Introduction to Software Engineering Methods

Requirements Analysis

Literature used

- Text book

  Chapter 5

Content

- Model and reality
- Activities during object modeling
- Class identification approaches
- Sequence diagrams
- State chart diagrams

Why modelling?

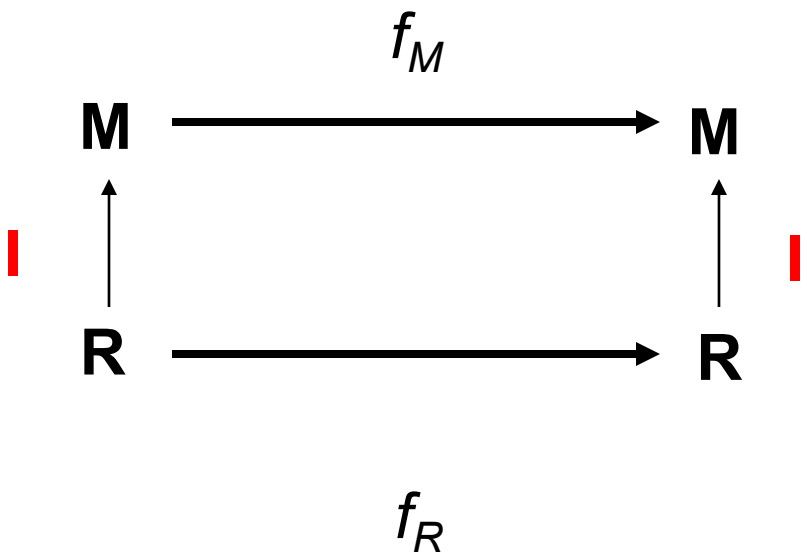
- We use models
  - To deal with complexity:
    - To abstract away from details in the reality, so we can draw complicated conclusions in the reality with simple steps in the model
  - To get insights into the past or presence and to make predictions about the future

Models

- A model is always an approximation
  - We must say “according to our knowledge”, or “with today’s knowledge”
  - We can only build models from reality, which are “true” until, we have found a counter example
- Principle of Falsification
  - ***demonstrating that relevant details of model are presented incorrectly or not presented at all***
- The falsification principle is the basis of software development
  - The goal of prototypes, reviews and system testing is to falsify the software system

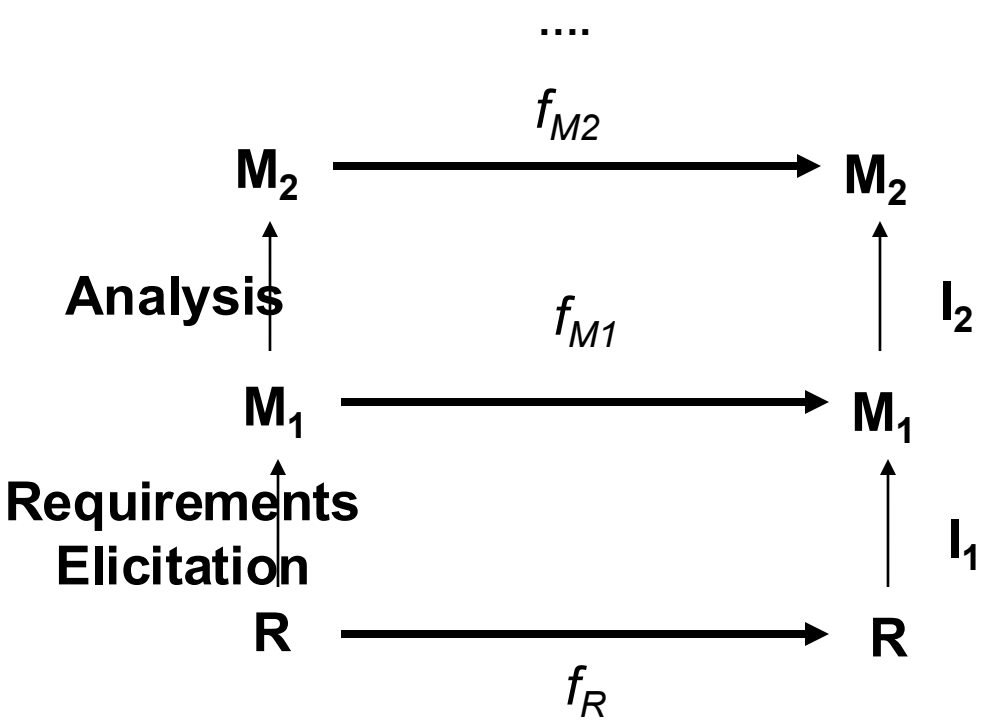
Models

- Relationships, which are valid in reality R, are also valid in model M.
  - I : Mapping of real things in reality R to abstractions in the model M (Interpretation)
  - f_M: relationship between abstractions in M
  - f_R: relationship between real things in R

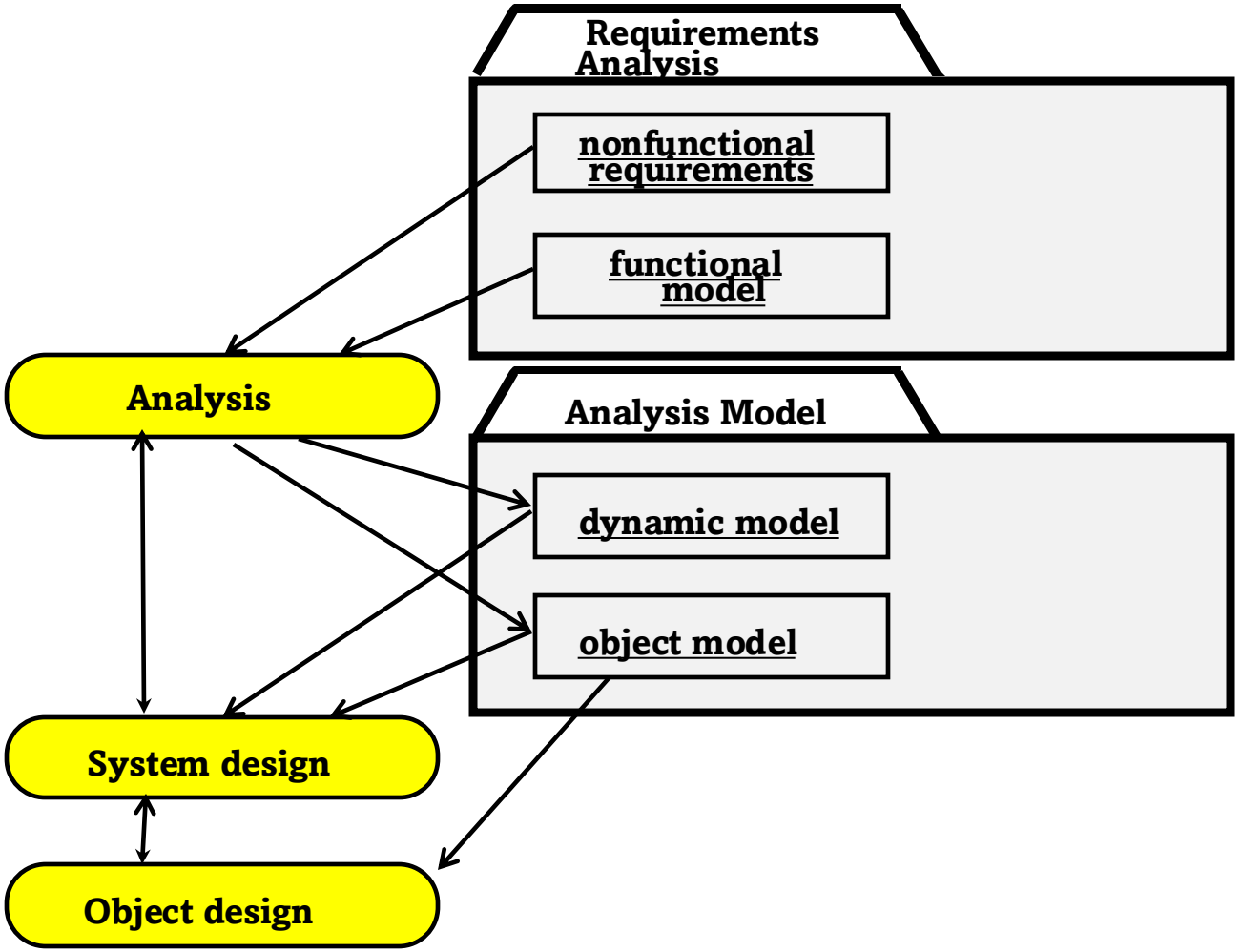


Models

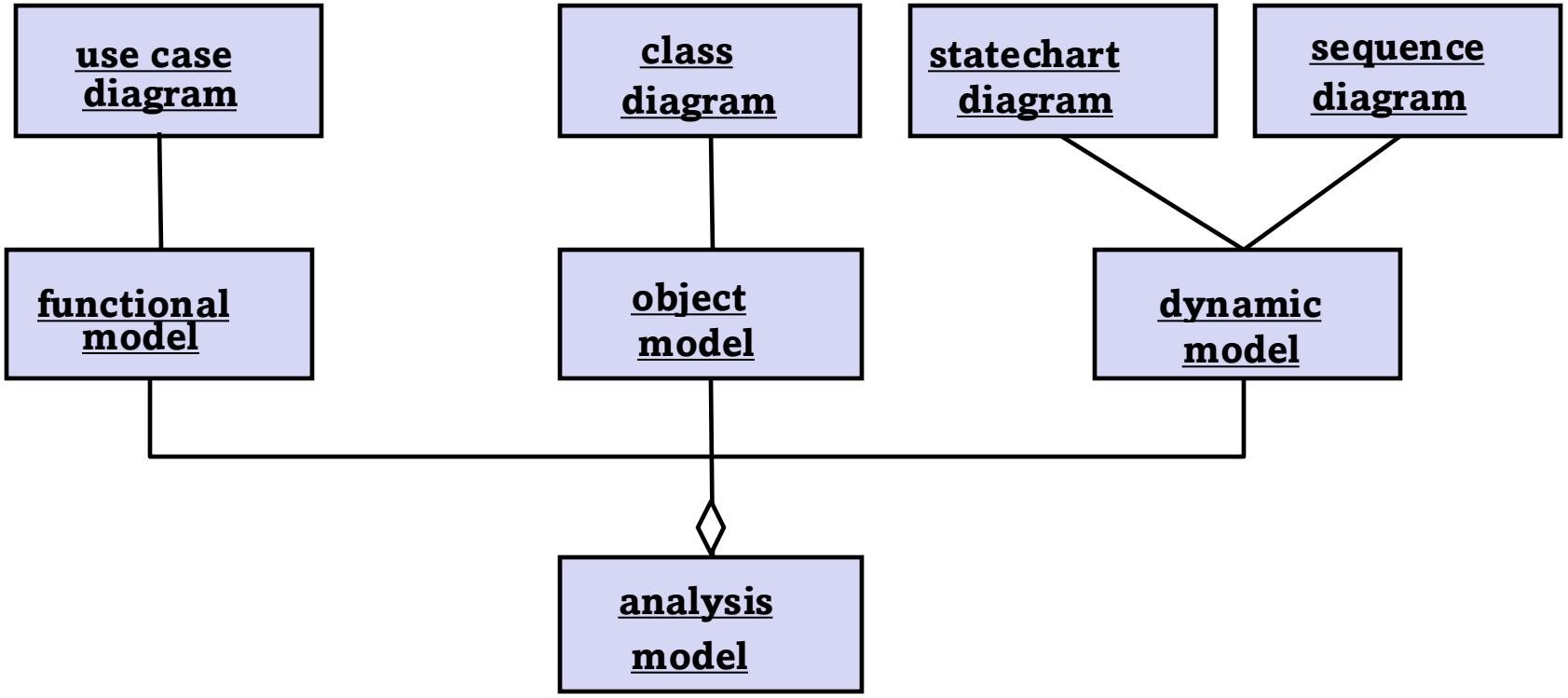
- Modeling is relative.
- We can think of a model as reality and can build another model from it (with additional abstractions)
- Development of a software system is a transformation of models



Products of Requirements Elicitation and Analysis



Analysis model



When is dominant a particular model?

- ***Object model*:** The system has objects with nontrivial state.
- ***Dynamic model:*** The model has many different types of events: Input, output, exceptions, errors, etc.
- ***Functional model*:** The model performs complicated transformations (e.g. computations consisting of many steps).

Which of these models is dominant in the following three cases?

- Compiler
- Database system
- Spreadsheet program

When is a model dominant?

- *Compiler:*
  - The functional model most important.
  - The dynamic model is trivial because there is only one type input and only a few outputs.
- *Database systems:*
  - The object model most important.
  - The functional model is trivial, because the purpose of the functions is usually to store, organize and retrieve data.
- *Spreadsheet program:*
  - The functional model most important.
  - The dynamic model is interesting if the program allows computations on a cell.
  - The object model is trivial, because the spreadsheet values are trivial and cannot be structured further. The only interesting object is the cell.

Components of an Object Model

- Classes
- Associations (Relations)
  - Generic associations
  - Canonical associations
    - Part of- Hierarchy (Aggregation)
    - Kind of-Hierarchy (Generalization)
- Attributes
  - Attributes in one system can be classes in another system
- Operations
  - Generic operations: Get/Set, General world knowledge, design patterns
  - Domain operations: Dynamic model, Functional model

Object Modeling

- Main goal:
  - Find the important abstractions
- What happens if we find the wrong abstractions?
  - Iterate and correct the model
- Steps during object modeling
  - Class identification
    - Based on the fundamental assumption that we can find abstractions
  - Find the attributes
  - Find the methods
  - Find the associations between classes

Components of a Dynamic Model

- Sequence diagrams
  - to model interactions between objects
- State-Chart diagrams
  - to model behavior of individual objects

Analysis activities

- Identify classes/objects
- Identify associations, identify aggregates
- Identify attributes
- Modeling inheritance relationships
- Map use cases to objects with Sequence Diagrams
- Model state-dependent behavior of individual objects
- Reviewing the analysis model

Class Identification

- Class identification is crucial to object-oriented modeling
- Basic assumption:
  - We can find the classes for a new software system and identify the classes in an existing system
- Why can we do this?
  - Philosophy, science, experimental evidence

Class Identification Approaches

- Application domain approach
  - Ask application domain expert to identify relevant abstractions
- Syntactic approach
  - Use noun-verb analysis (Abbot's technique) to identify components of the object model (from descriptions or from use cases - extract participating objects from flow of events)
- Common class pattern approach
  - Use generic classification theory of objects
- Design patterns approach
  - Use reusable design patterns
- Component-based approach
  - Identify existing solution classes
- Object types approach
  - Identify existing object types

Mapping parts of speech to object model components [Abbott, 1983]

Part of speech	*Model component*	*Example*
Proper noun	object	Jim Smith
Improper noun	class	Toy, doll
Doing verb	method	Buy, recommend
being verb	inheritance	is-a (kind-of)
having verb	aggregation	has an
modal verb	constraint	must be
adjective	attribute	3 years old
transitive verb	method	enter
intransitive verb	method (event)	depends on

Example

Flow of events:

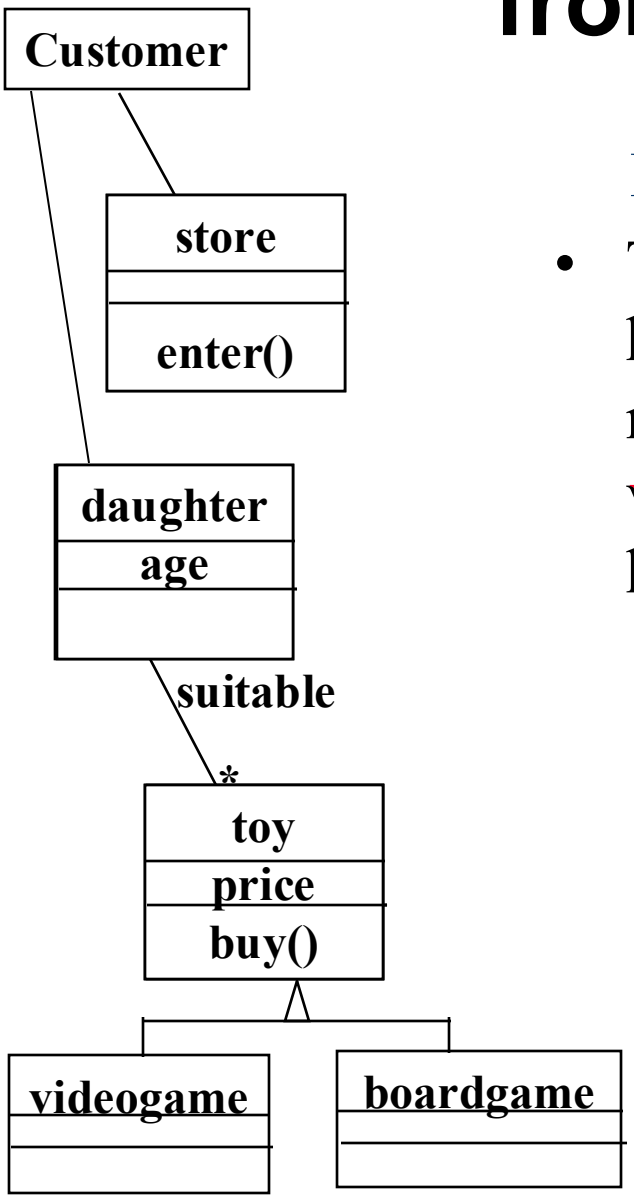
- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

An assistant helps him.
The suitability of the game depends on the age of the child.
His daughter is only 3 years old.
The assistant recommends another type of toy, namely the boardgame. The customer buy the game and leaves the store

Textual analysis (Abbot‘s technique)

Example	*Grammatical construct*	*UML Component*
“Monopoly"	Concrete Person, Thing	Object
“toy"	noun	class
"3 years old"	Adjective	Attribute
“enters"	verb	Operation
“depends on…."	Intransitive verb	Operation (Event)
“is a" ,“either..or", “kind of…"	Classifying verb	Inheritance
"Has a ", “consists of"	Possessive Verb	Aggregation
“must be", “less than…"	modal Verb	Constraint

Generation of a class diagram from flow of events



Flow of events:

- The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store

Noun-phrase approach

- Irrelevant classes are those that are outside the problem domain
- Relevant classes are those that manifestly belong to the problem domain
- Fuzzy classes are those that the analyst cannot confidently and unanimously classify as relevant

Common class pattern approach (Bahrami,1999)

Classification theory is a part of science concerned with partitioning the world of objects into useful groups so that it is possible reason about them better

- **Concept class** – a concept is a notion that a large community of people share and agree on (e.g. reservation)
- **Events class** – an event is something that does not take time relative to our time scale (e.g. arrival, departure)
- **Organization class** – organization is any kind of purposeful grouping or collection of things (e.g. travel_agency)
- **People class** – "people" is understood here as a role that a person plays in the system (e.g. passenger)
- **Places class** – places are physical locations relevant to the information system (e.g. agency_office)

Rumbaugh et al. (1999) propose a different classification scheme

- physical class (e.g. Airplane);
- business class (e.g. Reservation);
- logical class (e.g. FlightTimetable);
- application class (e.g. ReservationTransaction);
- computer class (e.g. index);
- behavioral class (e.g.ReservationCancellation).

Finding Participating Objects in Use Cases

- Pick a ***use case*** and look at its ***flow of events***
  - Find terms that developers or users need to clarify in order to understand the flow of events
  - Look for recurring nouns (e.g., Incident),
  - Identify real world entities that the system needs to keep track of (e.g., FieldOfficer, Dispatcher, Resource),
  - Identify real world procedures that the system needs to keep track of (e.g., EmergencyOperationsPlan),
  - Identify data sources or sinks (e.g., Printer)
  - Identify interface artifacts (e.g., PoliceStation)

- Be prepared that some objects are still missing and need to be found:
  - Model the flow of events with a sequence diagram

- Always use the user's terms

Managing Interactions among Objects with CRC (class-responsibility-collaborators) Cards

- The CRC approach involves brainstorming sessions, made easy by the use of specially prepared cards
- The cards have three compartments:
  - Class name
  - Responsibilities
  - Collaborations

- The CRC approach is an animated process during which the developers "play cards“
- Developers fill cards with class names and assign responsibilities and collaborators while "executing" a processing scenario (e.g. a use-case scenario).

Mixed approach

- In practice, the process of class discovery is likely to be guided by different approaches at different times
- The process is neither top-down nor bottom-up; it is middle-out.
- A **possible** scenario
  - The initial set of classes may be discovered from the generic knowledge and experience of analysts
  - The common class patterns approach can provide additional guidance
  - Other classes may be added from the analysis of high level descriptions of the problem domain using the noun phrase approach
  - If use-case diagrams are available, then the use-case driven approach can be used to add new and verify existing classes
  - Finally, the CRC approach will allow brainstorming the list of classes discovered so far.

Separate concepts of solution domain in analysis model

- Domain concepts should be represented in the analysis object model
  - Universal time
  - Time zone
  - Location
- Software classes should not be represented in the analysis object model
  - TimezoneDatabase
  - GPSLocation
  - UserID

Class Identification

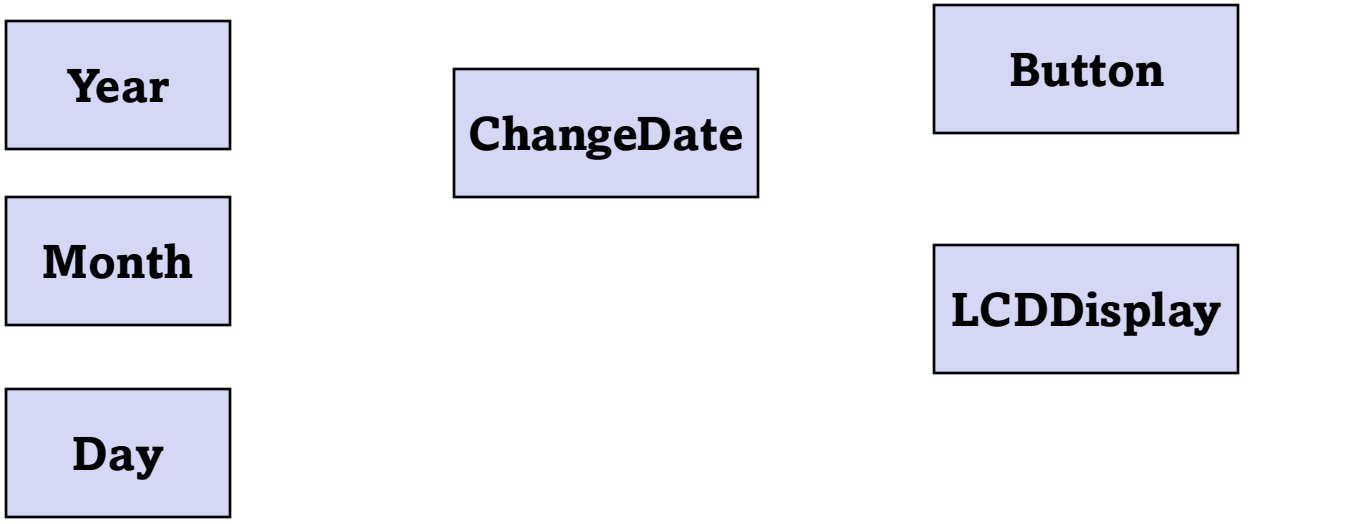
– Design patterns approach (Apply design knowledge):
  - Distinguish different types of objects
  - Apply design patterns

– Component-based approach:
  - Identify existing solution classes

Object Types

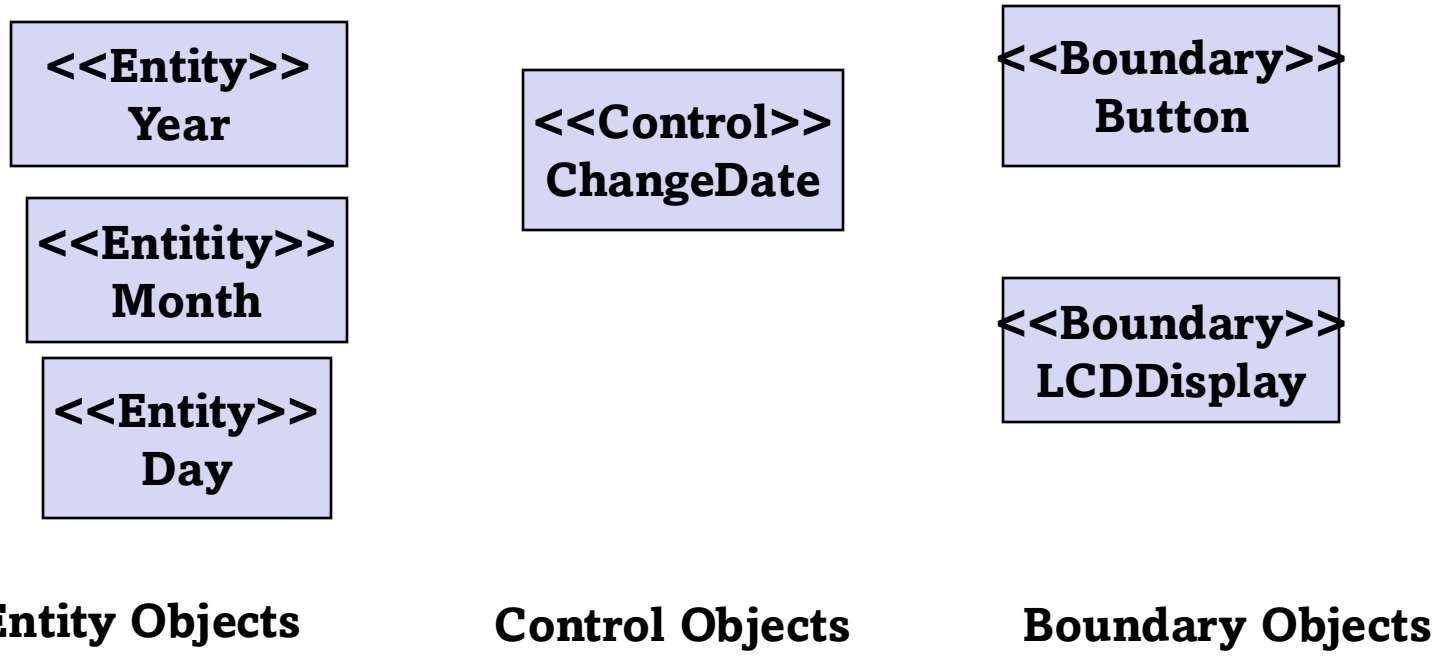
- Entity Objects
  - Represent the persistent information tracked by the system (Application domain objects, "Business objects")
- Boundary Objects
  - Represent the interaction between the user and the system
- Control Objects:
  - Represent the control tasks performed by the system
- Having three types of objects leads to models that are more resilient to change.
  - The interface of a system changes more likely than the control
  - The control of the system change more likely than the application domain

Object Types



Naming of Object Types in UML

- UML provides the stereotype mechanism to present new modeling elements



Identifying Entity Objects

- Heuristics to be used together with Abbot's heuristics:
  - Terms that developers or users need to clarify in order to understand the use case
  - Recurring nouns in the use cases
  - Real-world entities that the system needs to track
  - Real-world activities that the system needs to track
  - Data sources or sinks

Identifying Boundary Objects

- Heuristics for identifying boundary objects
  - Identify user interface controls that the user needs to initiate the use case
  - Identify forms the users need to enter data into the system
  - Identify notices and messages the system uses to respond to the user
  - When multiple actors are involved in a use case, identify actor terminals to refer to user interface under construction
  - Do not model the visual aspects of the interface with boundary objects (user mock-ups are better suited for that)
  - Always use the end user's terms for describing interfaces; do not use terms from the solution or implementation domains.

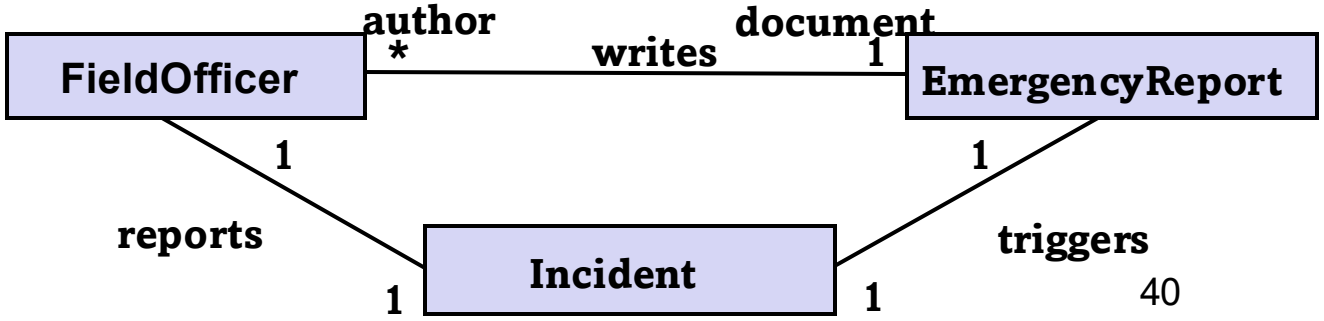
Identifying control objects

- Heuristics for identifying control objects
  - Identify one control object per use case.
  - Identify one control object per actor in the use case.
  - The life span of a control object should cover the extent of the use case or the extent of a user session.

Identifying associations

Heuristics for identifying associations

- •Examine verb phrases (e.g., has, is part of, manages, reports to, is triggered by, is contained in, talks to, includes).
- •Name associations and roles precisely.
- •Use qualifiers as often as possible to identify namespaces and key attributes.
- •Do not worry about multiplicity until the set of associations is stable.
- •Too many associations make a model unreadable.
- •Eliminate any association that can be derived from other associations.

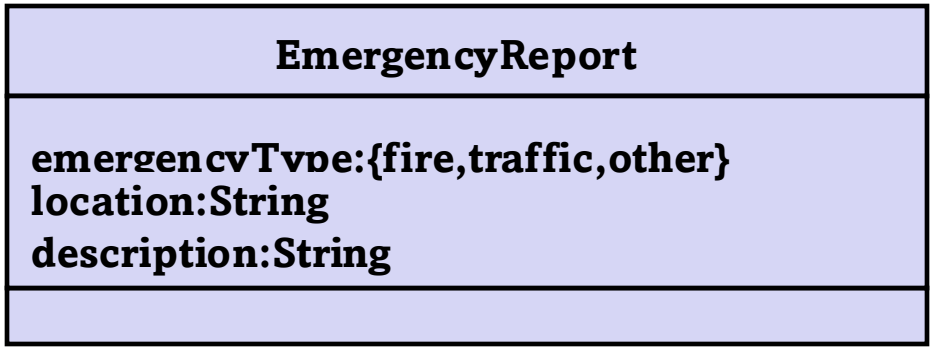


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Identifying Attributes

Heuristics for identifying attributes

- Examine possessive phrases or adjective phrases.
- Represent stored state as an attribute of the entity object.
- Describe each attribute.
- Do not waste time describing fine details before the object structure is stable.



Who uses class diagrams?

- The **customer** and the **end user** are often not interested in class diagrams. They usually focus more on the functionality of the system.
- **The application domain expert** uses class diagrams to model the application domain
- The **developer** uses class diagrams during the development of a system, that is, during analysis, system design, object design and implementation.

Different users of class diagrams

- According to the development activity, the developer plays different roles.
  - Analyst
  - System-Designer,
  - DetailedDesigner
  - Implementor.
- In small systems some of the roles do not exist or are played by the same person.
- Each of these roles has a different view about the models.

Why do we distinguish these different users of class diagrams?

- Models often don‘t distinguish between application classes (“address book") and solution class (“array", “tree").
  - **Reason:** Modelling languages like UML allow the use of both types of classes in the same model.
  - **Preferred** : No solution classes in the analysis model.
- Many systems don‘t distinguish between specification and implementation of a class.
  - **Reason:** Object-oriented programming languages allow the simultaneous use of specification and implementation of a class.
  - **Preferred**: The object design model does not contain implementations.
- The key for creating high quality software systems is the exact distinction between
  - Application and solution domain classes
  - Interface specification and implementation specification

Dynamic Modeling

- Dynamic model consists of:
  - A collection of multiple state chart diagrams (one state chart diagram for each class with important dynamic behavior).
  - A collection of communication diagrams between objects
- Purpose:
  - Detect and supply methods for the object model
- How do we do this?
  - Start with use case or scenario
  - Model interaction between objects => sequence diagram
  - Model dynamic behavior of a single object => statechart diagram

Sequence Diagram

- A sequence diagram is a graphical description of objects participating in a use case or scenario using a graph notation
  - Sequence diagrams obtained from the flow of events in the use case or scenario
- Relation to object identification:
  - Objects/classes have already been identified during object modeling
  - New objects are identified as a result of dynamic modeling
- Heuristic:
  - An event (represented as a message in sequence diagrams) always has a sender and a receiver.
  - Find objects for each event/message

    => These are the objects participating in the use case

An Example

- Flow of events in a “GetPosition” use case :

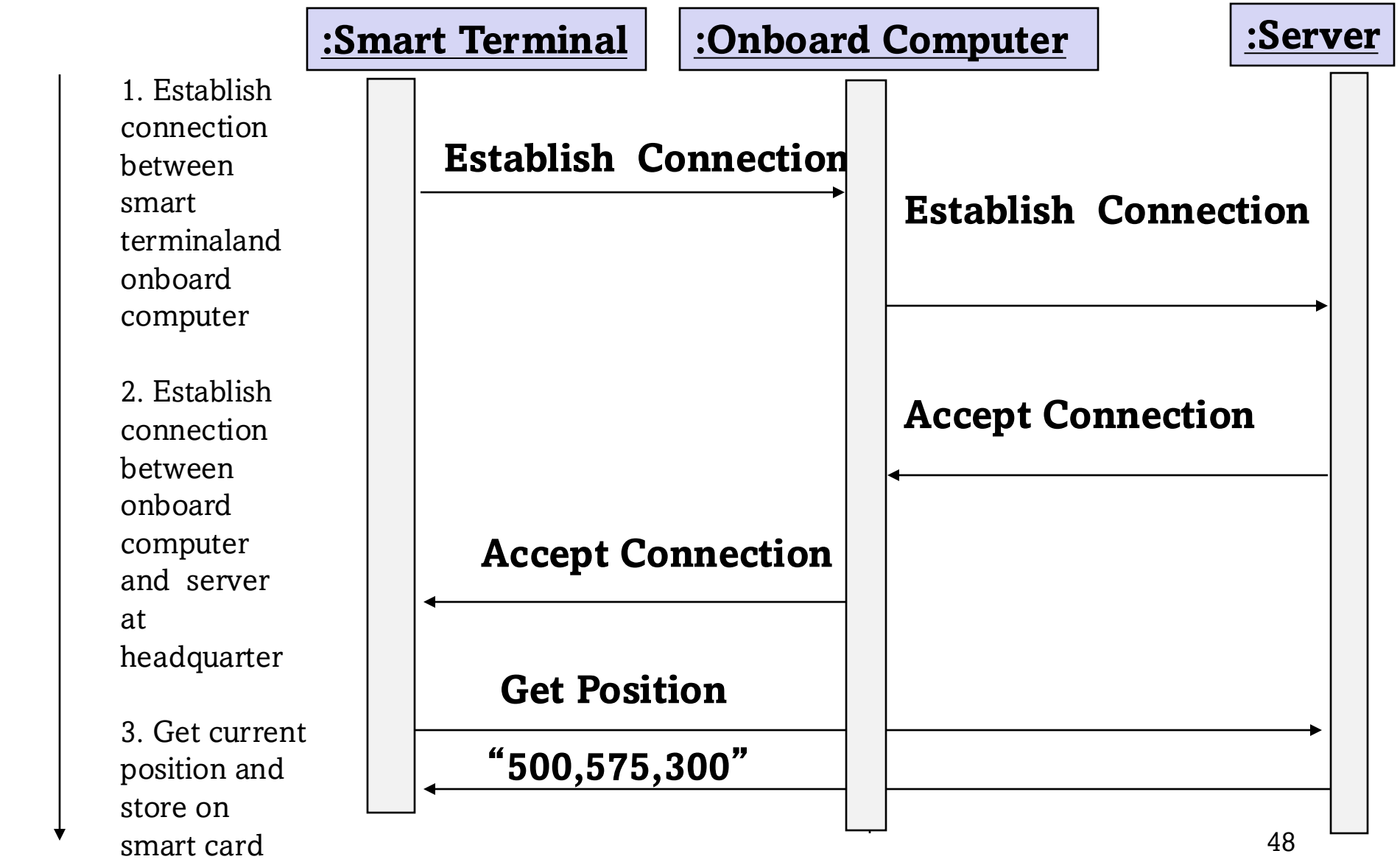
  1. Establish connection between smart terminal and onboard computer

  2. Establish connection between onboard computer and server at Headquarter

  3. Get current maintenance expert position and store in smart terminal

- Which are the objects?
  - Onboard computer, smart terminal, server

Sequence Diagram for "GetPosition"



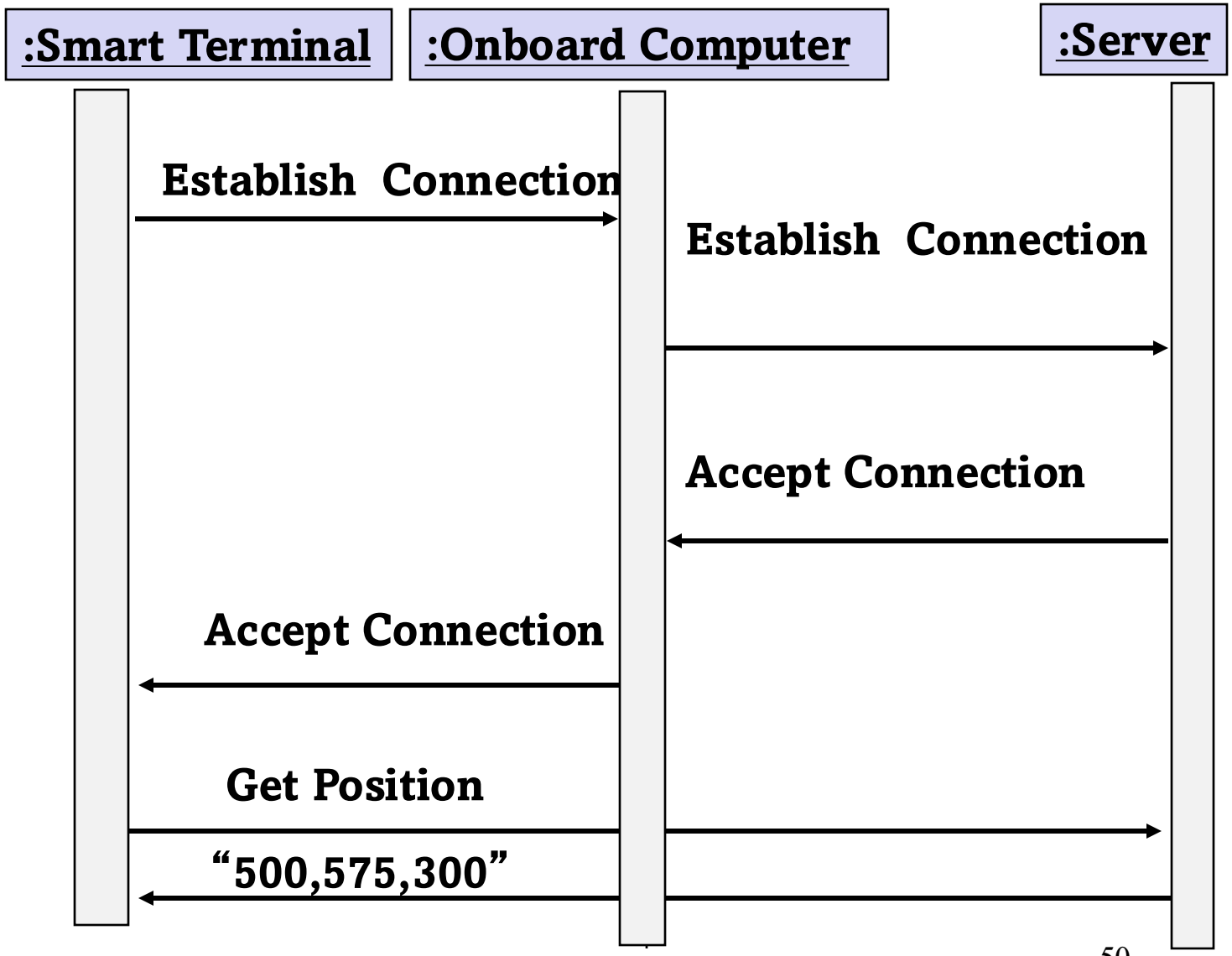
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Heuristics for Sequence Diagrams

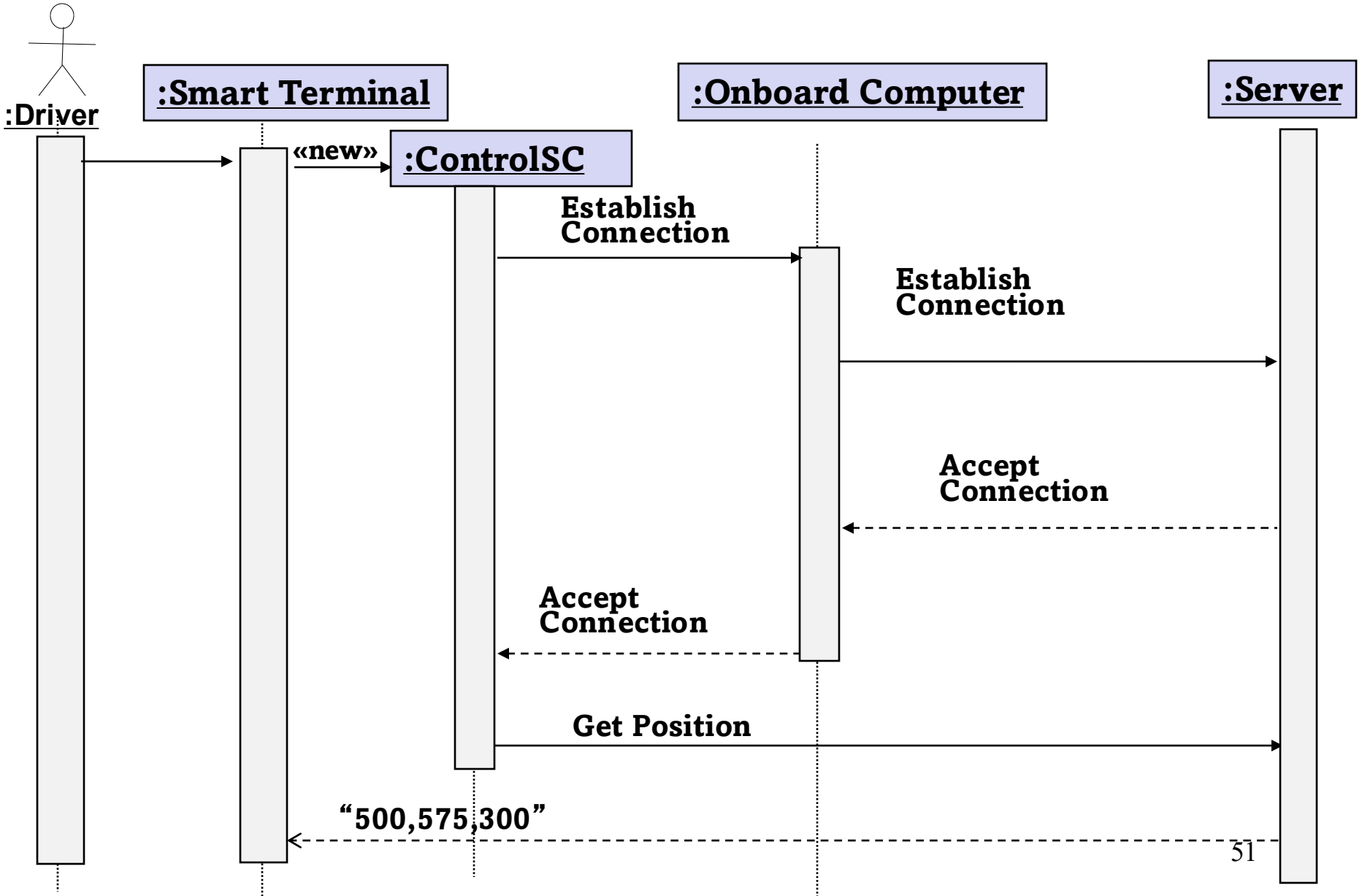
- Layout:
  - 1st column: Should correspond to the actor who initiated the use case
  - 2nd column: Should be a boundary object
  - 3rd column: Should be the control object that manages the rest of the use case
- Creation:
  - Control objects are created at the initiation of a use case
  - Other then initial Boundary objects are created by control objects
- Access:
  - Entity objects are accessed by control and boundary objects,
  - Entity objects should never call boundary or control objects: This makes it easier to share entity objects across use cases and makes entity objects resilient against technology-induced changes in boundary objects.

Is this a good Sequence Diagram?

- First column is not the actor
- It is not clear where the boundary object is
- It is not clear where the control object is

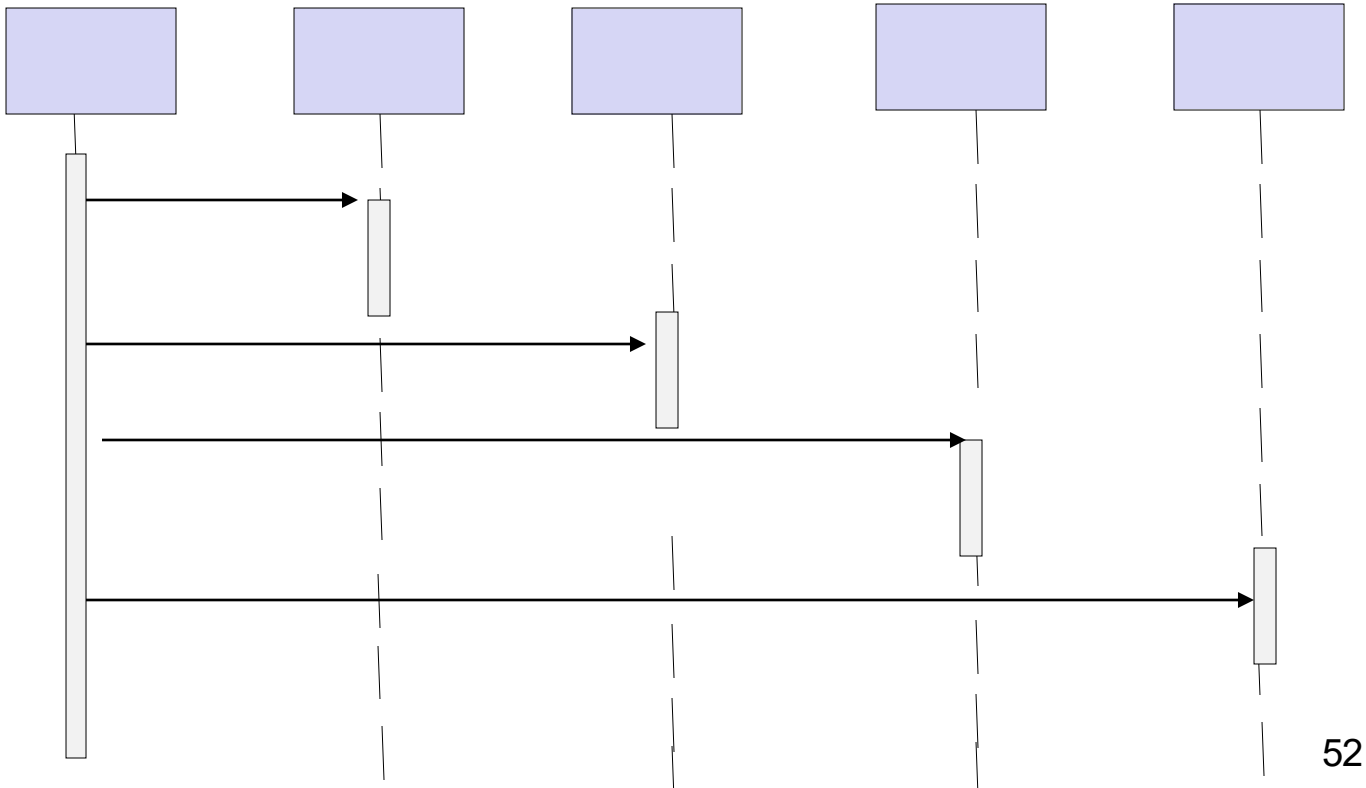


A Better Sequence Diagram for "GetPosition"



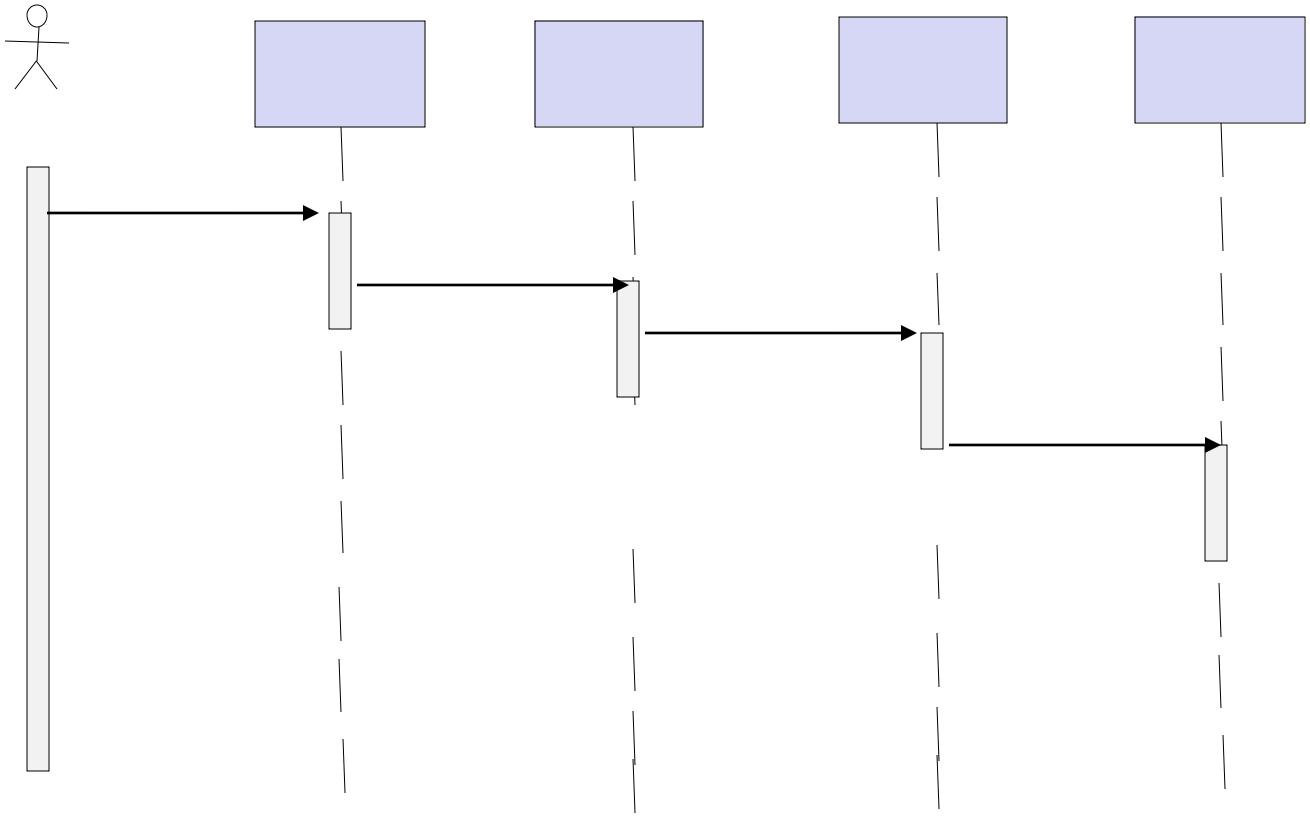
Fork Diagram

- Much of the dynamic behavior is placed in a single object, usually the control object. It knows all the other objects and often uses them for direct questions and commands.



Stair Diagram

- The dynamic behavior is distributed. Each object delegates some responsibility to other objects. Each object knows only a few of the other objects and knows which objects can help with a specific behavior.



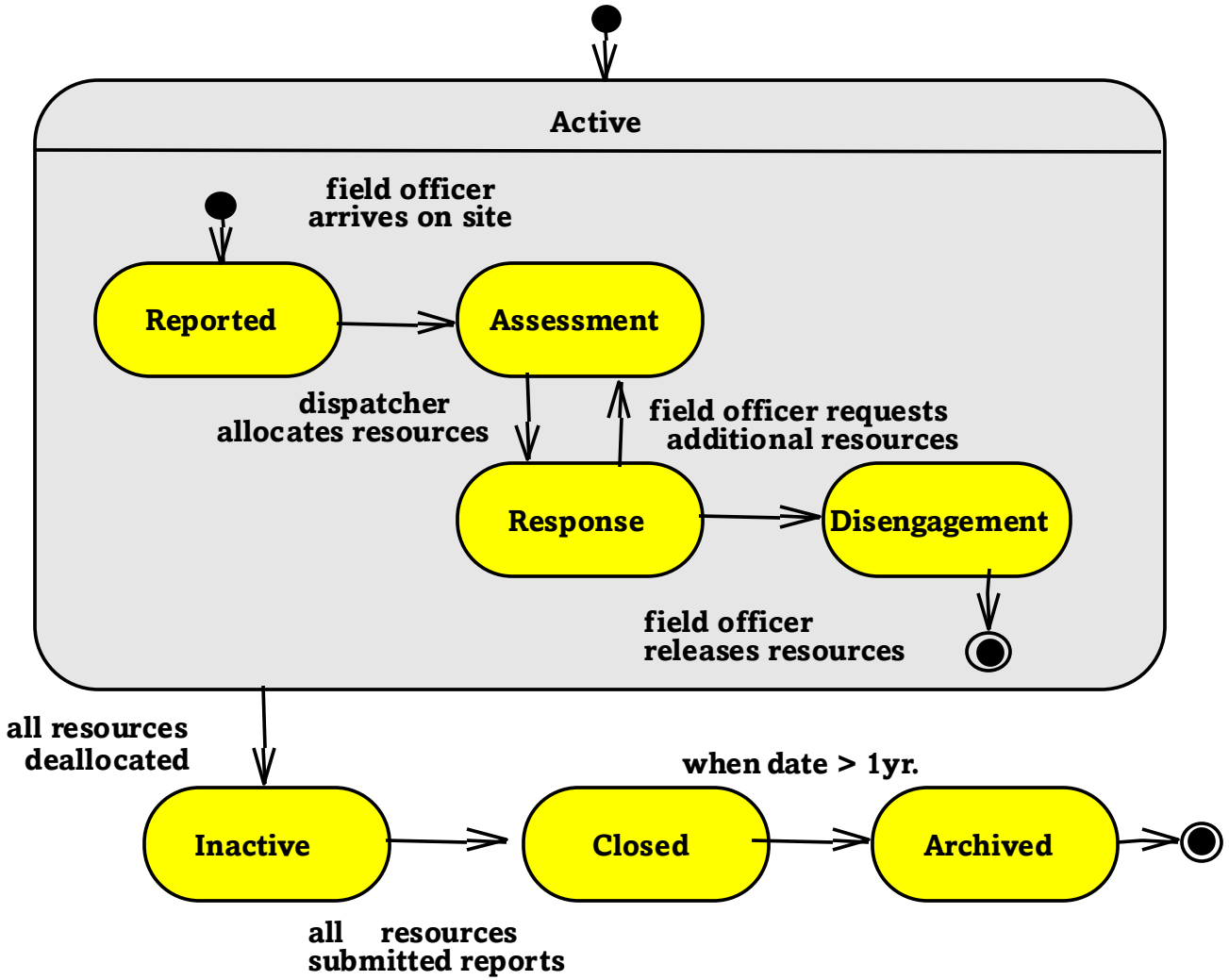
Fork or Stair?

- Decentralized control structure
  - The operations have a strong connection
  - The operations will always be performed in the same order
- Centralized control structure better support of change (because of encapsulation in one object)
  - The operations can ***change*** order
  - New operations can be inserted as a result of new requirements

State Chart Diagram vs Sequence Diagram

- Sequence diagrams help to identify
  - The *temporal relationship* between objects over time
  - *Sequence of operations* as a response to one or more events
- State chart diagrams help to identify:
  - *Changes* to an individual object over time

Modeling State-Dependent Behavior of Individual Objects (Incident)



Modeling Checklist for the Review

- **Is the model correct?**

A model is correct if it represents the client's view of the the system: Everything in the model represents an aspect of reality

- **Is the model complete?**

Every scenario through the system, including exceptions, is described.

- **Is the model consistent?**

The model does not have components that contradict themselves (for example, deliver contradicting results)

- **Is the model unambiguous?**

The model describes one system (one reality), not many

- **Is the model realistic?**

The model can be implemented

Summary

1. What are the functions/transformations?  **Functional Modeling**

   Create *scenarios and use case diagrams*

   Talk to client, observe, get historical records, do thought experiments

2. What is the structure of the system?  **Object Modeling**

   Create *class diagrams*

   Identify objects.

   What are the associations between them? What is their multiplicity?

   What are the attributes of the objects?

   What operations are defined on the objects?

3. What is its behavior?  **Dynamic Modeling**

   Create *sequence diagrams*

   Identify senders and receivers

   Show sequence of events exchanged between objects. Identify event dependencies and event concurrency.

   Create *state diagrams*

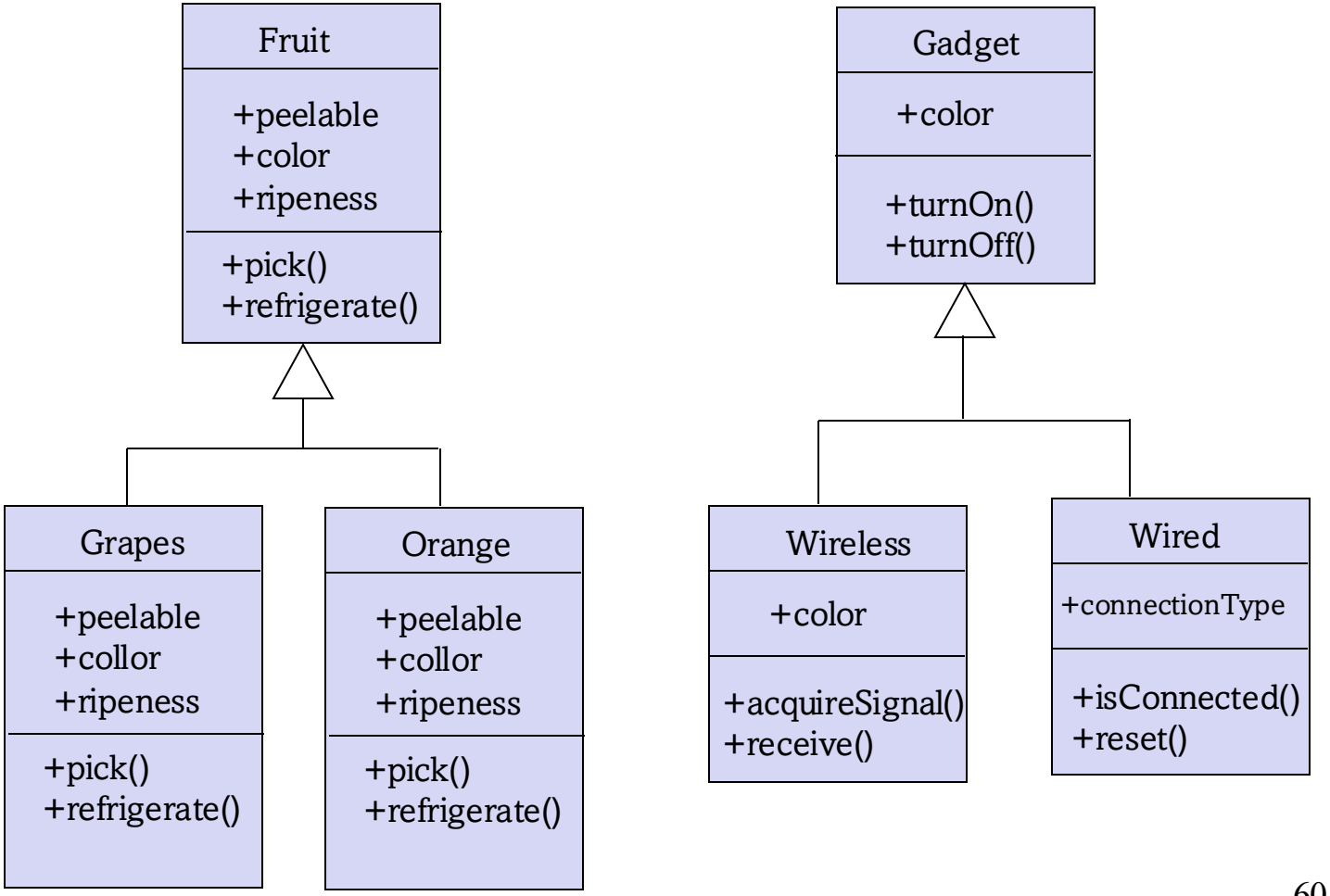
   Only for the dynamically interesting objects.

Aspect-Oriented Programming (AOP)

- OOP – assumes knowledge of all needed interfaces for the developer and interfaces are not easy to change
- Some problems are not attached to any particular domain, but they are spread across them
- OOP composes components vertically while we might need to compose them horizontally

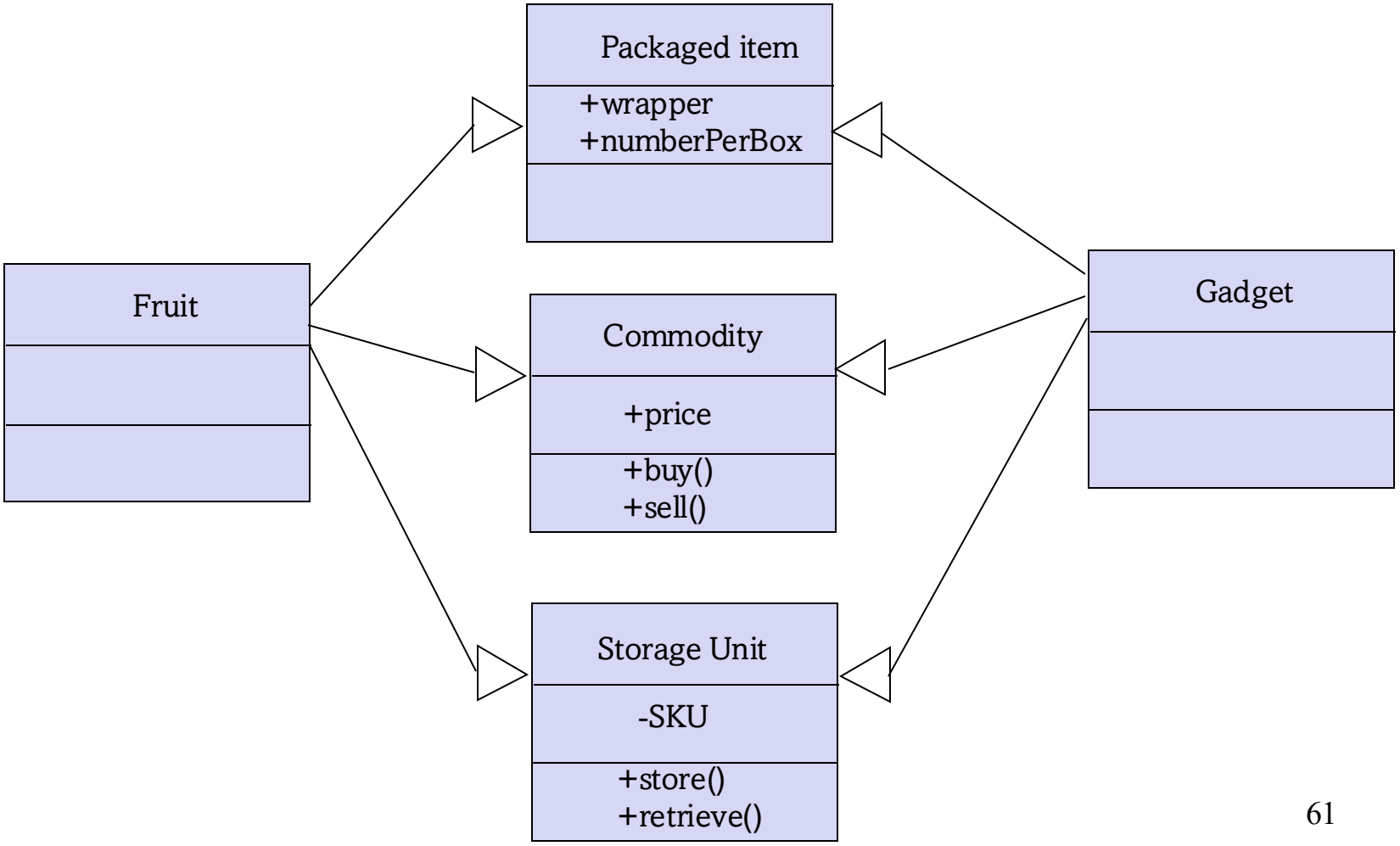
Adopted from Ivan Kiselev. Aspect-Oriented Programming with AspectJ

Fruit and Gadget hierarchy



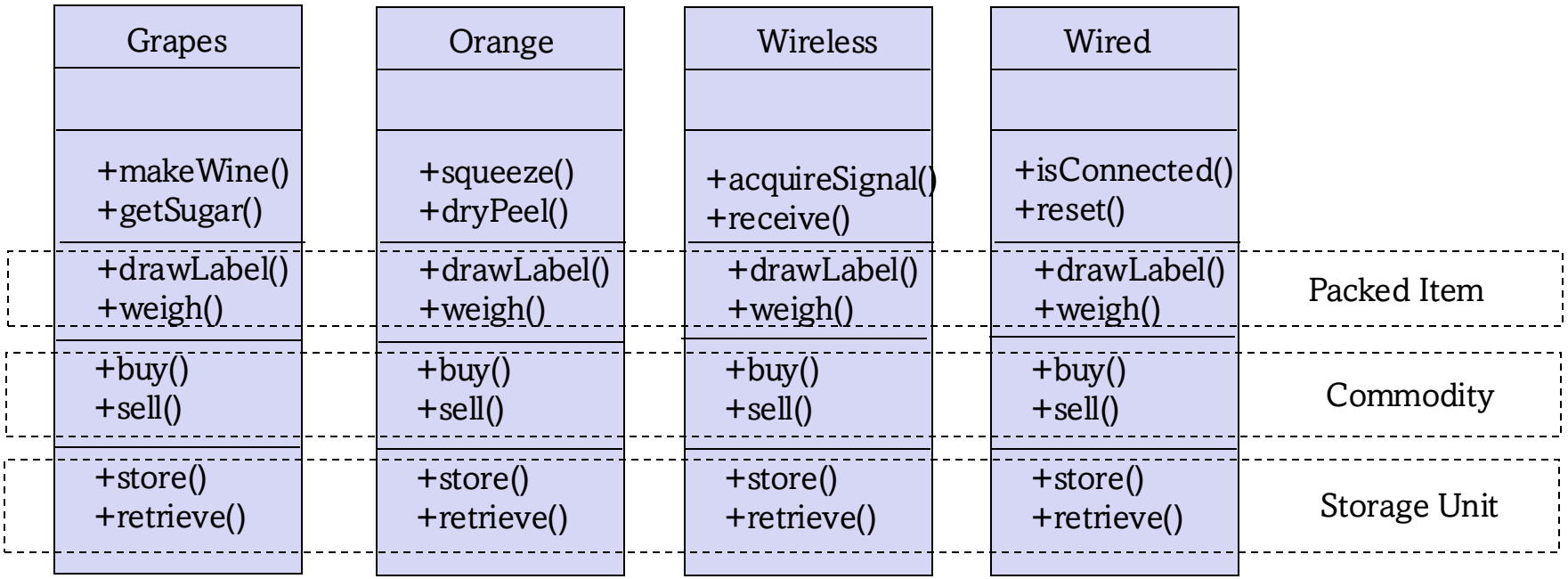
Adopted from Ivan Kiselev. Aspect-Oriented Programming with AspectJ

Fruits and Gadgets as commodities



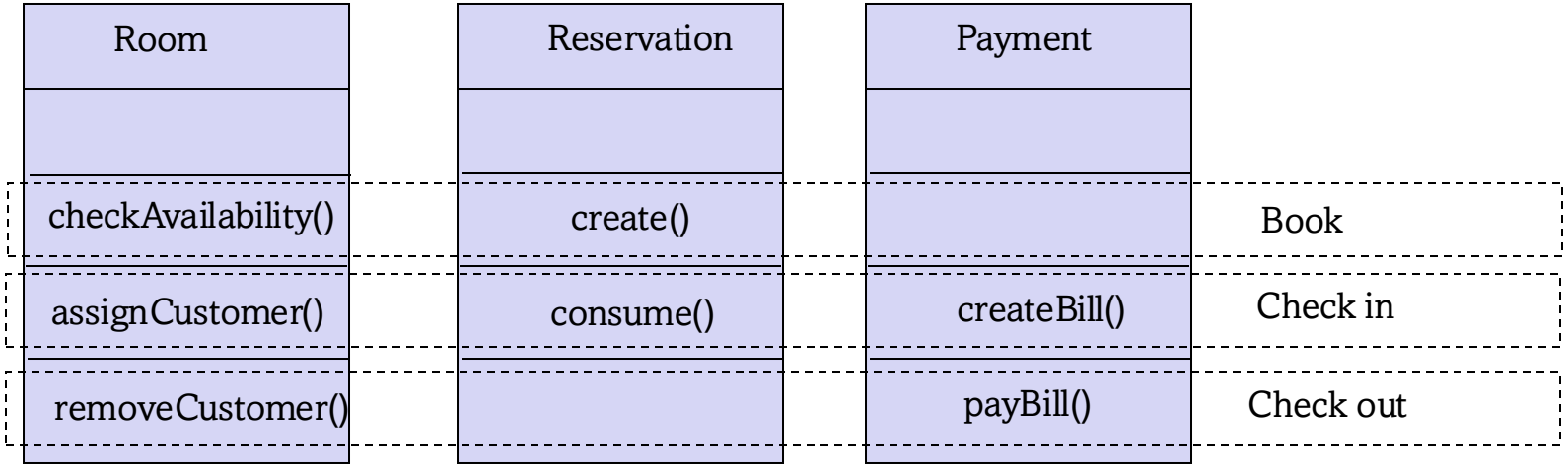
Adopted from Ivan Kiselev. Aspect-Oriented Programming with AspectJ

Crosscutting concerns



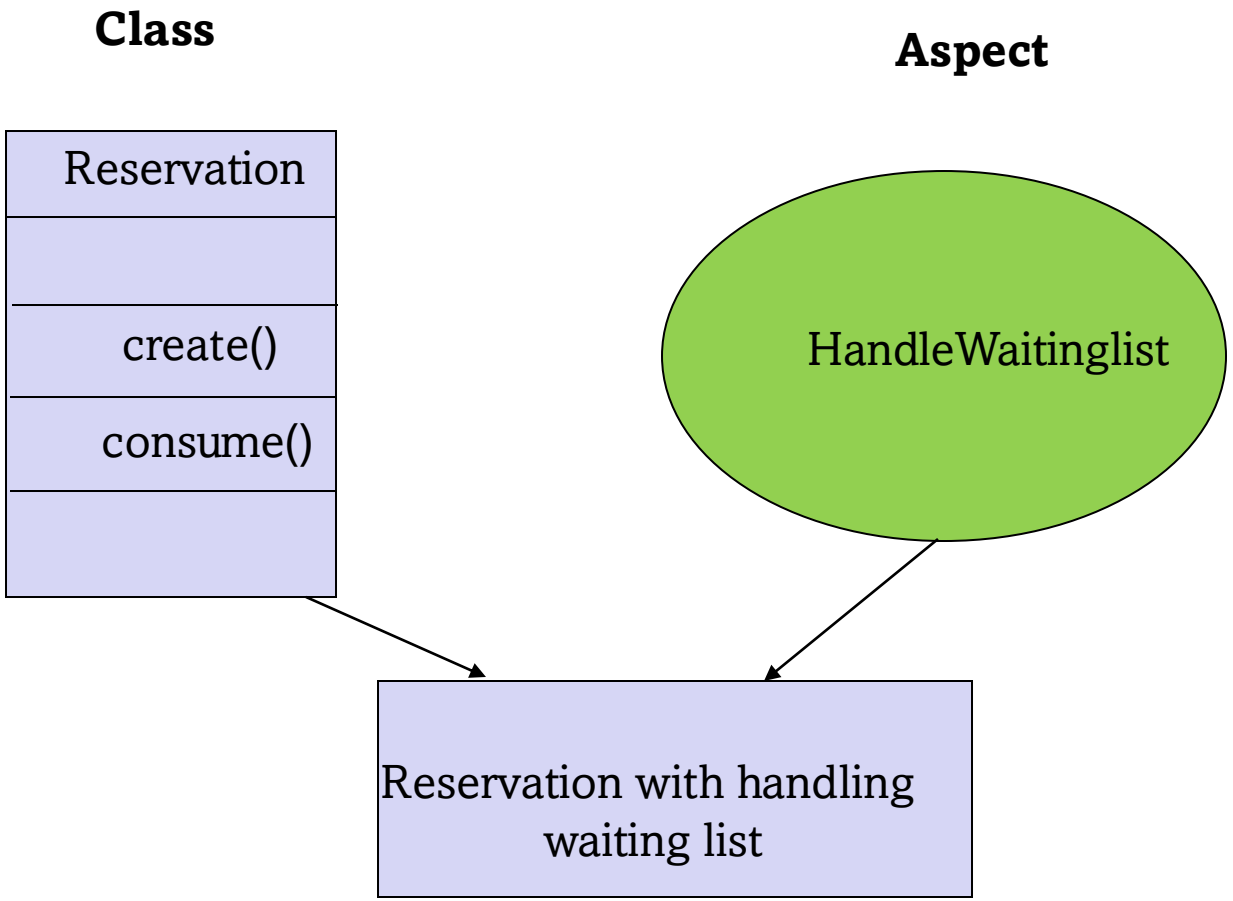
Adopted from Ivan Kiselev. Aspect-Oriented Programming with AspectJ

Room, Reservation and Payment



Adopted from Jacobson, Ng. Aspect-Oriented Software Engineering

A Simplified Example
(Handling Waiting List)



Implementation

- Join Points – special well-defined points in the program flow where horizontal extension/bundling appears in the code
- Pointcuts – predicates that match joint points
- Advices – a code which is executed when a given pointcut is reached

Example continue

```
class Reservation {
   public void create {
     ...
        if(theRoom.getQuantityAvalable()<=0){
            throw new NoRoomException() {
   ...}
```

```
aspect HandleWaitingList {
...
   pointcut makingReservation():
        execution(void Reservation.create());
   after throwing (NoRoomException e): makingReservation(){
       //code to add customer into waiting list
...
}
```

Next lecture

- System Design

Text book

Chapters 6 & 7