Course "Softwaretechnik"
Book Chapter 5
Analysis Model: Objects

Lutz Prechelt, Bernd Bruegge & Allen H. Dutoit

Freie Universität Berlin, Institut für Informatik

- Use Cases versus Objects
- Object identification with Abbott's technique
  - Nouns may indicate classes
  - Verbs may indicate operations
  - Adjectives may indicate attributes
  - Proper nouns may indicate object instances
  - "is a" may indicate inheritance
  - etc.
- Checklists
- Analysis vs. design model
  - roles, views, model differences

Where are we?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - Entwurf (Lösungsraum)
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - *Abstraktion*
  - Wiederverwendung
  - Automatisierung
- **Methodische Ansätze ("weich")**
  - ***Anforderungsermittlung***
  - **Entwurf**
  - **Qualitätssicherung**
  - **Projektmanagement**

Where are we?: Anforderungsermittlung

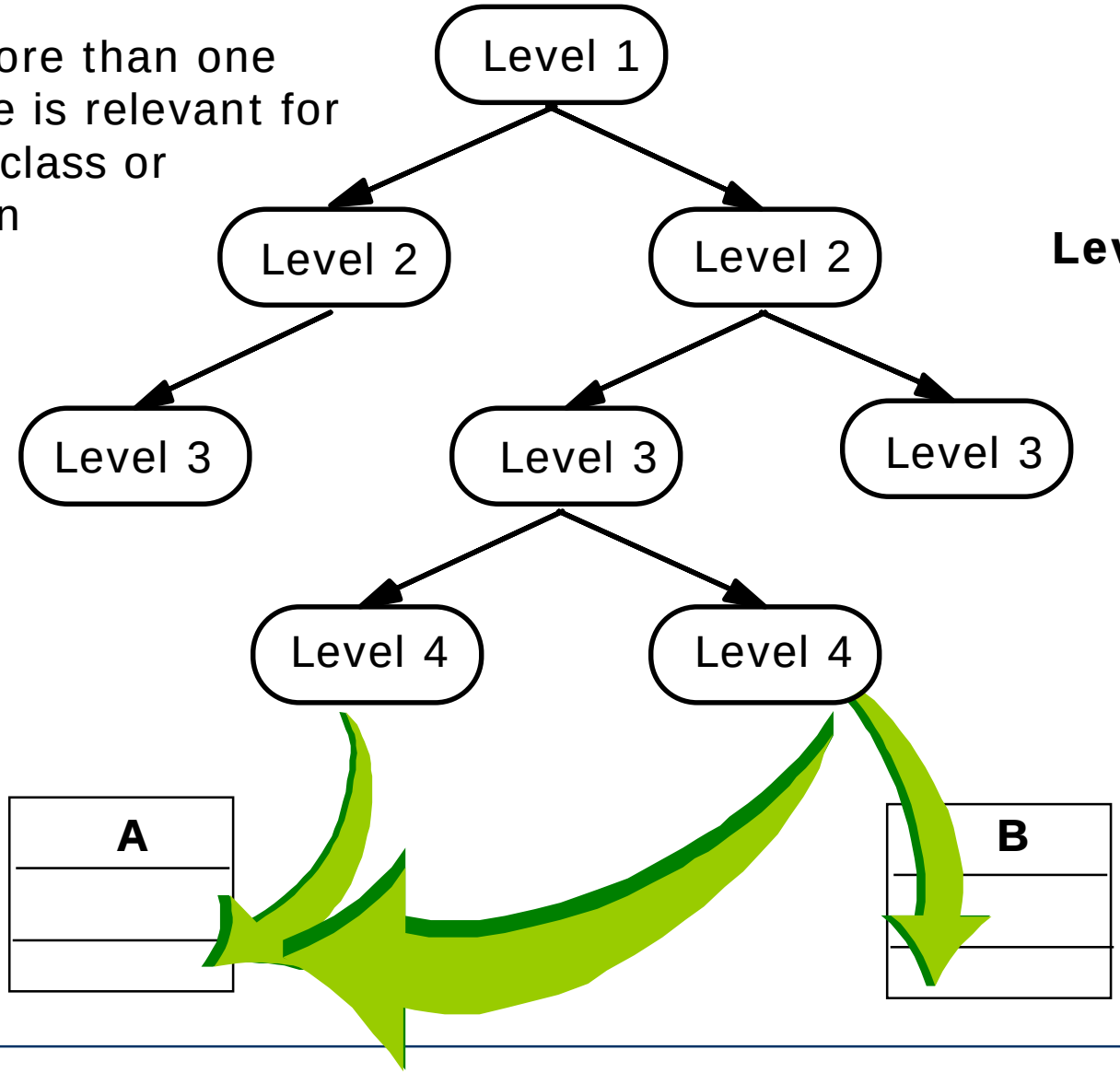
Freie Universität Berlin

- Einsicht: Man darf sich nicht auf intuitiven Eindruck darüber verlassen, was gebaut werden sollte
  - sondern sollte die Anforderungen systematisch ermitteln
- Prinzipien:
  - **Erhebung** der Anforderungen bei allen Gruppen von Beteiligten
  - **Beschreibung** in einer Form, die die Beteiligten verstehen
  - **Validierung** anhand der verschriftlichten Form
  - ***Spezifikation****: Übertragung in zur Weiterverarbeitung günstige Form (Analysemodell)*
  - **Trennung von Belangen**: Anford. möglichst wenig koppeln
  - ***Analyse auf Vollständigkeit****: Lücken aufdecken und schließen*
  - ***Analyse auf Konsistenz****: Widersprüche aufdecken und lösen*
  - **Mediation**: Widersprüche, die auf Interessengegensätzen beruhen, einer Lösung zuführen (Kompromiss oder Win-Win)
  - **Verwaltung**: Übermäßige Anforderungsänderungen eindämmen, Anforderungsdokument immer aktuell halten

From Use Cases to Objects: Classes may not be obvious

Freie Universität Berlin

Often more than one Use Case is relevant for a single class or operation



Overview

Level 1 Use Cases

Level 2 Use Cases

Operations

Participating Objects

Activities during Object Modeling

Goal: Find the abstractions important in the application domain

- Steps during object modeling
  - 1. Class identification
  - 2. Find attributes
  - 3. Find methods
  - 4. Find associations between classes

- Note: Order of steps
  - Goal: find the desired abstractions
  - The order of steps is flexible (the above is only a heuristic)
  - Iteration helps
- What happens if we find the wrong abstractions?
  - ➔ Must detect inconsistencies, then correct the model
- Resulting model reflects application domain and requirements
  - **It is not meant to be a solution design!**

Pieces of an Object Model

- Classes
  - With or without subclasses
- Associations (class or object relationships)
  - Generic/canonical associations
    - Part-of Hierarchy (Aggregation, on object level)
    - Kind-of Hierarchy (Generalization, on class level)
  - Domain-specific associations
- Attributes
  - Domain-specific
- Operations
  - Generic operations (create etc.): General world knowledge
  - Domain operations: Dynamic model, Functional model

Object vs. Class

- **Object** (**instance, dt.: *Exemplar***): Exactly one thing
  - E.g. this lecture on Software Engineering today
  - The term "instance" (german: "Exemplar") is preferable, because "object" is sometimes also used to mean a class
- A **class** abstractly describes a category of objects that share similar properties
  - e.g. Game, Tournament, mechanic, car, database
  - A class consists of a data type and operations
- **Object diagram**: A graphic notation for modeling objects, classes and their relationships ("associations"):
  - **Class diagram**: Describes all possible states of data
  - **Instance diagram**: A particular set of objects relating to each other. Useful for discussing scenarios, test cases and examples
- During modeling, we use class diagrams for *specification* and instance diagrams for *illustration*

How do you find classes?

Methods (one should apply several):

- Learn about problem domain:
  - Observe, talk to your client
- Apply general world knowledge and intuition
  - Try to establish a taxonomy
- Do a syntactic analysis of problem statements or scenarios: Abbott Textual Analysis (1983), also called noun-verb analysis
  - Nouns are good candidates for classes
  - Verbs are good candidates for operations
  - Adjectives are often candidates for attributes
- Apply design knowledge:
  - Distinguish different types of objects
  - Apply design patterns (→ lecture on design patterns)
  - Identify existing solution classes
  - Often amounts to solution design (not requirements analysis)

Finding participating objects in Use Cases

Freie Universität Berlin

- 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
  - Identify real world entities that the system needs to keep track of
  - Identify real world procedures that the system needs to keep track of
  - Identify data sources and data sinks

  All these are candidates for becoming objects in your model

- Be prepared that some objects are still missing
  - Model the flow of events with a sequence diagram
- Always use the user's terms
  - and be consistent

Object kinds

Freie Universität Berlin

- **Entity Objects**
  - Represent the persistent information tracked by the system ("business objects", "Geschäftsobjekte")
- **Control Objects:**
  - Represent the control tasks performed by the system ("logic")
- **Boundary Objects**
  - Represent the interaction between the user and the system

- Having three kinds of objects leads to models that are more resilient to change.
  - Change frequencies are highest for Boundary Objects and lowest for Entity Objects
- Sometimes known as Model, View, Controller (MVC)
  - but that term is more commonly applied within GUIs only
  - Model ≈ Entity, View ≈ Boundary, Controller ≈ Control

And please remember:

- We are still in the application domain!
  - These are application domain classes (problem domain classes)
  - not solution domain classes (design or code classes)

Example: 2BWatch Objects

Hour

Minute

Second

ChangeTime

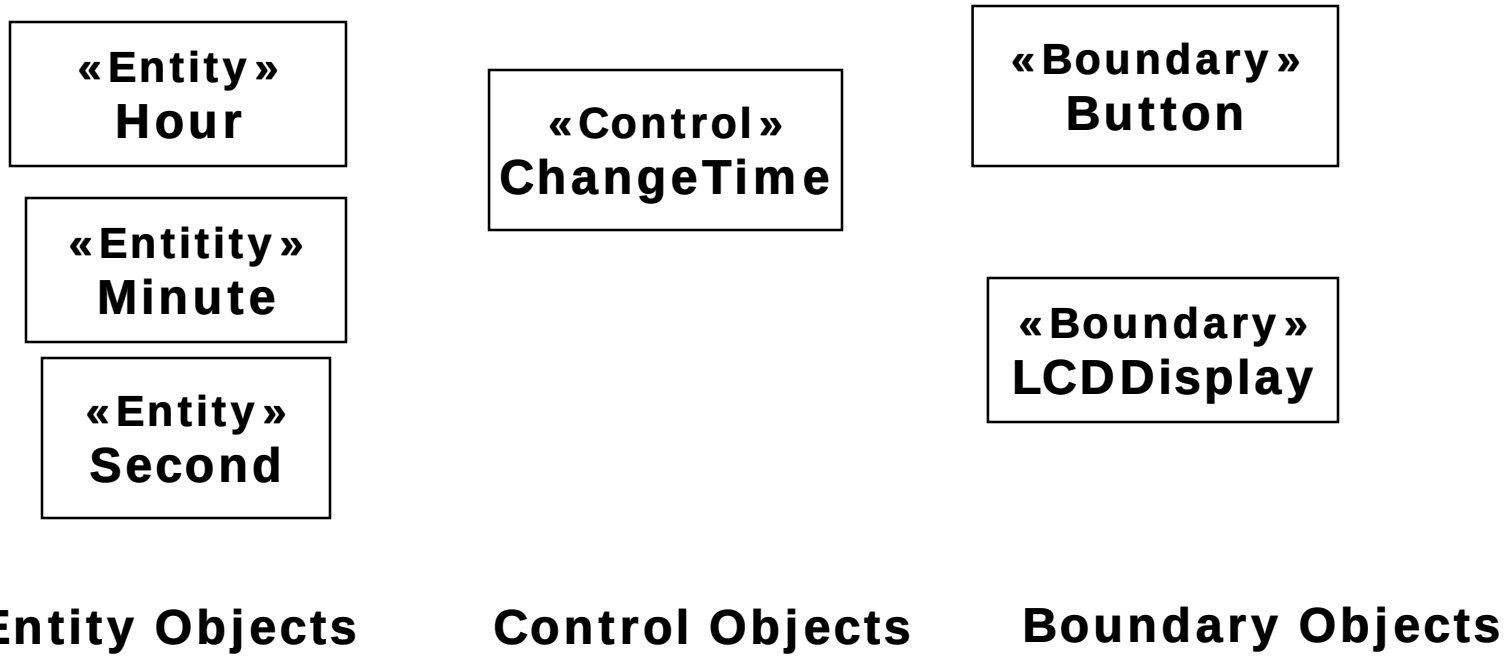
Button

LCDDisplay

Entity Objects **Control Objects** **Interface Objects**

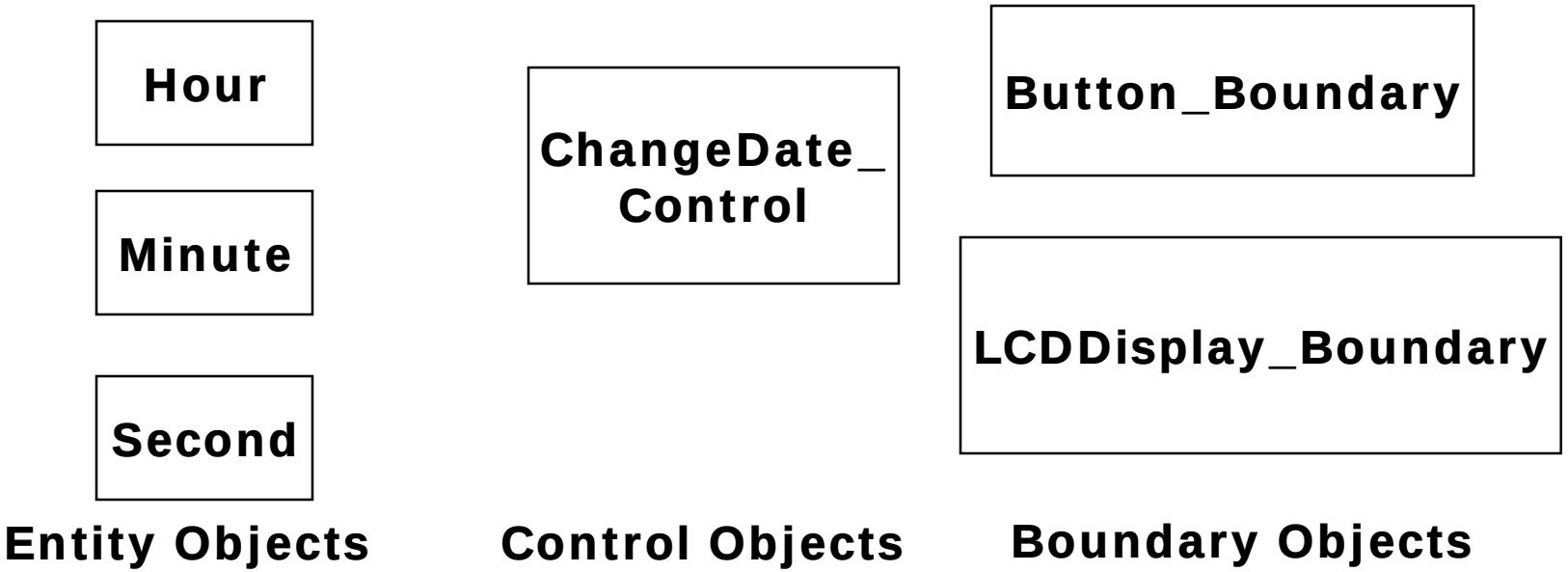
Tagging of object kinds in UML: stereotype

- UML provides several mechanisms to extend the language
  - In particular the *stereotype* mechanism to represent new modeling elements
  - Stereotypes are defined in a UML *profile*
  - An appropriate profile can be used to tag classes with the three kinds



Possible naming convention for object kinds

- To distinguish the different object kinds on a syntactical basis, one may use name suffixes:
  - Object names ending with "Boundary" suffix represent boundary objects
  - Objects names ending with "Control" suffix represent control objects
  - Objects names ending without either suffix represent entity objects



Abbott's textual analysis technique

Freie Universität Berlin

Flow of events:

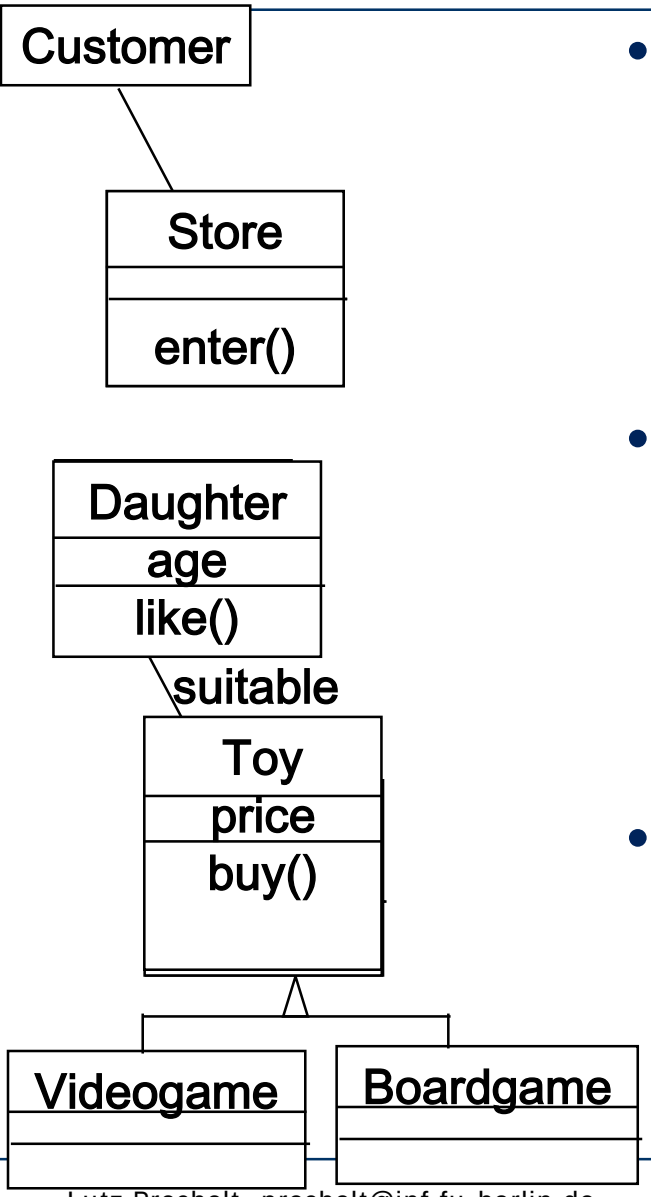
- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display.
  He likes it.
- An assistant helps him. The suitability of the game depends on the age of the child.
- His daughter is only 3 years old.
- The assistant recommends another type of toy, the boardgame "Monopoly".

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

Freie Universität Berlin

Example	***Grammatical construct***	***(perhaps) 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



- 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 buys the game and leaves the store.

[There is more information left to be analyzed in this narrative.]

Order of activities in object modeling

Freie Universität Berlin

- Formulate scenarios
  - with help from the end user and/or application domain expert
- Extract the use cases from the scenarios
  - with the help of application domain expert
- Analyze the flow of events
  - for example with Abbott's textual analysis
- Generate the class diagrams.
  This includes the following steps:
  - Class identification (textual analysis, domain experts).
  - Identification of attributes and operations
    - sometimes before the classes are found!
  - Identification of associations between classes
  - Identification of multiplicities
  - Identification of roles
  - Identification of constraints

Checklist for identifying **classes**

- Identify a category for the class
  - Concrete things (e.g. Toy)
  - Persons and their roles (e.g. Customer)
  - Information about actions (e.g. Receipt)
  - Places (e.g. Shop)
  - Organizations (e.g. Company)
  - Containers (e.g. Shelf)
  - Events (e.g. Payment)
  - Contracts (e.g. Purchase)
- Find a suitable name
  - A user term
  - Noun, singular
  - Not confusable with some other class name
- Check abstraction level
  - Avoid classes that are too fine-grained or too simple
- Is this really an application domain class?
  - Or is it a solution domain class e.g. for technically managing a set of objects?

Checklist for identifying **associations**

Freie Universität Berlin

- Start with a simple line only
- Check for association type:
  - A is a physical part of B
  - A is a logical part of B
  - A is a description of B
  - A uses B
  - A owns B
  - other
- Check for restrictions:
  - Is it {ordered} ?
  - If there are several associations: {xor}?, {subset}?
- Check for roles of classes:
  - Name roles or name the association if this adds clarity
  - In particular if there are multiple associations at a class
  - Always name reflexive assocs
  - Role names are nouns
  - If assoc. names are nouns, they refer to abstractions
    - e.g. *authorship*, not *author*
- Check 1:1 associations
  - If the association is mandatory, should the classes be united?
- Check for multiple associations between the same classes
  - Are they really different?
    - Probably yes if they have different multiplicities
    - Often no if they do not

Checklist for identifying **attributes**

- Check abstraction level
  - Use elementary types only where appropriate
  - Complex attributes should become classes, not multiple elementary attributes
  - Implementation details (e.g. for realizing an association, for optimizations) should not be modeled!
- Check location:
  - If the class had no associations, would this attribute still be required?
    - Yes: OK
    - No: It may be an attribute of an association. Think about forming an association class.
- Find a suitable name:
  - Noun or adjective+noun
  - Do not repeat name of class
  - Avoid abbreviations (unless well-known in the domain)
- Is it a class attribute?
  - Should the value always be the same for all instances?

Checklist for **inheritance**

- Is it natural?
  - During analysis, inheritance should describe a type taxonomy present in the *problem domain*
- Is it redundant?
  - It is if two subclasses need the same set of attributes and operations
- Is it misaligned?
  - It is if some subclasses inherit operations that make no sense for them
    - **Very dangerous!**

- Note:
  Inheritance in the analysis model needs not always be implemented as inheritance in the design model or in the final program.

Checklist for identifying **operations**

Freie Universität Berlin

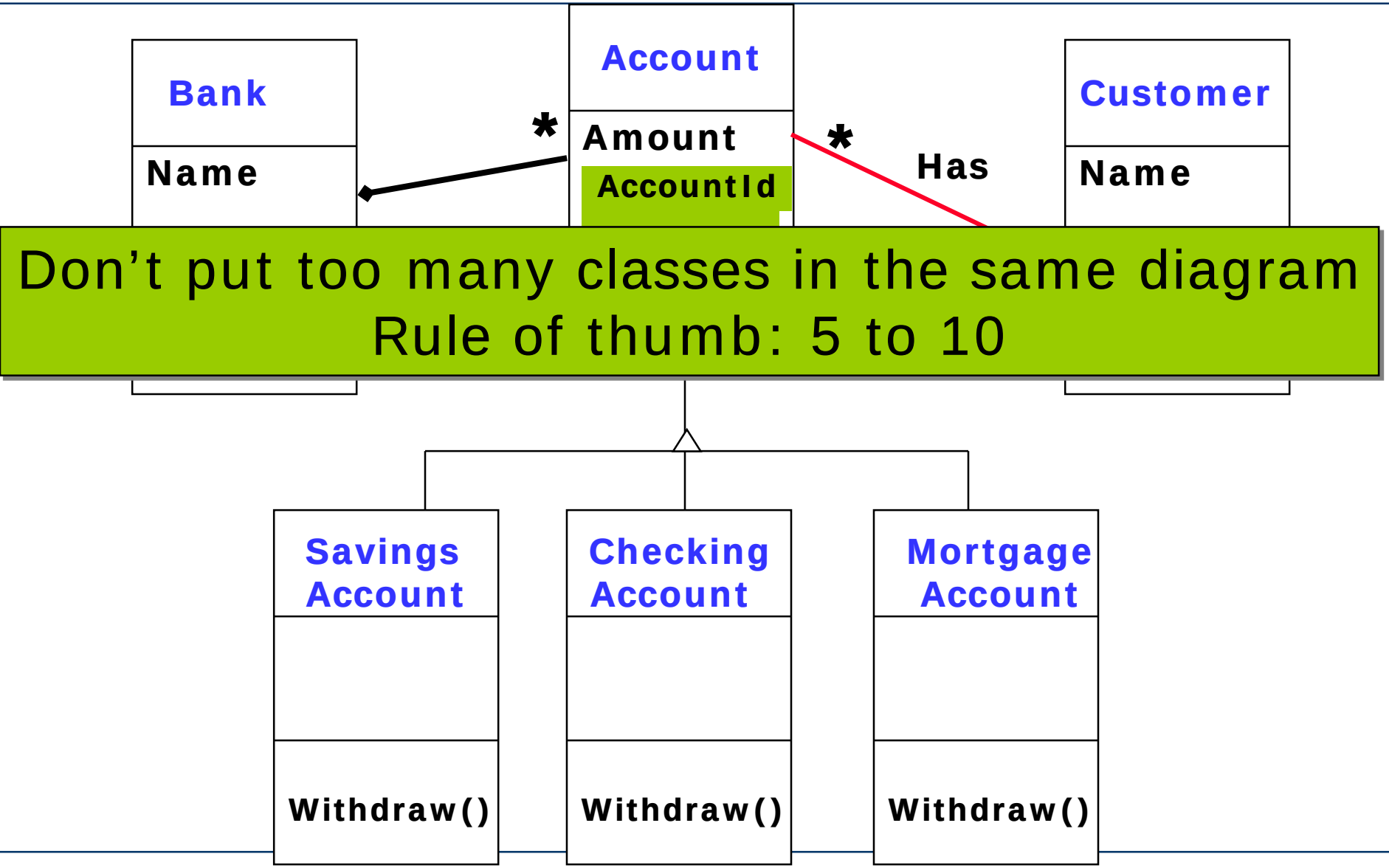
- Is this the right class?
  - In an inheritance hierarchy, move operations as far up as makes sense
- Find a suitable name:
  - For procedures: describes the effect of the operation
    - Starts with imperative verb
  - For functions: describes the result returned
    - A noun
- Check granularity:
  - Does the operation serve some purpose completely?
    - If no, join it with others
  - Does it serve more than one?
    - If yes, split it in several
- Check class cohesion:
  - Are there attributes that are not used by any operation?
    - If yes, an operation is missing
- Does it have too many parameters?
  - If yes, you may need to introduce auxiliary classes to group some of them together

Practical hints

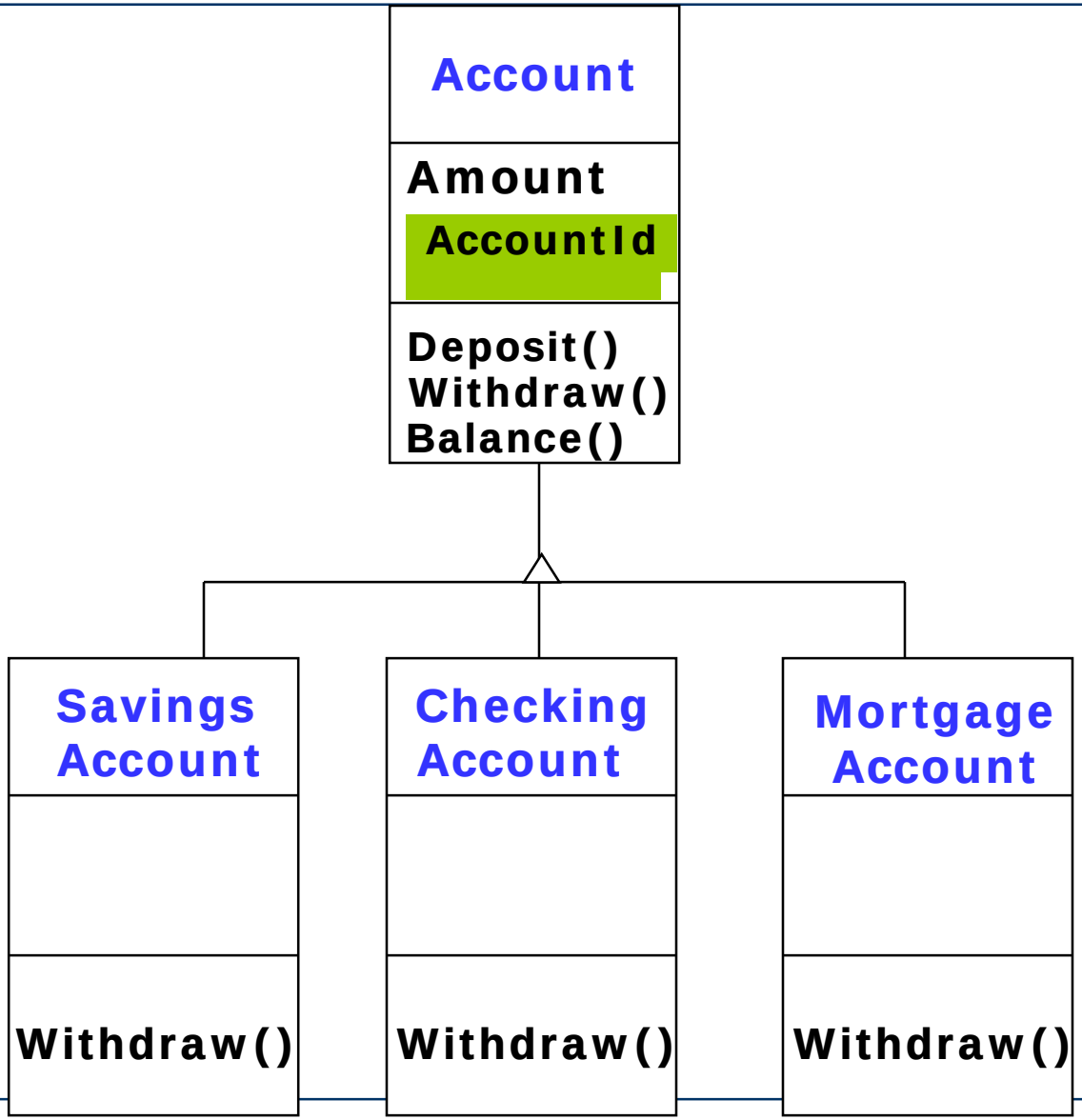
The next few slides will give some heuristics regarding:

- The readability of class diagrams
  - DOs and DON'Ts
- Managing object modeling
  - how to approach the process
- The different users of class diagrams
  - different types and needs of users mean you should have different types of diagrams as well.

Avoid Ravioli Models



One rule of thumb: put taxonomies on a separate diagram



Project management heuristics

Explicitly schedule meetings for object identification

1. First just find objects
2. Then differentiate them between entity, boundary, and control objects
3. Find associations and their multiplicity
   - Start from instance diagrams of concrete situations
   - Unusual multiplicities often lead to new objects or categories
4. Identify inheritance: Categorize and look for a taxonomy
5. Identify aggregation
6. Identify important methods and attributes

- Allow time for brainstorming at each stage
- Iterate, iterate, iterate

Who uses class diagrams?

- Customers and end users are rarely interested
  - They usually focus more on the behavior of the system
  - (➔ Use Cases)
- Application domain experts use class diagrams to model the application domain
  - ➔ Analysis model
- Developers use class diagrams during analysis, system design, object design, and implementation.
  - ➔ Design models
  - Design models extend and modify the analysis model
  - Never assume your analysis model is a design model!

Class diagrams have different types of users

Freie Universität Berlin

- Developers play different roles
  - (Often one person fills more than one role)
  - Analyst
  - System-level designer
  - Detailed-level designer
  - Implementor
- Each of these roles has a different view of the models
- To understand these views, we need to distinguish between
  - application domain classes and
  - solution domain classes

Application domain vs. solution domain

Freie Universität Berlin

- **Application domain (problem domain):**
  - The "home" domain of the problem to be solved
  - Examples: financial services, meteorology, the health system
- Application domain class (analysis & design models):
  - An abstraction in the application domain
    - In business applications often called business objects
  - Example: Contract, Customer, Order

- **Solution domain:**
  - Technical domains that help in constructing software
  - Examples: telecommunication, databases, compiler construction, operating systems
- Solution domain class (in design models only!):
  - An abstraction that is introduced for technical reasons
    - not directly due to application domain requirements
  - Examples: OrderQueue, DatabaseConnection, Scheduler

Analyst view

- The analyst is interested
  - in application domain classes: The associations between classes are relationships between abstractions in the application domain
  - whether the use of inheritance in the model reflect the taxonomies in the application domain
    - A taxonomy is a hierarchy of abstractions
- The analyst is not interested
  - in the exact signature of operations
  - in solution domain classes

Designer view

- Designers focus on the solution of the problem
  - that is, the solution domain
- Designers consider application domain classes as largely given (and not to be meddled with)
  - in particular the Entity objects
  - to a lesser degree the Boundary and Control objects
- and search for appropriate solution domain classes
  - such that the overall system can be built

- The central design problem is the specification of appropriate interfaces
  - First of subsystems (architectural design), then of classes,
  - such that all functional and non-functional requirements can be fulfilled
  - and that the design is easy to implement, test, understand, and modify

Three types of implementor views

- Class implementor:
  - Implements the class
  - Chooses appropriate data types (for the attributes) and
  - algorithms (for the operations), and
  - realizes the interface of the class in a programming language

- Class extender:
  - Extends the class by a subclass which is needed for a new problem or a new application domain

- Class-user (client):
  - Wants to use an existing class (e.g. a class from a class library or from another subsystem)
  - Is only interested in the interface of the class (signatures, preconditions, postconditions)
  - Should not need to be interested in the class implementation

Model interpretation and roles

- Depending on our role (analyst, designer, implementor), we may be interested in limited aspects of a model only
  - Separate models reduce confusion and information overflow

- A helpful habit for creating high quality software systems is the exact distinction of
  1. Application domain classes versus solution domain classes
  2. Interface specification versus implementation specification

- Depending on our role and the model we have different interpretations for certain UML constructs:
  - Different interpretations of associations
  - Different interpretations of attributes
  - Different interpretation of inheritance

- Let us look at these different interpretations:

Interpretations in analysis vs. design model

- Different interpretations of associations
  - Analysis model: Relationships between objects in reality
  - Design model: Reachability of instances
- Different interpretations of attributes
  - Analysis model: Characteristics of object instances
  - Design model: State storage, basis for decisions/control flow
- Sometimes different interpretation of inheritance
  - Analysis model:
    Type taxonomy; objects that can take the role of a superclass object
  - Design model:
    [ditto] or type extension or superclass code reuse
    - but beware: Reuse-without-taxonomy often creates huge problems.

Very important!

Summary

- The analysis object model reflects concepts from
  - the application domain and
  - the requirements
- It can be found by systematic analysis of use cases
  - plus other techniques
- The subsequent design model is usually quite different!
  - It often leaves out a number of application domain classes
    - because they are not relevant for the technical system
  - It usually contains many additional solution domain classes

Thank you!