



# 20 Design Methods - An Overview

1. From Requirements to Design
2. What is a Design Method?
3. Overview of Design Methods
  1. Functional Development
  2. Action-Based Development
  3. Component-Based Development
  4. Data-Oriented Development
  5. Object-oriented Development
  6. Transformative Development
  7. Generative Development
  8. Model-Driven Software Development
  9. Formal Methods
  10. Aspect-oriented Development
4. Other Architectural Styles

- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediатеchnik
- Gruppe Softwaretechnologie
- <http://st.inf.tu-dresden.de/teaching/swt2>
- Version 11-0-2, 10.12.11



## Obligatory Readings

- Pfleeger Chapter 5
- Ghezzi Chapter 3



## Secondary Literature

- [Thayer] Richard Thayer. Software Engineering. A curriculum book. IEEE Press
- [Budgen] David Budgen. Software Design: An Introduction. In [Thayer]
- [Thayer&McGettrick] Richard Thayer, Andrew McGettrick. Software Engineering - A European Perspective. IEEE Press
- [Parnas] David Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM Dec. 1972. The classic article on modularity
- [Brooks] Frederick P. Brooks jr. No Silver Bullet. Essence and Accidents of Software Engineering. In [Thayer]. Wonderful article on what software engineering is all about
- Heise Developer Podcast <http://www.heise.de/developer/podcast/>



## Literature

- [Budgen] David Budgen. Software Design. Addison-Wesley. Expands on the Budgen paper. Pretty systematic.
- [Shaw/Garlan] Software Architecture. 1996. Prentice-Hall. Great book for architects.
- [Shaw/Clements] M. Shaw, P. Clements. A Field Guide to Boxology.
- [Endres/Rombach] A. Endres, D. Rombach. A Handbook of software and systems engineering. Empirical observations, laws and theories. Addison-Wesley. Very good collection of software laws. Nice!



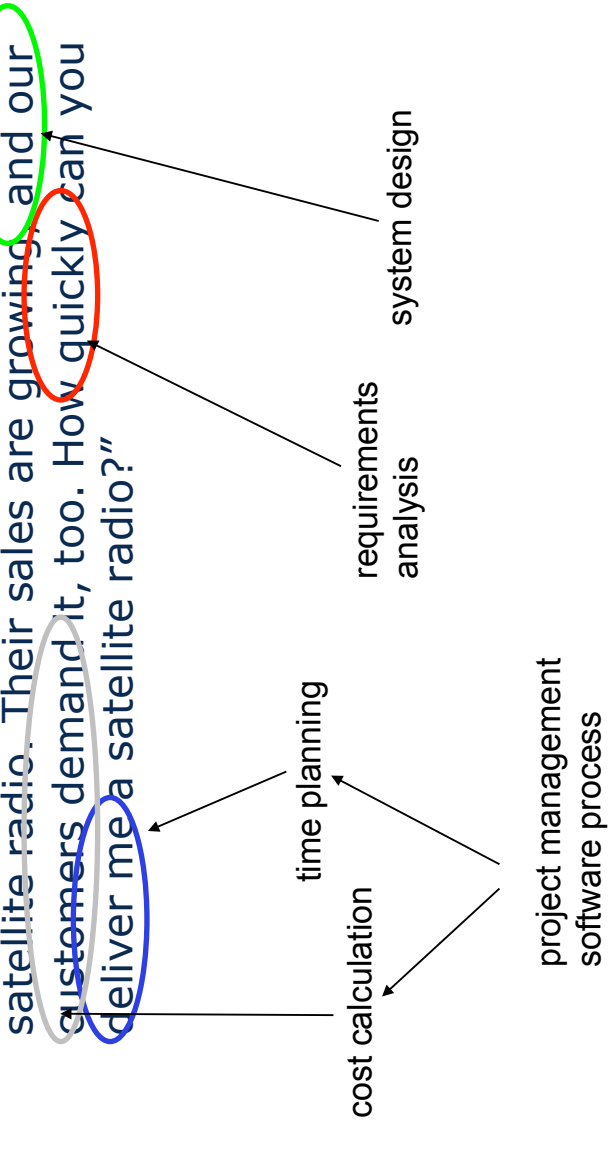
## Goals

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

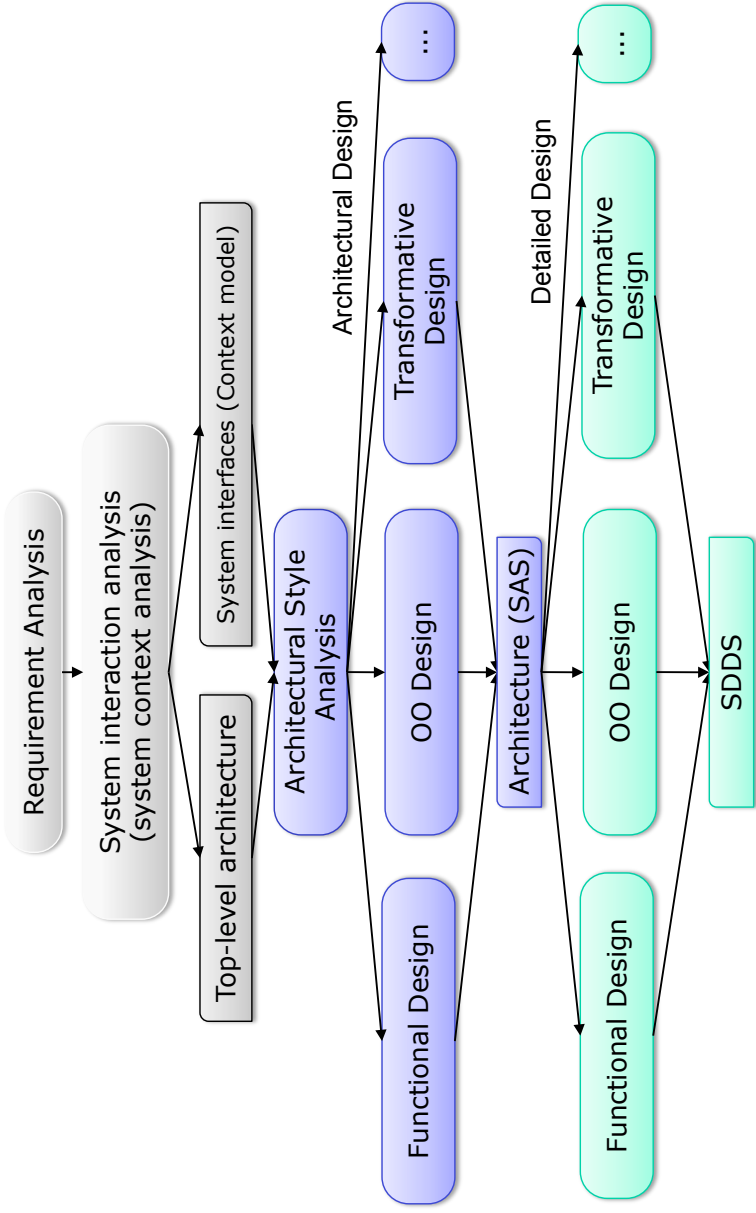


## Scenario

- 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?"

- 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?"
- 
- cost calculation      time planning      requirements analysis      system design
- project management software process

- "Design produces a workable solution to a given problem" [Budgen]
- "Design is the description of a solution" [Pfleeger]
- "The Design Process is the creative process of transforming the problem into a solution" [Pfleeger]
- 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.)

- The Software Requirement Specification (SRS) contains a list of things the system has to fulfill
- Example [Richard Fairley, Software Engineering]
- Usually, specification languages are the same or similar for requirements and design
- 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



- **Conceptual abstraction level**
  - Conceptual instead of technical
  - Coarse grain instead of detailed
- **Design dimensions**
  - Structure (part-of relations, is-a relations)
  - Function (types, interfaces)
  - Behavior
- **System components and their interfaces**
  - Contract specifications of modules: how to use a module?
  - What should it take, what deliver (pre- and postconditions)
- **Component relations**
  - Uses, is-a, part-of, behaves-like
  - Connections
- **Architectural styles (architectural patterns)**
  - Coarse grain patterns of the architecture in terms of control flow and data flow
  - Constraints of modules, relations, and connections
- **Design patterns**
  - Micro-structures in the design model, mostly on the collaboration of 2-5



- **SDDS = Software Detailed Design Specification**
- **Fine-grained design**
  - Technical instead of conceptual
  - Sketch of the implementation with pseudo code, statecharts, petri nets, or other design notations
  - Behavioral model
  - Tells more about the HOW, without giving the implementation

## 20.2 DESIGN METHODS

### A Software Design Method (aka Development Method)

... has 3 components [Budget]:

1. Representation part (notation, language)
  - Set of notations in (informal) textual, (semi-formal) diagrammatic, or mathematic (formal) form
2. Process model (“Vorgehensmodell”, “Prozessmodell”)
  - Design strategy: A basic design question (focus of refinement)
  - Restructuring methods
  - Consistency checking
3. Set of heuristics
  - General rules of thumb
  - Process-specific rules
  - Process patterns
  - Design patterns
  - Adaptation rules

	Text	Graphics	Math
Paper Specification Languages	Informal Natural language Pseudo-code	Flow chart Data-flow Diagram Entity-Relationship Diagram ER	Vienna Development Language VDL/VDM Z B
Executable Specification Languages	Parseable natural language	Colored Petri nets State machines UML Structure Diagram Statecharts	Larch (with prover) CSP CCS
Programming Languages	ELAN SETL Java Scala C#	Workflow languages (BPEL, BPMN)...	



Generic steps

## 20.2.2 DESIGN PROCESSES

### 20.2.3 Design Process

- A **design process** is a structured algorithm (or workflow) to achieve a design model from a requirement specification
  - A sequence of steps
  - A set of milestones
- The design process starts from *the system's interfaces (context code/)* and refines its internals
- Every design process
  - Contains several central generic steps
  - Uses general design strategies
  - Ends up in a specific *architectural style*

Every design process contains some generic steps

- **Elaboration**
  - Work out a certain aspect of the design model, using an appropriate design notation
- **Refinement**
  - Refine an existing specification/model, replacing abstract parts by details, e.g., add platform-specific details
  - Retain *refinement conditions* such as embedding
  - Abstraction is the opposite of refinement
- **Checking Consistency**
  - Checking business rules and context constraints
- **Restructure (more structure, but keep semantics)**
  - Split (decompose, introduce hierarchies, layers, reducibility)
  - Coalesce (rearrange)
- **Symmetry operations (semantics-preserving, restructuring):**
  - Semantic refinement
  - Refactoring
  - Change Representation (Notation):
    - Simplification (factoring, transitive reduction, facading)
    - Change representation, but keep semantics
    - Transform a certain representation of the model into another one

- Every notation has elaboration, refinement, checking, and structuring operations
- **Hand operations**
  - Split (decompose, introduce hierarchies, layers, reducibility)
  - Coalesce (rearrange)
- **Automatic operations**
  - Graph analysis methods, such as constraints
  - Graph structuring methods, e.g., graph analysis or transformations
  - Text-based specifications can be transformed into ASGs and then structured by graph structuring methods
  - Some notations have specific automatic methods

- Ernst Denert. Software Engineering. Springer, 1991.
- Consequence of Denert's law:
  - if we can split off a concern in an application domain, we arrive at a new standard architecture (architectural style)

**Separation of concerns leads to standard architectures.**

**E. Denert, 1991**

- An **architecture style** employs certain types of concepts
  - Certain types of components with
  - Certain types of connections/connectors
  - And a certain relation between control and data flow
- Architectural styles enable us to talk about the coarse-grain structure of a system
  - Good for documentation and comprehension
  - Good for maintenance
- Architectural Styles vs Design Patterns
  - Design patterns have been called *microarchitectures*
    - They grasp a relationship between several classes of an application, but not of the entire architecture or subsystem
  - Architectural styles are *coarse-grain design patterns*

## What Is In a Style ?

- A style has 5 major concerns, in which it can vary [Shaw/Garlan]
  - **Structural Parts:** components, interfaces (ports), connectors
  - **Control flow**
    - Topology (in which form coordination taken place?)
    - Synchronisation (synchronous, asynchronous)
    - Binding time (When are the components organized?)
  - **Data flow**
    - Topology (How does the data flow?)
    - Continuity (singular, sporadic, continuous, strong, weak)
    - Modus (shared memory, messages, ..)
  - **Interaction** between control- and data flow
    - Isomorphic similar to a data structure
    - Direction (parallel, antiparallel)
  - **Invariants**
    - Features that never change
  - **Analysis features**
    - How can be architecture be analyzed?

- How do I derive at a design for the system?
  - How do I derive at an architectural style for the system?
  - How do I derive a detailed design?
- Most often, after reading the requirements, the system looks like in mist
  - Developers have a bad feeling in their stomach
  - They *feel* their way forward
  - Important is: which questions are asked?
- In design meetings, 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
  - If process gets stuck, change design method and try another one
    - However, be aware, which design method and process you use

- A central *viewpoint* with a *set of concerns*, according to which the system is elaborated
  - Decomposed
  - Refined
  - composed
- An elaboration strategy
- The central question

- A design method relies on a **elaboration strategy**, including a basic question the developer has to pose himself, or the team asks itself
- A different question gives a different design method
- 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
  - **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)
  - Layered abstract machines (see ST-1)
  - Use: when the system has a lot of different functions

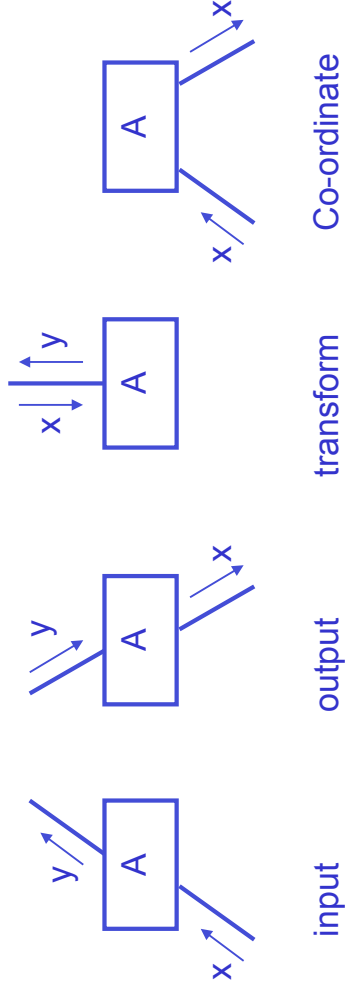
**What are the functions of the software and their subfunctions?**

- "Divide and Conquer" of function
- Decompose system into smaller and smaller pieces
  - Ideally, each piece can be solved separately
  - Ideally, each piece can be modified independent of other pieces
- Reality: each piece must communicate with other pieces
  - This communication implies a certain cost
  - At some point the cost is more than the benefit provided by the individual pieces
  - At this point, the decomposition process can stop

## 20.3.2) Action-Oriented Design

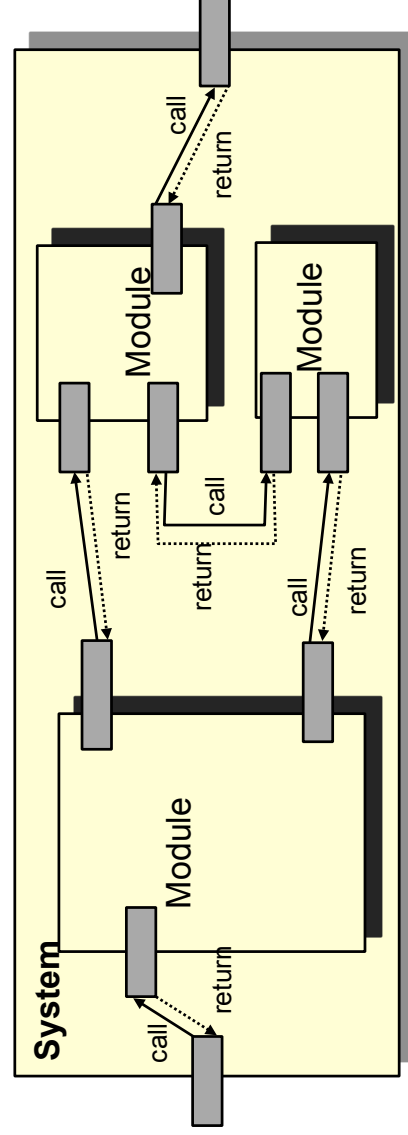
- Action-oriented design is similar to function-oriented design, but actions require *state* on which they are performed (imperative, state-oriented style)
- Divide: finding subactions
- Conquer: grouping to modules
- Examples:
  - Use-case-based development
  - SA, SADT
- Use: when the system maps to a state space, in which actions form the transitions

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



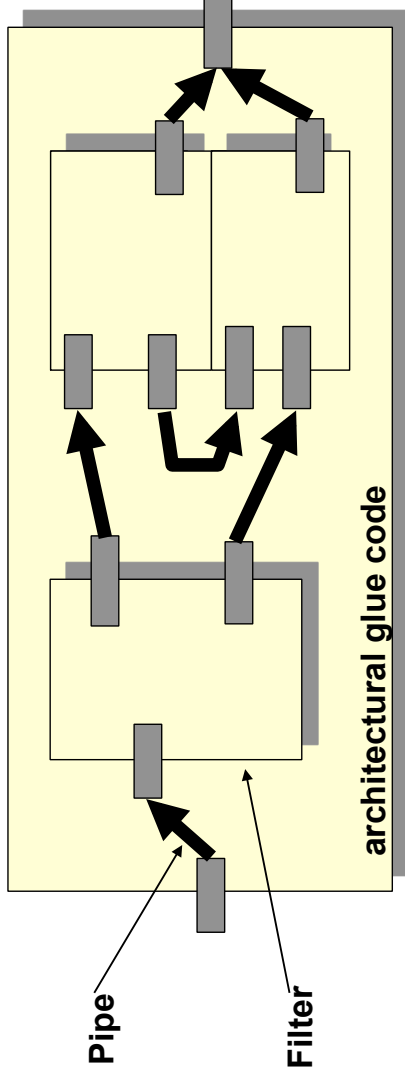
## Result 1: Call-Based Architectural Style

- Components denote procedures that call each other
- Control flow is symmetric (calls and symmetric returns)
- Data-flow can be
  - parallel 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





- 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*)



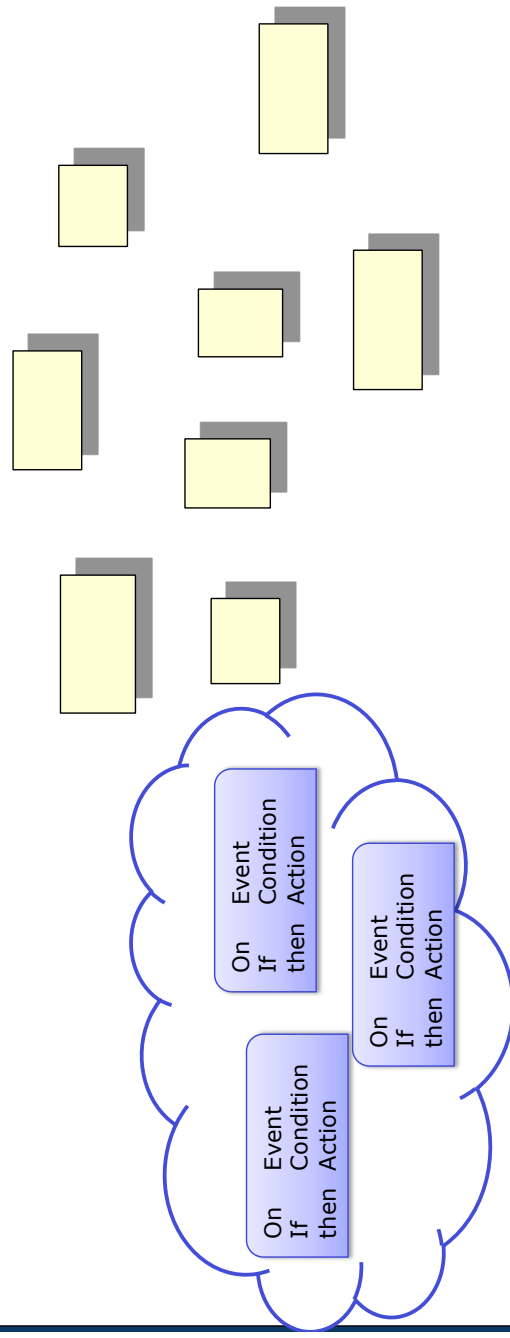
cat server.log | grep "Adding student"

- Data-flow based systems:
- Image processing systems
    - Microscopy, object recognition
  - Digital signal processing systems
    - Video and audio processing, e.g., the satellite radio
  - Content management systems (CMS)
    - Data is stored in XML or relational format
    - Pipelines produce display format
  - Batch-processing systems
  - UNIX shell scripts
  - Microsoft Power Shell
- Call-based systems:
- Object-oriented frameworks
  - Layered architectures

- 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 (SBVR)
- 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?**

- 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



## current contact details

Concept Type: role

Definition: contact details of rental that have been confirmed by renter of rental

## rental

Definition: contract that is with renter and specifies use of a car of car\_group and is for rental period and is for rental movement

## optional extra

Definition: Item that may be added to a rental at extra charge if the renter so chooses

Example: One-way rental, fuel pre-payment, additional insurances, fittings (child seats, satellite navigation system, ski rack)

Source: CRISG ["optional extra"]

## rental actual\_return\_date/time

Concept Type: role

Definition: date/time when rented car of rental is returned to EU-Rent

## rental requests car\_model

Synonymous Form: car\_model is requested for rental

Necessity: **Each rental requests at most one car model.**

Possibility: **The car\_model requested for a rental changes before the actual\_pick-up\_date/time of the rental.**

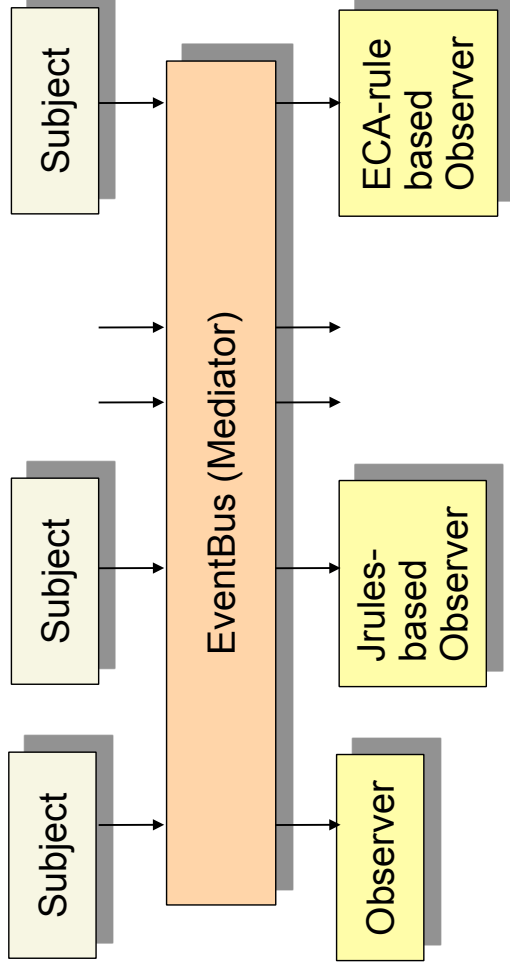
Necessity: **No car\_model requested for a rental changes after the actual\_pick-up\_date/time of the rental**

```

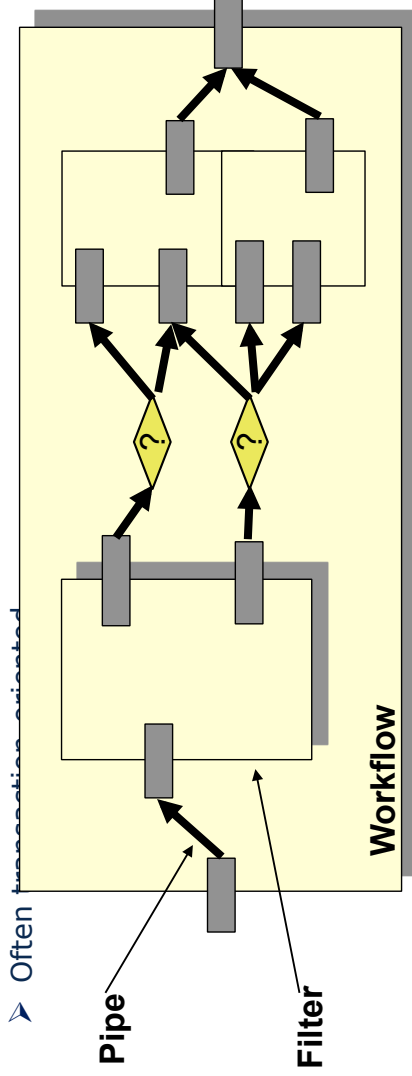
<rule name="Free Fish Food Sample">
<parameter identifier="cart">
  <java:condition>org.drools.examples.java.petstore.ShoppingCart</java:condition>
</parameter>
<parameter identifier="item">
  <java:condition>org.drools.examples.java.petstore.CartItem</java:condition>
</parameter>
<java:condition>cart.getItems(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



- 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



- **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
  - Heterogenous Rich Components (HRC)
  - EAST-ADL

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

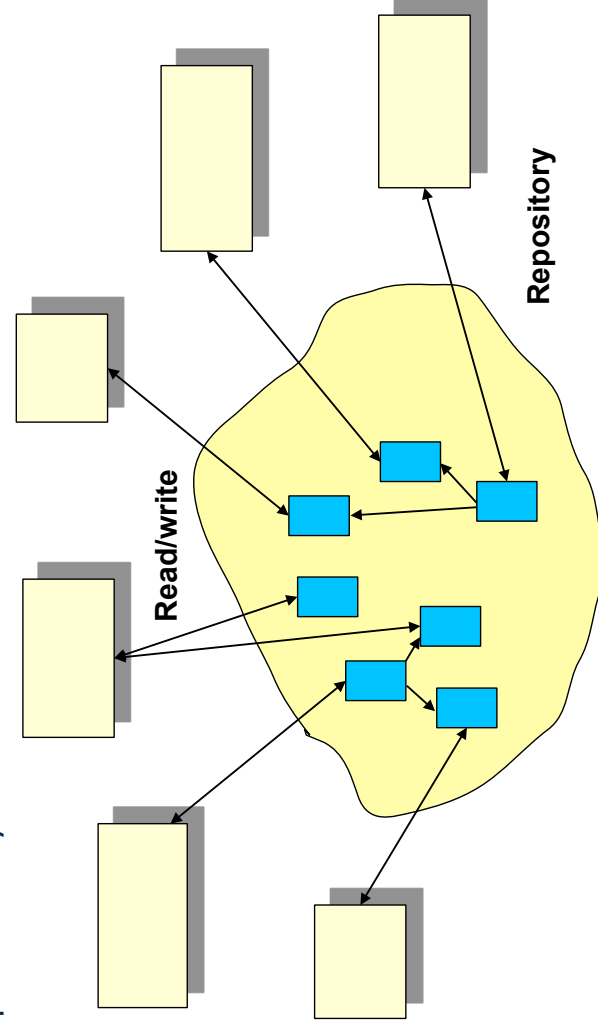
- Data-oriented design is grouped around a input/output/inner data structure
  - or a language for a data structure (regular expressions, finite automata, context-free grammars, ...)
- The algorithm of the system is isomorphic to the data and can be derived from the data
  - Input data (input-data driven 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)

**What does the data look like?**

- *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)

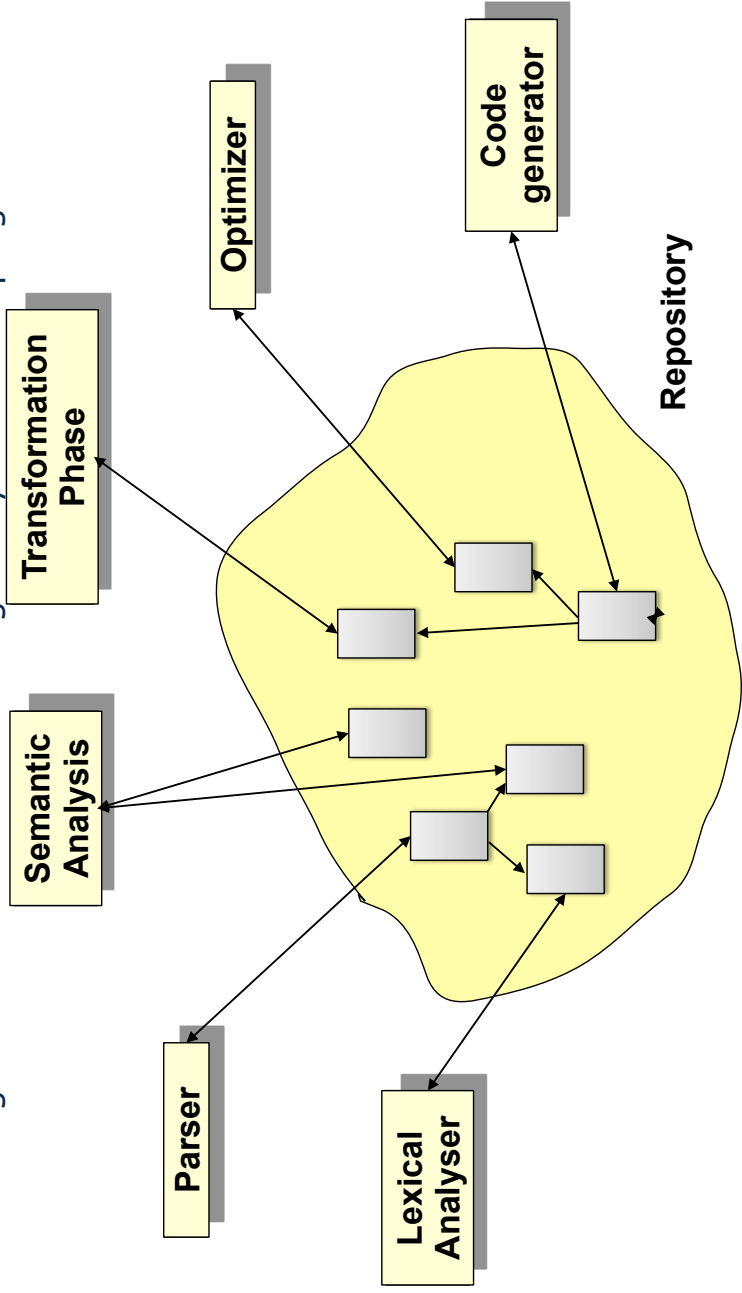
## Arch. Style: Repository Systems (Data Base Systems)

- Processing is data-oriented
- Free coordination the components, can be combined with call-based style or process-style



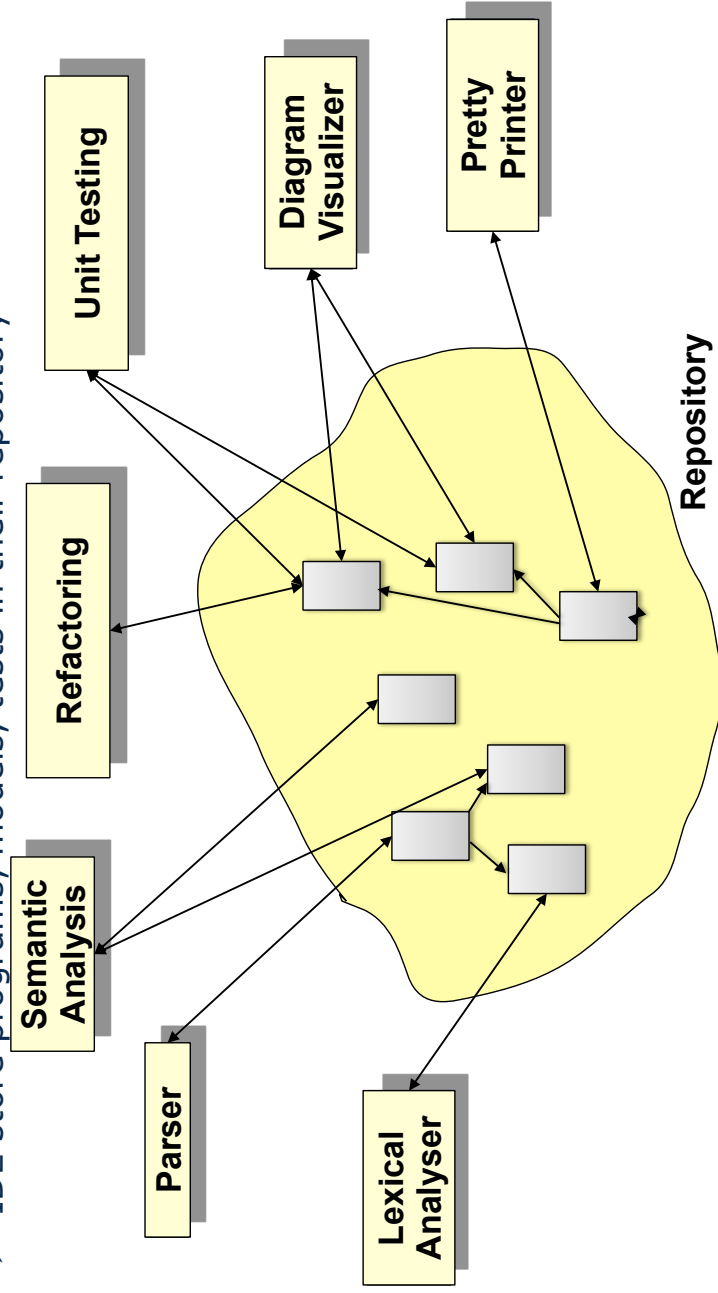
# Example: Repository Style in a Compiler

➤ The algorithms are structured along the syntax of the programs



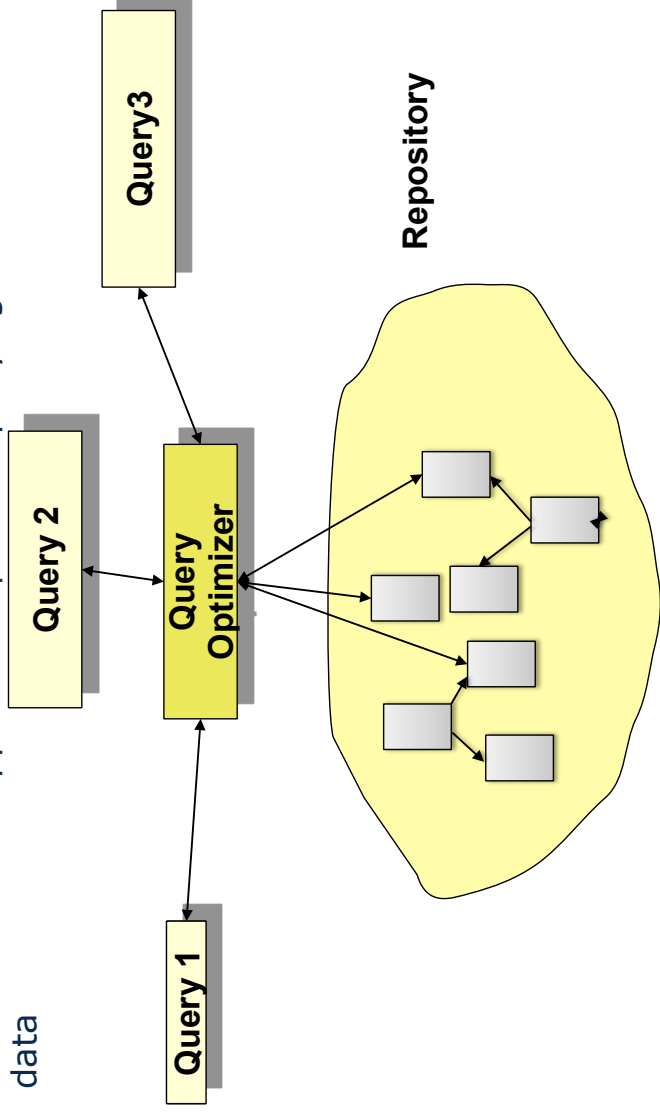
# Repository Style in an Integrated Development Environment

➤ IDE store programs, models, tests in their repository

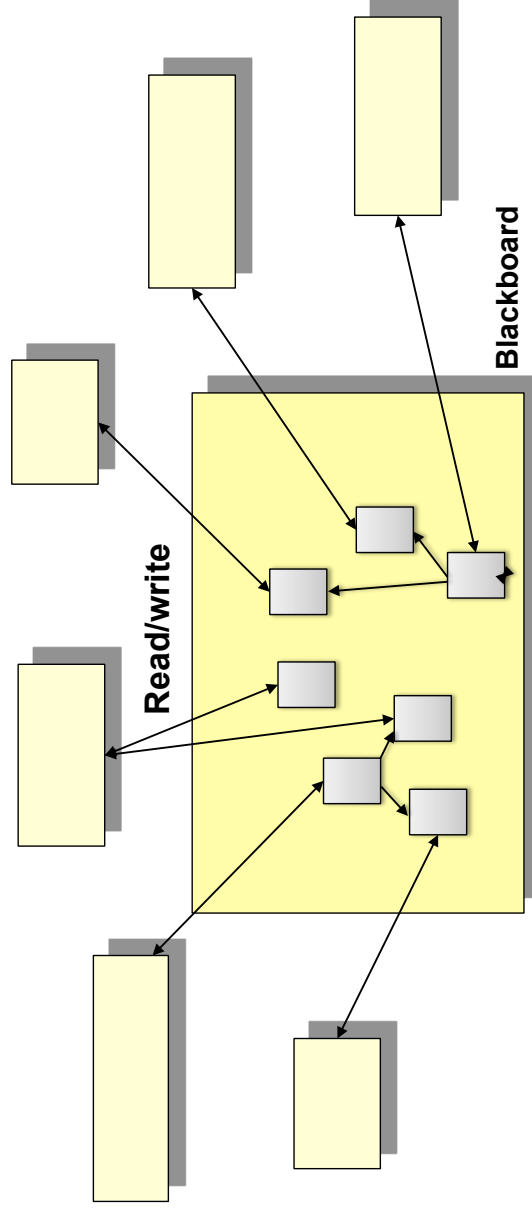




- Algorithms are structured along the relational data
- Data warehouse applications provide querying on multidimensional data



- 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
  - Design with classical component systems (components-off-the-shelf, COTS), such as CORBA
- However, many *component models* exist
- Separate course “Component-based software engineering (CBSE)”

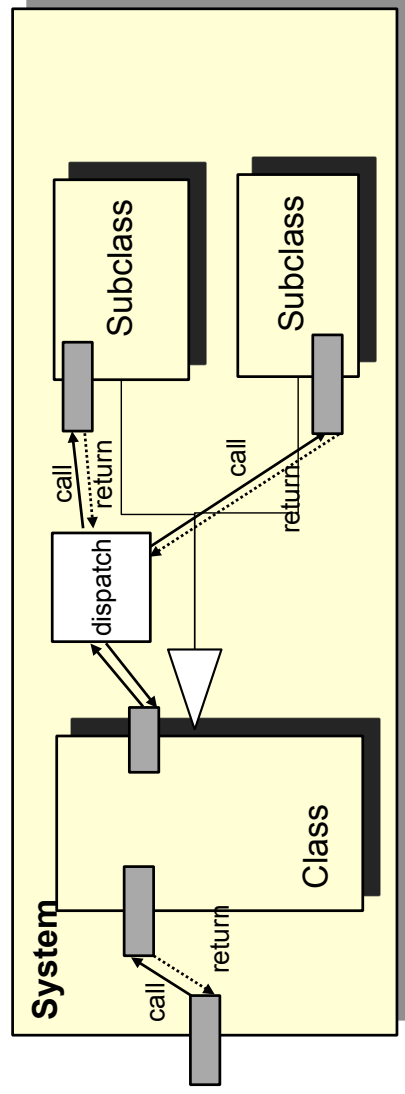
**What are the components (parts) of the system, their structure, and their relations?**

- Data and actions are grouped into *objects*, and developed together
  - Focus is on the is-a and the behaves-like relation
  - A part of the system is like or behaves like another part (similarity)
- Divide: finding actions with their enclosing objects
- Conquer: group actions to objects

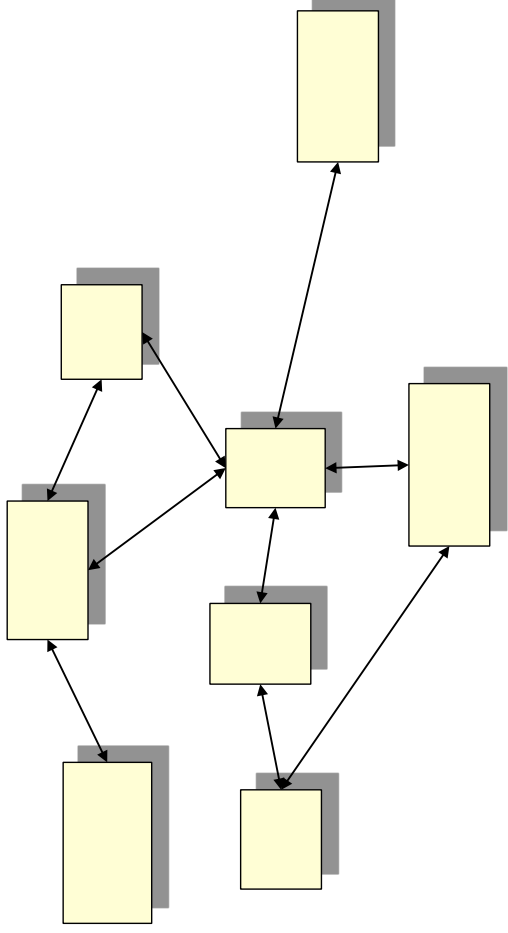
**What are the "objects" of the system?  
What are the actions and attributes of the objects?**

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

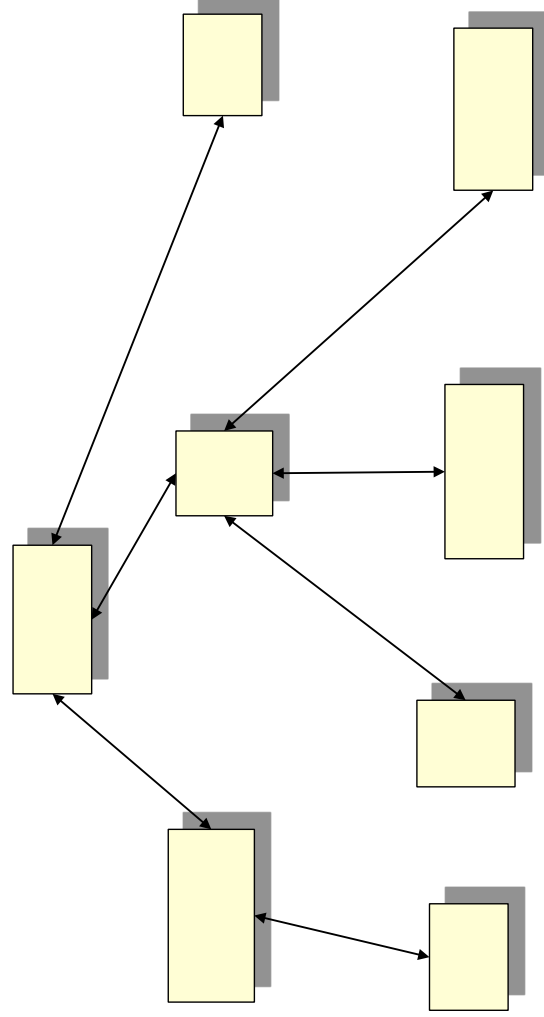
- Control flow is symmetric (calls and returns)
- Control flow is **not fixed** (dynamic architecture via polymorphism)
  - Control-flow can be sequential or parallel
- Data-flow can be parallel the call (*push-based system*) or antiparallel, i.e., parallel to the return (*pull-based system*)



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



- Processes 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: ?

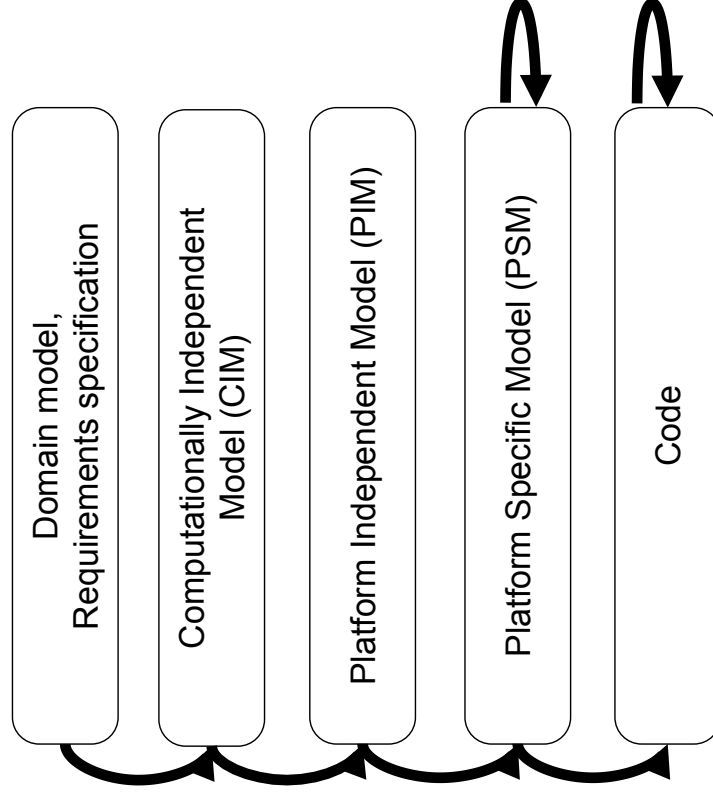
**How should I transform the current design to an better version and finally, the implementation?**

- **Wide spectrum languages** uses rule-based transformation systems and transformation planners
  - The **semantic refinements** are refactorings which **lower** expressive expressions to low-level
  - This starts at the requirement specification and refines (under proofs of correctness) expressive expressions to executable programs
  - Semantics can be proven in different forms, e.g., with Hoare logic, Dynamic logic, or denotational semantics
- **Examples**
  - CIP-L (Munich University)
  - F. L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner. Towards a wide spectrum language to support program specification and program development. SIGPLAN Notices, 13(12) 15-24, 1978.
  - SETL (J. Schwartz, New York University)
  - KIDS (Kestrel institute)

- More informal and incremental process: Extreme Programming (XP)
  - Based on refactorings for structural improvements, but not particularly for lowerings
  - Refactoring can be supported by refactoring tools
  - Every requirement is implemented and tested in separation
  - Continuous testing and continuous integration (test-driven development)
  - Customer is involved (customer-driven development)
  - Permanent review with pair programming

## Model-Driven Architecture as Transformational Design Method

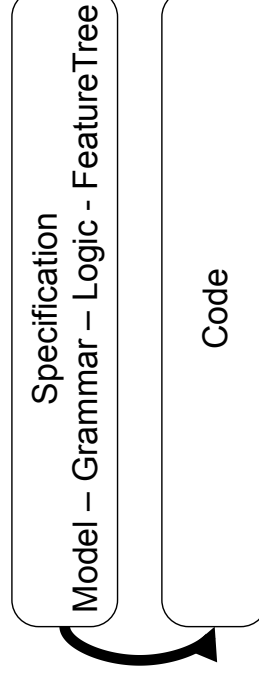
Model mappings



- (aka Automatic Programming, 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

**How can I derive the implementation from the design automatically?**

- Examples:
  - 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
    - Optimizers from graph rewrite systems
  - Feature-oriented development (FODA): specify *feature trees* and derive the components from them



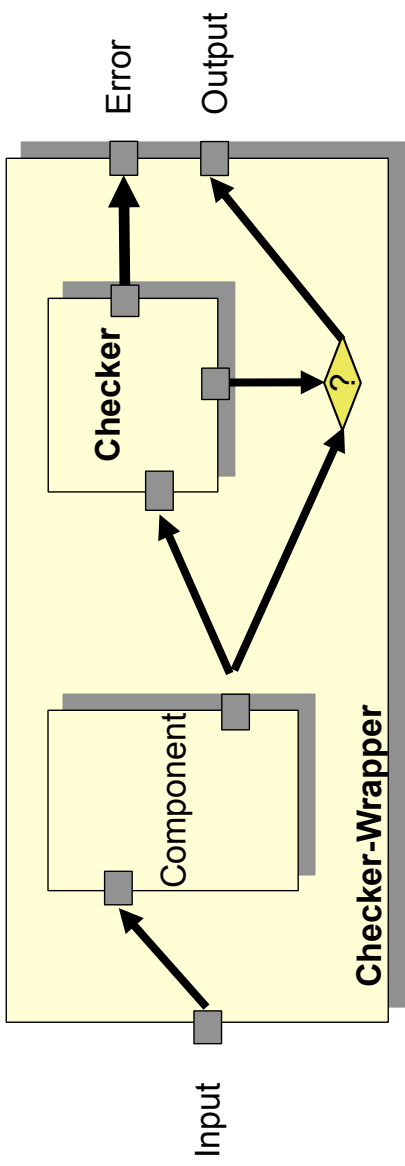
- MDSD blends Transformational and Generative design
- Models
  - represent partial information about the system
  - Are not directly executable
  - But can be used to generate parts of the code of a system
- Model-driven architecture (MDA®) of OMG blends Transformational Design and Generative Design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

- 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

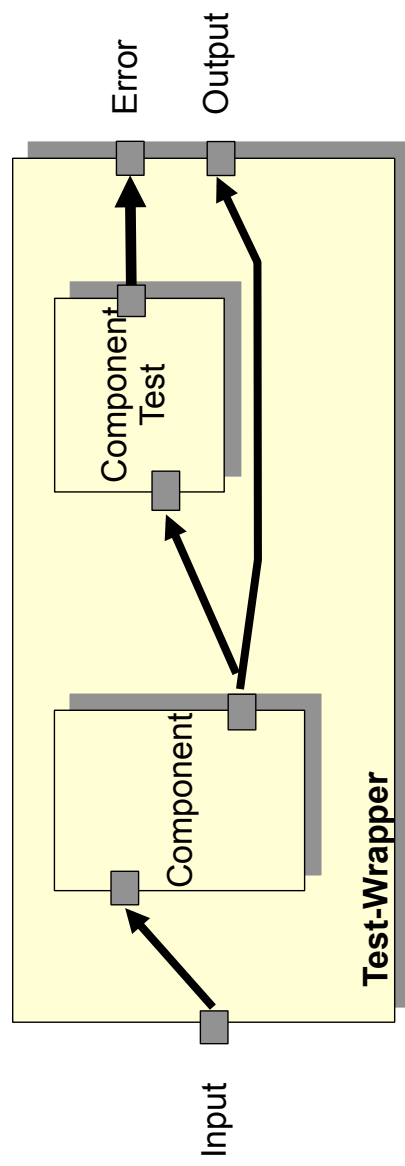
**How can I prove that my design is correct with regard to the requirements?**



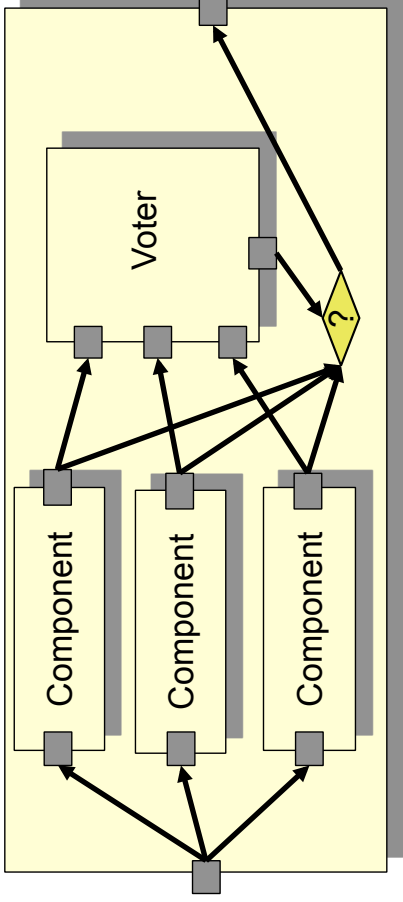
- A **checker-based system** is fault-tolerant in the sense that for every component, a *checker* exists that checks the correctness of an application
  - Also called a *monitor*
- Example: Verified compilers, fault-tolerant 24/7 systems



- A test-driven system maintains with every component a test component
- The test runs prior to the system
- Example: TDD (Test-Driven Development)

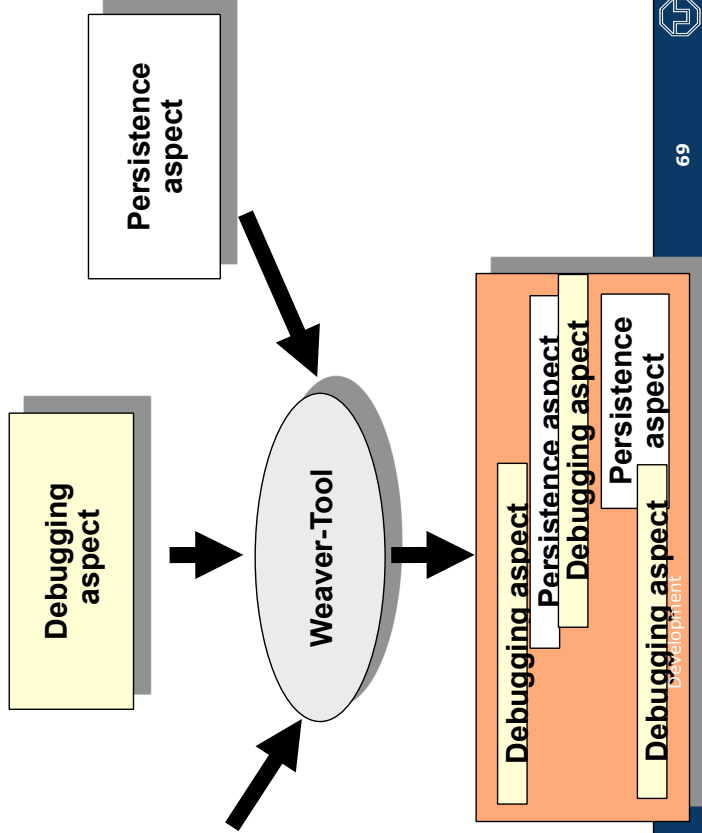


- In a *voting fault-tolerant architecture*, the run-time checker is a majority voter that compares the results of several instances of the component
- Example: Space Shuttle



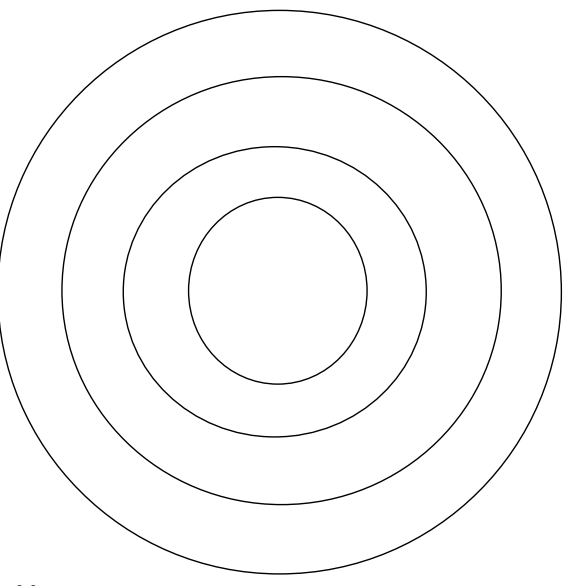
# 20.3.10 ASPECT-ORIENTED SOFTWARE DEVELOPMENT

- Aspect-oriented systems specify different aspects of a system in separation (separation of concerns)
  - The slices are reintegrated by *generative Aspect Weavers* (Aspect/J)
  - More in course CBSE

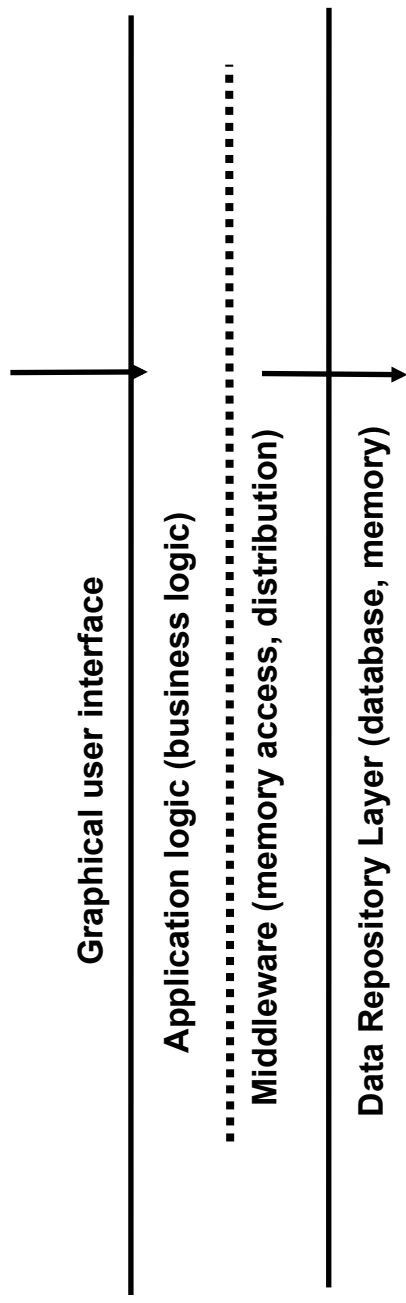


# 20.4 OTHER ARCHITECTURAL STYLES

- Connectors: procedure calls
- Ports: procedure interfaces
- Control flow mostly synchronous
- Data flow along the layers and the call graphs, mostly singulary
- Data- and control flow are isomorphic
- Dominating style for large systems



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



UNIX:      User Space  


---

Kernel

Apple-UNIX:      User Space  


---

Kernel  


---

Microkernel (Mach)

Windows NT/XP:      User Space  


---

Kernel  


---

Hardware Abstraction Layer (HAL)

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

- 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
- 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



- Functional and action design to call-based architectural style or component-based style
- Object-oriented design to object-oriented call style or actor style
- Action-based design (with data-flow) to data-flow architectures (pipe-and-filter architectures)

**A specific design method leads to a specific architectural style**

- 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!

- There is no single “the 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

- Don't be discouraged about the diversity of this lecture. There is something to win...
- A good object-oriented designer is not automatically a good software engineer
- A software engineer knows a large toolbox of *different methods* to be able to choose the right method!
- Usually, people stick to the methods in which they have been educated
  - COBOL programmers
  - Imperative vs functional programmers
  - Object-oriented programmers vs procedural programmers
- Do you want to get stuck?
- You will have a large advantage if you are open-minded



- In the following, we will see several examples for selected design methods
- With the concepts of simple graph-based models, we can see common concepts in all of them