



## 23 Action-Oriented Design Methods

1. Use Cases
2. Structured Analysis/Design (SA/SD)
3. Structured Analysis and Design Technique (SADT)

➤ Prof. Dr. U. Abmann  
➤ Technische Universität Dresden  
➤ Institut für Software- und Multimediatechnik  
➤ <http://st.inf.tu-dresden.de>  
➤ Version 11-0.1, 28.12.11



### Obligatory Reading

- Balzert, Kap. 14
- Ghezzi Ch. 3.3, 4.1-4, 5.5
- Pfleeger Ch. 4.1-4.4, 5



- Usually, action-oriented design is *structured*, i.e., based on hierarchical stepwise refinement.
- Resulting systems are
  - *reducible*, i.e., all results of the graph-reducibility techniques apply.
  - Often *parallel*, because processes talk with streams
- SA and SADT are important for *embedded systems* because resulting systems are parallel and hierarchic



# 23.1 ACTION-ORIENTED DESIGN

- Action-oriented design is similar to function-oriented design, but admits that the system has states.
  - It asks for the internals of the system
  - Actions require state on which they are performed (imperative, state-oriented style)
- Divide: finding subactions
- Conquer: grouping to modules and processes
- Example: all function-oriented design methods can be made to action-oriented ones, if state is added

**What are the actions the system should perform?**

Data-flow connects processes (parallel actions)

## 23.2 ACTION-ORIENTED DESIGN WITH SA/SD

- [DeMarco, T. Structured Analysis and System Specification, Englewood Cliffs: Yourdon Press, 1978]
- Representation
  - Function trees (action trees, process trees): decomposition of system functions
  - Data flow diagrams (DFD), in which the actions are called *processes*
  - Data dictionary (context-free grammar) describes the structure of the data that flow through a DFD
  - Pseudocode (minispecs) describes central algorithms
  - Decision Table and Trees describes conditions (see later)

- On the highest abstraction level:
  - Elaboration: Define interfaces of entire system by a top-level function tree
  - Elaboration: Identify the input-output streams most up in the function hierarchy
  - Elaboration: Identify the highest level processes
  - Elaboration: Identify stores
- Refinement: Decompose function tree hierarchically
- Change Representation: transform function tree into process diagram (action/data flow)
- Elaboration: Define the structure of the flowing data in the Data Dictionary
- Check consistency of the diagrams
- Elaboration: Minispecs (pseudocode)

# DFD- Modellierung

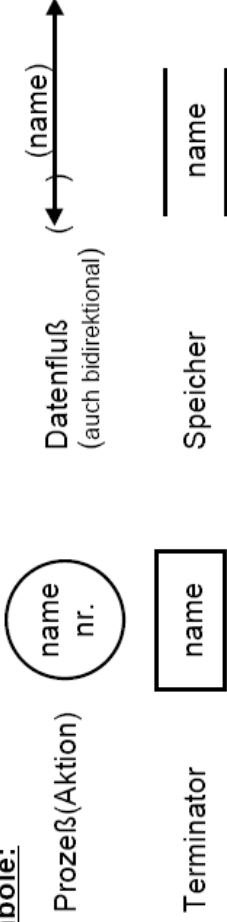
## ● Graphische Mittel: Datenflußdiagramme - DFD

Kontextdiagramm  
Parent-Diagramme  
Child-Diagramme

(mit Terminatoren)

hierarchisch gegliedert  
mit übereinstimmender  
Prozeß-/Diagramm-  
numerierung

### Symbole:



## ● Textliche Mittel:

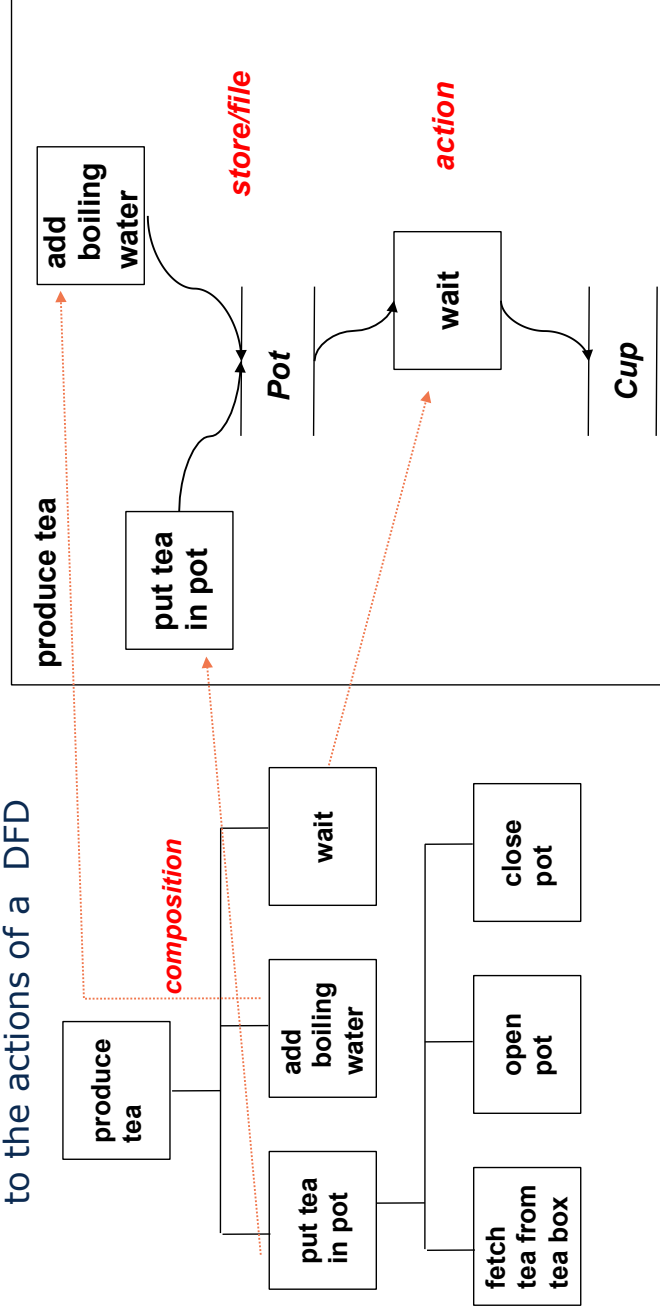
**Data Dictionary:** Im Datenkatalog ist jeder Speicher und jeder Datenfluß in seiner Zusammensetzung zu beschreiben. Es ist identisch mit dem der IM-Modellierung.

**Mini-Spezifikationen:** Dienen der näheren Beschreibung der in Elementarprozessen durchzuführenden (Daten-)Transformationen.

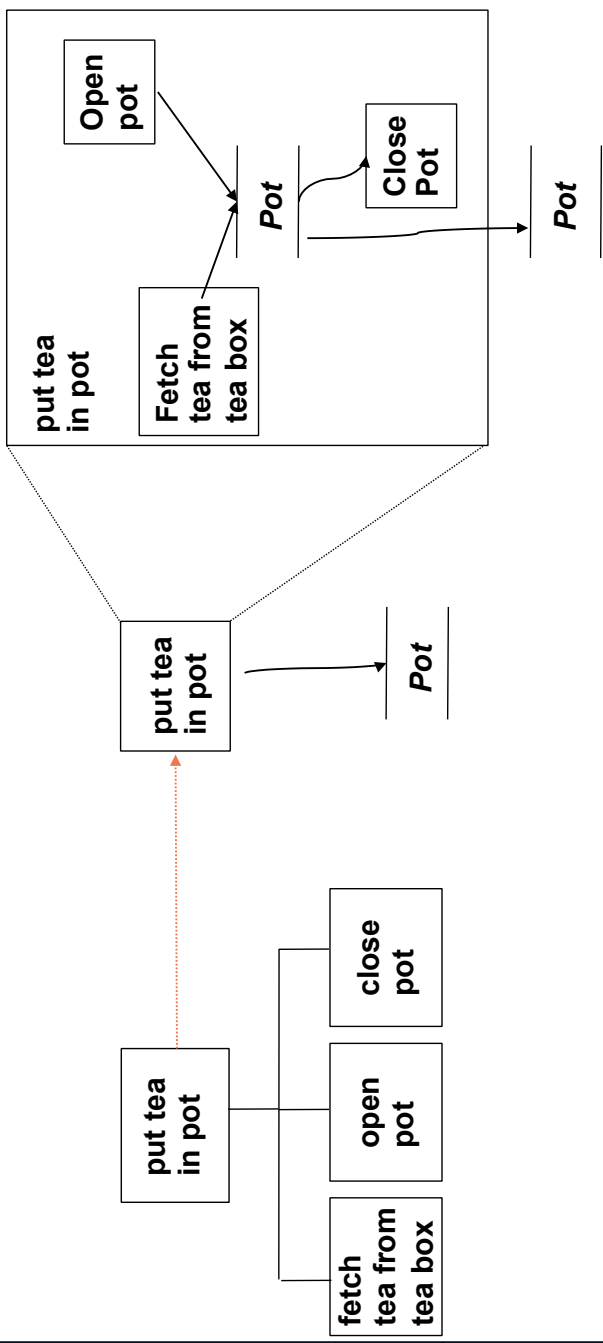


## Function Trees (Action Trees) and DFDs

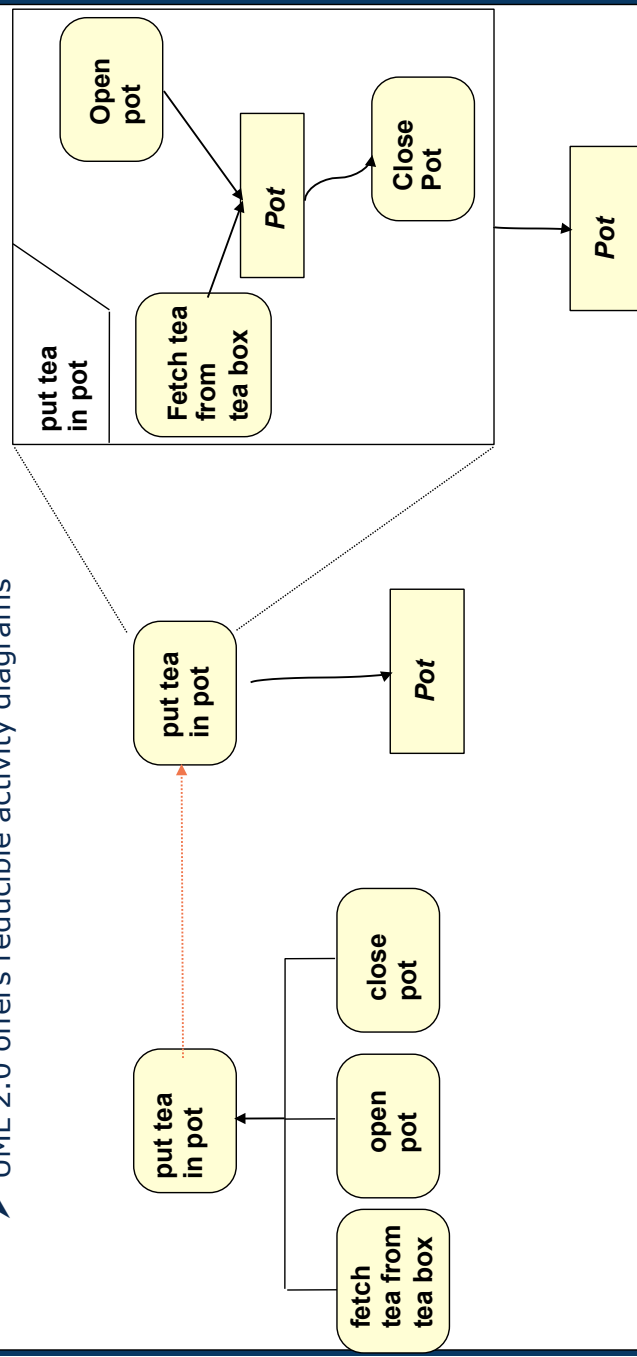
- Function trees are homomorphic to DFD
- RepresentationChange: Construct a function tree and transform it to the actions of a DFD



- Subtrees in the function tree lead to reducible subgraphs in the DFD



- UML function diagrams can be formed from actions, and aggregation
- Activity diagrams can specify dataflow
  - UML 2.0 offers reducible activity diagrams



# Regeln der DFD-Erstellung

(möglichst werkzeuggestützt prüfen)

- **Semantische Regeln** zur Namensgebung:
    - **Prozeßnamen:** *Verb\_Substantiv* zur aussagekräftigen Beschreibung einer Aktion (z.B. *berechne\_Schnittpunkt*)
    - **Datenflußnamen:** [*<Modifier>*]*Substantiv* beschreibt momentanen Zustand des Datenflusses (z.B. *<neue>Anschrift*)
    - **Speichernamen:** *Substantiv*, das den Inhalt des Speichers (identisch Entity im DD) beschreibt (z.B. *Adressen*)
  - **Syntaktische Regeln** zur graphischen DFD-Darstellung:
    - Jeder Datenfluß muß mit mindestens einem Prozeß verbunden sein.
    - **Datenflüsse** zwischen Terminatoren und direkt zwischen Speichern sind nicht erlaubt.
    - **Datenspeicher**, die nur einseitig beschrieben (ohne zu lesen) und nur einseitig gelesen (ohne zu beschreiben) werden, sind nicht erlaubt.
    - **Prozesse**, die Daten ausgeben, ohne sie erhalten zu haben oder umgekehrt, die Daten erhalten, ohne sie auszugeben oder zu verarbeiten, sind nicht erlaubt.
    - Im **Kontext** darf es keine Speicher geben, in **Verfeinerungen** keine Terminatoren
- Jeder Prozeß, Speicher und Datenfluß muß einen Namen haben. Nur in dem Fall, wo der Datenfluß alle Attribute des Speichers beinhaltet, kann der Datenflußname entfallen.  
Weiterführende Literatur: [2, S.437]



## Typing Edges with Types from the Data Dictionary

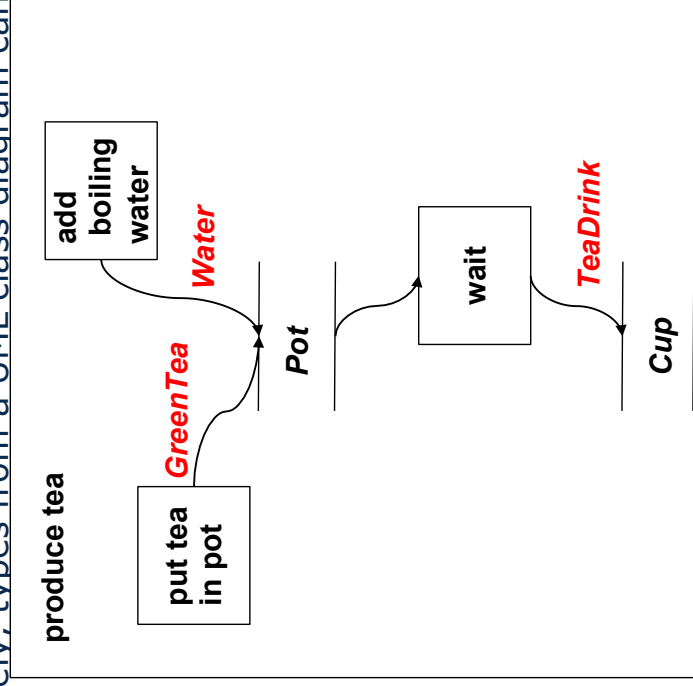
- In an SA model, the *data dictionary* describes the context free structure of the data flowing over the edges
  - For every edge in the DFDs, it contains a context-free grammar that describes the flowing data items
- Notation is also called Extended Backus-Naur Form (EBNF)

	Notation	Meaning	Example
Sequence	::= or =		A ::= B.
Sequence	+		A ::= B+C.
Selection	<blank>		A ::= B C.
Repetition	[   ]	Alternative	A ::= [ B   C ].
Limited repetition	{ }^n		A ::= { B }^n.
Option	{ } n	Repetition from m to n	A ::= 1{ B }10.
	( )	Optional part	A ::= B (C).

```

DataInPot ::= TeaPortion WaterPortion.
TeaAutomatonData ::= Tea | Water | TeaDrink.
Tea ::= BlackTea | FruitTea | GreenTea.
TeaPortion ::= { SpoonOfTea }.
SpoonOfTea ::= Tea.
WaterPortion ::= { Water }.
    
```

- Nonterminals from the data dictionary become types on flow edges
- (Alternatively, types from a UML class diagram can be annotated)





- Minispecs describes the processes in the nodes of the DFD in pseudo code. They describe the data transformation of every process

- Here: specification of the minispec attachment process:

```

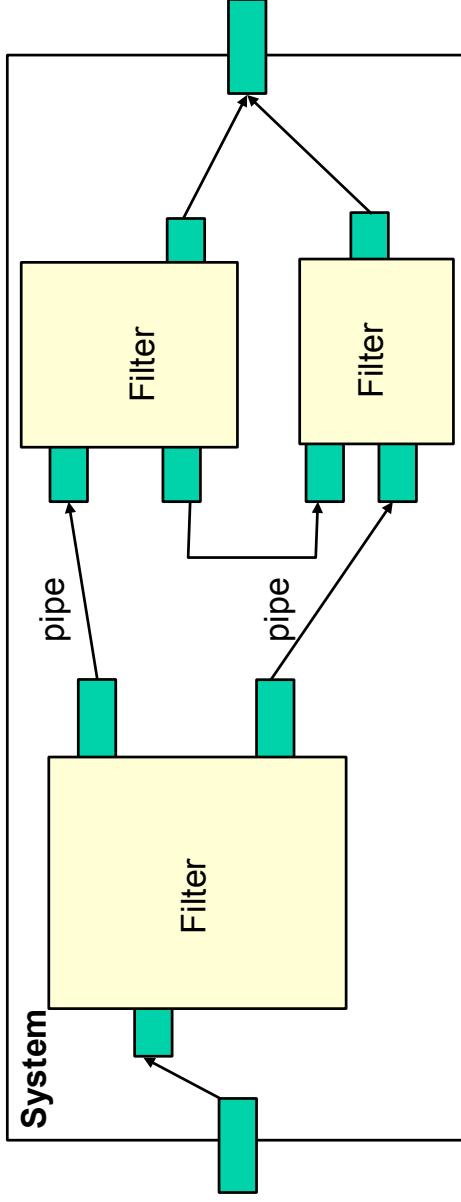
procedure: AddMinispecsToDFDNodes
  target.bubble := select DFD node;
  do while target.bubble needs refinement
    if target.bubble is multi-functional
      then decompose as required;
        select new target.bubble;
      add pseudocode to
  target.bubble;
    else no further refinement needed
      endif
    enddo
  end
  
```

- SETL (Schwartz, New York University)
  - Dynamic sets, mappings
  - Iterators
- PIKE (pike.ida.liu.se)
  - Dynamic arrays, sets, relations, mappings
  - Iterators
- ELAN (Koster, GMD)
  - Natural language as identifiers of procedures
- Smalltalk (Goldberg et.al, Parc)
- Attempto Controlled English (ACE, Prof. Fuchs, Zurich)
  - A restricted form of English, easy to parse

- Consistency checks
  - Several consistency rules between diagrams (e.g., between function trees and DFD)
  - Corrections necessary in case of structure clash between input and output formats
- Advantage of SA
  - Hierarchical refinement: The actions in the DFD can be refined, i.e., the DFD is a reducible graph
  - SA leads to a hierarchical design (a component-based system)

- SA focusses on actions (activities, processes), not functions
  - Describe the *data-flow* through a system
  - Describe stream-based systems with pipe-and-filter architectures
- Actions are processes
  - SA and SADT can easily describe parallel systems
- Function trees are interpreted as *action trees (process trees)* that treat streams of data

- SA/SD design leads to dataflow-based architectural style
- Processes exchanging streams of data
- Data flow forward through the system
- Components are called *filter*, connections are pipes

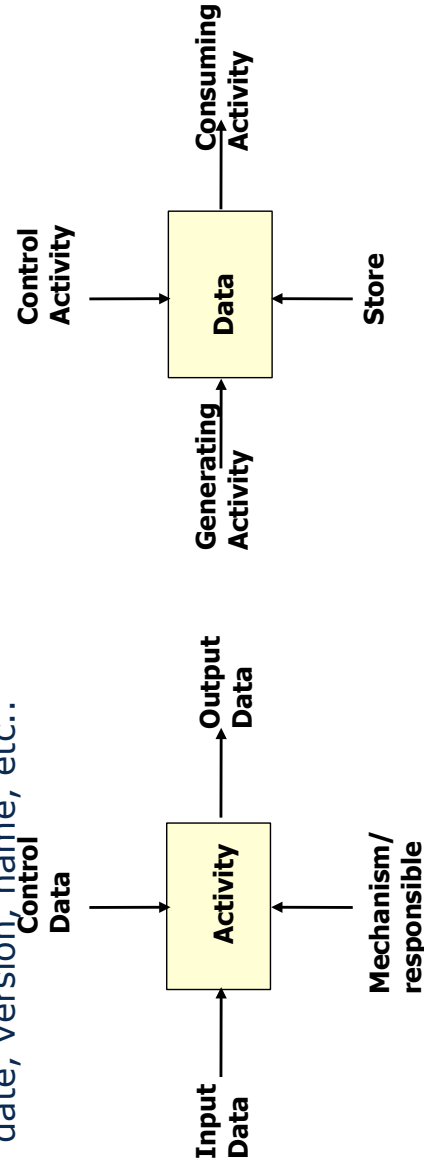


- Shell pipes-and-filters
- Image processing systems
- Signal processing systems (DSP-based embedded systems)
  - The satellite radio
  - Video processing systems
  - Car control
  - Process systems (powerplants, production control, ...)
- Content management systems (CMS)

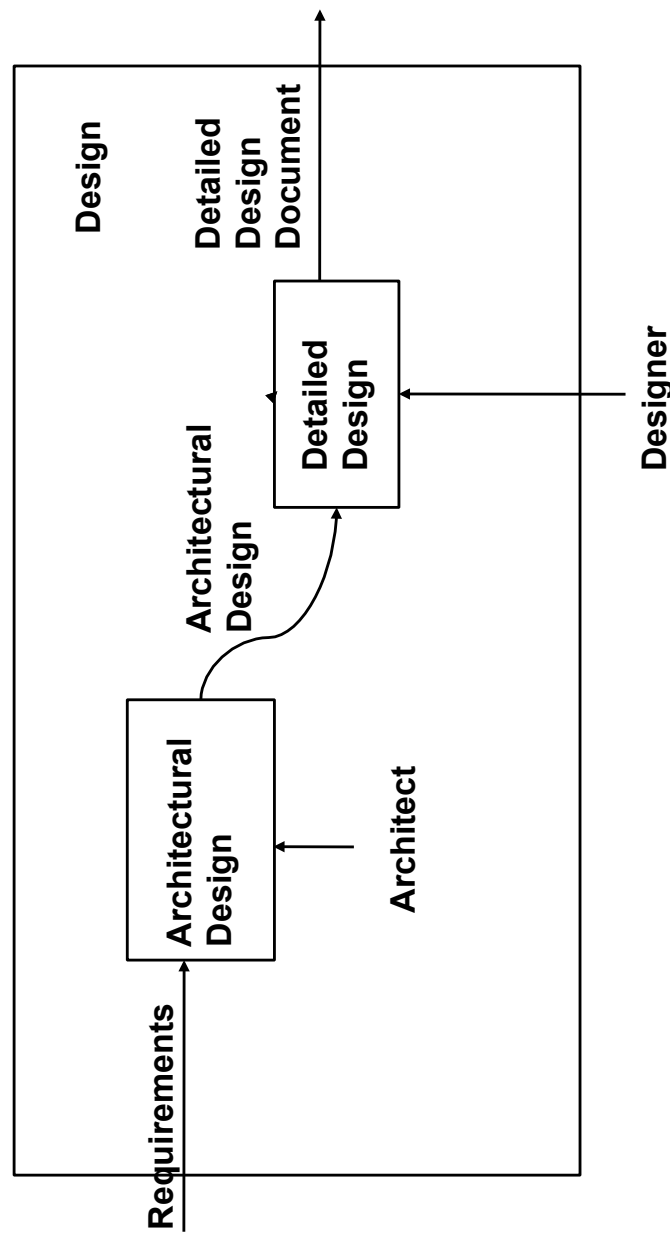
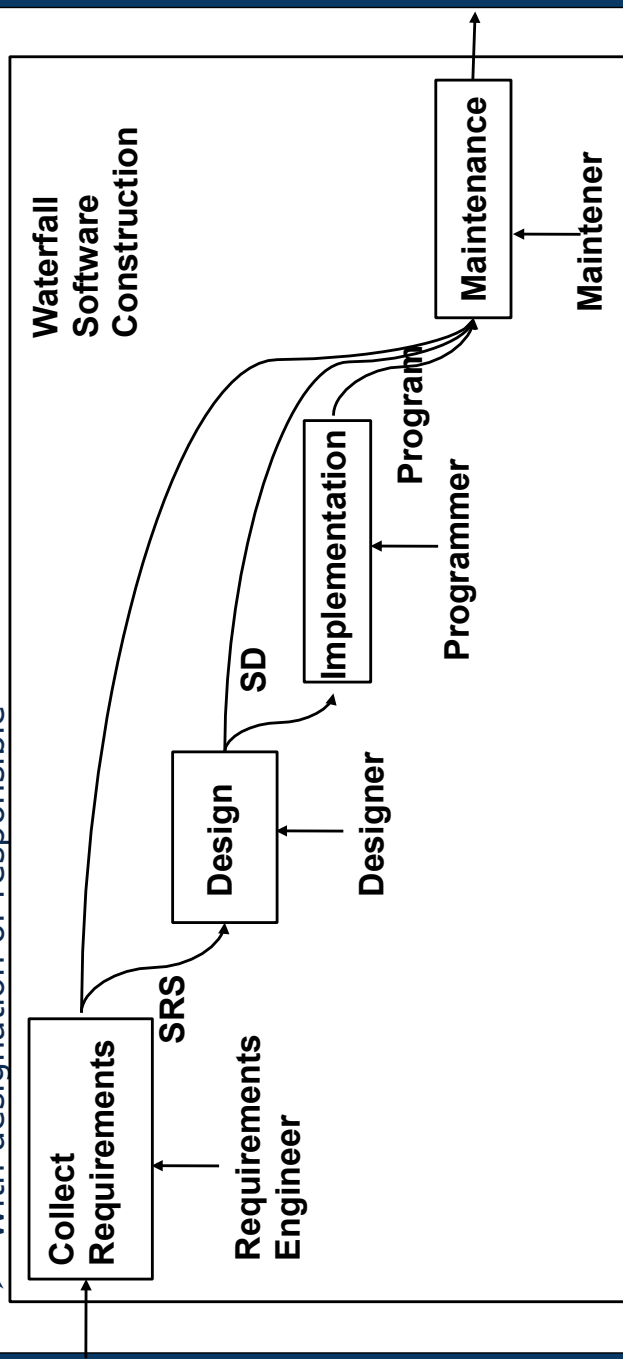
# 23.3 SADT

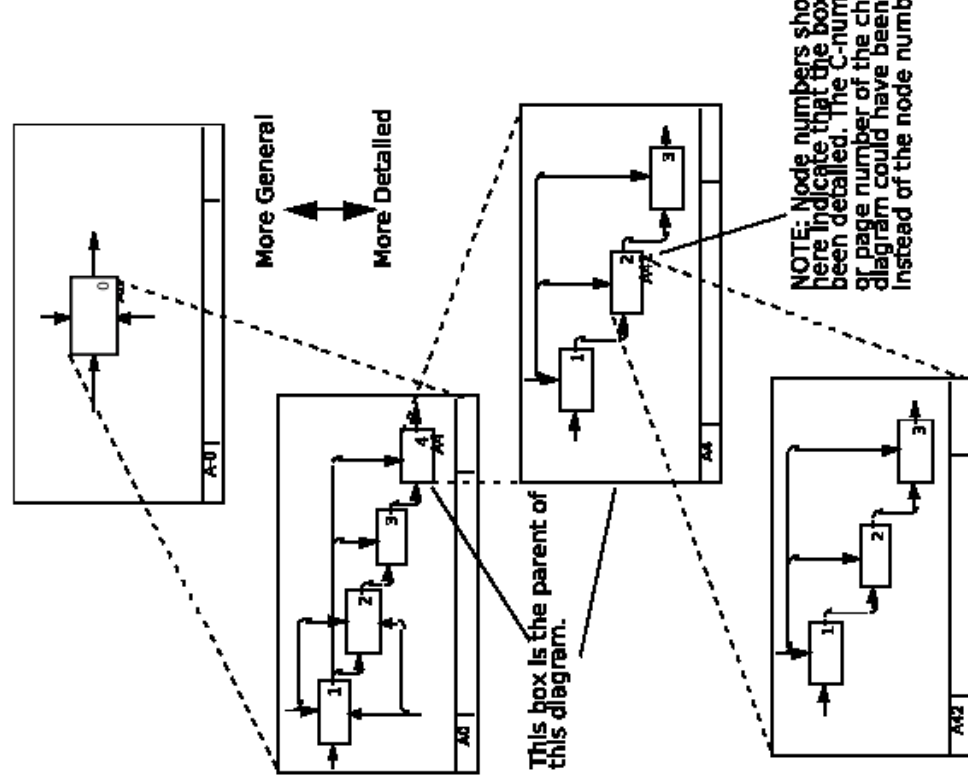
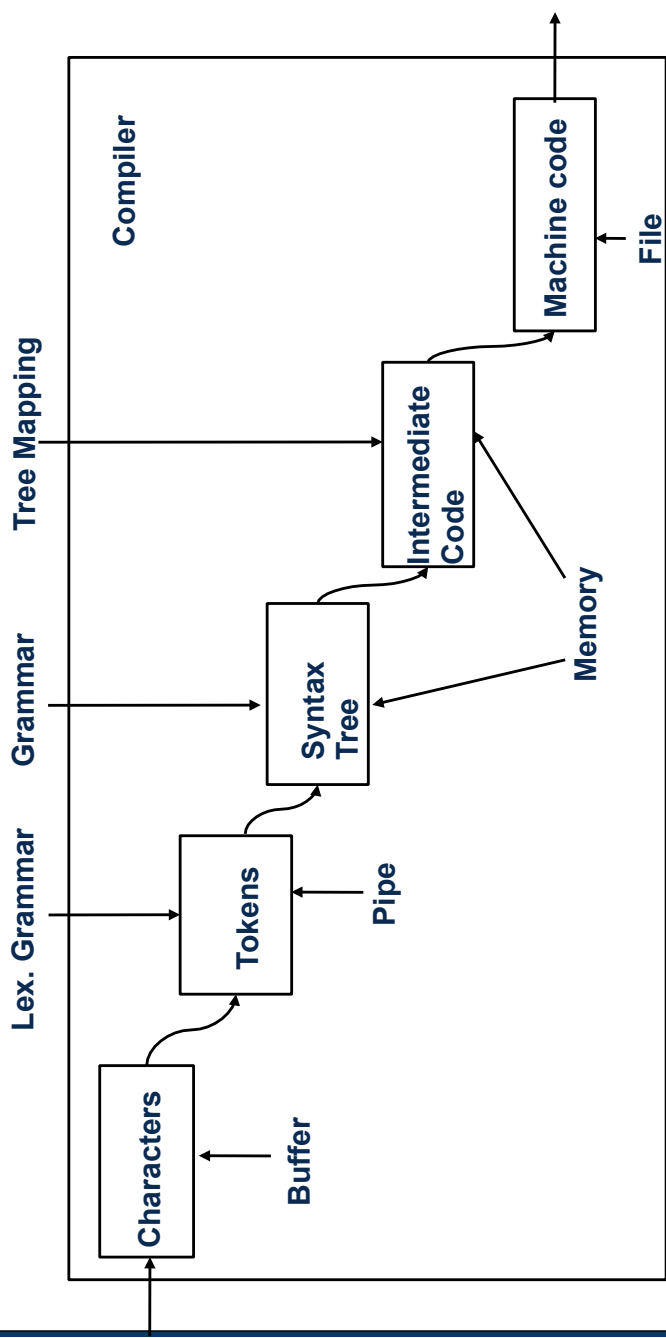
## Structured Analysis and Design Technique (SADT)

- SADT is an action- and data-flow-oriented method
- Reducible graphs with 2 main forms of diagrams:
  - Activity diagrams: Nodes are activities, edges are data flow (like DFD)
  - Data flow architectures result
- Data diagrams: Nodes are data (stores) and edges are activities
- Layout constraint: edges go always from left to right, top to bottom
- Companies used to have standardized forms, marked with author, date, version, name, etc..



- Activity Diagrams SADT – Similar to DFD
- Read direction left to right, top to bottom
- With designation of responsible





- SADT, SA/SD are system-oriented methods, known in other disciplines
  - *Action-oriented methods*
    - they only distinguish between actions (processes) and data
  - *Stream-oriented*, i.e., model streams of data flowing through the system
  - *System-oriented*, know the concept of a *subsystem*
- SA-DFDs are more flexible as SADT activity diagrams, since the layout is not constrained
  - Function trees and DDs may be coupled with SADT

- They lead to component-based systems (hierarchical systems)
  - Component-based systems are ubiquitous for many areas
  - Object-orientation is not needed everywhere
  - Other engineers use SADT also
- SA and SADT can easily describe parallel systems in a structured way
- SA and SADT are stream-based, i.e., for stream-based applications. When your context model has streams in its interfaces, SA and SADT might be applicable
- Use case actions can be refined similarly as SA and SADT actions!



## What Have We Learned

- Use case diagrams are an action-oriented diagram notation
  - that can be coupled with several design methods (action trees, communication diagrams)
- Besides object-oriented design, *structured*, *action-oriented design* is a major design technique
  - It will not vanish, but always exist for certain application areas
  - If the system will be based on stream processing, system-oriented design methods are appropriate
  - System-oriented design methods lead to *reducible systems*
- Don't restrict yourself to object-oriented design



## The End