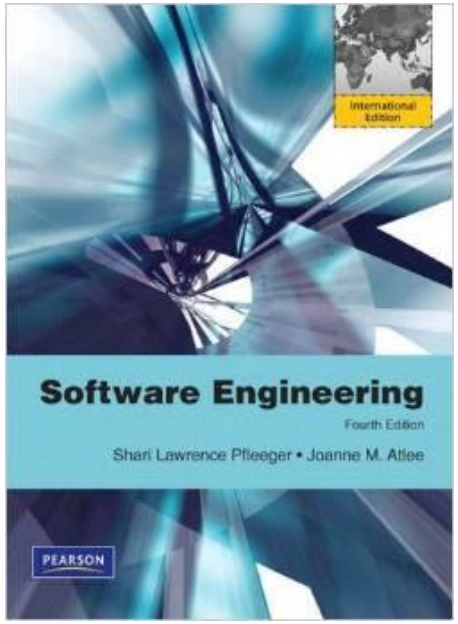
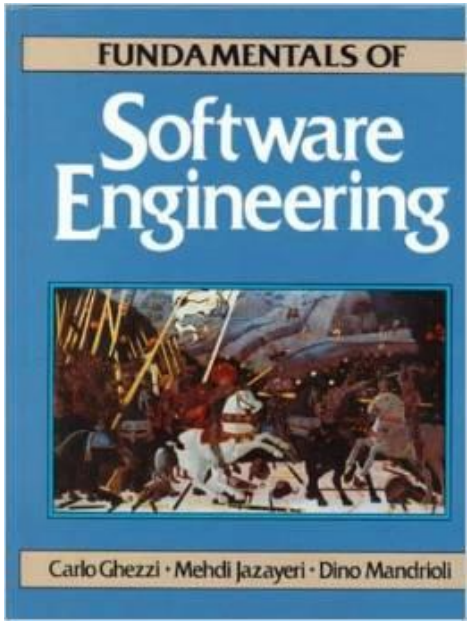
20 Design Methods - An Overview

- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediatechnik
- Gruppe Softwaretechnologie
- http://st.inf.tu-dresden.de/teaching/swt2
- Wintersemester 14/15, 16.12.2014

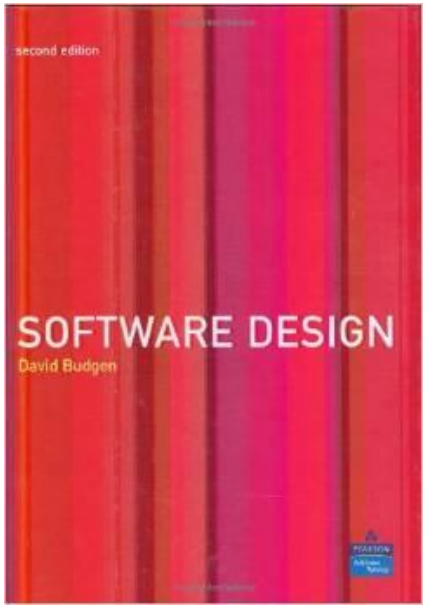
- **Lecturer**: Dr. Sebastian Götz

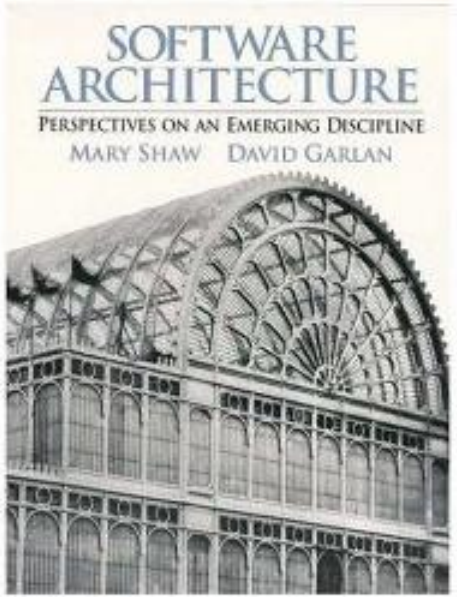
- S. L. Pfleeger and J. Atlee:
  Software Engineering: Theory and Practice.
  Pearson. 2009.
  - Chapter 5 (Designing the Architecture)
- C. Ghezzi, M. Jazayeri and D. Mandrioli:
  Fundamentals of Software Engineering.
  Prentice Hall. 1992.
  - Chapter 4 (Design and Software Architecture)





- D. Budgen:
  Software Design (2nd Edition).
  Addison-Wesley. 2003.
- M. Shaw and D. Garlan:
  Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.





- Get an **overview** on the available design methods to arrive at a design, starting from a requirements specification
- Understand that software engineers shouldn't get stuck by a specific design method

What is Design?

- "The purpose of design is simply to produce a solution to a problem." [Budgen, p. 18]
- "The design is the creative process of figuring out how to implement all of the customer's requirements." [Pfleeger, p. 224]
- "Design is the activity that acts as a bridge between requirements and the implementation of the software." [Ghezzi, p. 67]
- **Goal**: This lecture presents some systematic ways how to come to a workable solution for a given problem

Contents of the Software Requirements Specification (SRS) (rep.)

- Overview of Product
- *Background, Environment*
- *Interfaces of the System (context model)*
  - I/O interfaces, data formats (screens, protocols, etc.), Commands
  - Overview of data flow through system, Data dictionary
- *Functional requirements*
- Non-functional requirements
- *Error handling*
- Prioritization
- *Possible extensions*
- Acceptance test criteria
- Documentation guideline
- Literature
- Glossary

20.1 DESIGN METHODS

[Budgen, p. 34]

… has 2 components:

1. Representation part (notation, language)
   - Set of notations in (informal) textual, (semi-formal) diagrammatic, or mathematic (formal) form
2. Process part (“Vorgehensmodell”, “Prozessmodell”)
   - “… describing how […] transformations between the representation forms are to be organized […].”

… most design methods provide a third component:

1. Set of heuristics
   - “[…] provide guidelines on the ways in which the activities defined in the process part can be organized […]”

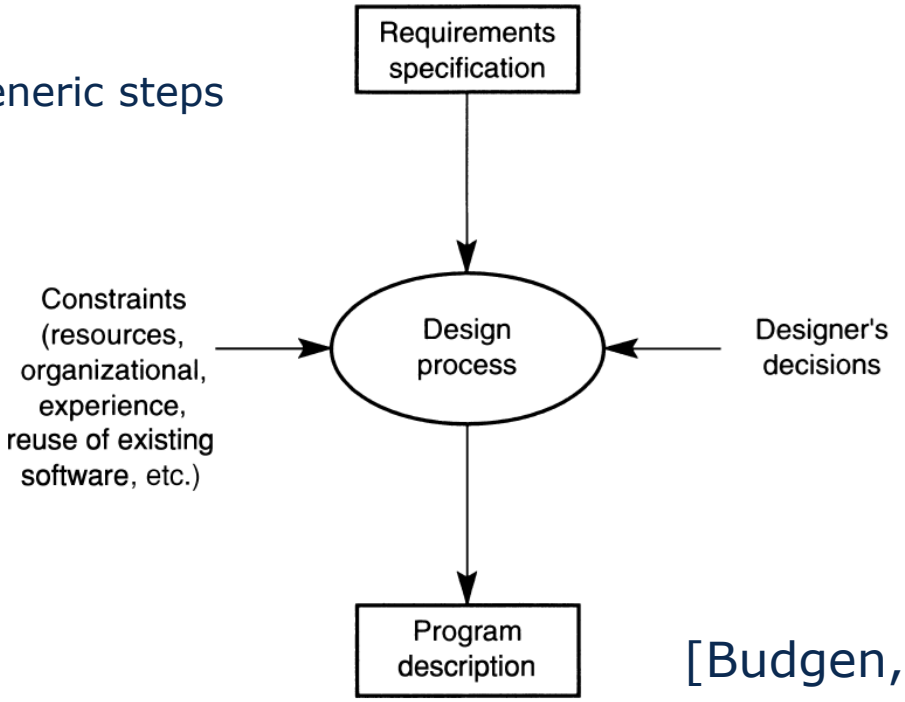
20.1.1 DESIGN REPRESENTATION

20.1.1 Design Representation

	Text	Diagrams	Math
Paper Specification Languages	Informal Natural language Pseudo-code	Flow chart Data-flow Diagram Entity-Relationship Diagram ER	Vienna Development Language Z B
Executable Specification Languages	Parseable natural language	Colored Petri nets State machines UML Structure Diagram	Process algebras (CSP, CCS)
Programming Languages	C Java Python	Workflow languages (BPEL) Choregraphe	Modelica Matlab Simulink

20.1.2 DESIGN PROCESSES

- A **design process** is a structured algorithm (or workflow) to achieve a workable solution from a requirement specification
  - A sequence of steps
  - A set of milestones
- The design process starts from *the system's interfaces (context model)* and refines its internals
- Every design process
  - Contains several central generic steps

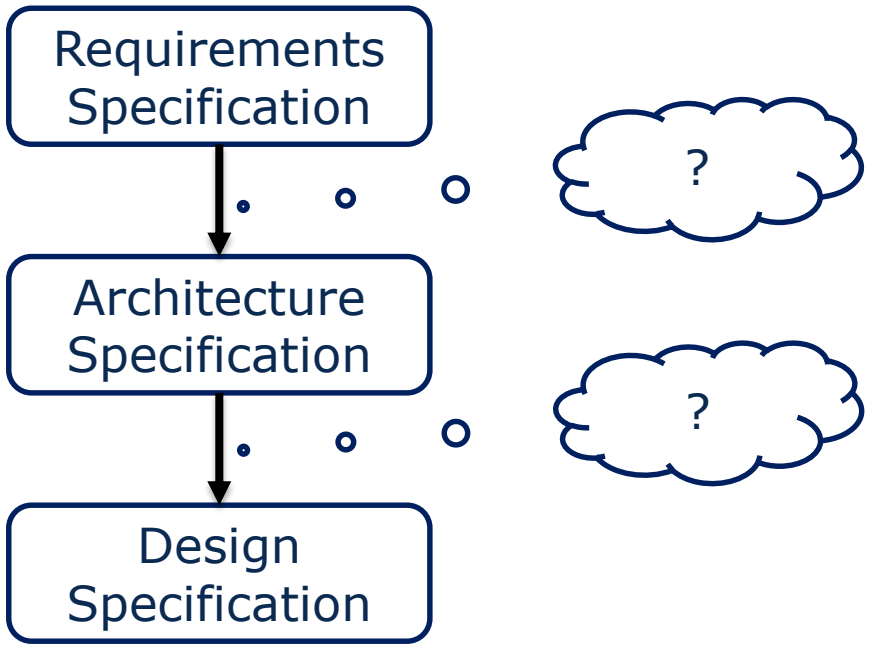


[Budgen, p. 29]

Operations/Actions of Design Methods

- Many methods have actions like elaboration, refinement, checking, and structuring
- Manual operations
  - Split (decompose, introduce hierarchies, layers, reducibility)
  - Merge (coalesce)
- Automatic operations
  - Graph analysis methods
  - Graph structuring methods, e.g., by graph transformations or edge addition rewrite systems
  - Remember: text-based specifications can be transformed into graphs

- How to get a workable solution starting with a requirements specification?



- An **architecture style** provides
  - Certain types of components
  - Certain types of connections/connectors
  - Invariants/constraints among them
- Architectural styles provide a vocabulary to talk about the coarse-grain structure of a system
  - Good for documentation and comprehension
  - Good for maintenance
- Architectural styles compared to design patterns
  - Design patterns describe the relationship between several classes or objects of an application, but not of the entire system
  - Architectural styles describe what kinds of building blocks and glue exists

What Is In a Style ?

- A style can be approached by answering 7 questions [Shaw/Garlan]
  1. What is the design vocabulary/the types of components and connectors?
  2. What are the allowable structural patterns?
  3. What is the underlying computational model?
  4. What are the essential invariants of the style?
  5. What are some common examples of its use?
  6. What are the advantages and disadvantages of using that style?
  7. What are some of the common specializations?

- Example: Pipes and Filters

```
> cat server.log | grep timeout | wc -l
```

- How do I derive a design for the system?
  - How do I find the best architectural style for the system?
  - How do I derive a detailed design?

- In design meetings, the basic design questions are posed in a structured way
  - Select a design method
  - Pose the design method's basic question
  - Perform the design method's process
    - Perform the design method's steps
  - If process gets stuck, change design method and try another one
    - However, be aware, which design method and process you use

20.2 OVERVIEW OF DESIGN METHODS

- Methods can be grouped according to their focus of decomposition and the design notation they use.
  - **Function-oriented**: function in focus
  - **Action-oriented, event-action-oriented:** Action in focus
  - **Data-oriented**: A data structure is in focus
  - **Component-oriented (structure-oriented)**: parts in focus
  - **Object-oriented**: objects (data and corresp. actions) in focus
  - **Transformational**: basic action is the transformation
  - **Generative:** basic action is a special form of transformation, the generation. Also using planning.
  - **Formal methods**: correct refinement and formal proofs in focus
    - **Refinement-based:** basic action is the point-wise and regional refinement, with verification of conformance
  - **Aspect-oriented methods:** refinement according to viewpoints and concerns

- Design with functional units which transform inputs to outputs
  - Minimal system state
  - Information is typically communicated via parameters or shared memory
  - No temporal aspect to functions

- Functions/operations are grouped to *modules* or *components*
- Divide: finding subfunctions
- Conquer: grouping to modules

- Examples
  - Parnas' change-oriented design (information-hiding based design, see ST-1)
- **Use**: when the system has a lot of different functions

What are the functions of the software?

Action-Oriented Design

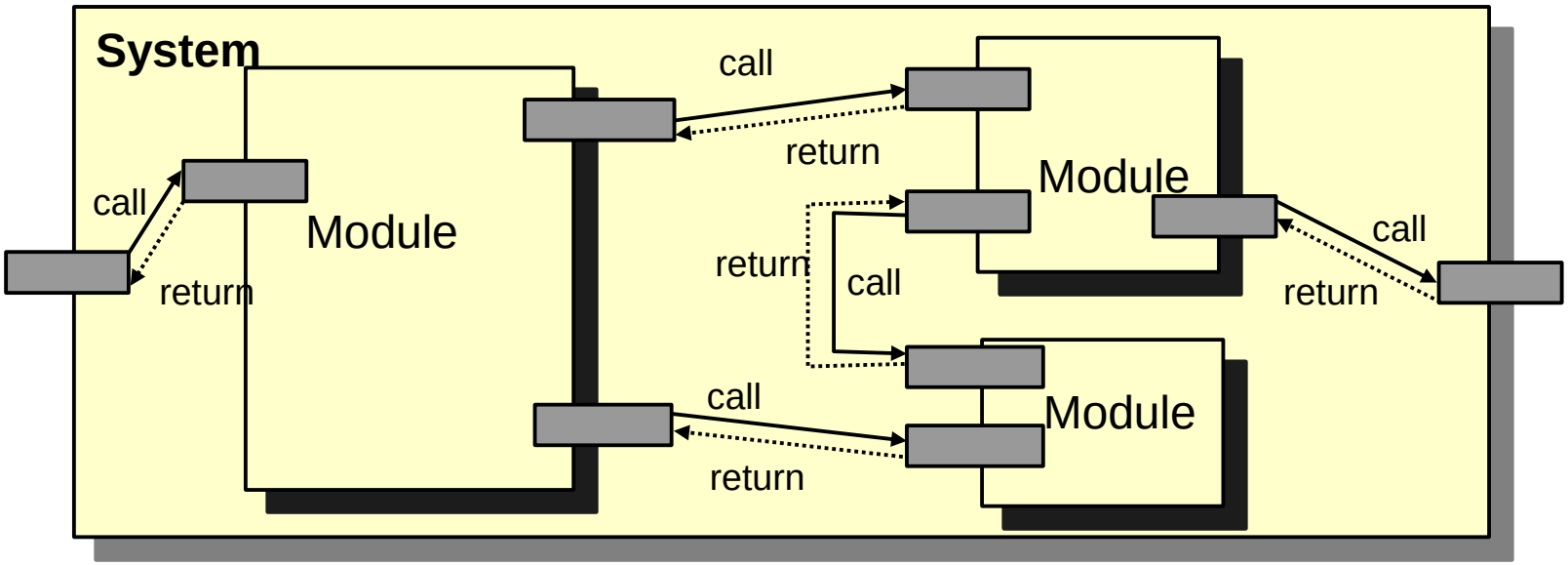
- Action-oriented design is similar to function-oriented design, but actions require *state* on which they are performed (imperative, state-oriented style)

- Divide: finding subactions
- Conquer: grouping to modules
- Examples:
  - Petri Nets
  - Use-case-based development
  - Data-flow based development SA, SADT
- **Use**: when the system maps to a state space, in which actions form the transitions

What are the actions the system should perform?

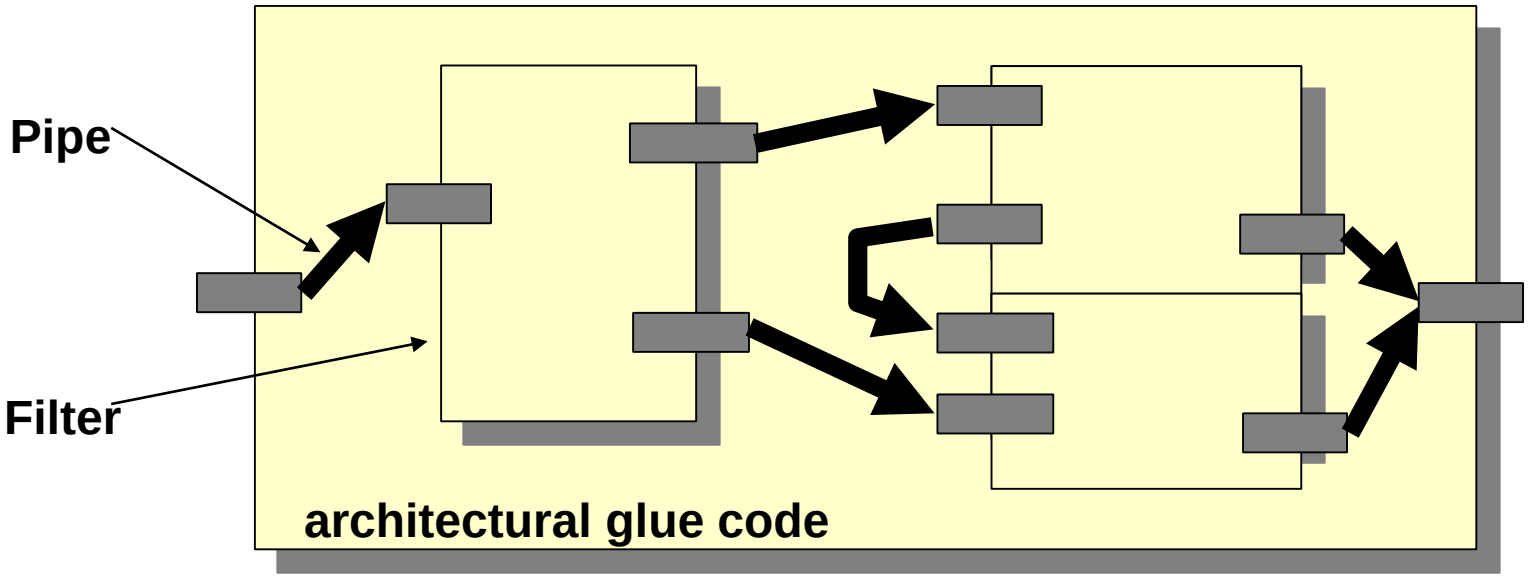
Result 1: Call-Based Architectural Style

- Components denote procedures that call each other
- Control flow is symmetric (calls and symmetric returns)
- Data-flow can be
  - parallel to the call (*push-based system*): caller pushes data into callee
  - antiparallel, i.e., parallel to the return (*pull-based system*): caller drags out data from callee
- Aka "Client-Server" in loosely coupled or distributed systems



Result 2: Data-Flow Based Systems (Pipe-and-Filter, Channels, Streams)

- If data flows in streams, call-based systems are extended to *stream-based systems*
- Components: processes, connectors: streams
- Control flow is asynchronous, continuous
- Data-flow graph of connections, static or dynamic binding
- Data-flow can be parallel to the control-flow (*push-based system*) or antiparallel (*pull-based system*)



Example: Linux shell: cat server.log | grep timeout | wc -l

Data-flow based systems:

- Image processing systems
  - Microscopy, object recognition
- Digital signal processing systems
  - Video and audio processing, e.g., telephony
- Batch-processing systems

Call-based systems:

- Object-oriented frameworks

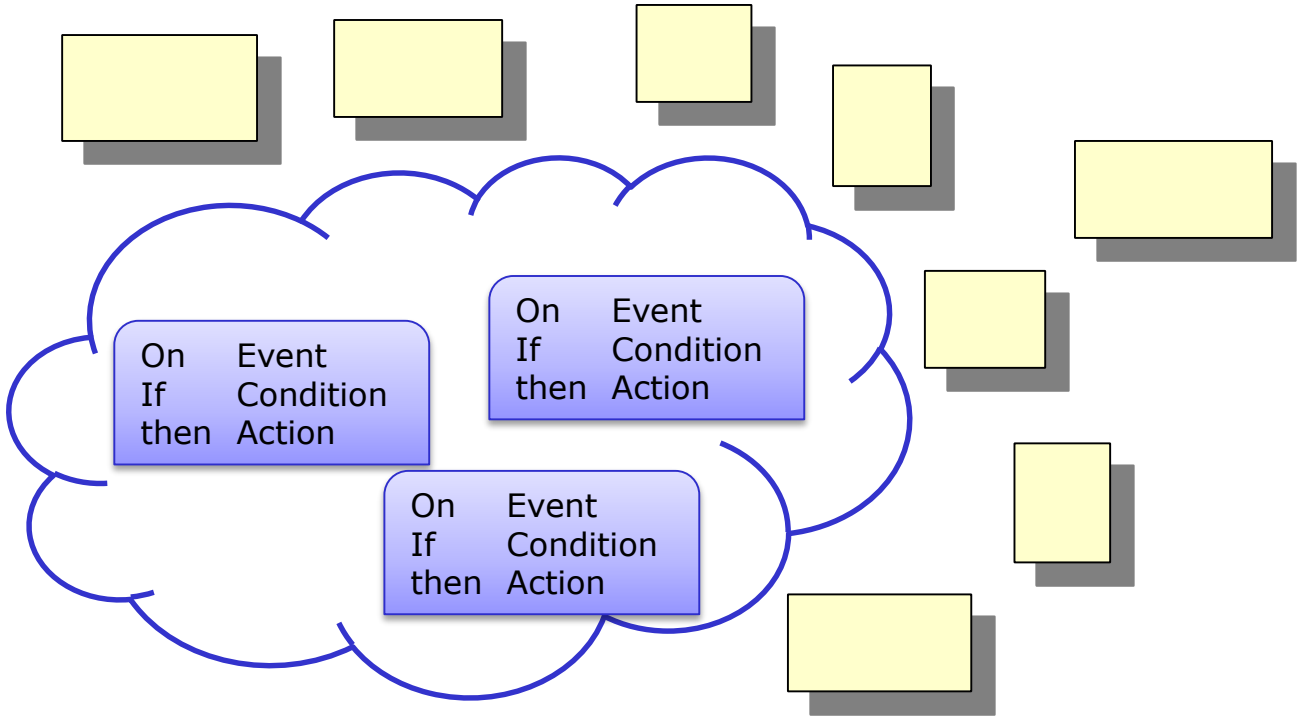
Event-Condition-Action-Oriented Design

- Event-condition-action rules (ECA rules)
  - On which event, under which condition, follows which action?
- Divide: finding rules for contexts
- Conquer: grouping of rules to rule modules
- Example:
  - Business-rule-based design
- **Use**: when the system maps to a state space, in which actions form the transitions and the actions are guarded by events

What are the events that may occur and how does my software react on them?

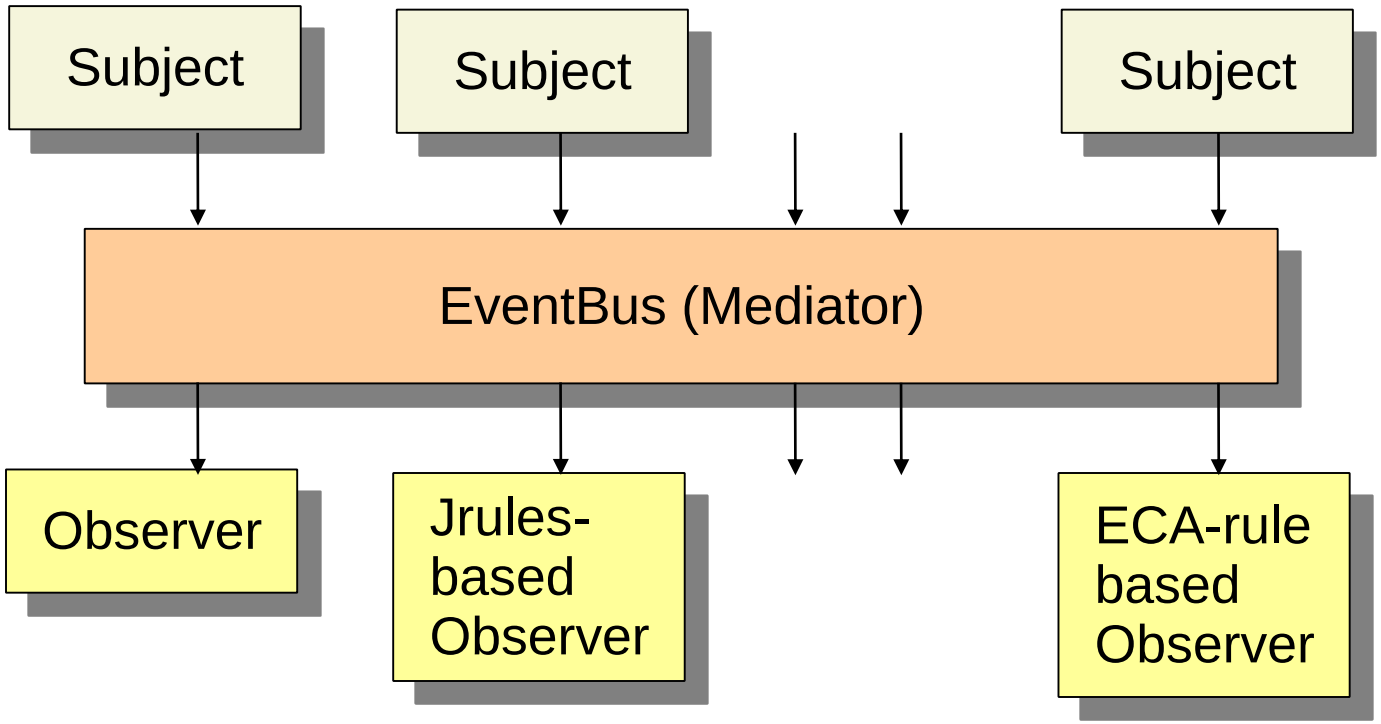
Arch. Style: Event-based Architectural Style (Implicit Invocation Style)

- Components: processes or procedures
- Connectors: Anonymous communication by events
  - Asynchronous communication
  - Dynamic topology: Listeners can dynamically register and unregister
  - Listeners are *implicitly invoked* by events



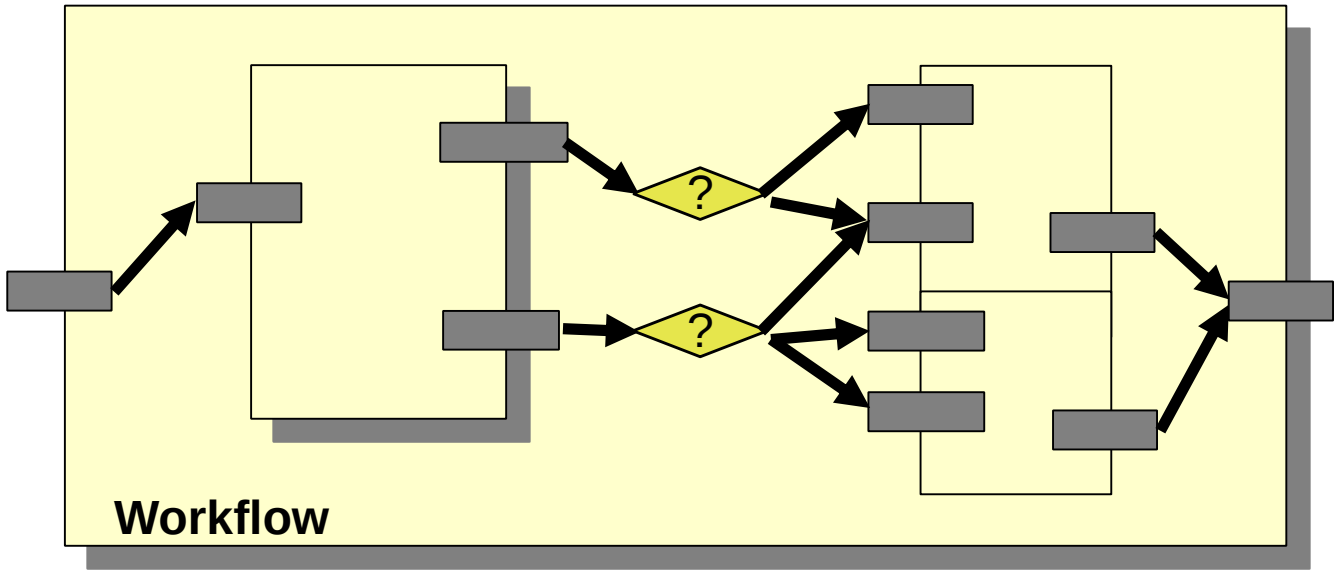
```
<rule name="Free Fish Food Sample">
    <parameter identifier="cart">
        <java:class>org.drools.examples.java.petstore.ShoppingCart</java:class>
    </parameter>
    <parameter identifier="item">
        <java:class>org.drools.examples.java.petstore.CartItem</java:class>
    </parameter>

    <java:condition>cart.getItems( "Fish Food Sample" ).size() == 0</java:condition>
    <java:condition>cart.getItems( "Fish Food" ).size() == 0</java:condition>
    <java:condition>item.getName().equals( "Gold Fish" )</java:condition>

    <java:consequence>
        System.out.println( "Adding free Fish Food Sample to cart" );
        cart.addItem( new org.drools.examples.java.petstore.CartItem( "Fish Food Sample", 0.00 )
     );
        drools.modifyObject( cart );
    </java:consequence>
</rule>
```

- Basis of many interactive application frameworks (XWindows, Java AWT, Java InfoBus, ...)
- See design pattern Observer with Change Manager



Arch. Style: Workflow-Based Systems

- A *workflow* describes the actions on certain events and conditions
  - Formed by a decision analysis, described by ECA rules
- Instead of a data-flow graph as in pipe-and-filter systems, or a control-flow graph as in call-based systems
  - A control-and-data flow graph steers the system
  - The data-flow graph contains control-flow instructions (if, while, ..)
  - This *workflow graph* is similar to a UML activity diagram, with pipes and switch nodes
  - Often transaction-oriented



Application Domains of Workflow Architectures

- Business software
  - The big frameworks of SAP, Peoplesoft, etc. all organize workflows in companies
- Production planning software
- Web services are described by workflow languages (BPEL)
  - More in course “Component-based Software Engineering”

- Processes can be modeled with state machines that react on events, perform actions, and communicate
- Model checking can be used for validation of specifications
- Languages:
  - Esterelle, Lotos, SDL
  - UML and its statecharts
  - Heteregenous Rich Components (HRC)
  - EAST-ADL

- *Protocol engineering*
  - Automatic derivation of tests for systems
- Telecommunication software
- Embedded software
  - In cars
  - In planes
  - In robots

- Data-oriented design is grouped around a input/output/inner data structure
  - or a language for a data structure (regular expressions, finite automata, context-free grammars, ...)
- The algorithm of the system is isomorphic to the data and can be derived from the data
  - Input data (input-data driven design)
  - Output data (output-data driven design)
  - Inner data
- Divide: finding sub-data structures
- Conquer: grouping of data and algorithms to modules
- Example:
  - Jackson Structured Programming (JSP)
  - ETL processing in information systems

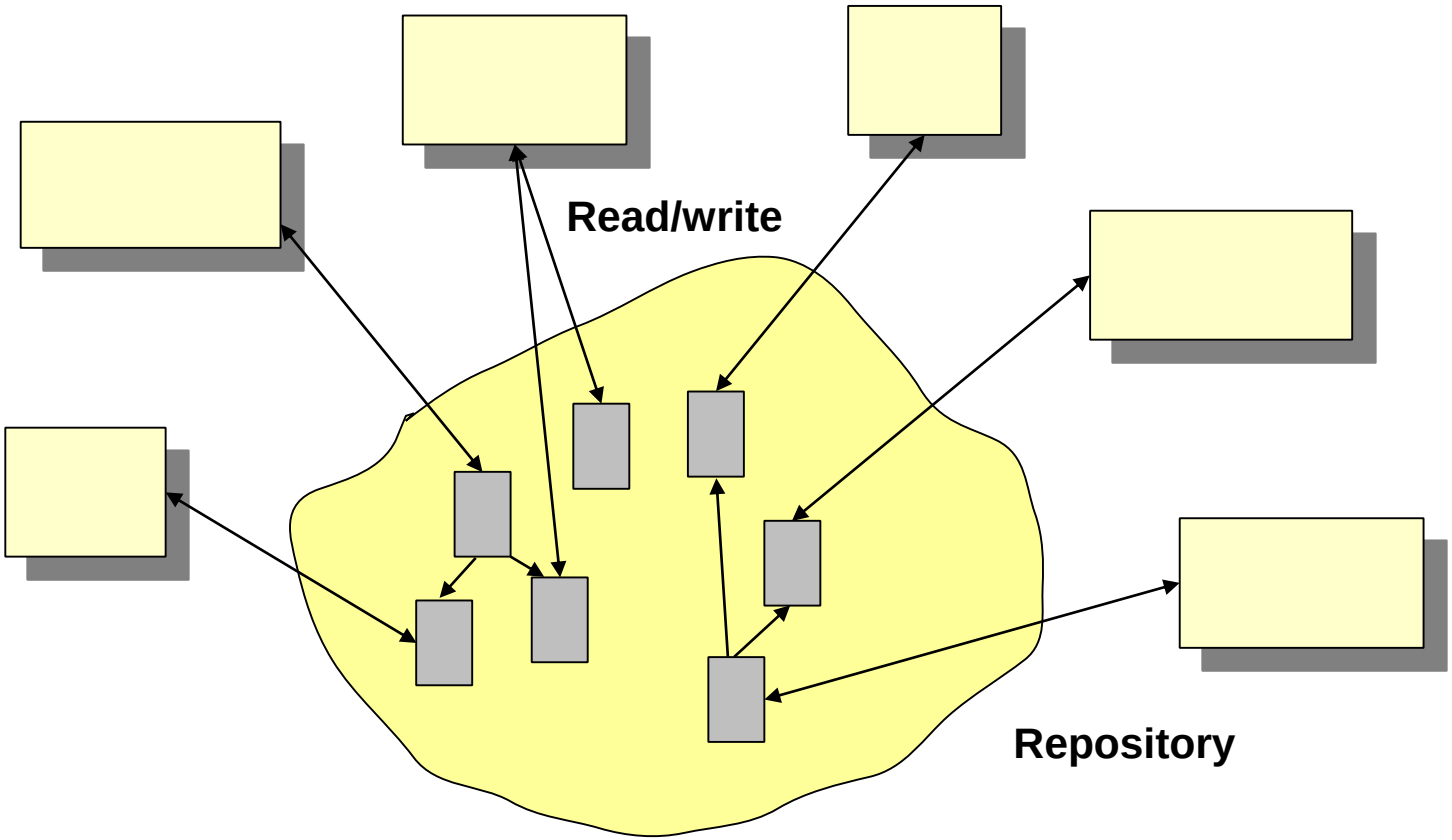
How does the data look like?

Data-Flow Style: Regular Batch Processing (ETL Processing)

- *Regular Batch Processing* is a specific batch-processing style. In such an application, regular domains are processed:
  - Regular string languages, regular action languages, or regular state spaces
- The form of the data can be described by a
  - Regular expression, regular grammar, statechart, or JSP diagram tree
- Often transaction-oriented
- Example:
  - Record processing in bank and business applications:
    - Bank transaction software
    - Database transaction software for business
  - Business report generation for managers (controlling)

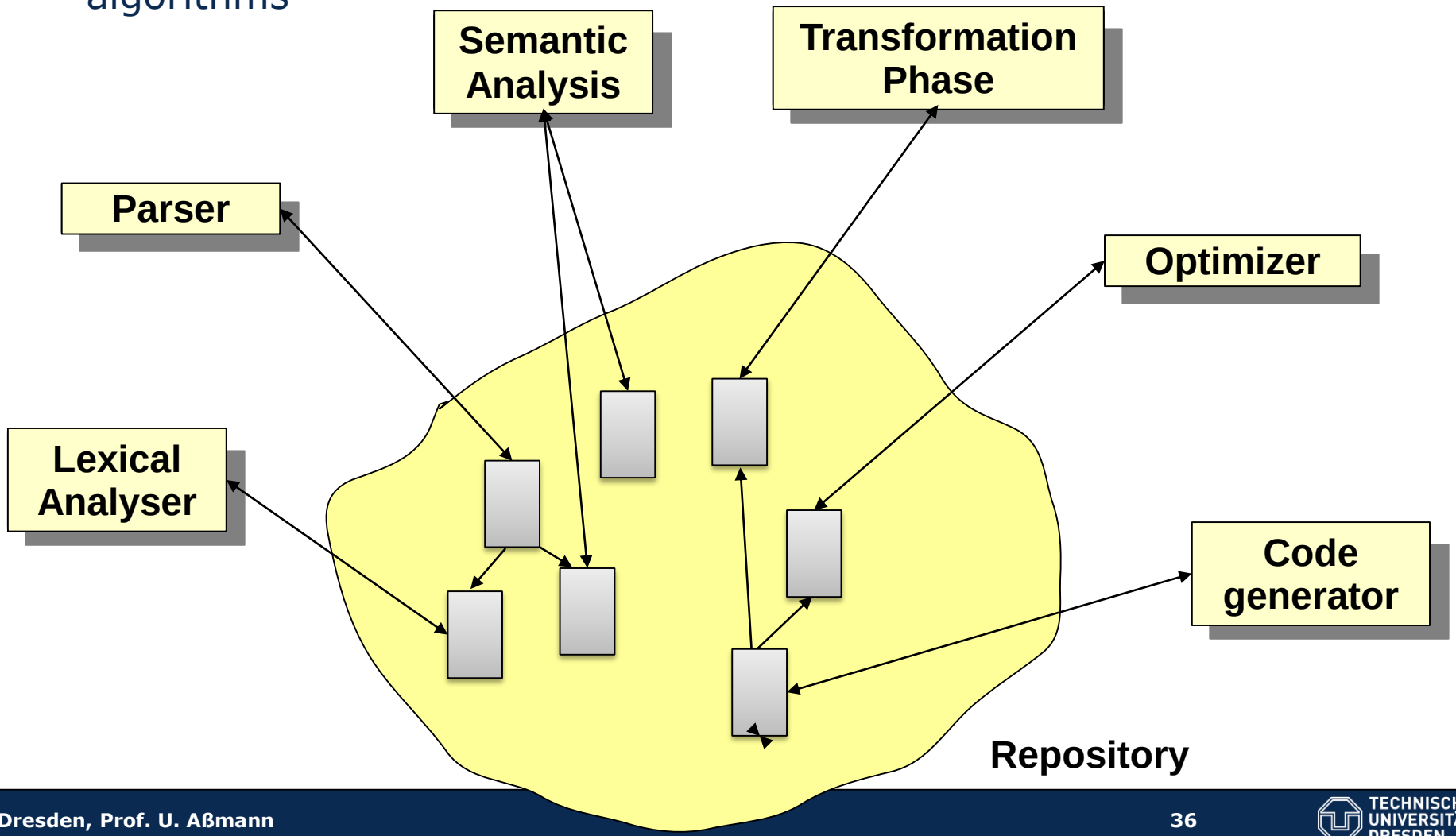
Arch. Style: Repository Systems

- Processing is data-oriented
- Free coordination of components
- Can be combined with call-based style
- Often also state-oriented

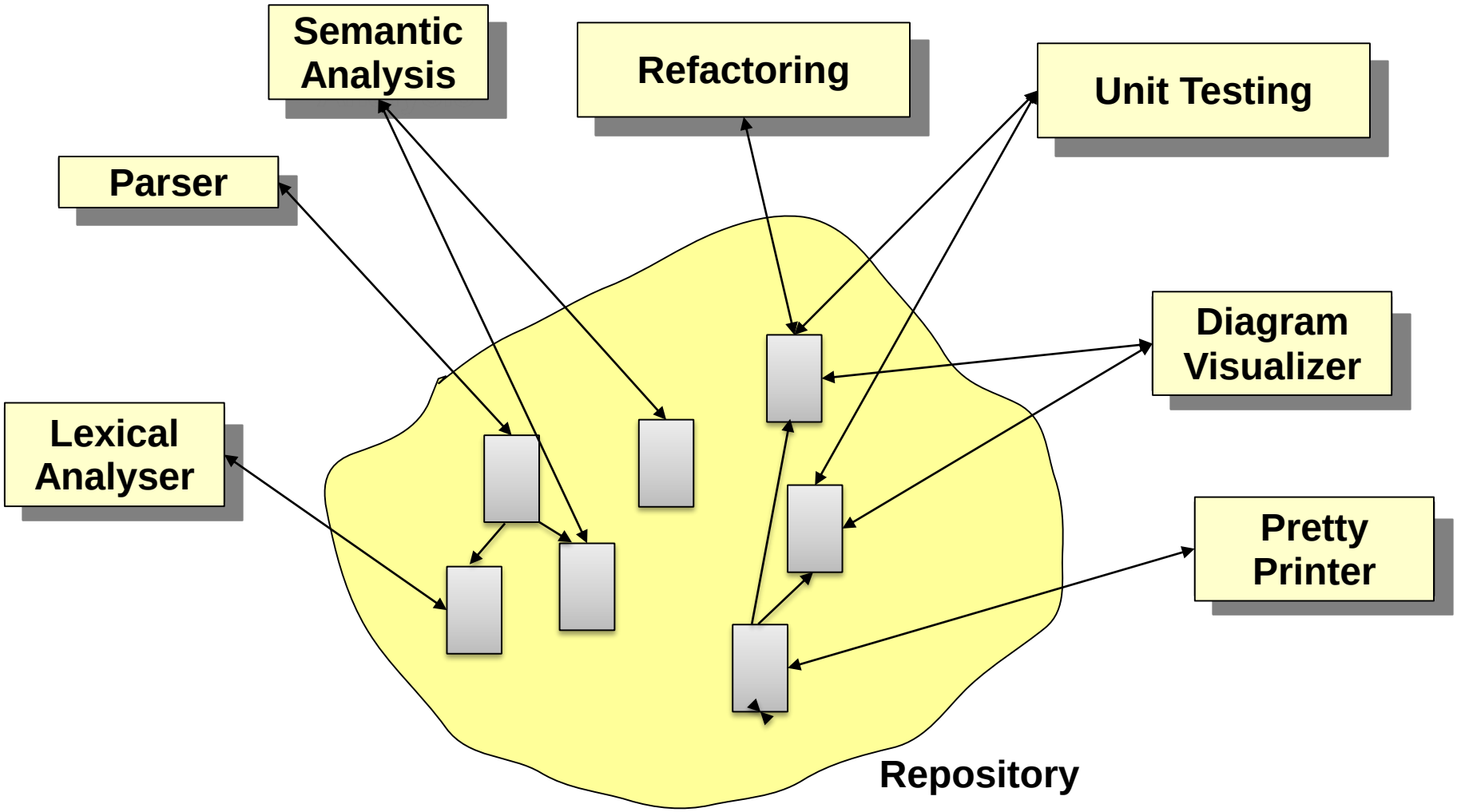


Example: Repository Style in a Compiler

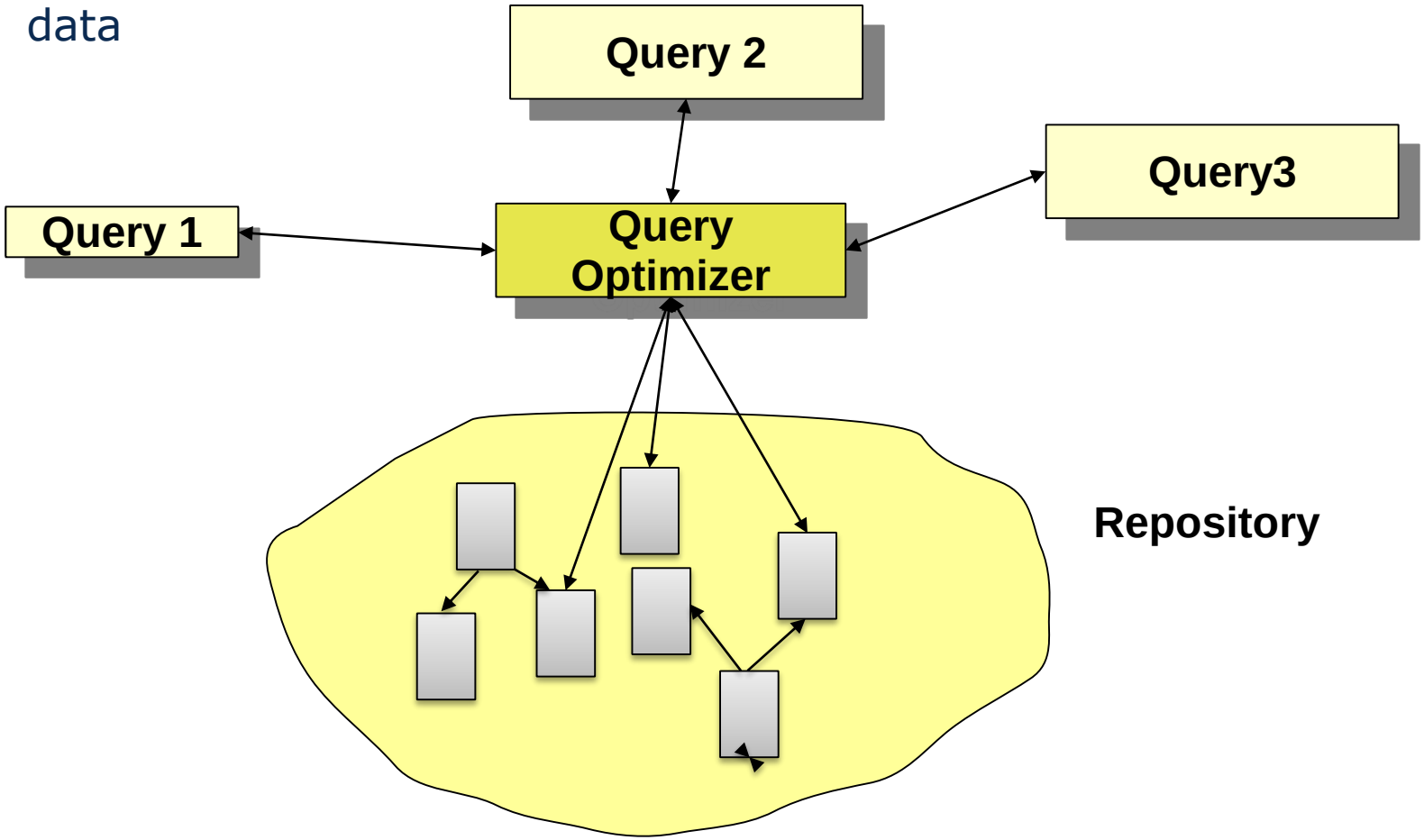
- The algorithms are structured along the syntax of the programs
- The Design Pattern "Visitor" separates data structures from algorithms



- IDE store programs, models, tests in their repository

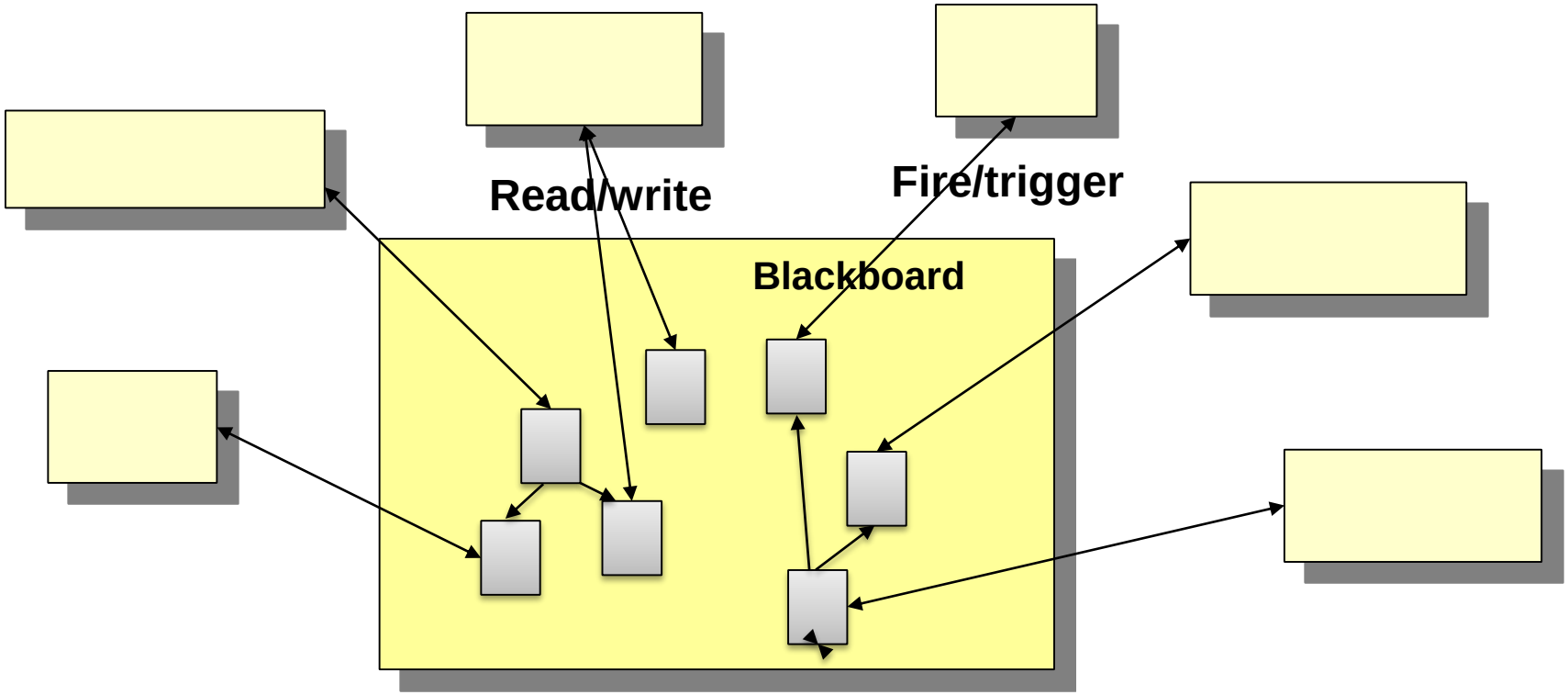


- Algorithms are structured along the relational data
- Data warehouse applications provide querying on multidimensional data



Blackboard Style

- The blackboard is an active repository (i.e., an active component) and coordinates the other components
  - by event notification or call
- Dominant style in expert systems



- Focus is on the HAS-A (PART-OF) relation
  - Focus is on *parts*, i.e., on an hierarchical structure of the system
- Divide: finding subcomponents (parts)
- Conquer: grouping of components to larger components
- Example:
  - Design with architectural languages (such as EAST-ADL)
  - Design with classical component systems (components-off-the-shelf, COTS), such as CORBA, EJB or AutoSAR
- However, many *component models* exist
- Separate course "Component-based software engineering (CBSE)"

What are the components (parts) of the system, their structure, and their relations?

- Data and actions are grouped into *objects*, and developed together

- Divide: finding actions with their enclosing objects
- Conquer: group actions to objects

What are the "objects" of the system?

What are the actions and attributes of the objects?

Object-Oriented Design Methods

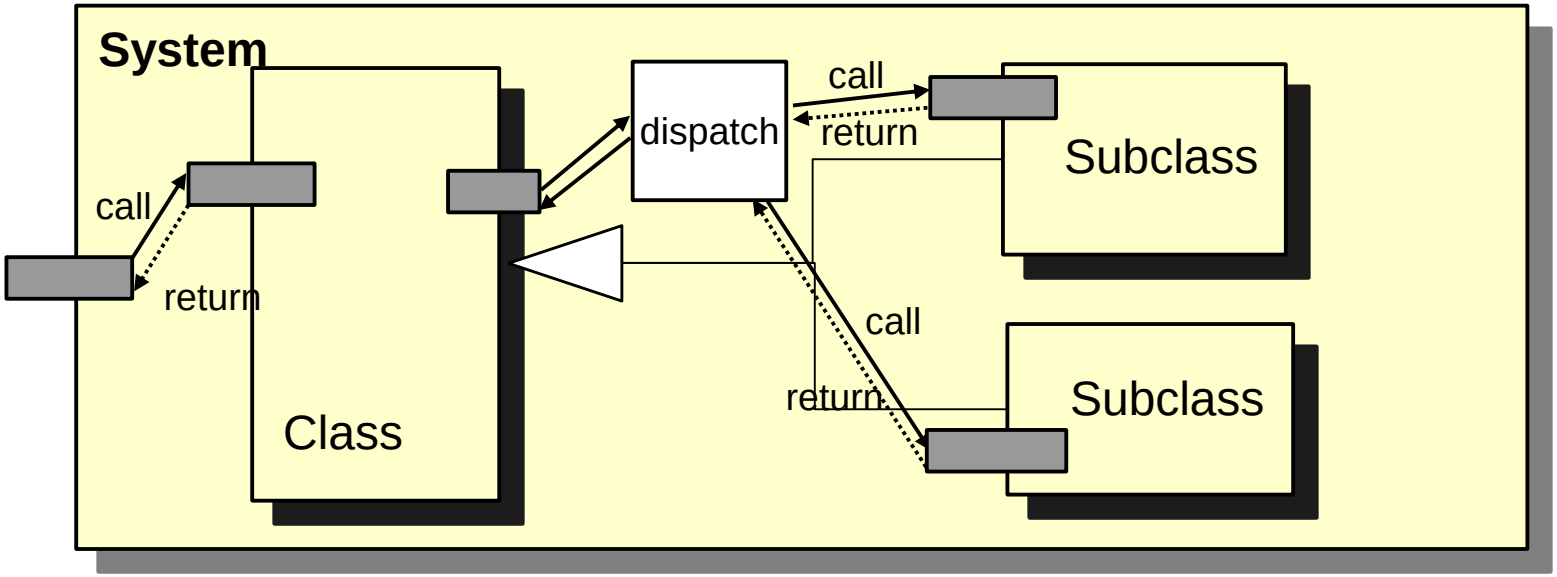
- CRC cards (ST-1)
- Verb substantive analysis (ST-1)
- Collaboration-based design and CRRC (ST-1)

- Booch method
- Rumbaugh method (OMT)
- (Rational) Unified Process (RUP, or Unified Method)
  - uses UML as notation

- Often, OO is used, when the real world should be simulated (simulation programs)

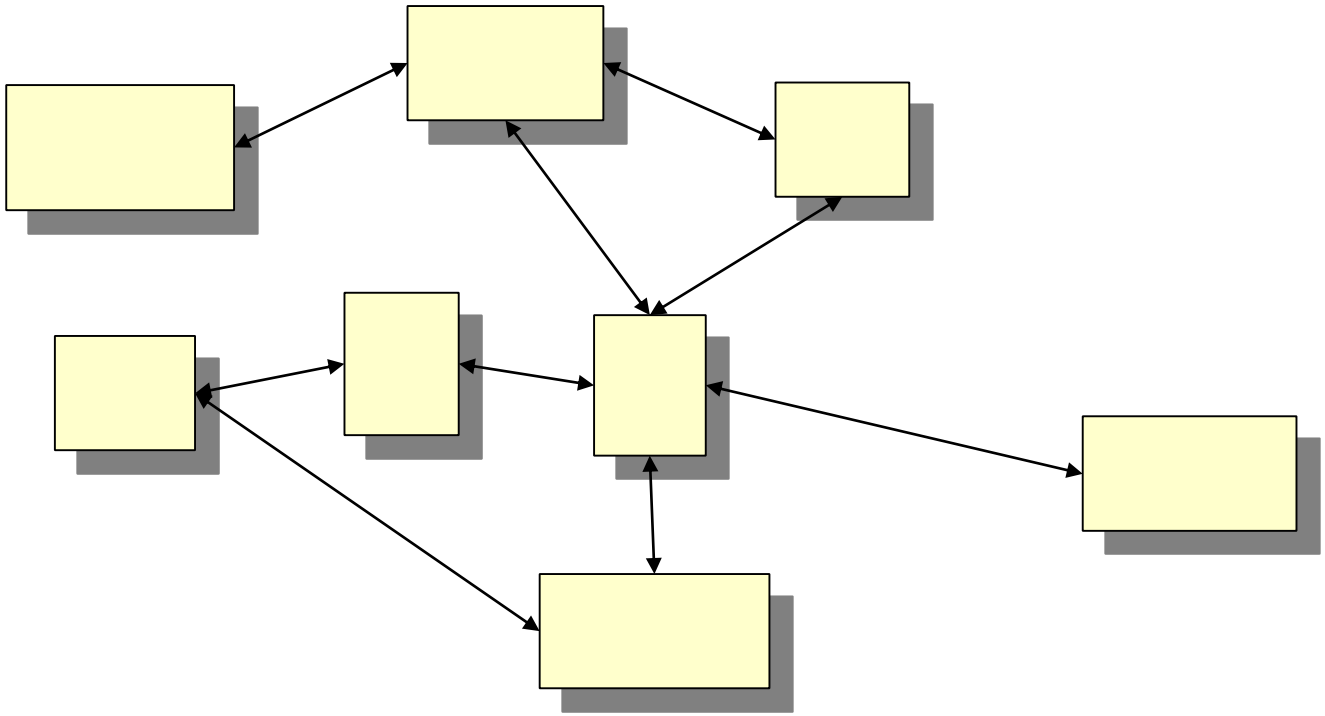
Arch. Style: Object-Oriented Call-Based Architectural Style

- Control flow is symmetric (calls and returns)
- Control flow is **not fixed** (dynamic architecture via polymorphism)
  - Control-flow can be sequential or parallel
- Data-flow can be parallel the call (*push-based system*) or antiparallel, i.e., parallel to the return (*pull-based system*)



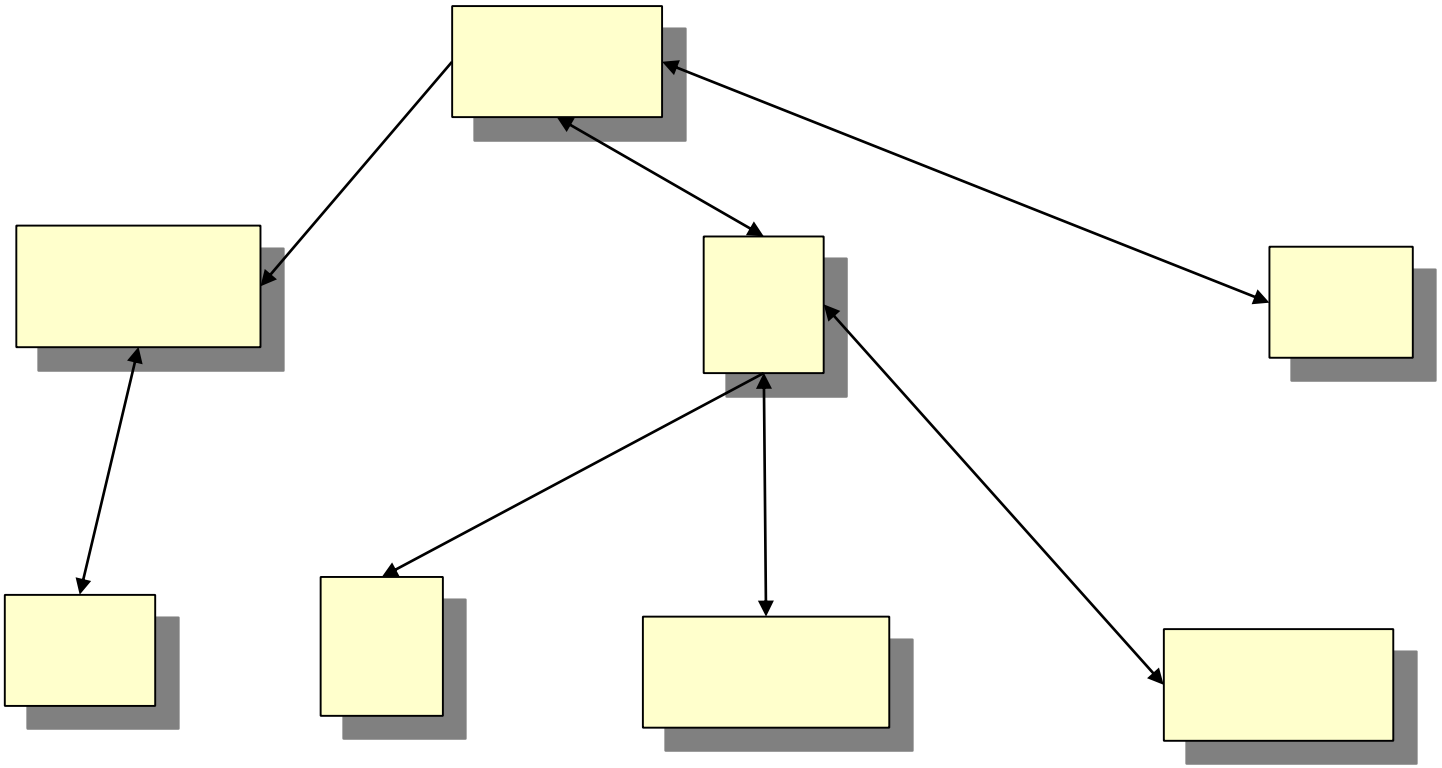
Arch. Style: Object-Oriented Process Systems

- Object-oriented systems can be parallel
- *Actors* are parallel communicating processes
  - Processes talk directly to each other
  - Unstructured communications



Arch. Style: Process Tree Systems (UNIX-Like)

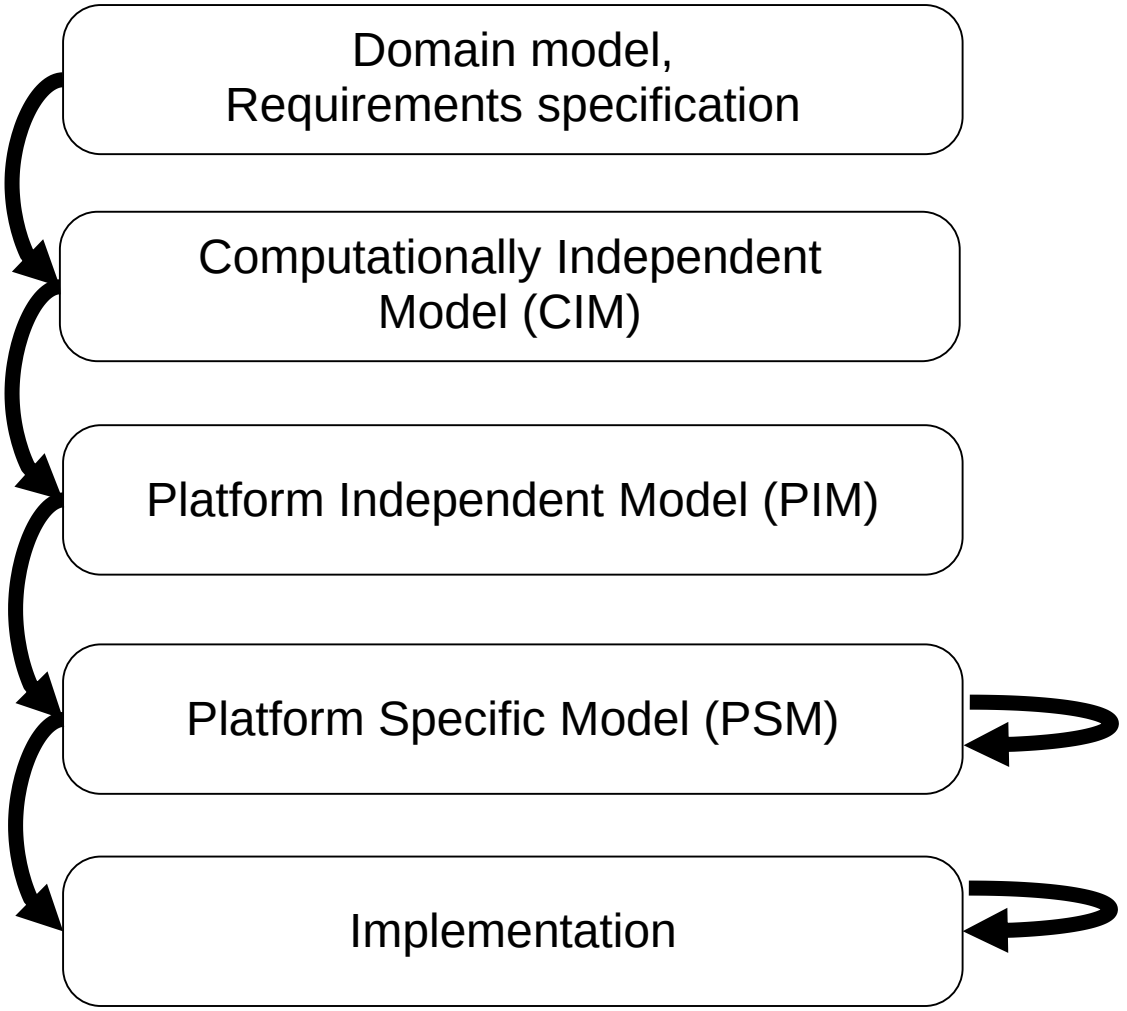
- Processes (parallel objects) are organized in a tree
  - and talk only to their descendants



- We start with an initial, abstract design that meets the requirements
  - The context model and the top-level architecture
- The implementation is achieved by an iterative transformation process, starting from an initial design
  - Refinement-based development
  - Refactoring-based development uses symmetry operations (refactorings)
  - Semi-automatically deriving a final design
- Divide: find steps from the initial to the final design
- Conquer: chain the steps
- Note: this design method is orthogonal to the others, because it can be combined with all of them

How should I transform the current design to a better version and finally, the implementation?

Model mappings

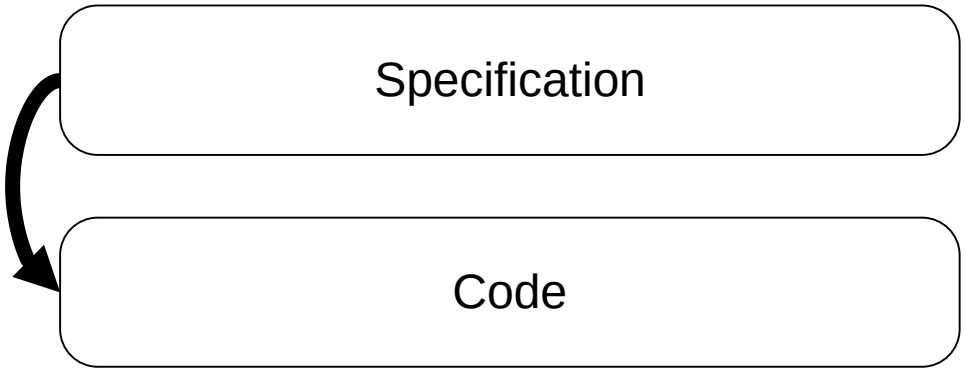


- (aka Generative Programming)
- Specify the solution in
  - a "formal method", a specification language
  - a template which is expanded (generic programming)
  - In UML, which is generated into code by a CASE tool
- Generate a solution with a generator tool that plans the solution
  - Planning the composition of the solution from components
  - Synthesizing the solution

- Divide: depends on the specification language
- Conquer: also
- Fully generative programming is called Automatic Programming

How can I derive the implementation from the design automatically?

- Developing a specification in one of these languages is simpler than writing the code
  - Grammar-oriented development (*grammarware*)
    - Finite automata from regular grammars
    - Large finite automata from modal logic (model checkers)
    - Parsers from Context-free grammars
    - Type checkers, type inferencers from Attribute grammars
    - Type checkers and interpreters from Natural semantics



Automatic Programming

- In automatic programming, a planner plans a way to generate the code from the requirement specifications
  - Using a path of transformations
- A.P. is generative, and transformative, and formal method.

Model-Driven Software Development (MDSD)

- MDSD blends Transformational and Generative design
- Models
  - represent partial information about the system
  - Are not directly executable
  - But can be used to generate parts of the code of a system
- Model-driven architecture (MDA® of OMG) blends Transformational Design and Generative Design
  - See also Chapter "Model-Driven Architecture"

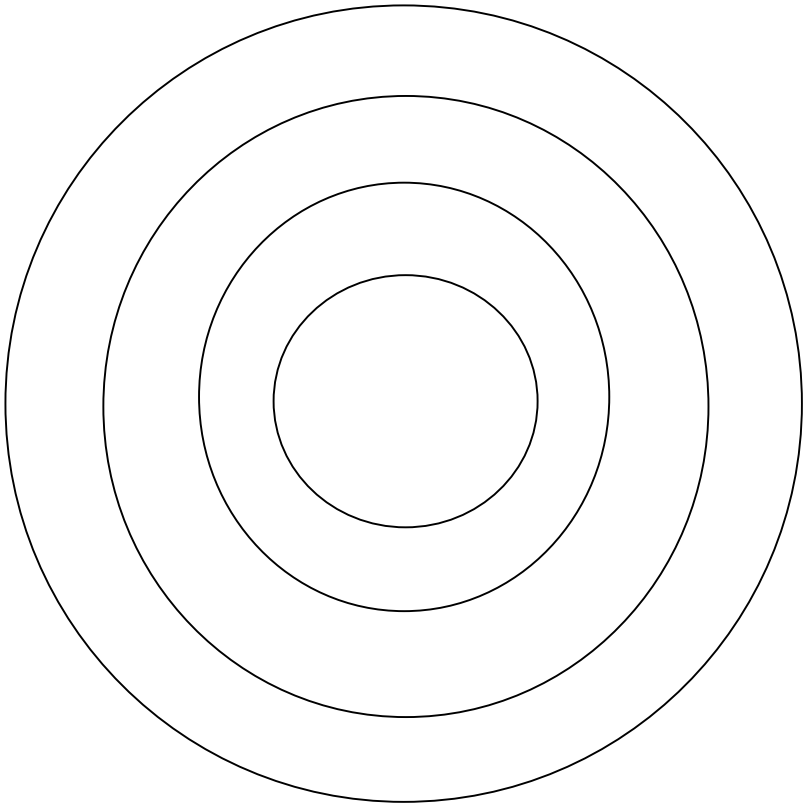
- A *formal method* is a design method that
  - Has a formal (mathematical) specification of the requirements
  - Develops a formal specification of the design
  - The design *can be verified* against the requirements specification
- A formal method allows for *proving a design correct*
  - Very important for safety-critical systems
- Formal methods are orthogonal to the other methods: every method has the potential to be formal
- Important in safety-critical application areas (power plants, cars, embedded and real-time systems)
  - Ex. Petri nets (separate chapter), B, Z, VDM, CSP, CML, …

How can I prove that my design is correct with regard to the requirements?

20.3 ARCHITECTURAL STYLES SPECIFIC TO LAYERS

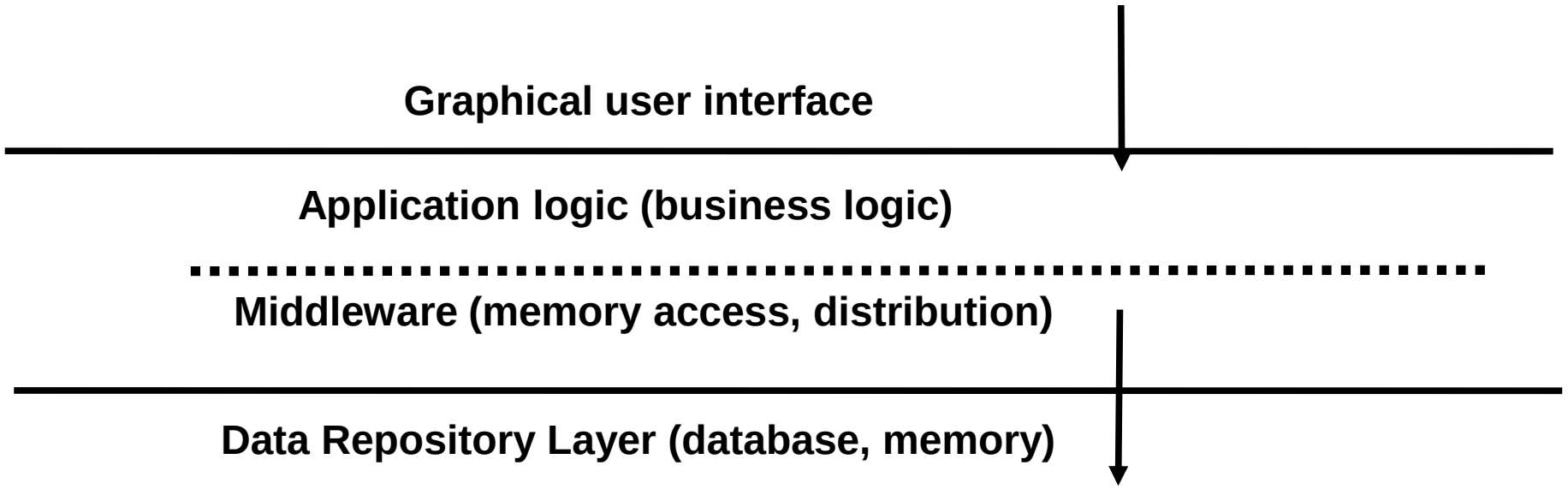
Layered Architecture

- A general approach to reduce the complexity of large systems is to decompose it into **layers**
- Layers can be combined with many architectural styles
- Dominating style for large systems



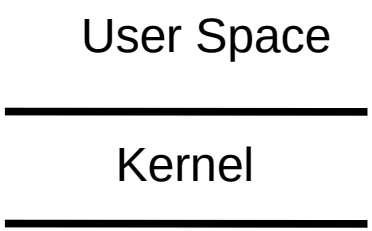
Example: 4-Tier Architectures in GUI-based Applications (BCED)

- Already presented in ST-1
- **Acyclic USES Relation, divided into** 3 (resp. 4) layers:
  - GUI (graphic user interface)
  - Middle layer (Application logic and middleware, transport layer)
  - Data repository (database)

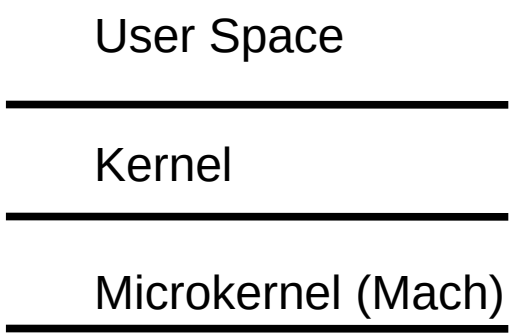


Example: Operating Systems

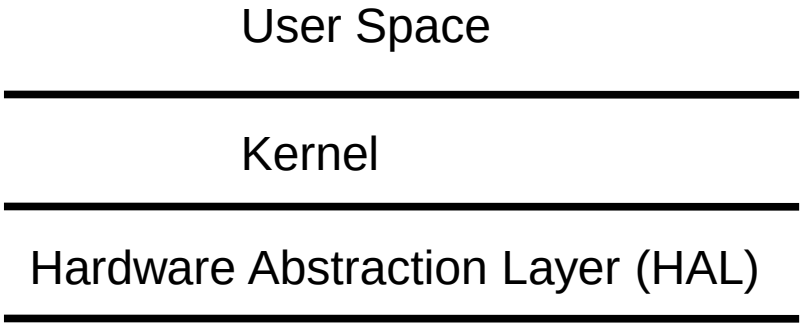
UNIX:



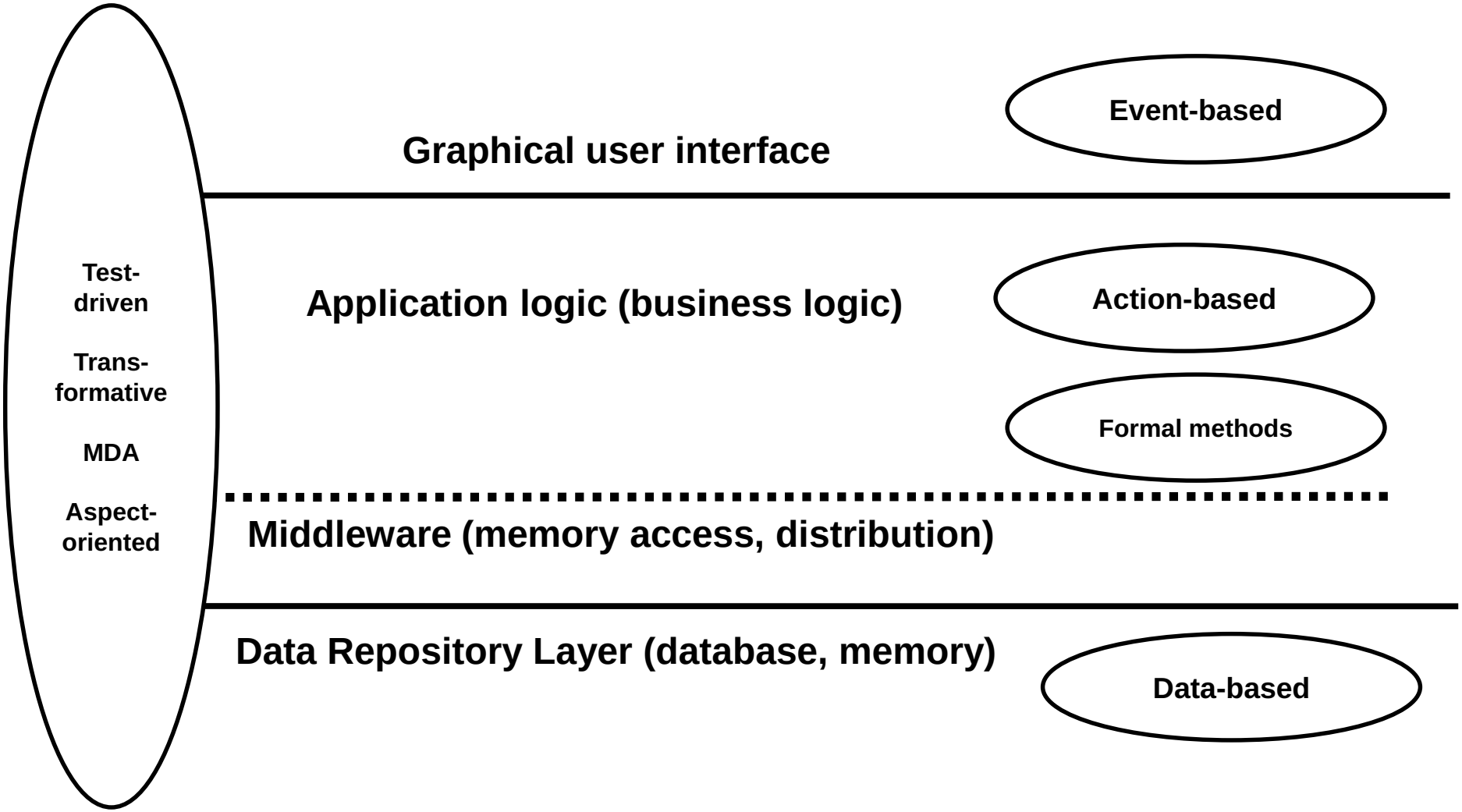
Apple-UNIX:



Windows NT/XP:



Architectural Styles Can Be Layer Specific



Domain-Specific Architectural Styles

- Often an application domain needs its own style, its *reference architecture*
- It's hard to say something in general about those

Important

- An architectural style results from a specific development method
  - Functional, modular design: call-based style
  - Action design: data-flow style, workflow style, regular processing, process trees
  - Object-oriented design: object-oriented call-based systems, client-server, actors (process systems)
  - Uses-oriented design: layered systems
- Specific layers need specific styles
- Reliable systems need specific styles
- The dedicated engineer knows when to apply what

Summary: Most Important Architecture Styles

- Data flow styles
  - Sequential pipe-and-filter
  - Data flow graph/network
  - Workflow systems (mixed with control flow)
- Call-style
  - Modular systems
  - Abstract data types
  - Object-oriented systems
  - Client/service provider
- Hierarchical styles
  - Layered architecture
  - Interpreter
  - Checker-based Architectures
- Interacting processes (actors)
  - Threads in a shared memory
  - Distributed objects
  - Event-based systems
  - Agenda parallelism
- Data-oriented (Repository architectures)
  - Transaction systems (data bases)
    - Query-based systems
  - Blackboard (expert systems)
  - Transformation systems (compilers)
  - Generative systems (generators)
  - Data based styles
    - Compound documents
    - Hypertext-based

- You are a project manager in Miller Car Radios, Inc
- Your boss comes into your office and says:

"Our competitor Smith Car Radios has a new satellite radio. Their sales are growing, and our customers demand it, too. How quickly can you deliver me a satellite radio?"

Which Design Method for the Satellite Radio?

- Real world objects must be simulated
  - Object-oriented design?
- Events in the real world
  - Event-condition-action based design?
- Flow of data from the satellite to the radio to the user
  - Data-oriented design? data-flow architecture!

General Strategies in Design Processes

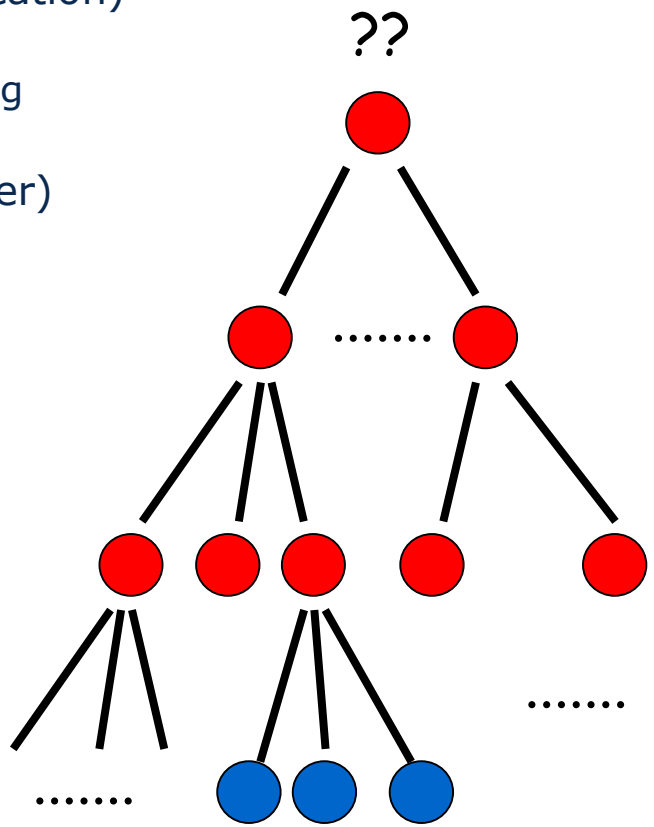
20.4 DESIGN HEURISTICS AND BEST PRACTICES

Heuristic: Lazy or Eager Design

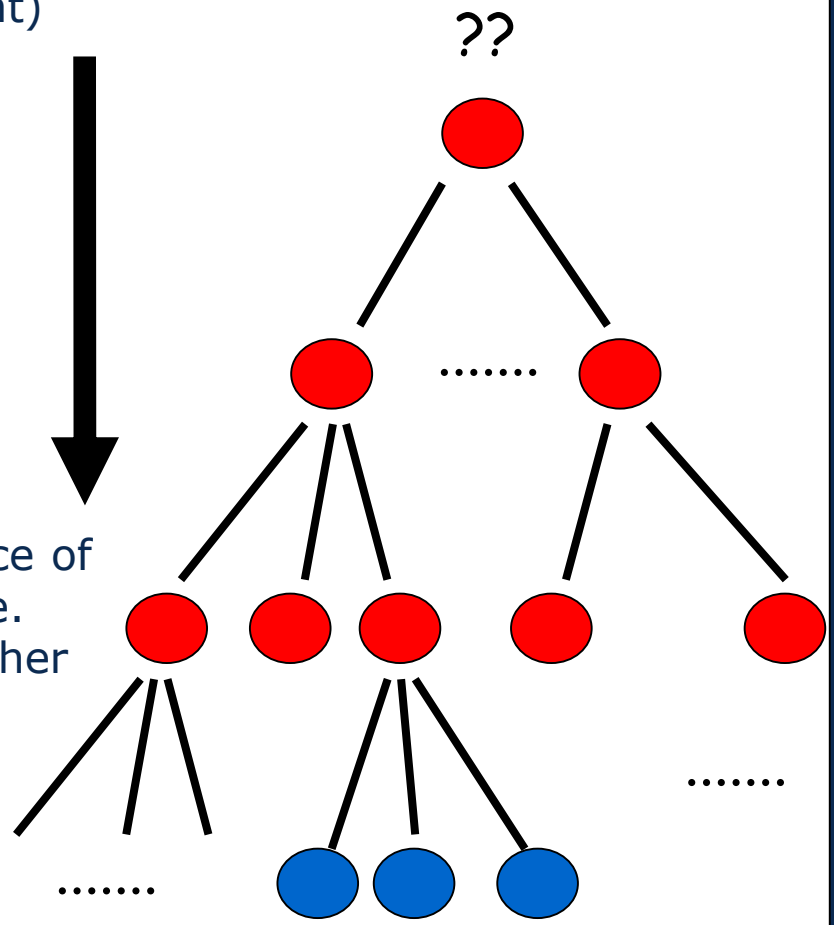
- In case of a difficult design decision
  - (when elaborating, refining, refactoring or changing representation)
  - …**defer** it (lazy design)
    - Iterative Software development methods such as Extreme Programming
  - …**decide** it (eager design)
  - …**anticipate** further developments in the design (anticipatory design)
- Time-boxed design: (SCRUM XP process)
  - Do iterations in fixed time-slots (1 month)
  - Fix requirements only for one time-slot
  - Have it running under all circumstances
  - Update requirements with customer after the time-slot

Heuristic: Divide and Conquer Strategy

- Divide et impera (from Alexander the Great)
  - **divide**: problems into subproblems (simplification)
    - To find solutions in terms of the abstract machine we can employ. When this mapping is complete, we can conquer
  - **conquer**: solve subproblems (hopefully easier)
  - **compose** (**merge**): compose the complete solution from the subsolutions
    - Reuse of partial solutions is possible (then the tree is a dag)
- Where do we begin?
  - Stepwise refinement (top-down)
  - Assemblage (bottom-up)
  - Design from the middle (middle-out, yo-yo)

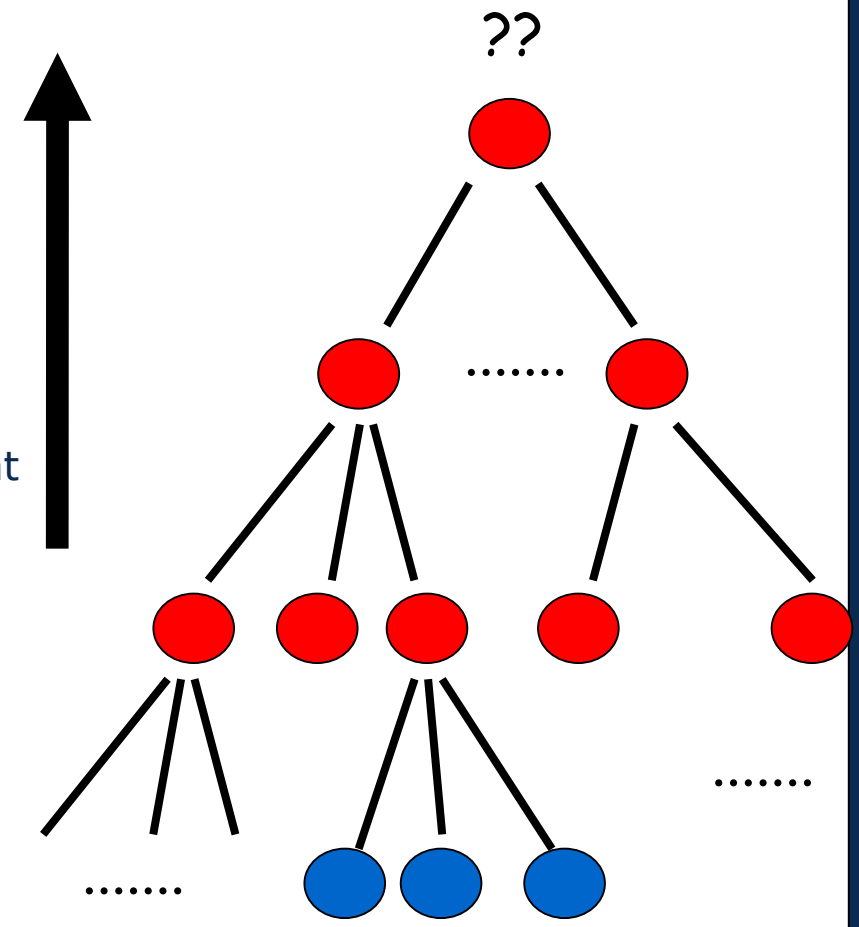


- **Pointwise refinement**
- **Fragment refinement**
- **Control refinement** (operation refinement)
  - We guess the solution of the problem in terms of a higher-level abstract machine
  - We refine their operations until the given abstract machine is reached
- **Data refinement**
  - We may also refine the data structures of the abstract machine
- **Syntactic refinement** does not respect semantics of the original model
- **Semantic refinement** proves conformance of the refined model to the original model, i.e. whether it is semantically equivalent or richer than the original model
- Disadvantage:
  - We might never reach a realization
  - Often "warehouse solutions" are developed, that are inappropriate

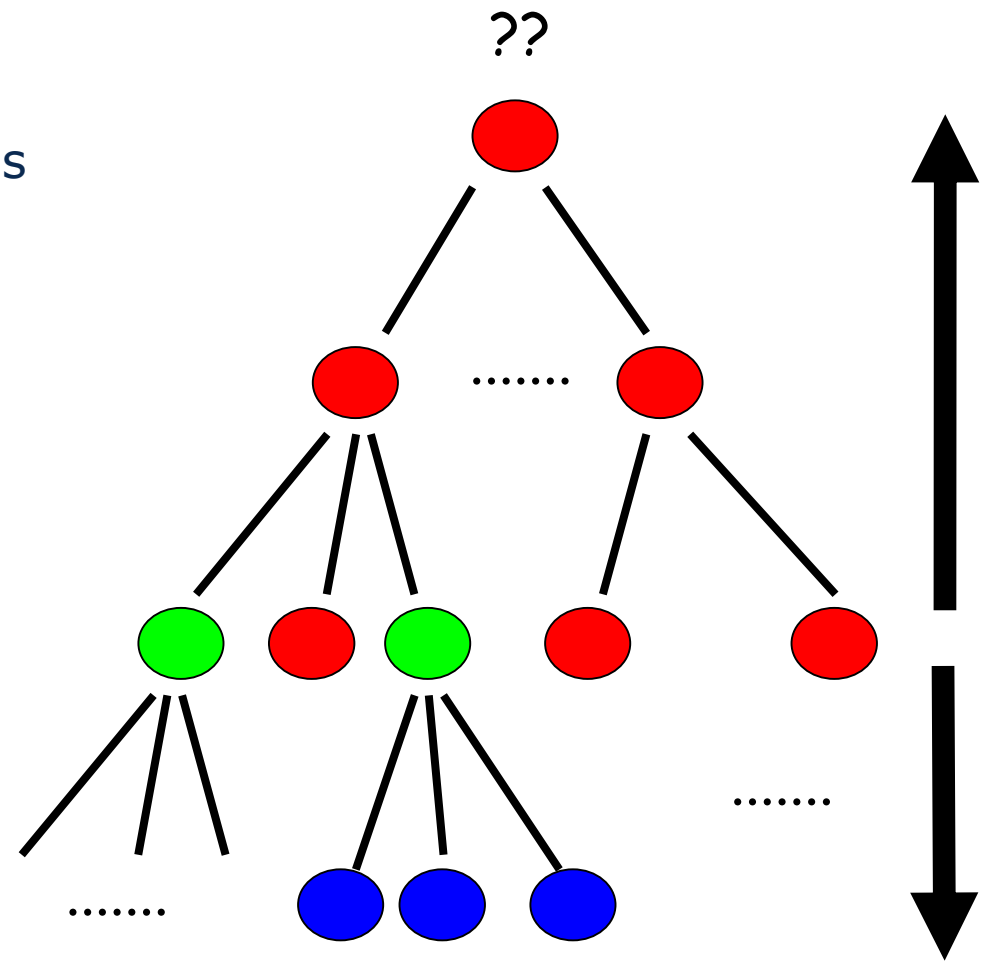


Stepwise Construction (Bottom-up)

- In this case we start with a given abstract machine and
  - assemble more complex operations of a higher-level abstract machine
  - or assemble the more complex data structures
- Good:
  - Always realistic
  - A running partial solution
- Bad:
  - Design might become clumsy since global picture was not taken into account

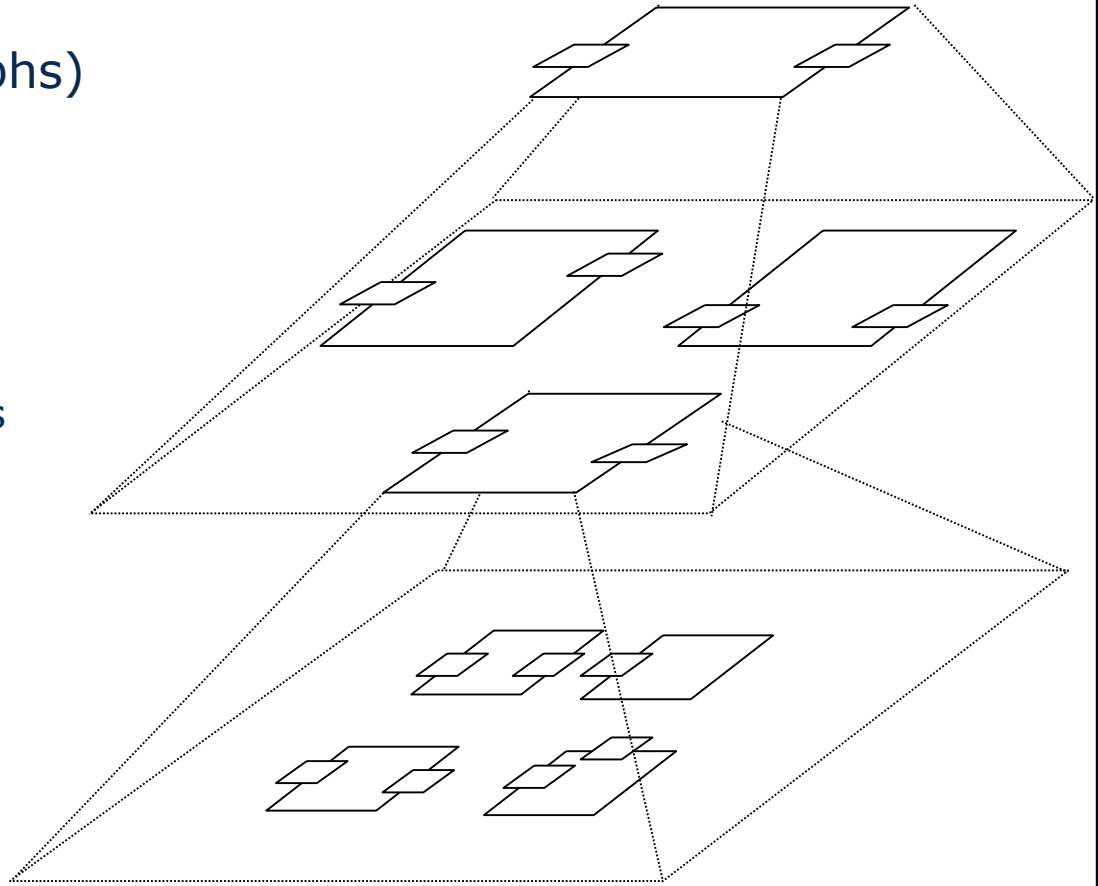


- Fix some subproblems in the middle and solve them by refinement
- Then work your way up
- Often avoids the disadvantages of top-down and bottom-up



Heuristic: Use Hierarchies and Reducible Graphs

- Trees
- Dags (directed acyclic graphs)
  - Can be layered
- Reducible graphs
  - Can be layered too, on each layer there are cycles
  - Every node can be refined independently and abstracts the lower levels



Heuristics on Size

- Limit yourself to a small number of items
  - Never use more than 5 items
    - on an abstraction level of a specification or model
- KISS (keep it simple stupid)
  - Remove all superfluous things, make it fit on 1 page
  - Simplification takes a long time “I didn't have the time to make it shorter”
  - Einstein: "Make things as simple as possible, but not simpler.”
- *Abstraction* is *neglecting unnecessary detail*
  - Focus at one problem at a time and temporarily forget about others
  - Display only essential information
  - Change representation if development strategy changes
  - This leads to design methods or decomposition methods

- Separation of Concerns (SoC)
  - Different concepts should be separated so that they can be specified independently
  - Dimensional specifications: specify from different viewpoints
  - If separated, then concerns can be varied independently

- Example of SoC: Separate Policy and Mechanism
  - Mechanism: The way how to technically realize a solution
  - Policy: The way how to parameterize the realization of a solution
  - If separated, then policy and mechanism can be varied independently

What Have We Learned?

- There is no single “way to the system”
  - Every project has to find its path employing an appropriate design method
- The basic design questions are posed over and over again, until a design is found
  - Select a design method
  - Pose the design method's basic question
  - Perform the design method's process
  - Perform the design method's steps: elaborate, refine, structure, change representation, ...
- If process gets stuck, change design method and try another one!
- Architectural styles are the result of a design process
  - They give us a way to talk about a system on a rather abstract level
  - Architectural styles can be distinguished by the relation of data-flow and control-flow (parallel vs antiparallel)
  - .. and the type of system structuring relation they use