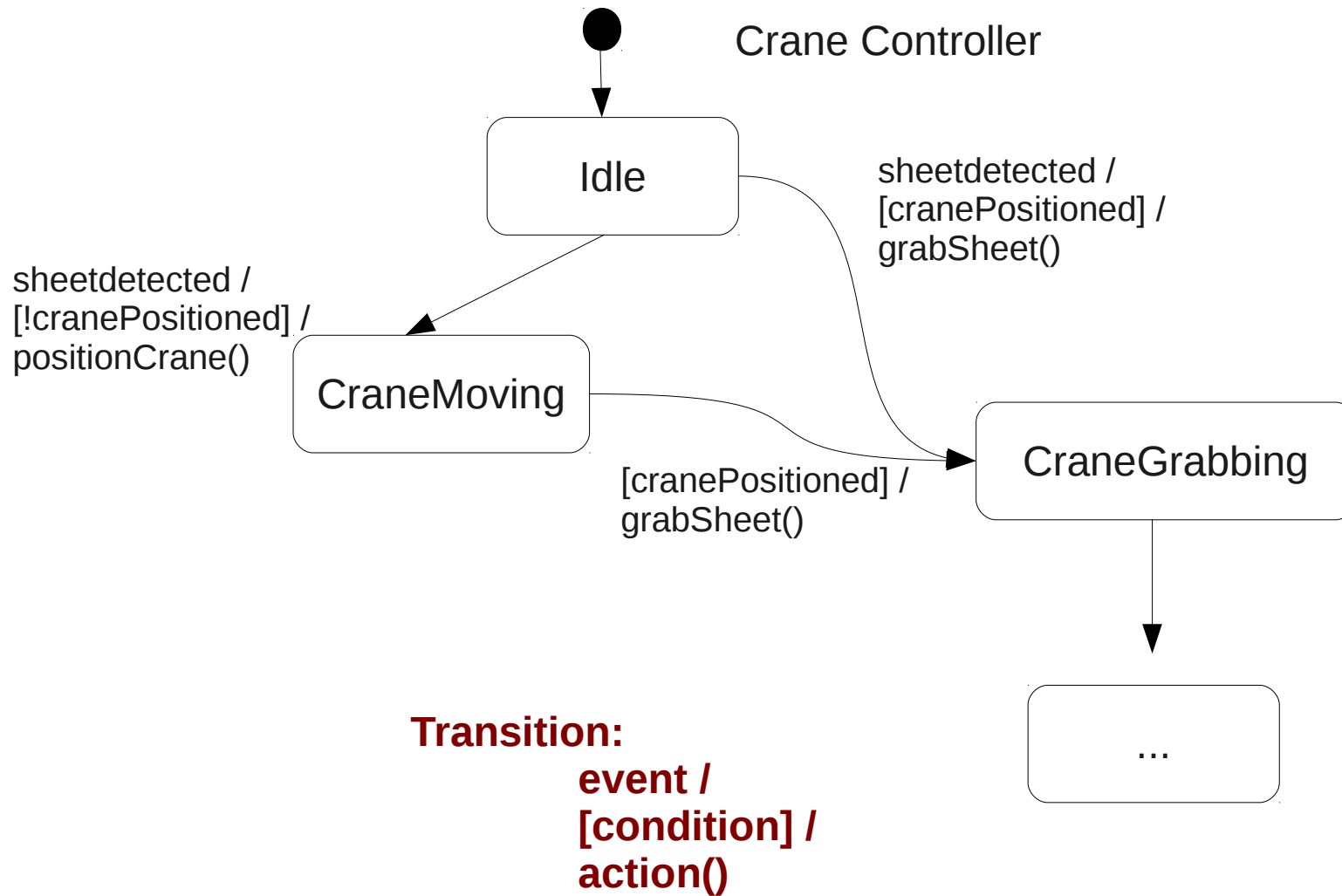
- The OO methodology describes processes using objects
- We can distinguish two important kinds of classes
 - **Entity** Objects: represent the problem domain
 - **Boundary** Objects: input/output of the system, messages
- We introduce an extra kind of objects to handle the functionality of a use case: **Control** objects
- The flow of events in a Use case can then be described using entity, boundary, and control objects:
 - The primary actor inputs data through boundary objects
 - The data is collected by control objects that pass it to entity objects, or other control objects
 - Control objects pass data to boundary objects that pass it to other actors.

Message numbering is UML 1.x
(ignore)

Example - ATM

State Diagram

Control Class



4 – System Design

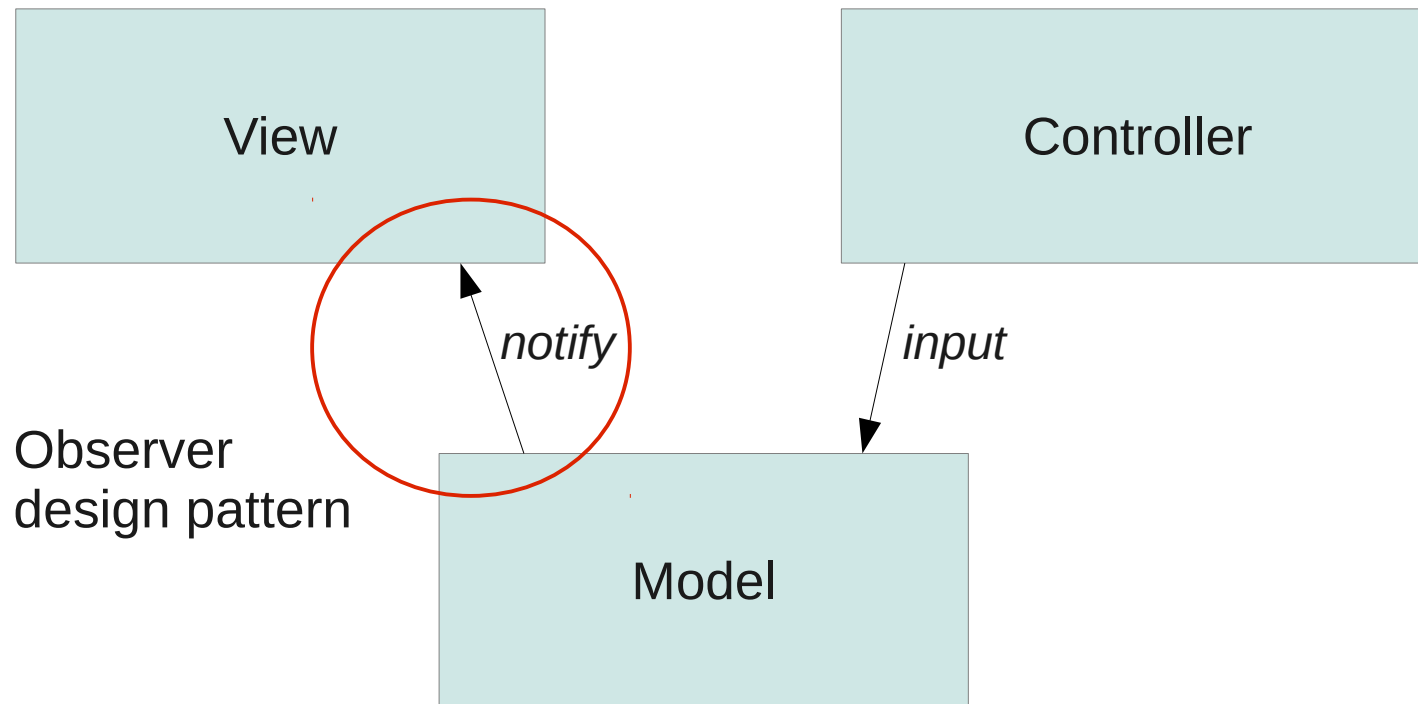
System Design

- From requirements to solution
- Software architecture: subsystems
 - Subsystem responsibilities,
 - Dependencies, interface contracts
 - Mapping to hardware
 - Code reuse
- Policies for access control, data storage
- Control Flow
- Boundary use cases: startup, shutdown, exception handling

System Design

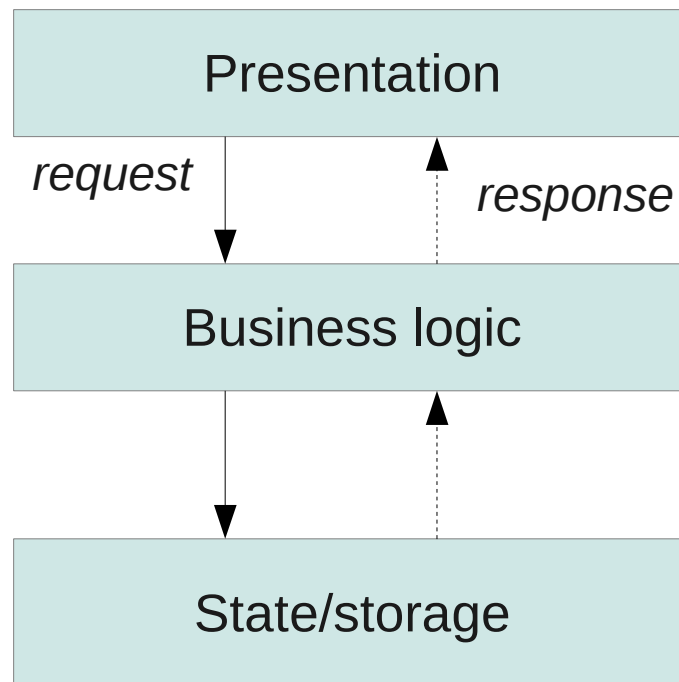
- Architectural Styles:
 - Client /Server
 - Layered
 - Pipe and filter
- Architectural Patterns
 - Model/View/Controller
 - 3-tier (presentation – business logic – storage)
 - Sense-compute-control

Model/View/Controller (MVC)



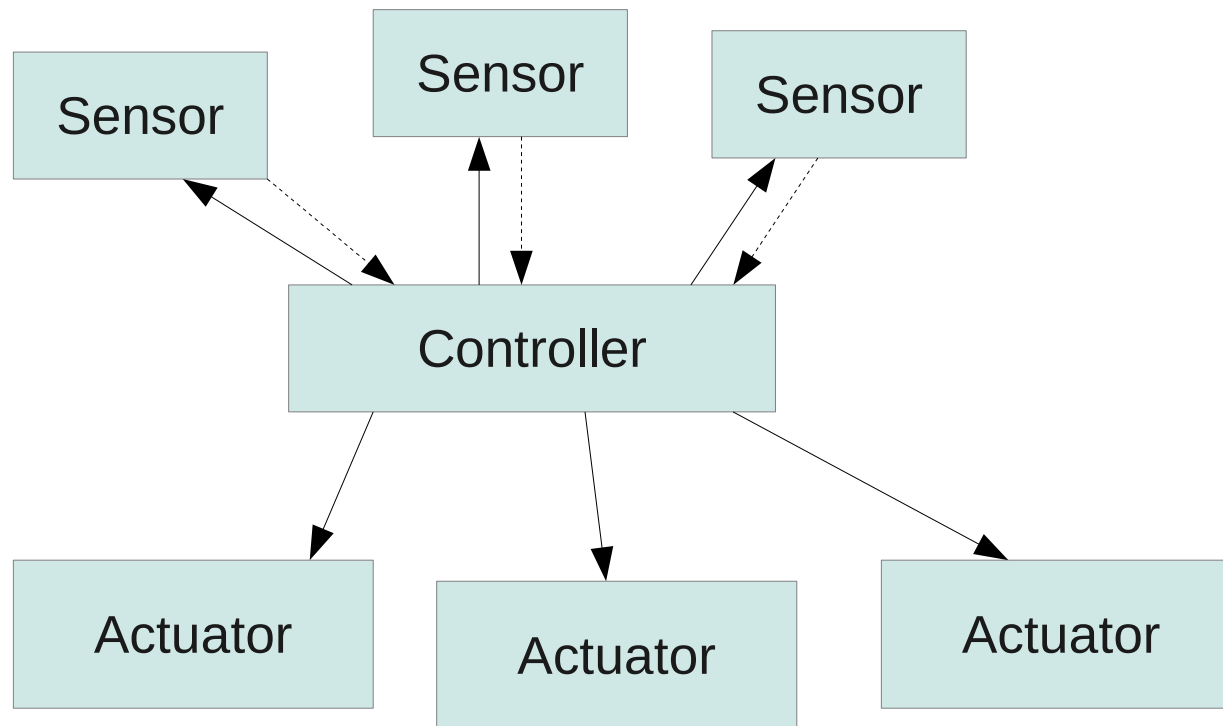
3-Tier

- Layered (3 layers), client/server

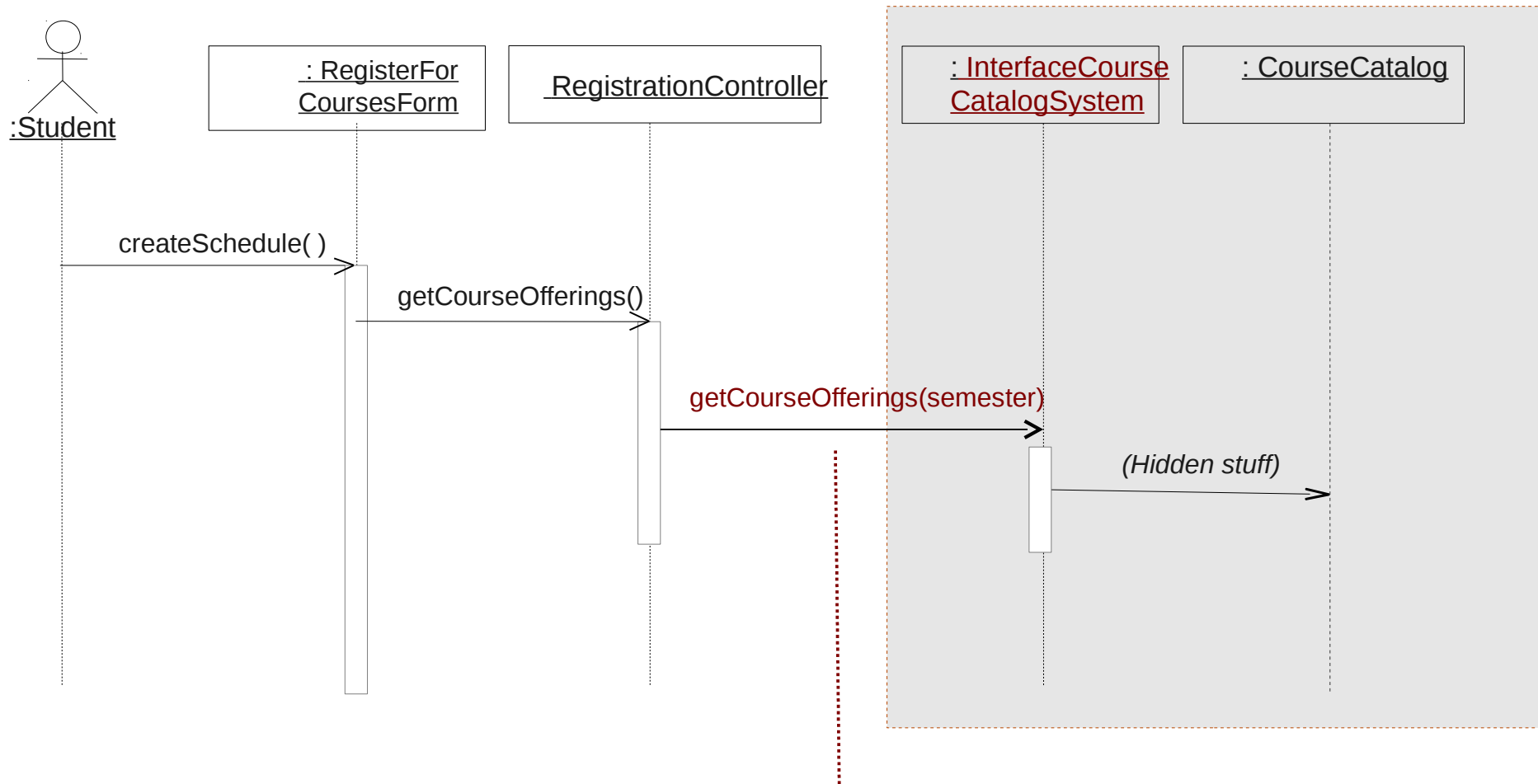


Sense – compute - control

- For control systems, agents, robots, etc.
 - Sensors: get data from environment
 - Actuators: act on environment



Subsystem API

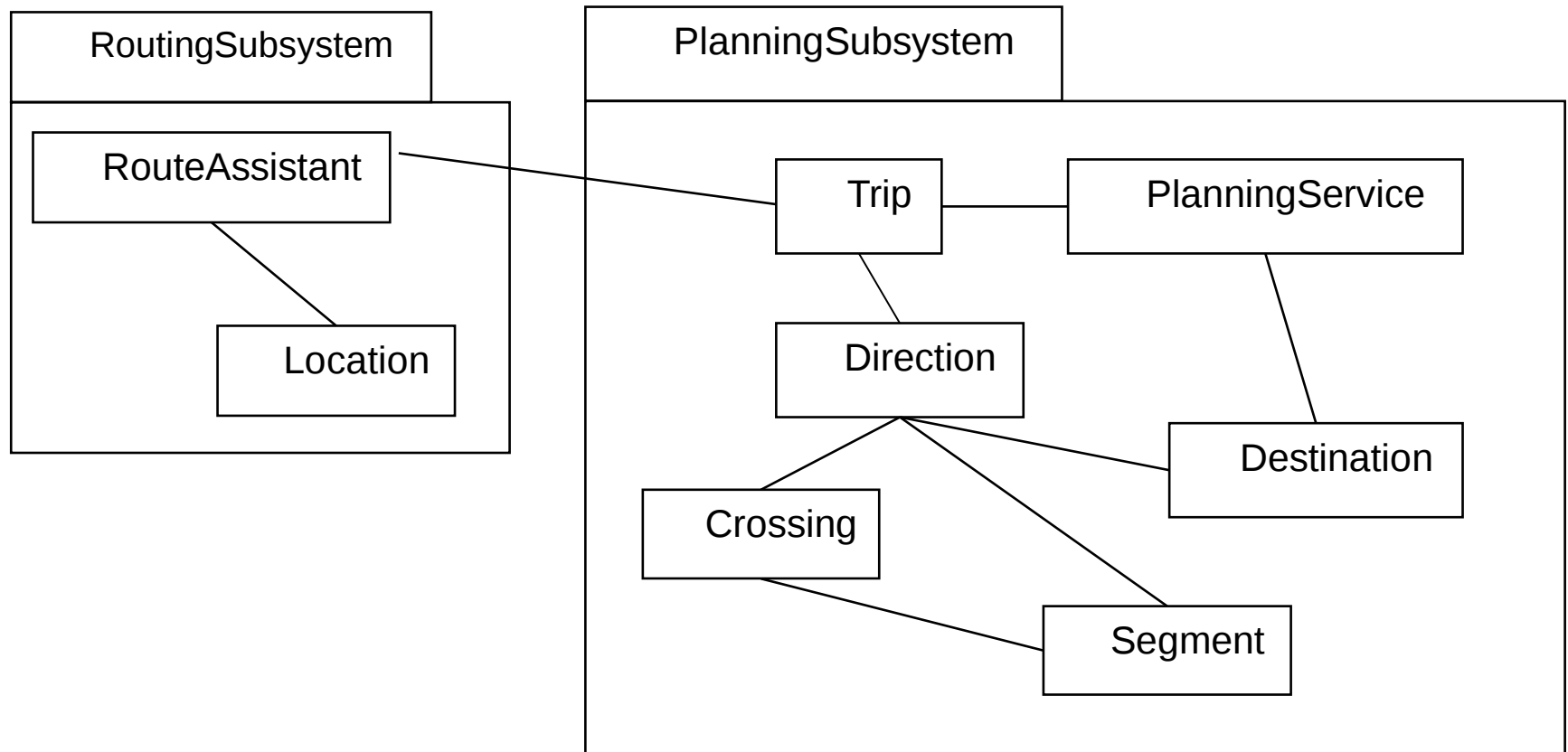


API operation

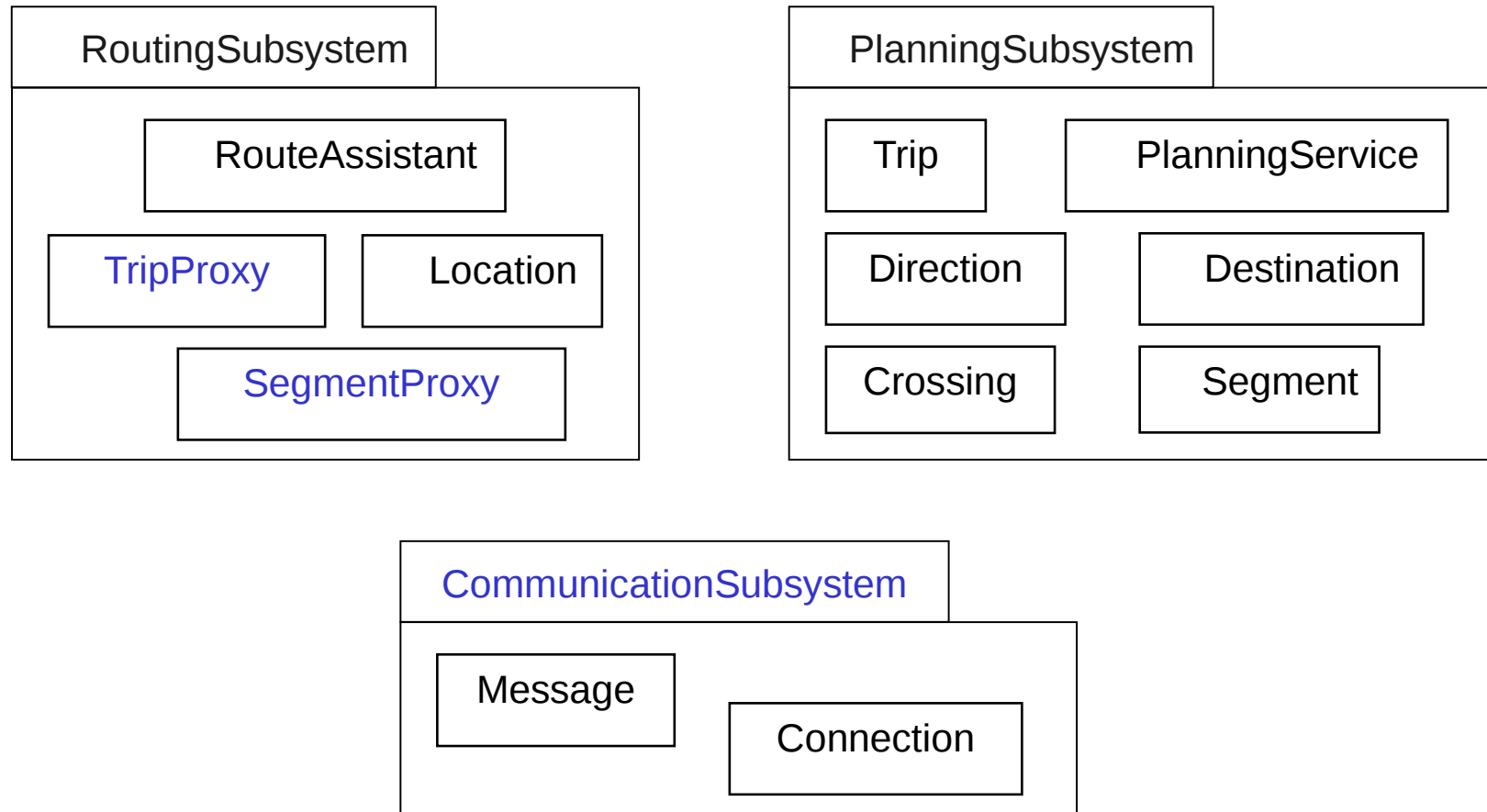
Mapping to hardware

- Large applications usually distributed
- Potential Efficiency....:
 - ...Gains (Parallel processing)
 - ...Losses (messaging between subsystems)
- Avoid:
 - Processing bottlenecks
 - Excessive messaging
- Use:
 - Load balancing
 - Prioritize tasks

MyTrip Subsystems



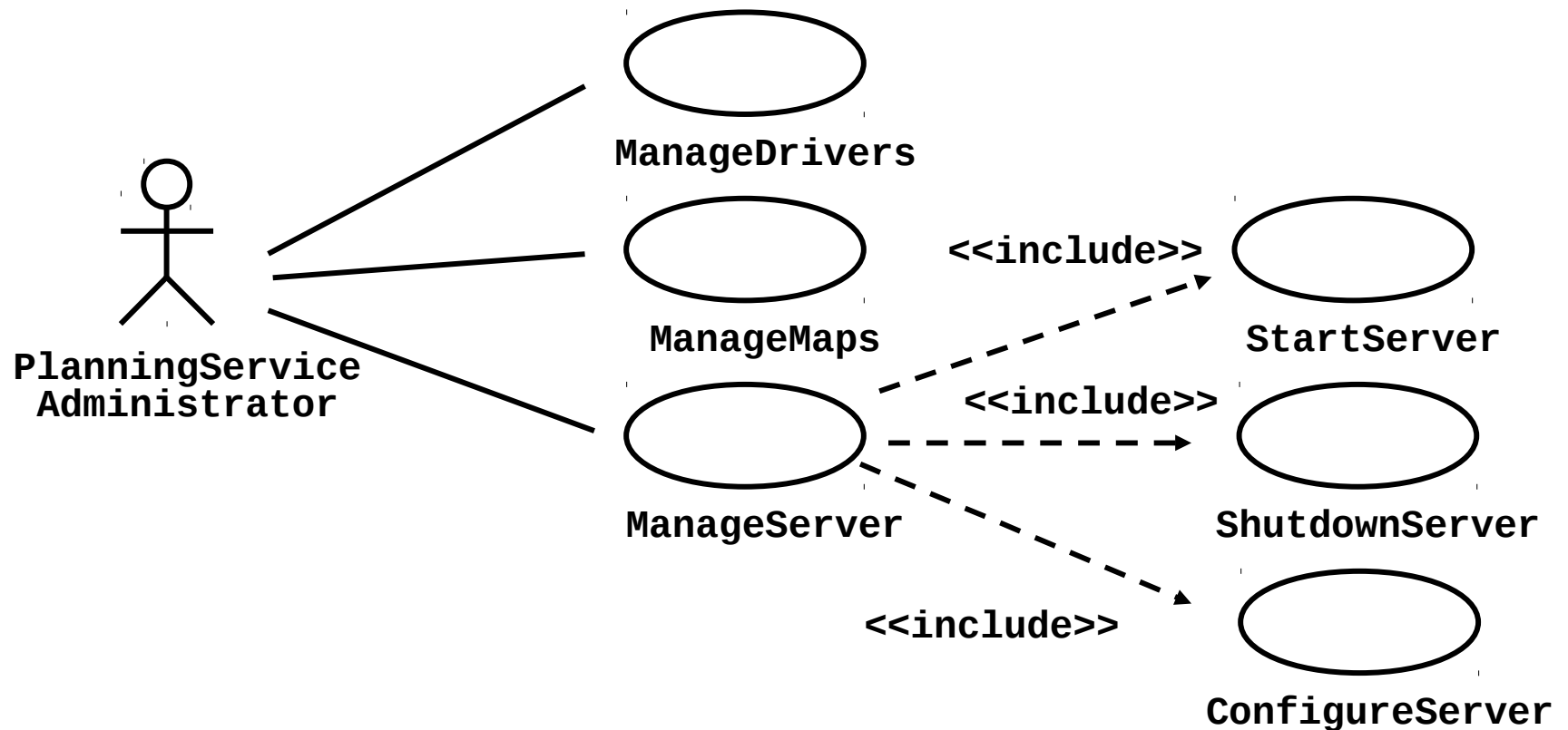
New Classes and Subsystems



Boundary Conditions

- Most system design effort is concerned with steady-state behavior.
- However, system design phase must also address boundaries of system
 - **Initialization**
 - Describes how the system is brought from a non initialized state to steady-state ("startup use cases").
 - **Termination**
 - Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").
 - **Failure (Exception Handling)**
 - An exception is an unexpected event or error that occurs during the execution of the system
 - Sources: User error, external problems (network or hardware failure – power supply), software bug
 - Good system design foresees fatal failures ("failure use cases").
 - Exception handling: catch exceptions, treat them to minimize damage

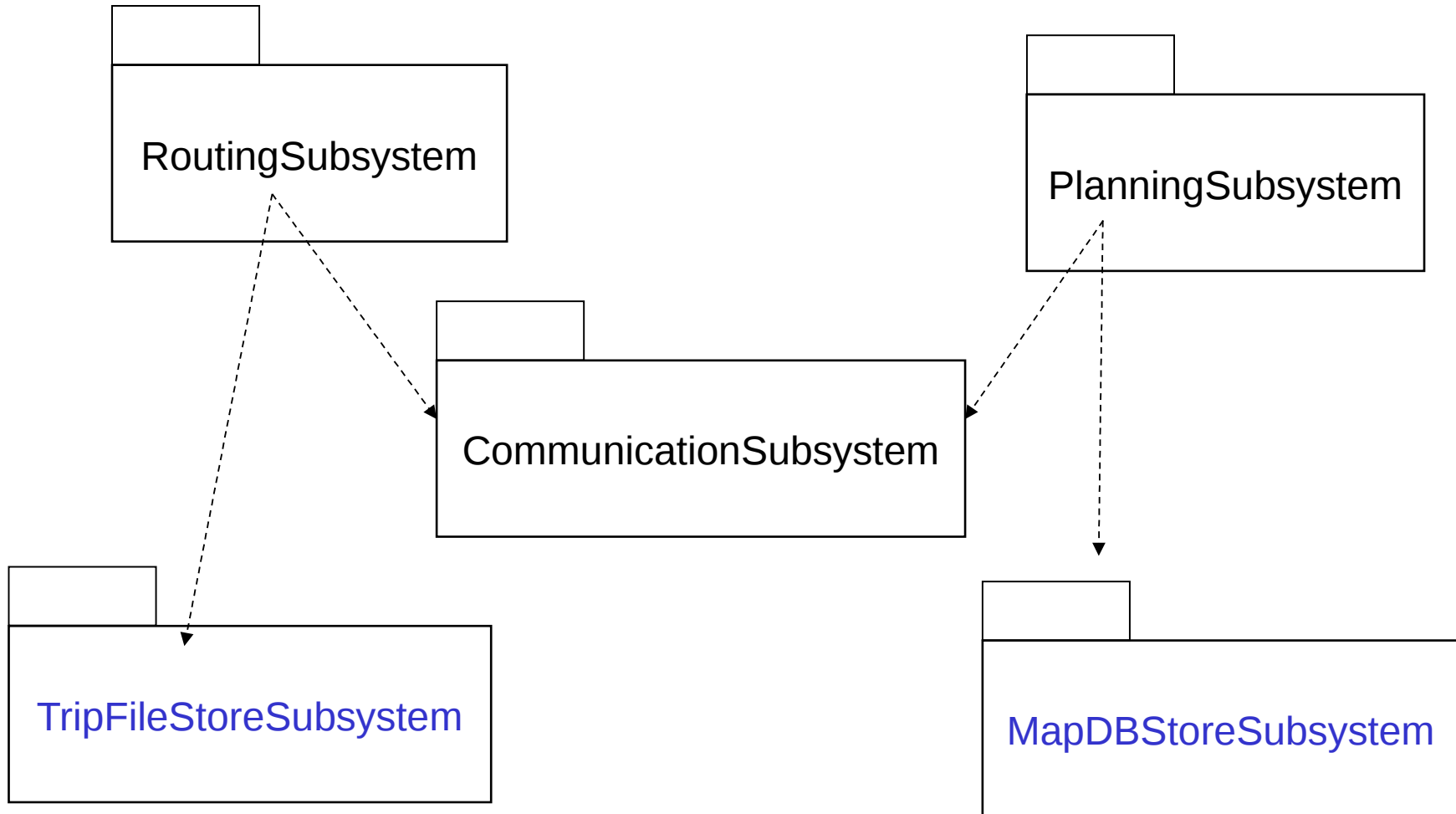
MyTrip Administration



Access Control Policy: Example

- Bank information system
- A Teller may debit or credit an account (operations on class Account) up to a predefined amount
- If this amount is exceeded, a Manager needs to approve the transaction
- Managers and Tellers can only access accounts in their own branch
- Analysts can access information across branches but cannot post transactions on individual accounts

MyTrip Data Storage



Databases

- Relational Databases
 - Properties (design goals)
 - Relational Data Modeling
 - Queries: SQL
 - ACID properties
 - Data Warehousing
- NoSQL databases
 - Design Goals
 - Different models

The Relational Model

- Data as tables

relational schema

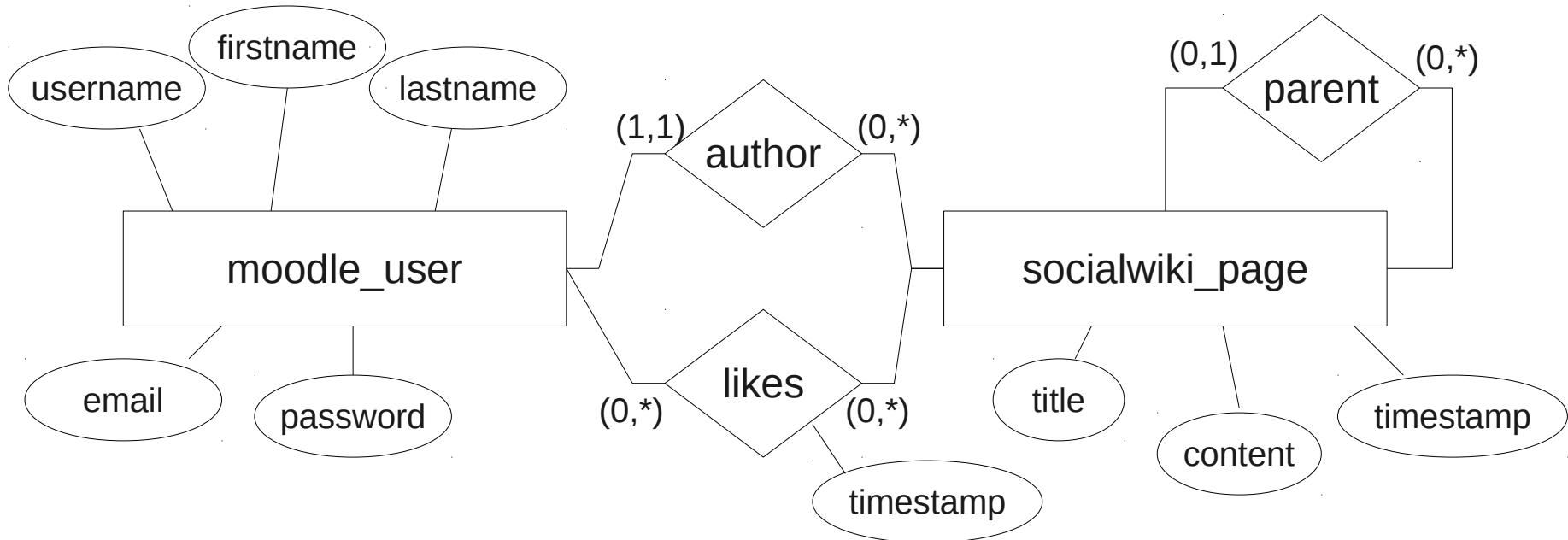
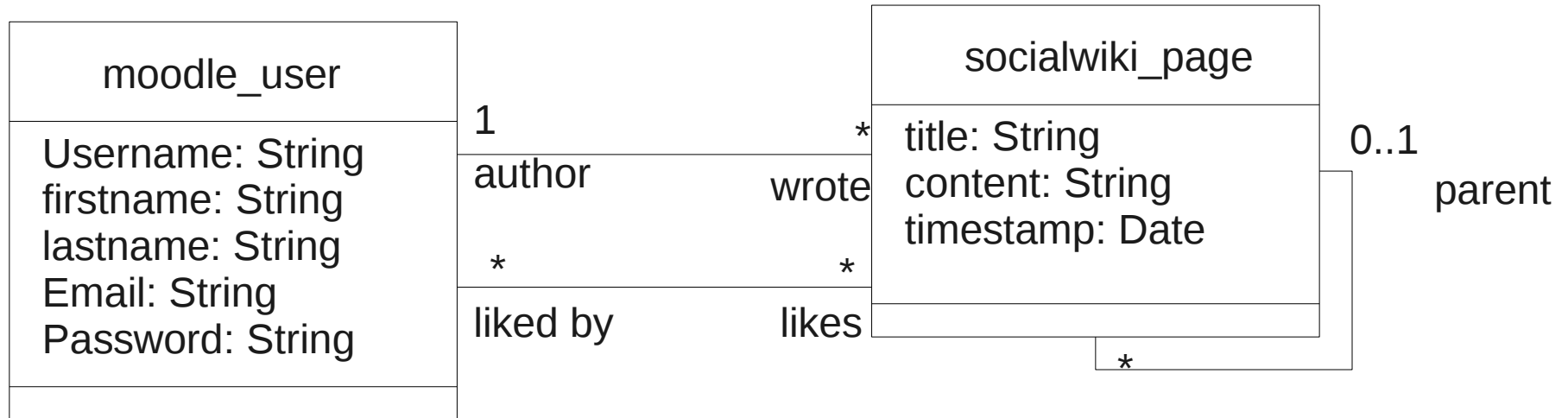
table name

fixed column headers

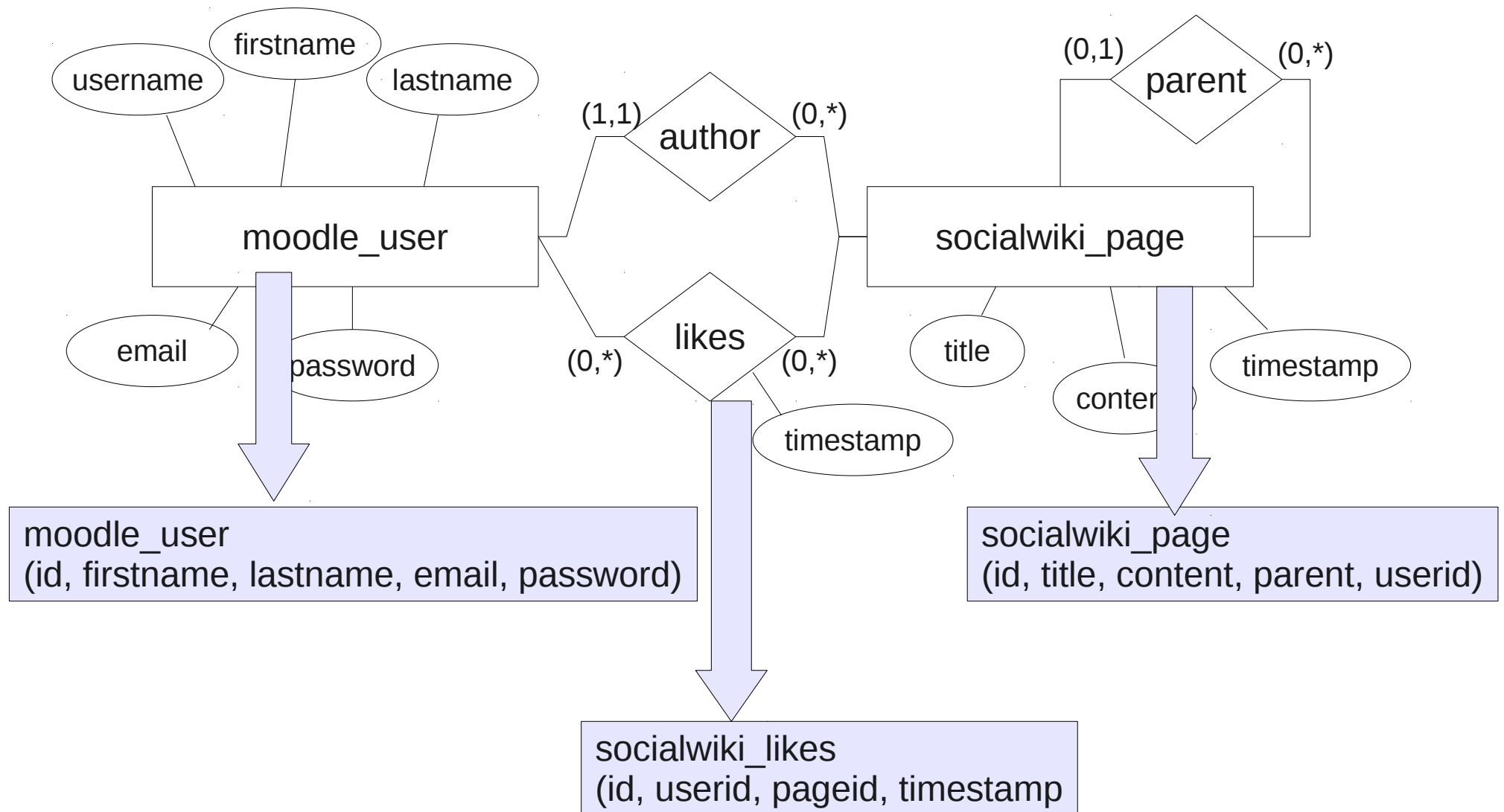
moodle_users					
id	username	firstname	lastname	email	password
1	alicesmith	Alice	Smith	a@bbc.co	himum!99
2	bobjones	Robert	Jones	bob@a.ca	hidad%90
3	charlie	Charles	Chan	ccd98@a.b	mycat01#
4	evajoli	Eva	Joly	ej@cc.ca	mydog!99

rows of data

The Entity-Relation Model



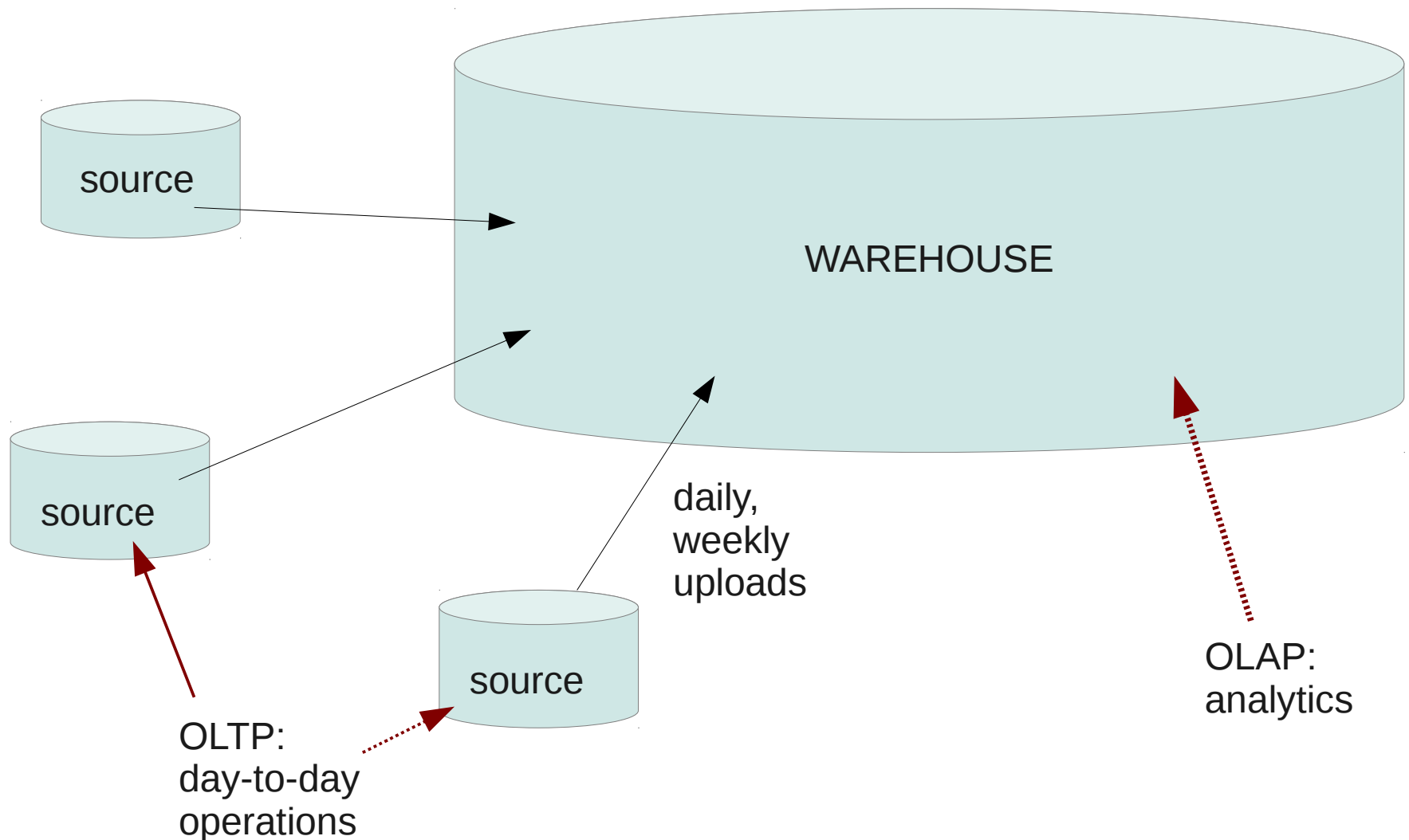
ER model to relational schema



ACID

- “transactional processing” = ACID properties
- Atomicity (of transactions)
- Consistency (of DB states)
- Isolation (concurrency management)
- Durability (data not lost over crashes...)

Data Warehousing



NOSQL databases

- Amazon, Google, Facebook, Twitter...
- “Internet age” performance requirements:
 - PB+ (1 petabyte = 10^{15} bytes = 1M GB) of data
 - Very high frequencies of operations, on logically centralized data

=> Limits of traditional (relational) DBs.
- Elasticity
- => Often “natively” distributed

NOSQL database Models

- Key-Value stores
- BigTable
- Document
- Graph

5 – Object Design

Code reuse

- Cost Trade-off
- **Pros:**
 - Less re-inventing the wheel
 - Reliability
 - Direct financial cost may be low (or zero)
- **Cons:**
 - Cost of adapting
 - COTS: How long will product be supported?
 - Open-source: licensing issues

Frameworks

- **Infrastructure frameworks** aim to simplify the software development process
 - e.g. Java Swing, OSGi, Ruby on Rails...
- **Middleware frameworks**
 - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- **Enterprise application frameworks**
 - Exist for: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities...

White Box and Black Box Reuse

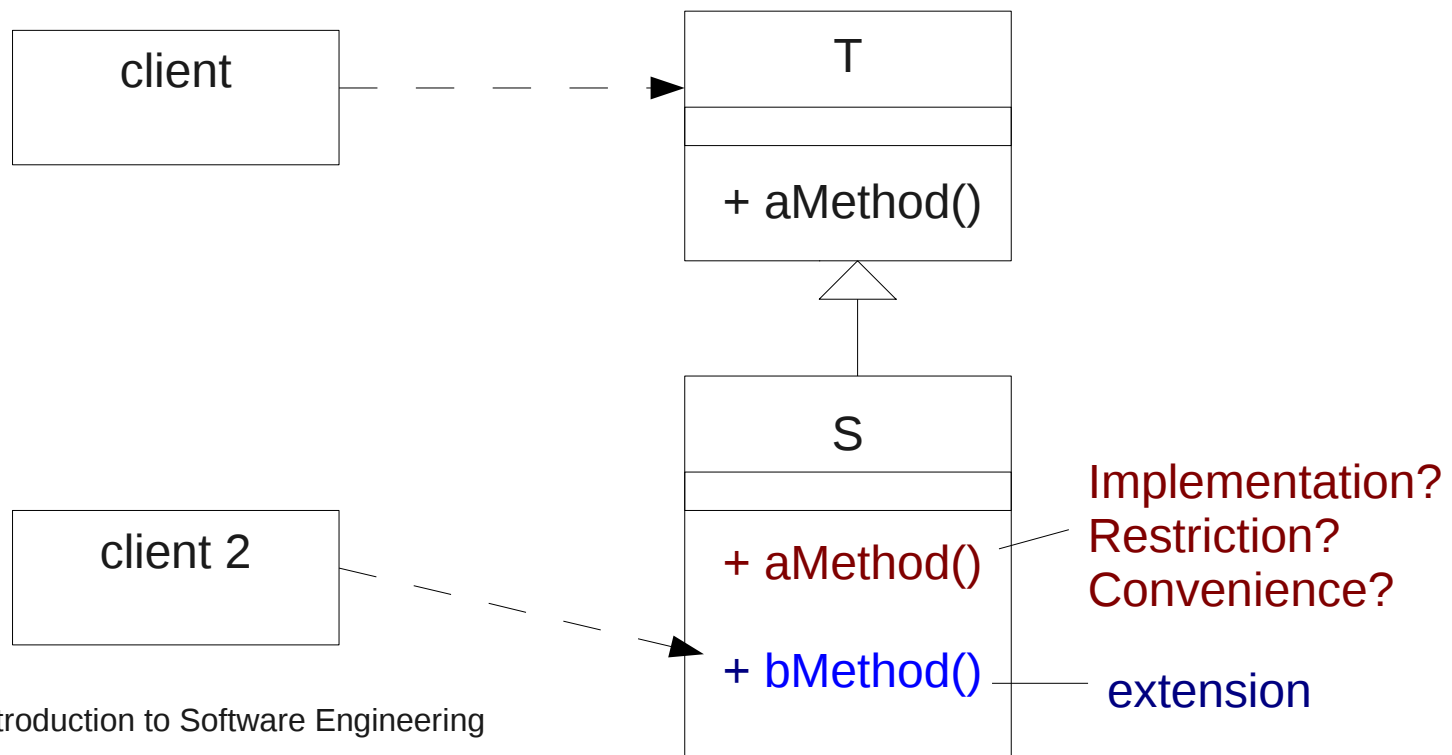
- **White box reuse**
 - Reuse known code
 - Add functionality with inheritance
- **Black box reuse**
 - Access to models and designs is not available, or models do not exist
 - Worst case: Only executables (binary code) are available
 - Better case: A specification of the system interface is available.
 - Add functionality by aggregation

Types of Inheritance

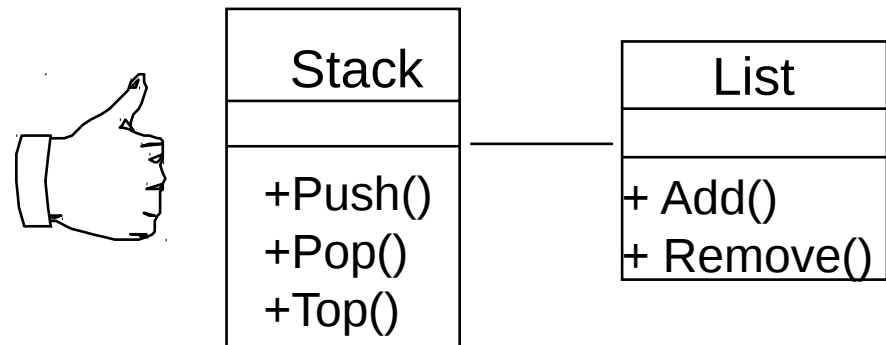
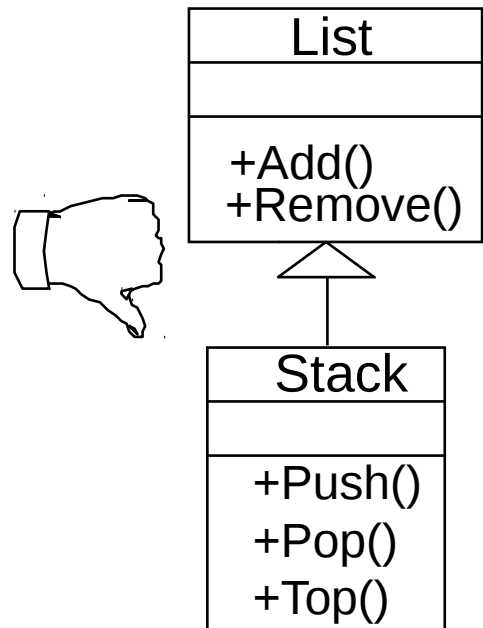
- Interface inheritance
 - Reuse only the interface
- Implementation inheritance
 - Override superclass behavior (sometimes **problematic**)
- Extension inheritance
 - Add some functionality
- **Restriction inheritance**
 - Remove some functionality: should be avoided
- **Convenience inheritance**
 - Reuse some useful property for a different purpose
 - => use delegation instead

Liskov substitution principle

- Class **S** is correctly defined as a **specialization** of class **T** if the following is true: *a client method written in terms of superclass T must be able to use instances of S without knowing whether the instances are of S or T.*
- S is said to be a subtype of T



Inheritance vs. Delegation



Design Patterns

- *Good* OO solutions to common problems:
 - Deal with complexity
 - Anticipate change
 - Use inheritance and/or delegation
- **A few patterns**
 - Adapter
 - Proxy
 - Facade
 - Strategy/Bridge
 - Observer
 - Composite
 - Singleton
 - Factory

6 – Tools

Software Engineering Tools

- Modeling tools
- IDEs
- Version Control
- Build scripts
- Testing tools
- Code analysis
- Bug tracking
- The cloud
- Command line

**AUTOMATION
TRACKING
EFFICIENCY**

7 – Verification and Validation

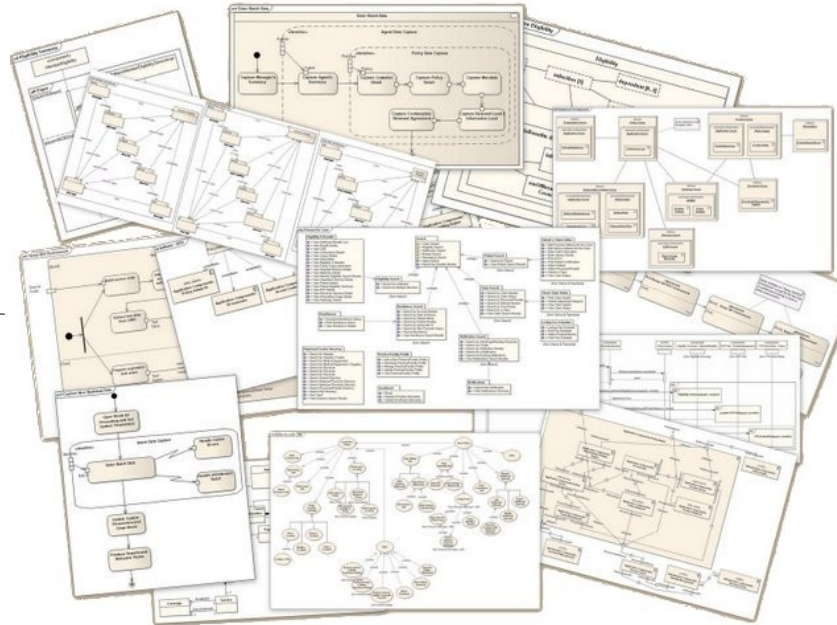
Cost of Testing

- You are going to spend about half of your development budget on testing, whether you want to or not.
 - For safety-critical systems, testing amounts to up to 80% of the cost
- In real world usage, testing is the main post design activity
- Restricting early testing usually increases costs

Verification vs. Validation

Verification

Different UML diagrams should be consistent



code should behave as specified (no bugs)

```
.getFromMap($map$value$1$0, a), a);  
}  
private int $dispatch$value$1$(int i, A a) {  
    switch(i) {  
        case 1: return value((A0)a);  
        case 2: return value((A1)a);  
        case 3: return value((A2)a);  
        default:  
            throw new RuntimeException("No such method");  
    }  
}  
private static final ConcurrentIntMap $map$value$1$0;  
static {  
    ConcurrentIntMap concurrentintmap=new ConcurrentIntMap(6)  
    concurrentintmap.insert(A0.class, 1);  
    concurrentintmap.insert(A1.class, 2);  
    concurrentintmap.insert(A2.class, 3);  
    $map$value$1$0=concurrentintmap;  
}  
}
```

Validation

The models reflect the functionality expected by the customer

the method specifications match the required functionality

Types of bugs

- Non-compliance with requirements
 - Functional/Non-functional
 - (validation)
- Crashes
- Memory Leaks
- Synchronization Problems (deadlocks...)
 - (verification)

V&V Techniques

- **Static Techniques:** i.e., without any execution of the system
 - Inspections
 - Code reviews
 - Static code analysis
 - Mathematical Proof:
 - Given **pre-condition**, **prove** that **post-condition** will be true
 - Model Checking
 - Prove that system cannot reach an incorrect state, based on formal model (e.g. state machine)
- **Dynamic Techniques:** i.e., executing the system (testing)
 - Functional testing
 - Integration testing
 - Performance testing

Basic Testing Definitions

- What's a test?
 - Do something to the system (stimulus)
 - See how the system responds

Test case

- Does it crash?

- It **outputs** something. is the **output** correct?

Incorrect
code

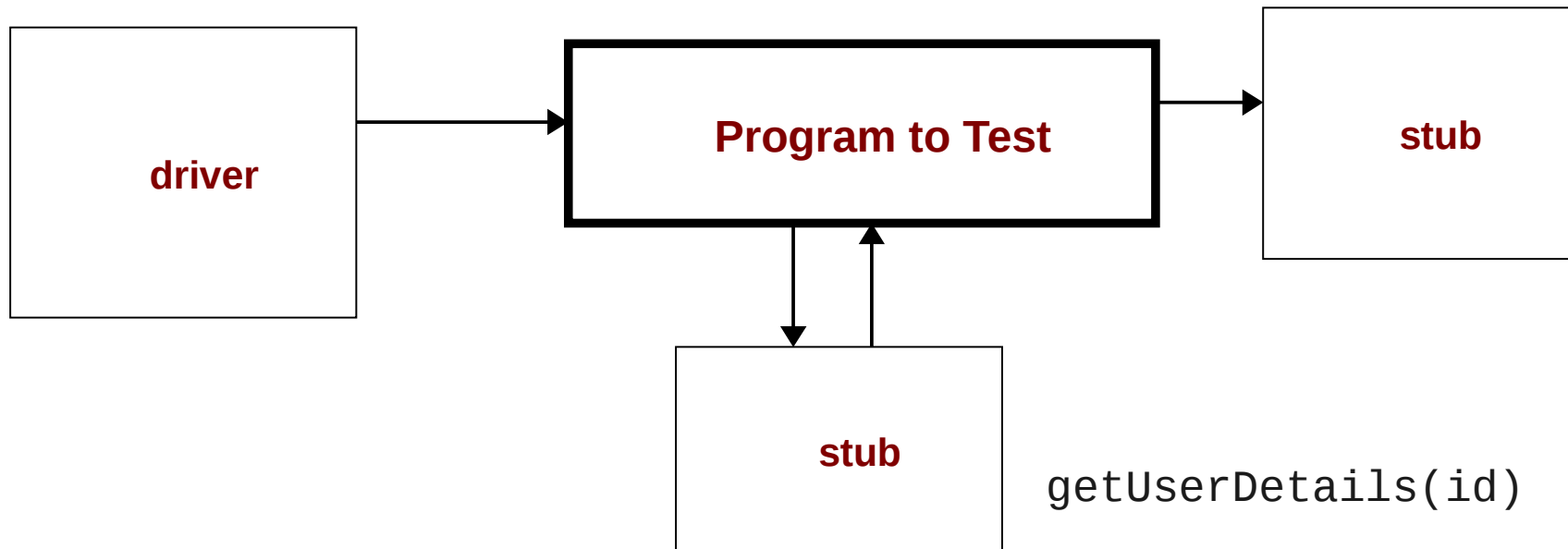
Incorrect
system state

Observed
Incorrect output

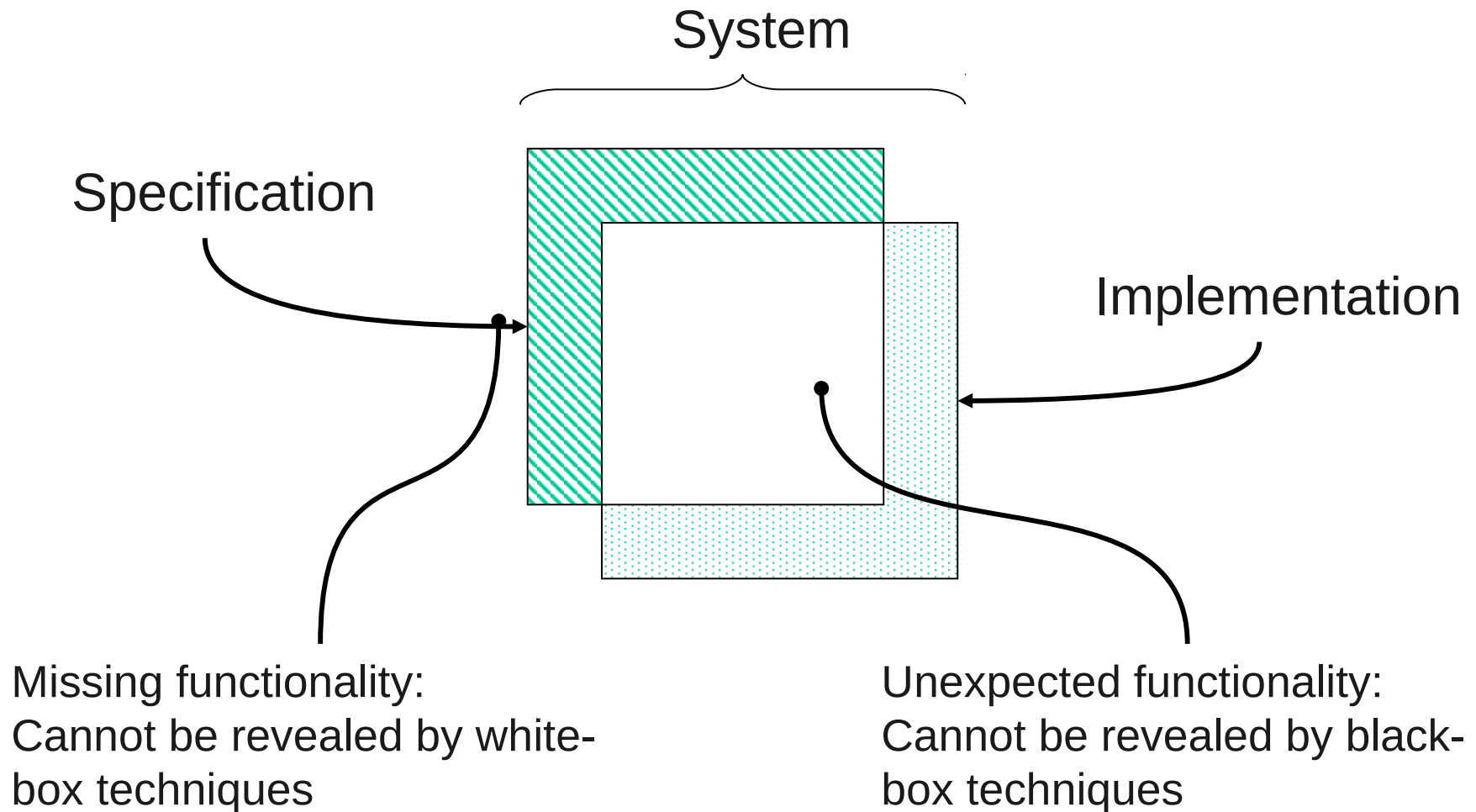
Oracle:
A function that gives
the “truth”
(*correct output*)

Fault → Error → Failure

Testing a subsystem



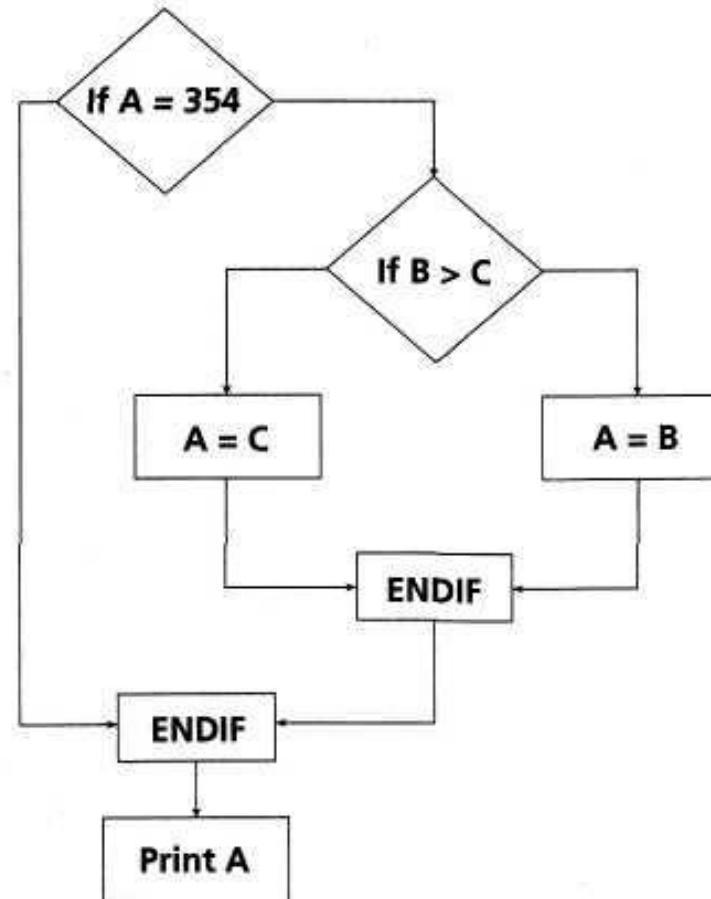
Black-box vs. White-box Testing



Code coverage

- Code as a flow graph

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
ENDIF
ENDIF
Print A
```



Input Space Partitioning

Test all possible inputs?

`int abs(x)`

`javac compiler`

=> Solution: Input space partitioning

- Determine equivalence classes
- 1-3 tests per equivalence class
- Test boundary/corner cases