21) Functional and Modular Design

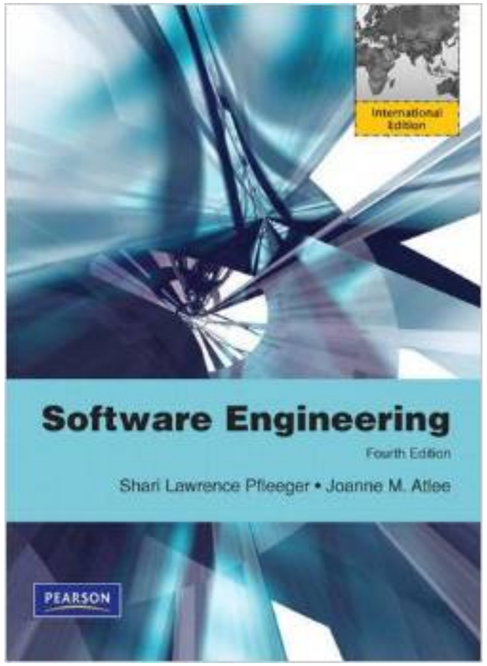
- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediatechnik
- http://st.inf.tu-dresden.de/teaching/swt2

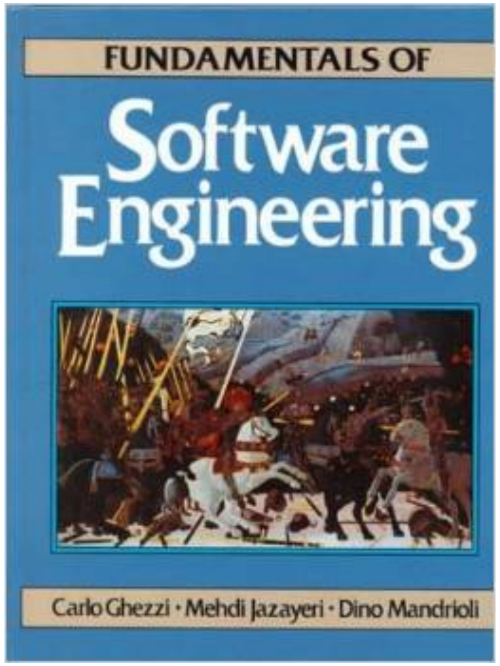
- Version 16-0.1 17.12.16

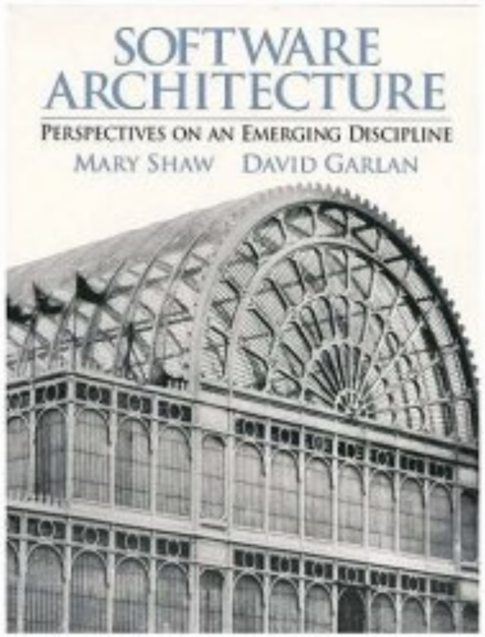
1. Functional Design
2. Modular Design (Change-Oriented Design)
3. Use-Case Based Design

Obligatory Readings

- S. L. Pfleeger and J. Atlee:
  Software Engineering: Theory and Practice.
  Pearson. 2009.
  - Chapter 5 (Designing the Architecture)
- C. Ghezzi, M. Jazayeri and D. Mandrioli:
  Fundamentals of Software Engineering.
  Prentice Hall. 1992.
  - Chapter 4 (Design and Software Architecture)
- M. Shaw and D. Garlan:
  Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.







Literature

- [Parnas] David Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM Dec. 1972 (15) 12.

21.1 FUNCTIONAL DESIGN

Function-Oriented Methods

- Examples:
  - **Stepwise function refinement** resulting in function trees
  - Modular decomposition with information hiding (Change-oriented modularization, Parnas)
  - Meyers Design-by-contract: Contracts are specified for functions with pre- and postconditions
    - (see OCL lecture)
  - Dijkstra’s and Bauer’s axiomatic refinement (not discussed here)

> **Which functionality will the system have?**
>
> **What are the subfunctions of a function?**

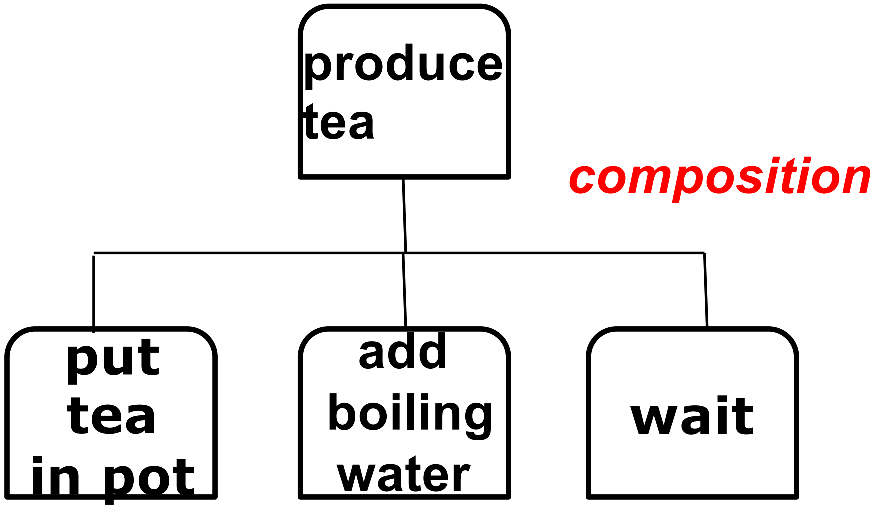
A Start for a Function Tree

- How to design the control software for a tea automaton?

```
Produce Tea
```

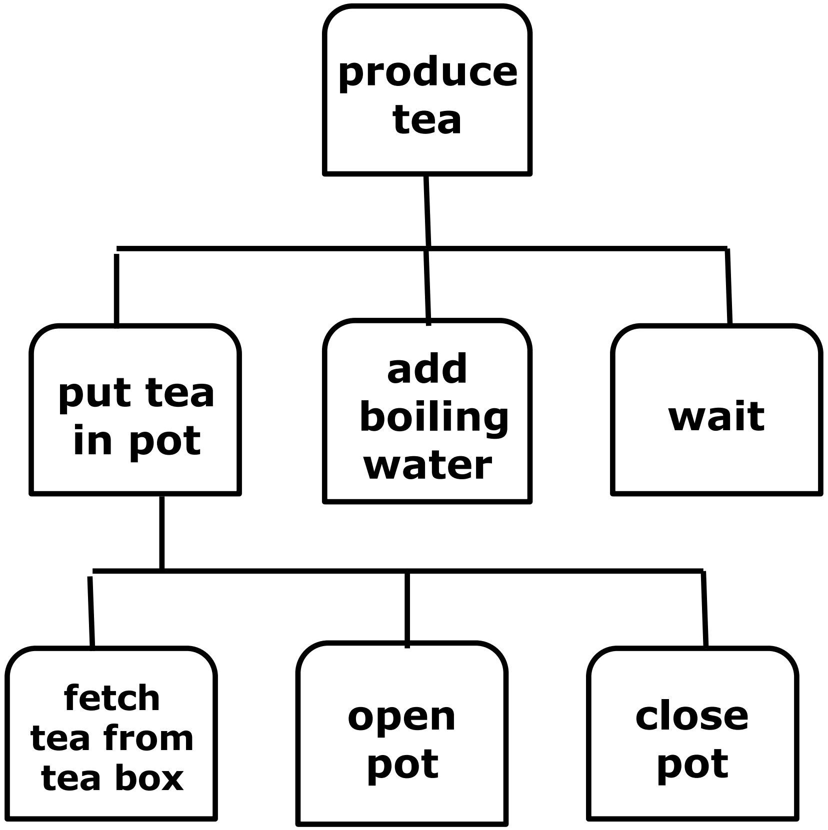
First Refinement of a Function Tree


```
Produce Tea
.. is composed of ..
Put tea in pot
Add boiling water
Wait
```



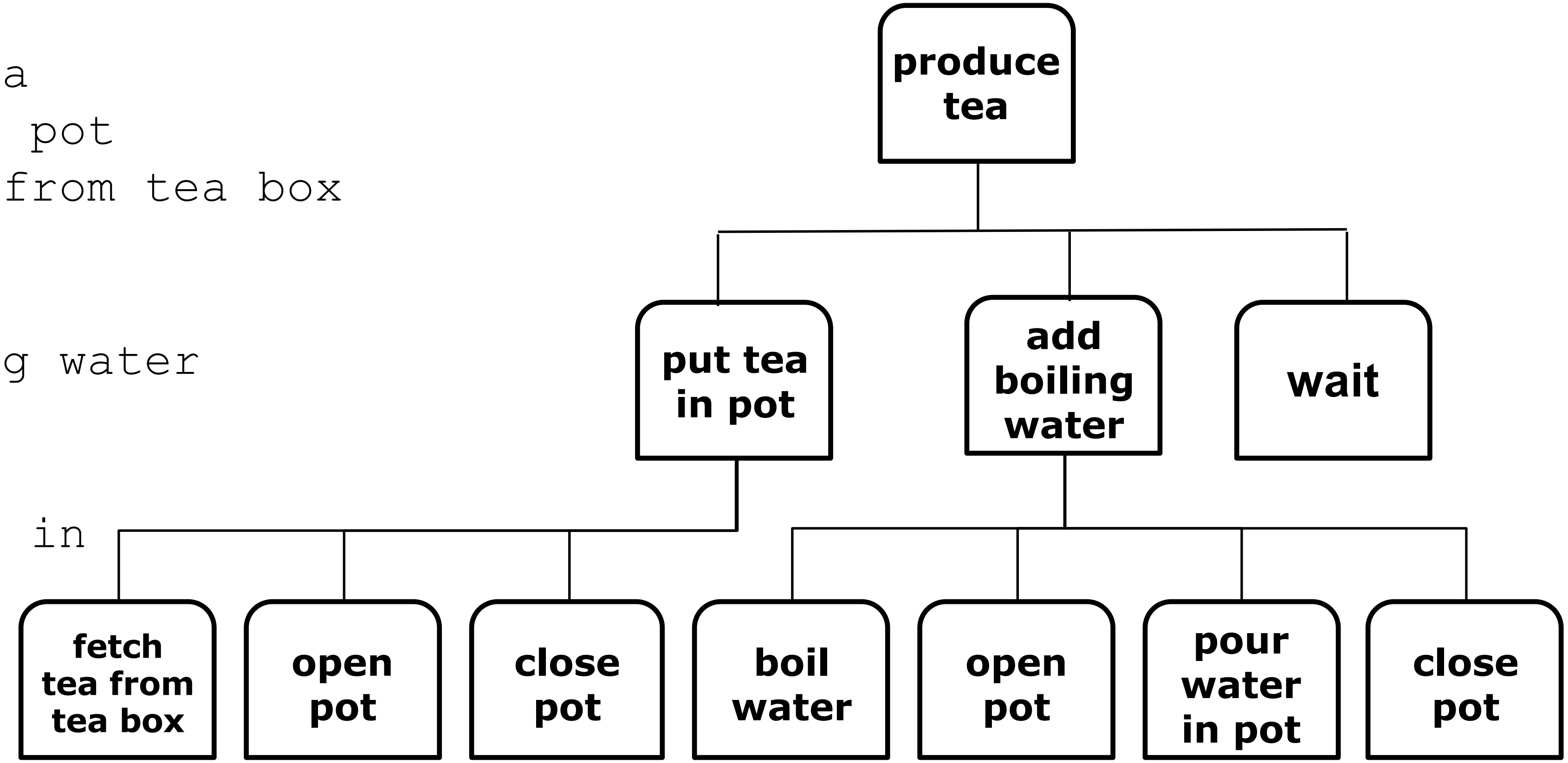
Second Refinement of a Function Tree


```
Produce Tea
Put tea in pot
Fetch tea from tea box
Open pot
Close pot
Add boiling water
Wait
```



Third Refinement of a Function Tree


```
Produce Tea
Put tea in pot
Fetch tea from tea box
Open pot
Close pot
Add boiling water
Boil water
Open pot
Pour water in
Close pot
Wait
```

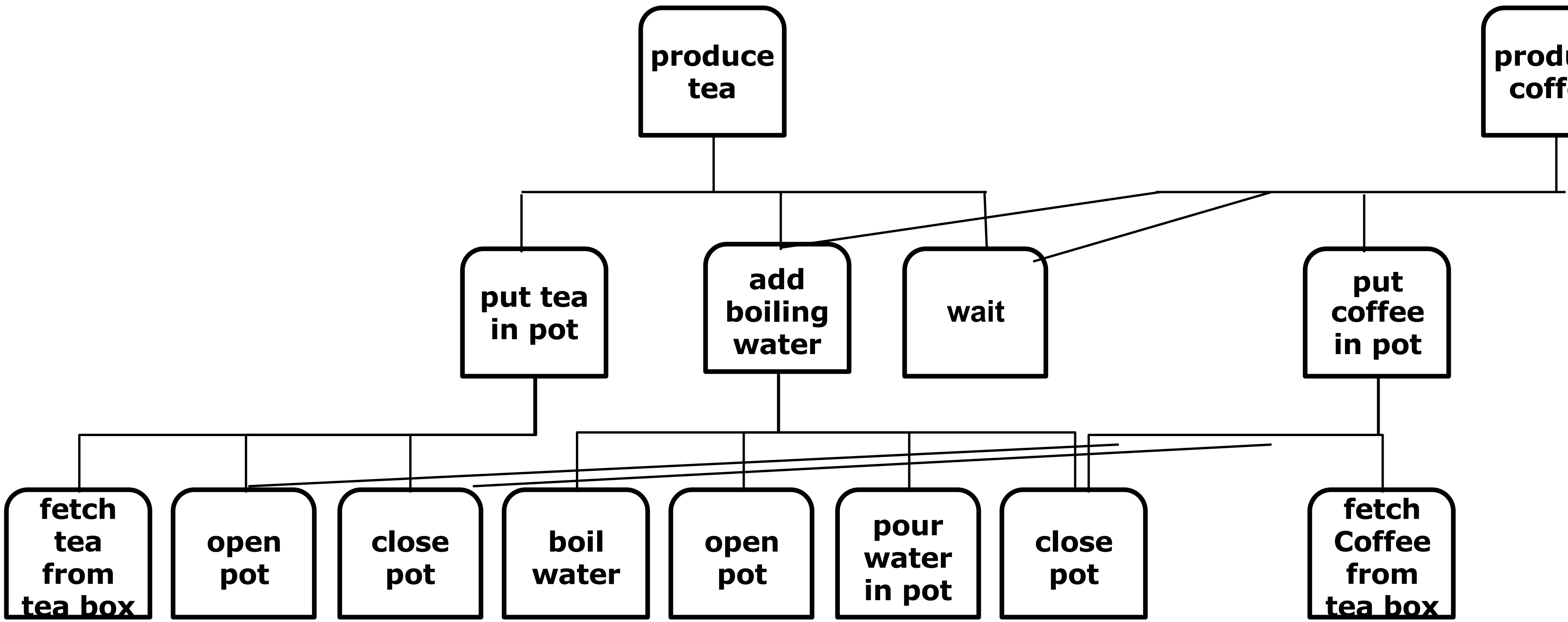


Function Trees

- Function trees can also be derived by a 1:1 mapping from a functional requirements tree (see ZOPP requirements lecture)
- Usually, for a system several function trees are develop, starting with top-level functions in the *context model*
- **Stepwise Refinement** works usually top-down (Hierarchic decomposition)
  - Bottom-up strategy (composition) possible
  - Middle-out strategy blends composition and decomposition
  - Development of the "subfunction-of" ("call") relationship: a part-of relationship for functions: the function has which parts (subfunctions)?
  - Usually implemented by call relationship (call graph)
- **Syntactic stepwise refinement** is indifferent about the semantics of the refined model
- **Semantic stepwise refinement** proves that the semantics of the program or model is unchanged
  - Systems developed by semantic refinement are **correct by construction**
- Functions are **actions**, if they work on *visible* state
  - In functional design, state is disregarded
  - State is important in action-oriented design, actions are usually related to state transitions!

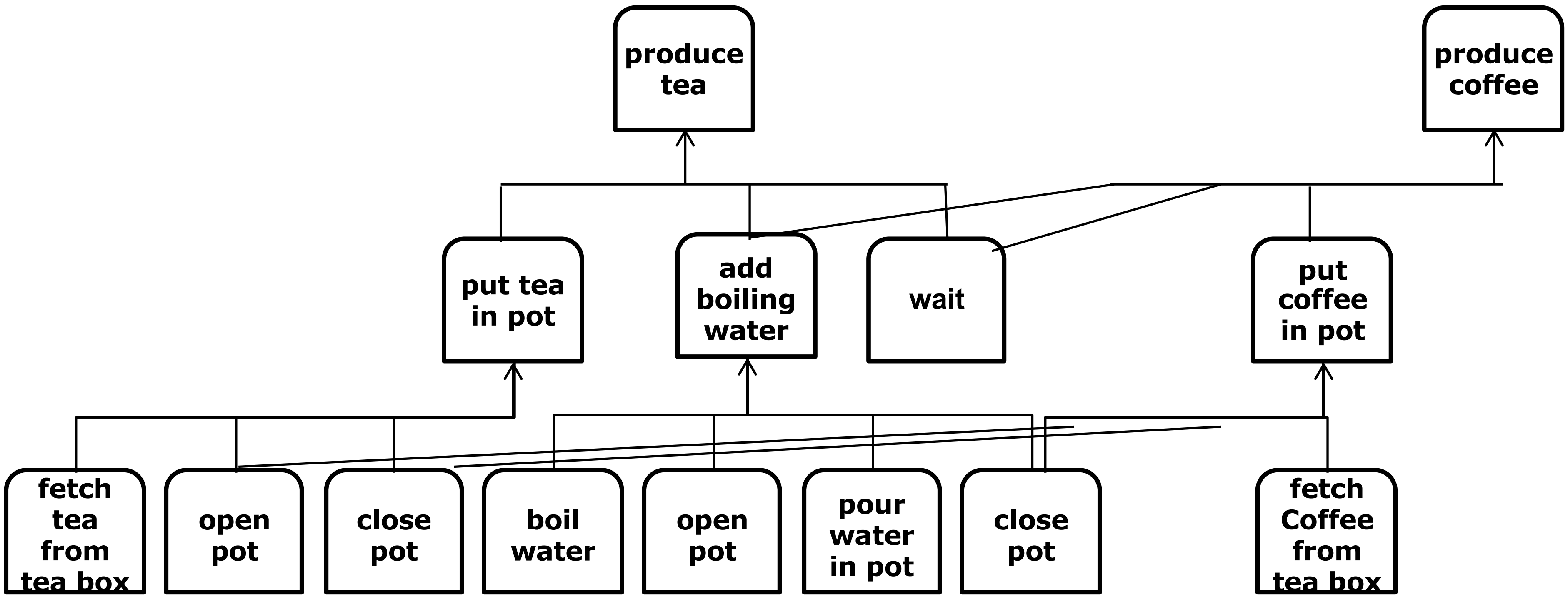
Function Polyhierarchies

- If subfunctions are shared, **polyhierarchies** result with several roots and shared subtrees



Function Nets

➢ Functions are arranged in a directed acyclic graph



Other Trees with Other Part-Of Relationships

- Many concepts can be stepwise refined and decomposed. Hierarchic decomposition is one of the most important development methods in Software Enineering:
- Problem trees
- Goal trees
- Acceptance test trees
- Requirements trees
  - Function trees
  - Feature trees (function trees describing grouping, variability and extensibility)
- Attack trees
- Fault trees
- ….
- The development is always by **divide and conquer.**
- Think about: Which part-of relationships do they develop?

21.1.2 MODULAR COMPOSITION: GROUPING MODULES AND COHESION

Grouping Functions to Modules to Support Cohesion

- Group functions according to cohesion: "which function belongs to which other function?"
- Minimize coupling of modules
- Maximize coherence: encapsulate dependencies within a module

```
Module Tea Automaton {
  Produce Tea

  Add boiling water
  Wait
}
```

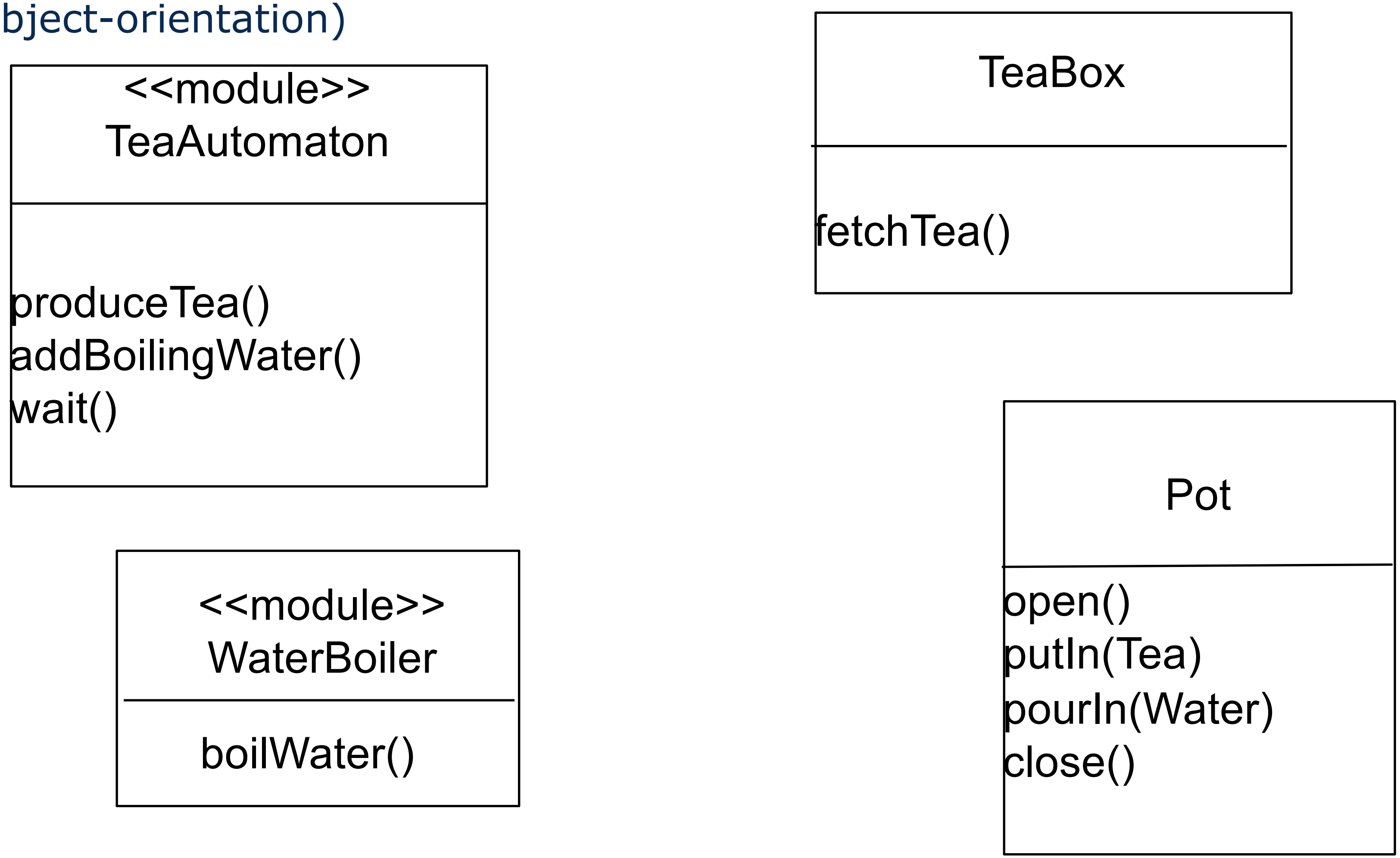
```
Module Tea Box {
  Fetch tea from tea box
}
```

```
Module Water Boiler {
 Boil water
}
```

```
Module Pot {
  Open pot
  Put tea in pot
  Pour water in pot
  Close pot
}
```

Grouping Functions to Modules *or* Classes in UML

- Functions can often be grouped to objects (object-oriented encapsulation)
- Then, they can be actions working on the state of the *object* (begin of object-orientation)

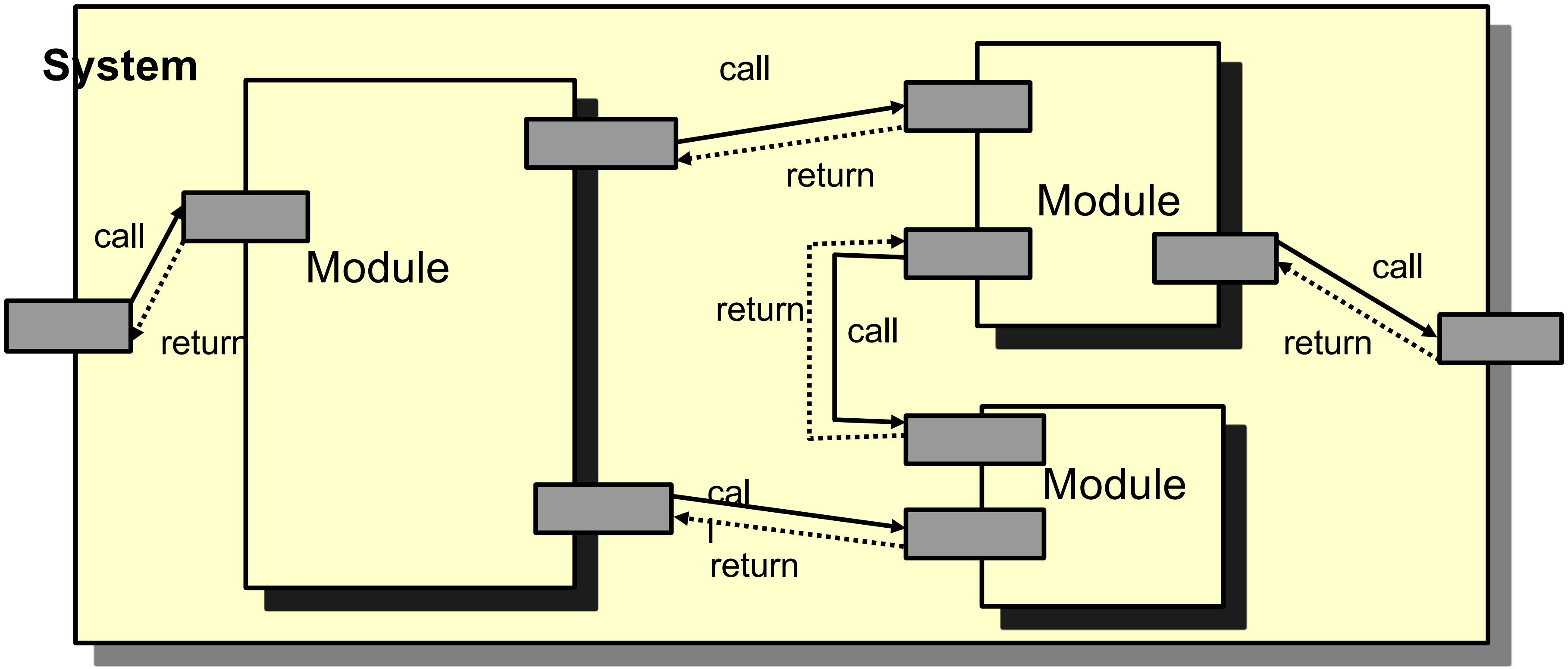


Heuristics and Best Practices

- Don't group too many items onto one abstraction level or into one module (**slim interface** principle)
- Technical modules or classes (classes that do not stem from domain modeling) can be found in similar ways, by grouping cohesive functions together
- Identify **material** modules or classes with CRUD interfaces (see TeaBox and Pot):
  - Create
  - Read
  - Update
  - Delete
- Identify **tool** modules or classes with "active functions":
  - List<Material>
  - Edit<Material>
  - Navigate<Material>
- Identify **command** modules or classes (Design Pattern Command)
  - Tools are specific commands, working on materials

Result: Call-Based Architectural Style

- Functional design leads to **call-based architectural style** with statically known callees (static call graph)



Prof. U. Aßmann

Grouping Other Trees with other Part-Of Relationships

- Any hierarchic relationship can be grouped to modules based on cohesion
- Problem trees ➔ problem modules
- Goal trees ➔ goal modules
- Acceptance test trees ➔ acceptance test modules
- Feature trees (describing variability, extensibility) ➔ Feature modules
- Attack trees ➔ attack modules
- Fault trees ➔ fault modules
- ....

Why is Function-Oriented Design Important?

- Implementation of function trees in a functional language
  - ... or a modular imperative language, e.g., Modula, C, or Ada-83.

In some application areas, object-oriented design and languages have severe disadvantages:

- Employment in safety-critical systems:
  - Proofs about the behavior of a system are only possible if the architecture and the call graph are *static.* Then they can be used for proofs
  - Due to polymorphism, object-oriented systems have dynamic architectures (don't program your AKW with Java!)
- In embedded and real-time systems:
  - Object-oriented language implementations usually are slower than those of modular languages
  - ... and eat up more memory
- In high-speed systems:
  - Operating systems, database systems, compilers, ...

(Rep. from ST-1, left out here)

21.2 CHANGE-ORIENTED MODULARIZATION WITH INFORMATION HIDING (VARIABILITY)

What is a Module?

- Software should, according to the divide-and-conquer principle, also physically be divided into basic parts, ***modules***
  - A module groups a *set of functions or actions*
  - A module can be developed independently
    - errors can be traced down to modules
    - modules can be tested before assembling
  - A module can be exchanged independently
  - A module can be reused
- The terms *module* and *component* mean pretty much the same
  - Often, a module is a programming-language supported component
  - Here: a module is a simple component
  - In the past, different component models have been developed
  - A component model defines features of components, their compositionality, and how large systems are built with them (architecture)
  - In course "Component-based SE", we will learn about many different component models

How To Modularize a System?

- Parnas principle of *change-oriented modularization (information hiding)* [Parnas, CACM 1972]:

- 1) Fix all design decisions that are likely to change
- 2) Attach each of those decisions to a new module
  - The design decision becomes the secret of a module (called *module secret)*
- 3) Design module interface that does not change if module secret changes

Information Hiding

- *Information hiding relies on module secrets*
- Possible module secrets:
  - How the algorithm works, in contrast to what it delivers
  - Data formats
  - Representation of data structures, states
  - User interfaces (e.g., AWT)
  - Texts (language e.g., gettext library)
  - Ordering of processing (e.g., design patterns Strategy, Visitor)
  - Location of computation in a distributed system
  - Implementation language of a module
  - Persistence of the data

Module Interfaces

- Should never change!
  - Well, at least be *stable*
- Should consist only of functions
  - State should be invisible behind interfaces
  - Direct access to data is efficient, but cannot easily be exchanged
    - e.g., emply set/get methods for accessing fields of objects
- Should specify what is
  - Provided (exported)
  - Required (imported)

Different Kinds of Modules

- Functional modules (without state)
  - sin, cos, BCD arithmetic, gnu mp,...
- Data encapsulators
  - Hide data and state by functions (symbol table in a compiler)
  - Monitors in the parallel case
- Abstract Data Types
  - Data is manipulated lists, trees, stacks, ..
  - New objects of the data type can be created dynamically
- Singletons
  - Modules with a singular instance of a data structure
- Data-flow processes (stream processors, filters)
  - Eating and feeding pipelines
- Objects
  - Modules that can be instantiated

What Have We Learned?

- When designing with functions, use function trees and subfunction decomposition
- When grouping to modules, fix module secrets
- The more module secrets, the better the exchange and the reuseability
  - Change-oriented design means to encapsulate module secrets
- Functional and modular design are still very important in areas with hard requirements (safety, speed, low memory)

Conclusion of Information-Hiding Based Design

We have seen how important it is to focus on describing *secrets* rather than interfaces or roles of modules.

When we have forgotten that, we have ended up with modules without clear responsibilities and eventually had to revise our design.

[Parnas/Clements, The Modular Structure of Complex Systems, CACM]

Conclusion of Information-Hiding Based Design (2)

- Product lines (product families) are a major business model for software companies.

Modularity is the basis of all product lines.

(repetition from ST-1)

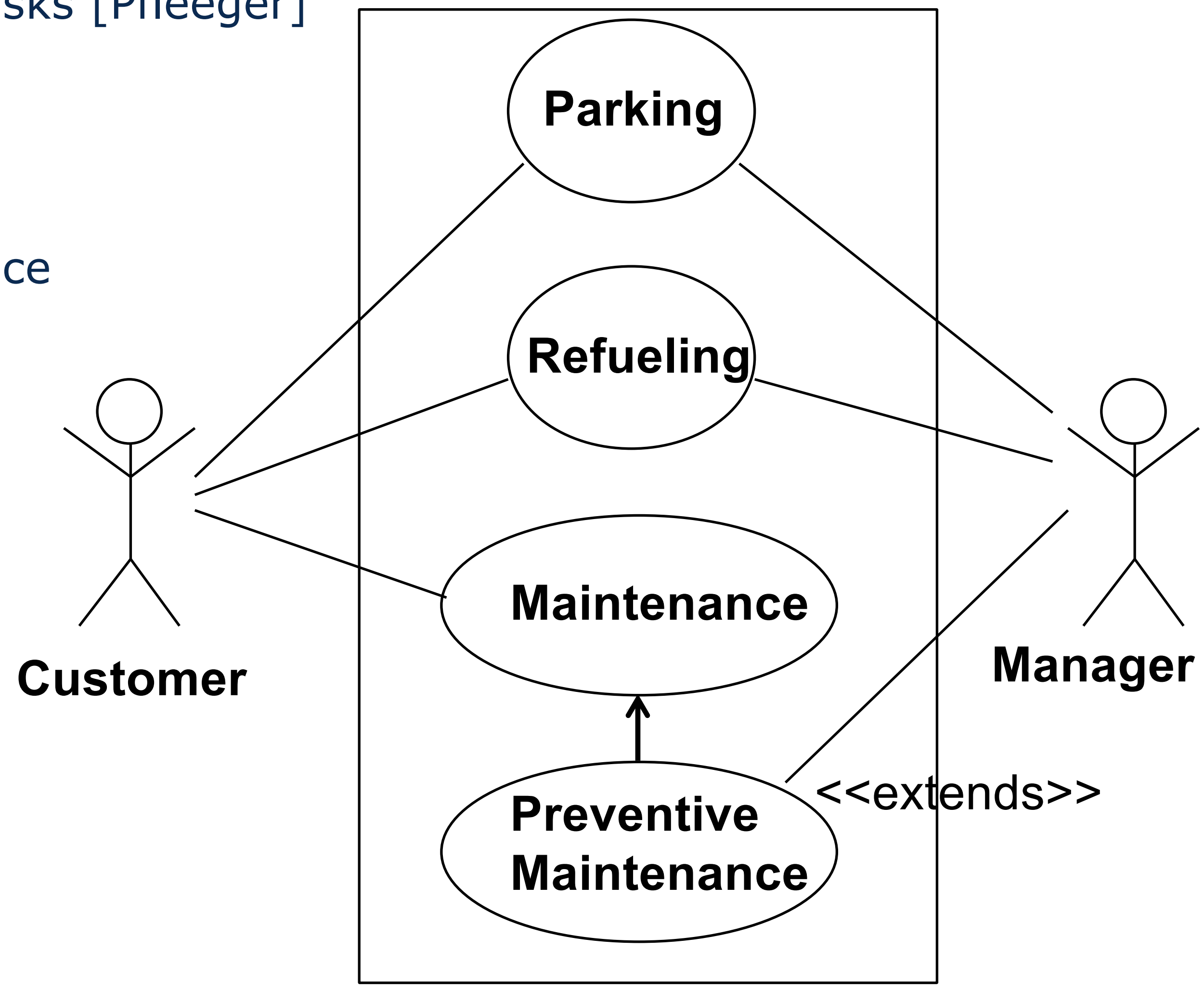
21.3 FUNCTION-ORIENTED DESIGN WITH USE-CASE DIAGRAMS

Use Case Diagrams

- Use Case Diagram (UCD) can be used in functional design
  - A Use Case Diagram consists of several use cases of a system
  - A use case describes an application, a coarse-grain function or action of a system, in a certain relation with actors
  - A use case contains a scenario sketch
    - Pseudocode text which describes the functionality
  - Use Case diagrams can be used in Function-Oriented, Action-Oriented, or in Object-Oriented Design
- From UCD, a function tree can be derived

Example Service Station

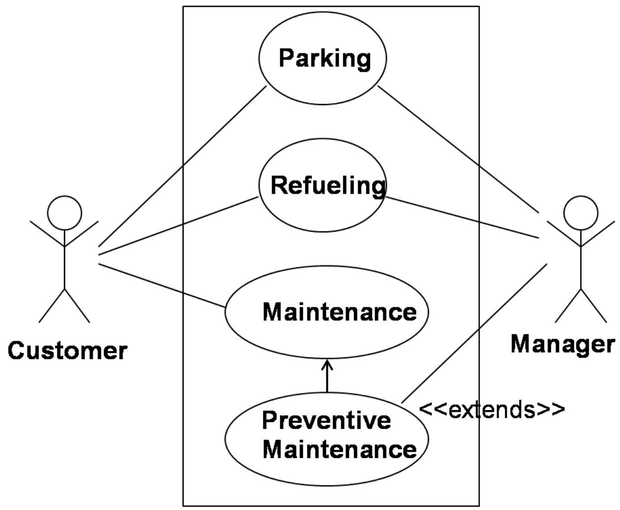
- A Service Station has 4 tasks [Pfleeger]
  - Parking
  - Refueling
  - Maintenance
  - Preventive Maintainance



Questions for Use Cases

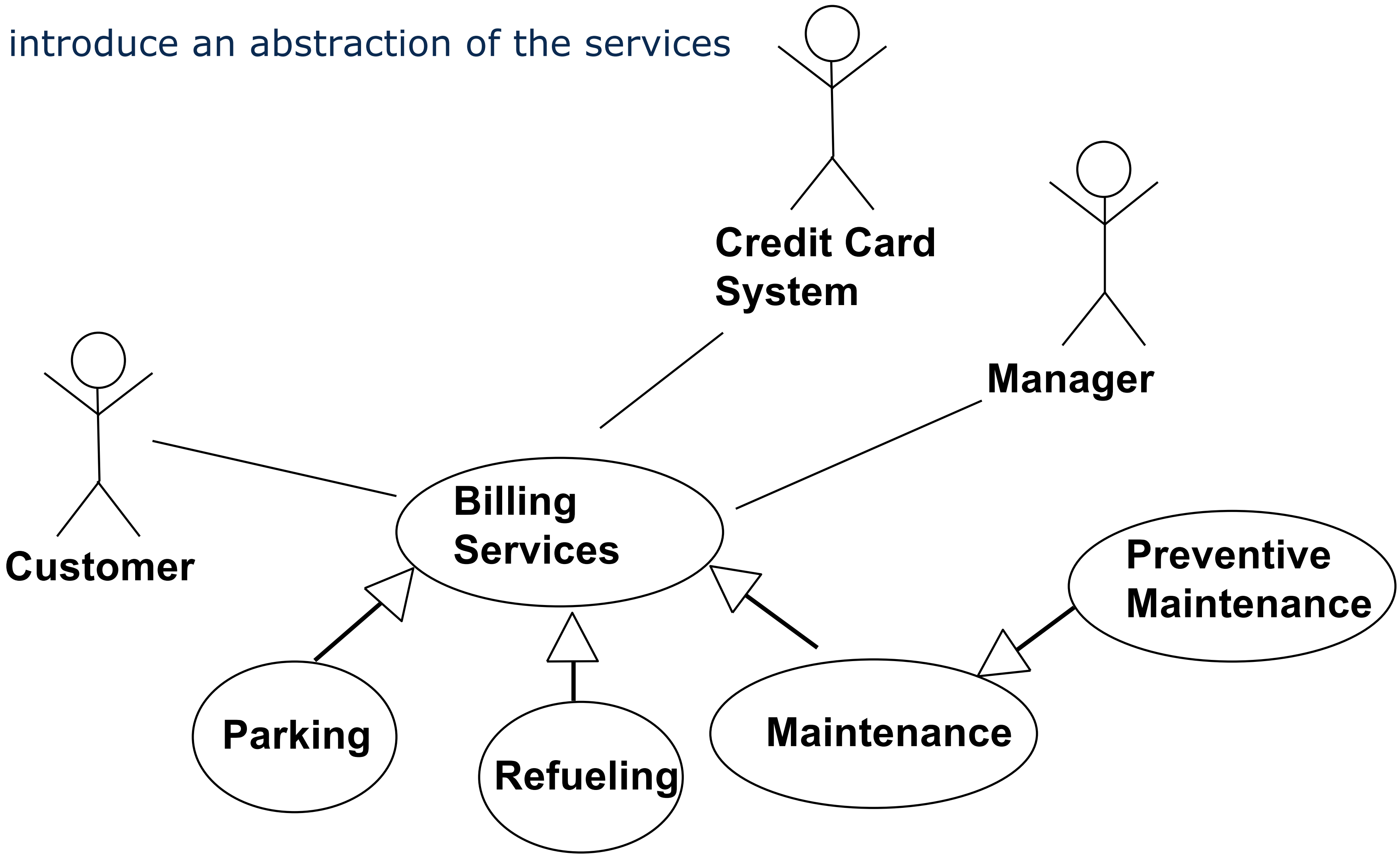
- What is the system/subsystem?
- Who is Actor?
  - A user
    - An active object
    - A person
    - A system
  - Must be external to the described system
- What are the Applications/Uses?
- What are the relations among Use Cases
  - Extends: Extend an existing use case (Inheritance)
  - Uses: Reuse of an existing use case (Sharing)

- Which
  - Users
  - External systems
  - Use
  - Need
- The system for which tasks?
- Are tasks or relations to complex?

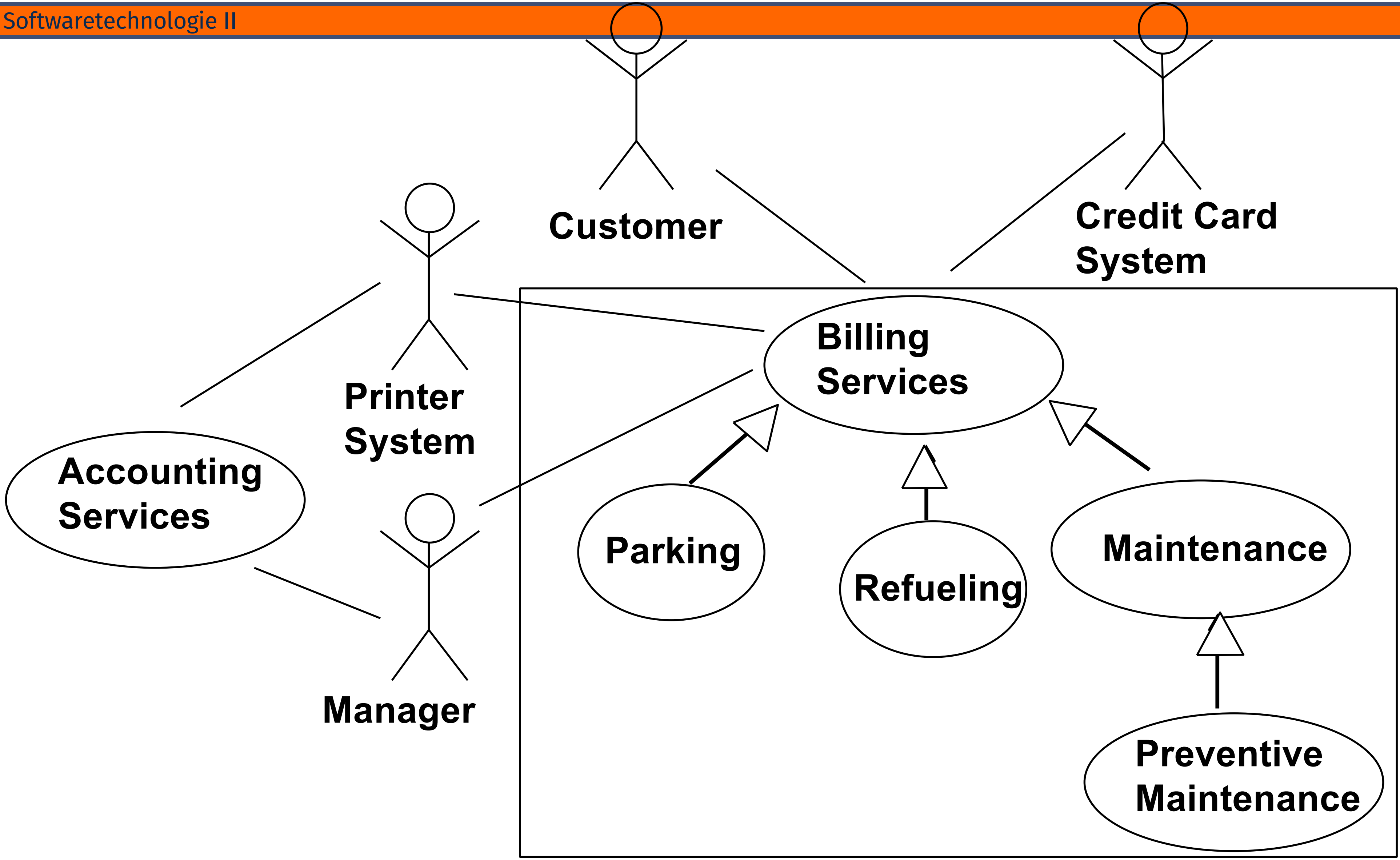


Refinement Service Station

- We introduce an abstraction of the services

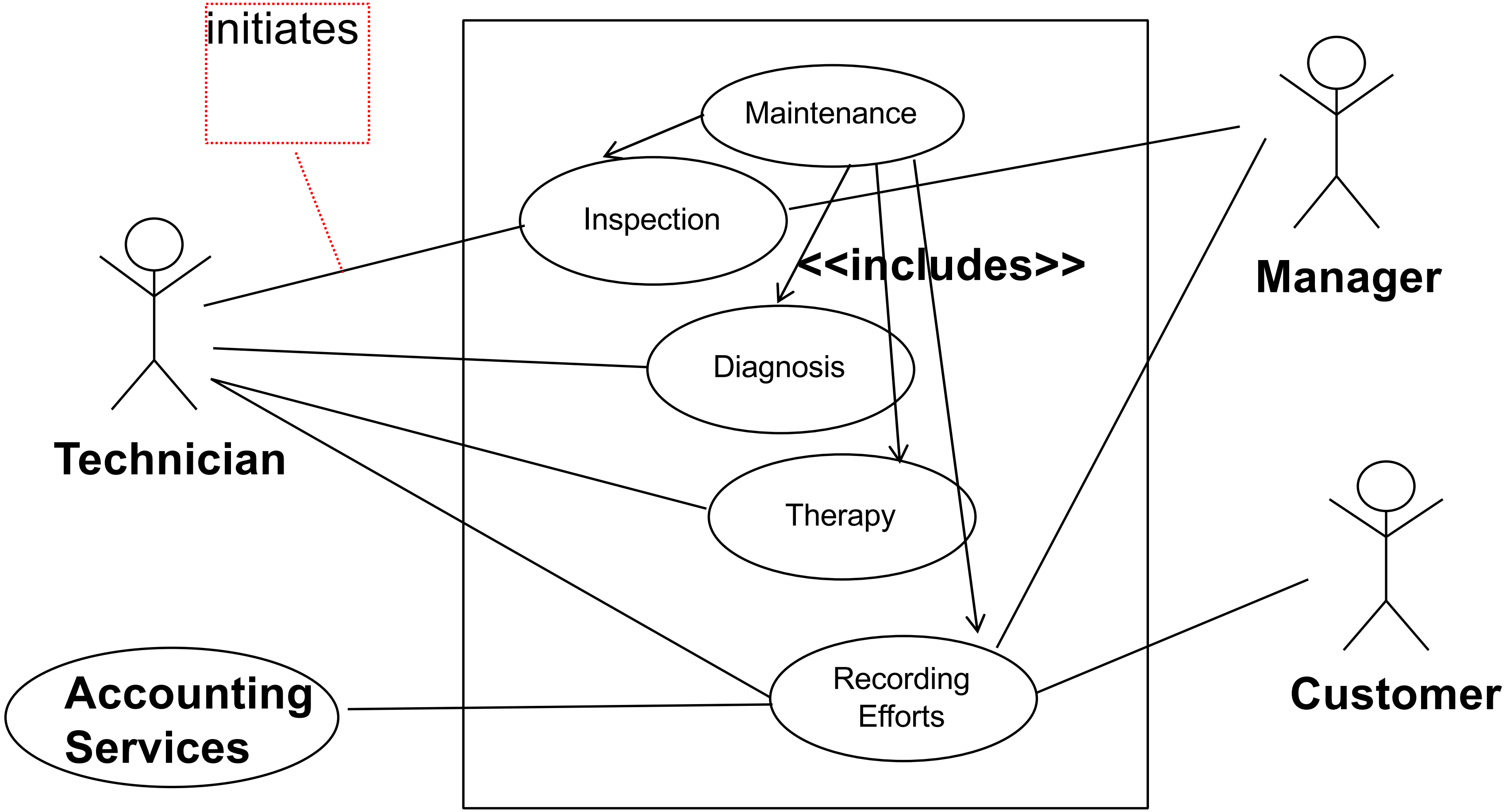


Second Refinement Service Station



Third Refinement Service Station

- The <<includes>> relationship allows for decomposition of a use case. <<includes>> is a form of <<part-of>>



Consistency Checking Check List Use Case Diagrams

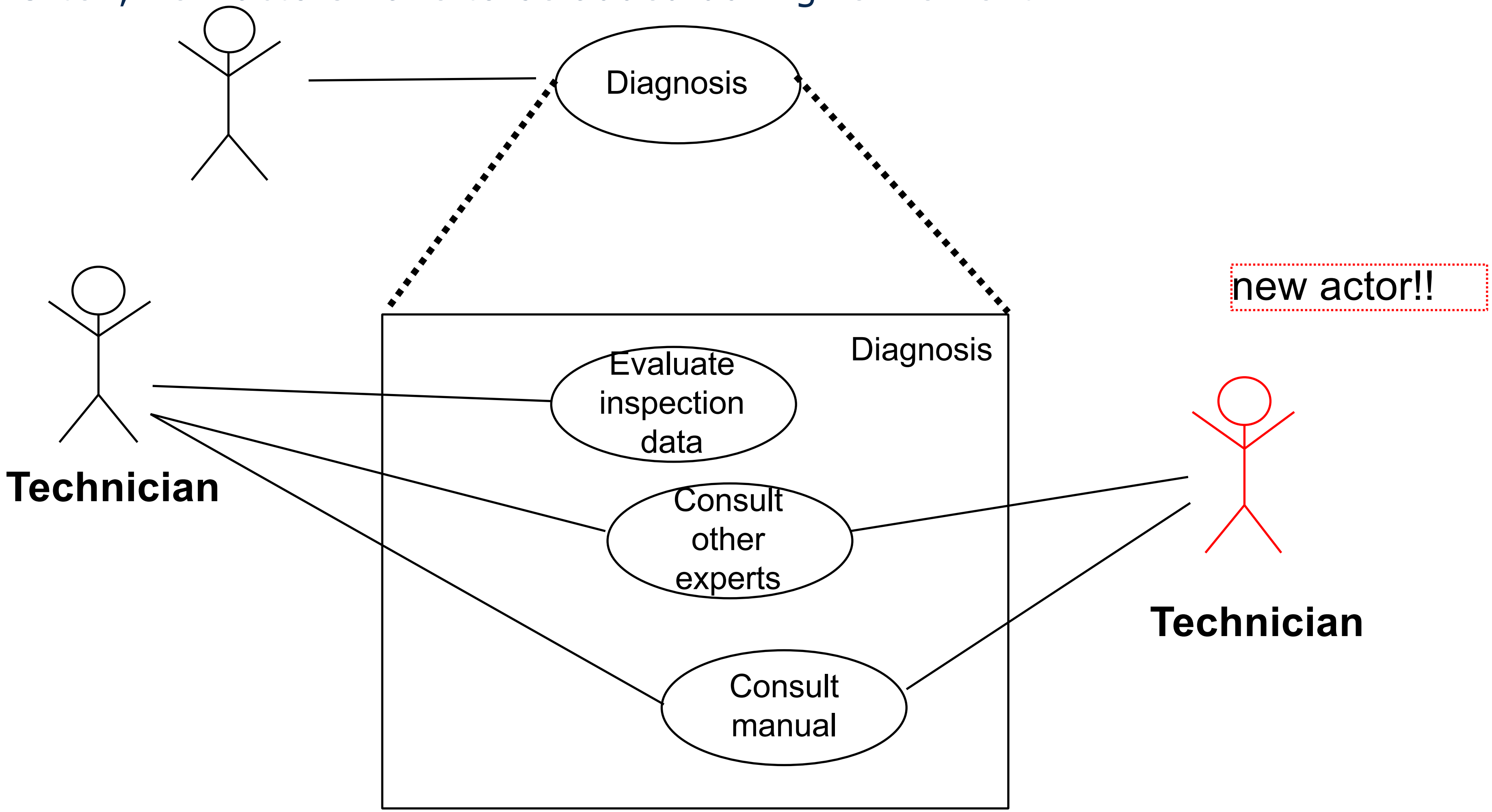
- One diagram
  - Clarity
  - Simplicity
  - Completeness
  - Match the stories of the customer?
  - Missing actors?
- Several diagrams
  - Which actions occur in several diagrams? Are they specified consistently?
  - Should actors from shared actions be replicated to other UCD?

How To Go On from a Use Case Diagram

- There are several ways how to reach a design from a use case diagram
  - *Hierarchical refinement* of the actions into UCD of second level, yielding a reducible specification
    - Disadvantage of UCD: Hierarchical refinement is sometimes difficult, because new actors have to be added
    - Leads to a correction of the top-level UCD
  - *Action tree method*: action-oriented method to refine the use case actions with an action tree
  - *Collaboration diagram method*: object-oriented method to analyse paths in the use case diagram with communication (collaboration) diagrams (see later)

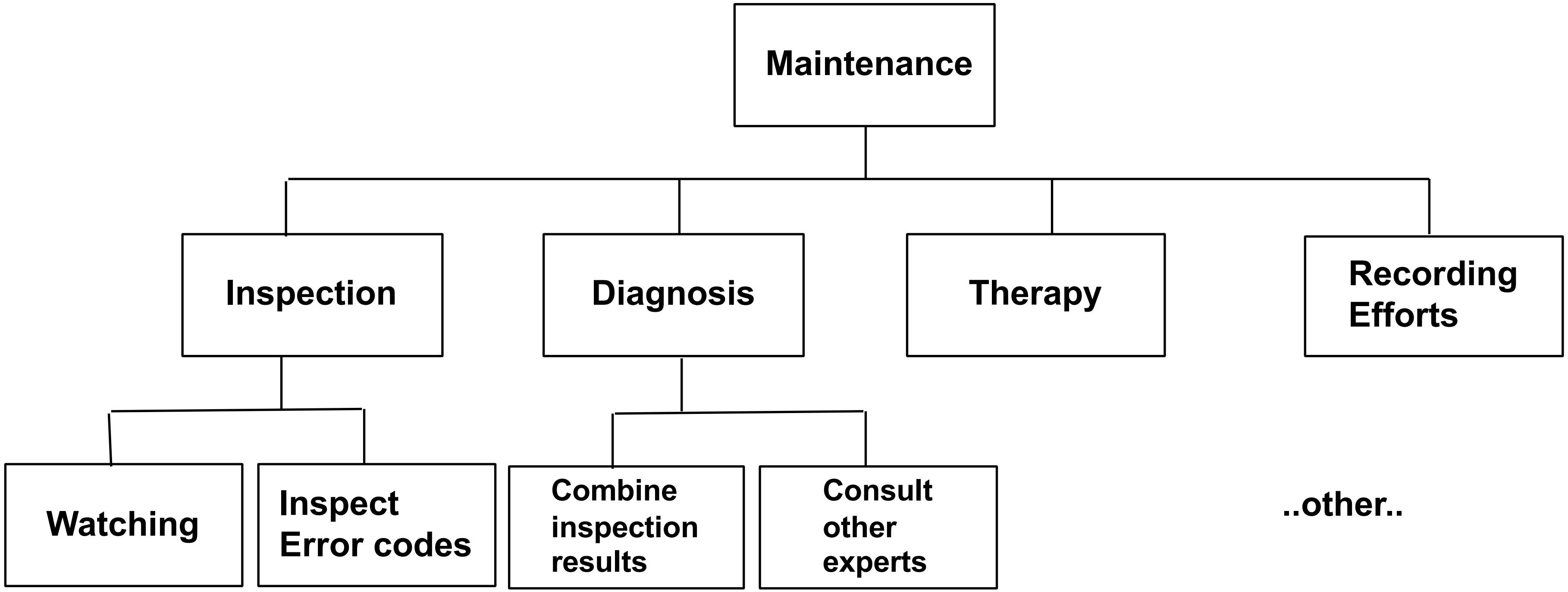
Hierarchical Refinement of a Use Case

- Often, new actors have to be added during refinement



Deriving an Function Tree from a Use Case

- DomainTransformation: From a UCD, set up a function or action tree
  - <<includes>> expresses a part-of hierarchy of function
- Refinement: Refine the functions by decomposition



Benefits of Use Cases

- Use cases are good for
  - Documentation
  - Communication with customers and designers -> Easy
  - Are started for the first layout of the structural model
    - To find classes, their actions, and relations
  - In eXtreme Programming (XP), use cases are called „stories“
    - which are written down on one muddy card
    - collected
    - and implemented one after the other
  - XP does not look at all use cases together, but implements one after the other

21.4 Extensibility of Function Trees

Change Points of Function Trees

- A function tree can have optional, variable and extensible parts (option points, variation points, extension points)

Prof. U. Aßmann

The End

Prof. U. Aßmann