TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik, Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

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

- Version 12-1.1, 15.12.12

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

---

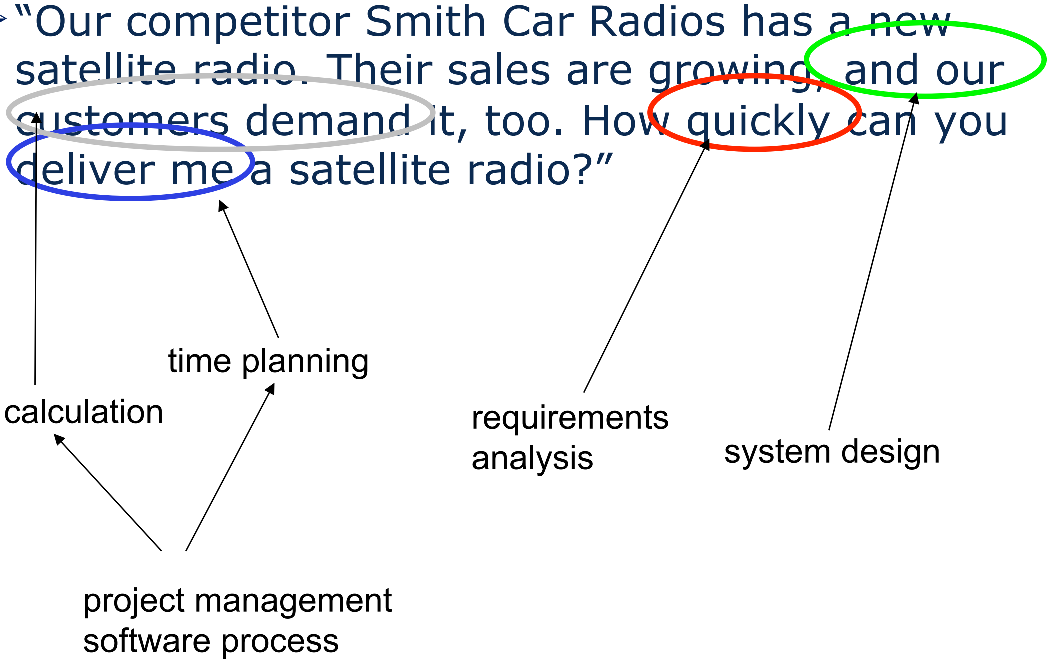
Scenario

- 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?"

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?"



time planning

cost calculation

requirements analysis

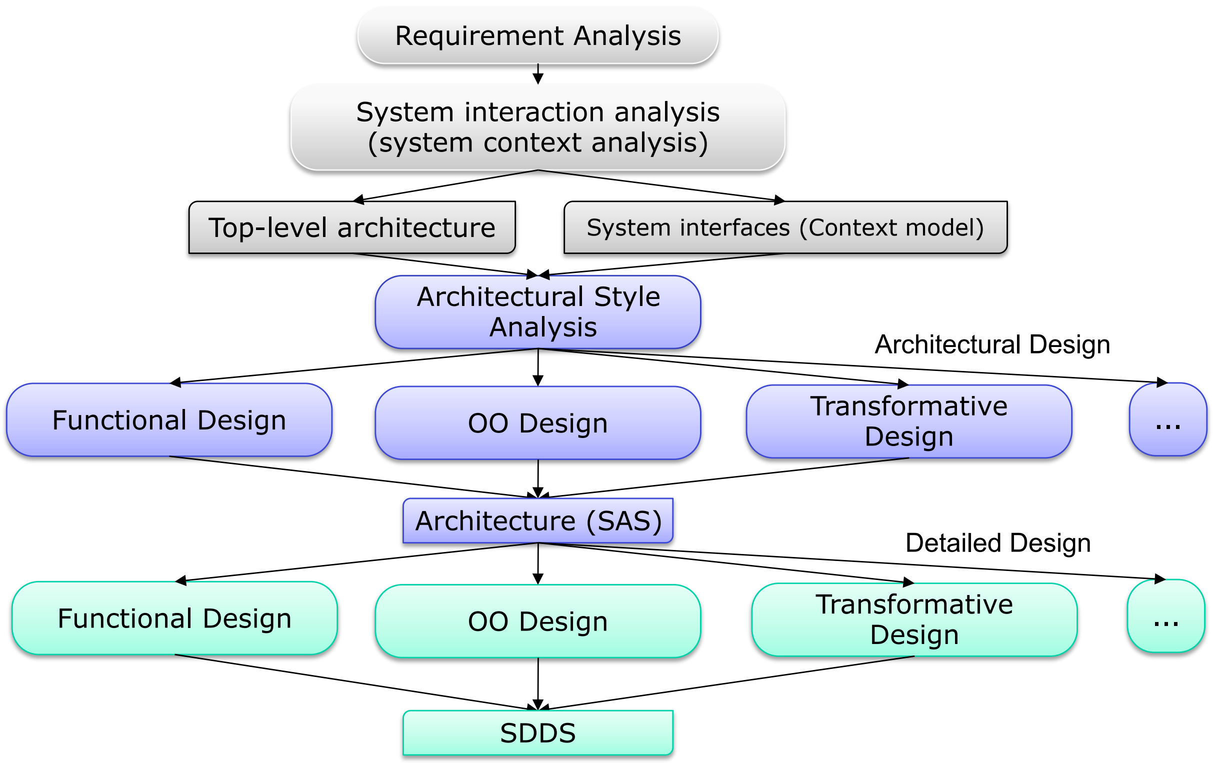
system design

project management software process

The Ideal Design 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

20.1 From Requirements to Design



Requirement Analysis

System interaction analysis (system context analysis)

Top-level architecture

System interfaces (Context model)

Architectural Style Analysis

Architectural Design

Functional Design

OO Design

Transformative Design

...

Architecture (SAS)

Detailed Design

Functional Design

OO Design

Transformative Design

...

SDDS

➢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

Contents of the Software Architectural Design Specification (SAD, SAS)

- 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

Contents of Detailed Design Document (SDDS)

- 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 [Budgen]:

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

20.2.1 Design Representations

Design Languages

		Text	Diagrams	Math
Specification Languages	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	Larch (with prover) CSP CCS
Programming Languages		ELAN SETL Java Scala C#	Statecharts Workflow languages (BPEL, BPMN)...	Modelica Metamodelica Matlab Simulink

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 codel)* and refines its internals
- Every design process
  - Contains several central generic steps
  - Uses general design strategies
  - Ends up in a specific *architectural style*
- Design processes belong to software development methods/ processes
  - Together with requirements, testing etc.

Repetition: Generic Steps in Design Processes

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

---

Development Operations of Design Methods

- 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

20.2.2.1 Architectural Styles as Results of Design Processes

Denert's Law on Architectural Styles

- 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**

Architectural Styles

- 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, classes, objects, modules
  - **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?

The Design Problem

- 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

---

Design Processes have a Focus of Elaboration and Refinement

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

20.3 Overview of Elaboration Strategies

- 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

---

20.3.1) Function-Oriented Design (Operation-oriented, Modular Design)

- 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

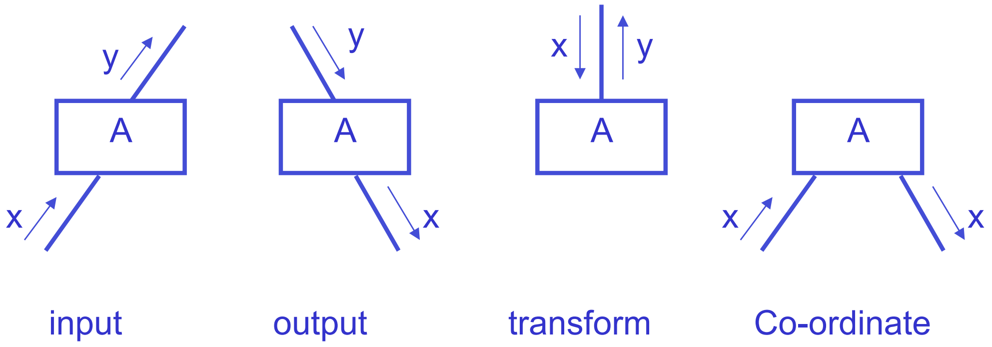
20.3.2) Action-Oriented Design

- 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:
  - 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

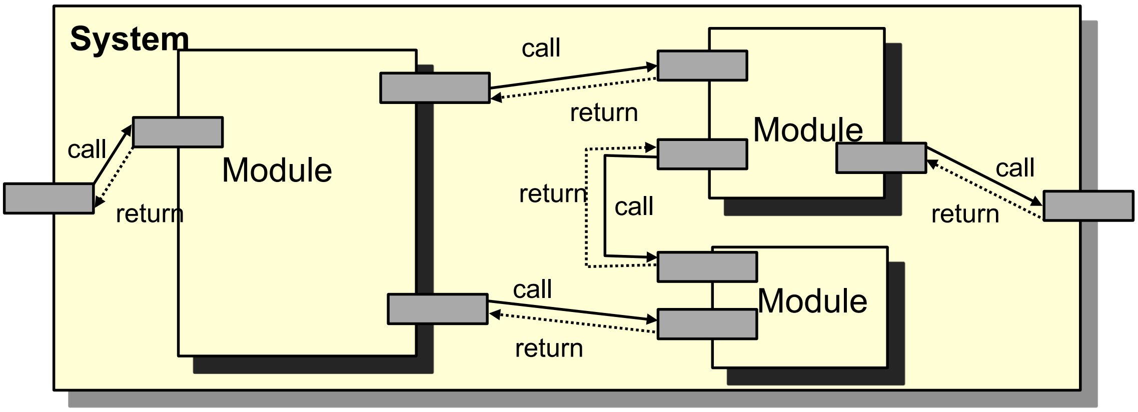
What are the actions the system should perform?

Structural Decomposition



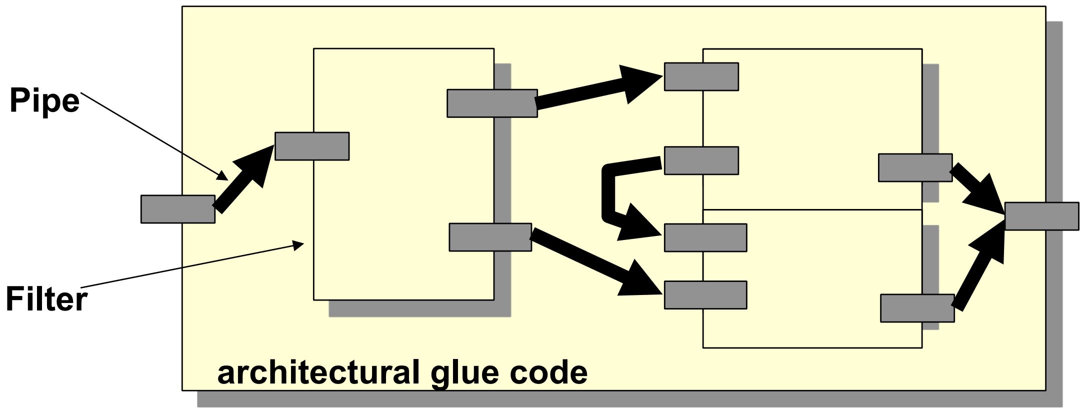
Result 1: Call-Based Architectural Style

- 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



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*)



```
cat server.log | grep "Adding student"
```

Examples

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 provides untyped data flow (texts)
- Microsoft Power Shell provides typed data-flow, typed in XML

Call-based systems:

- Object-oriented frameworks
- Layered architectures

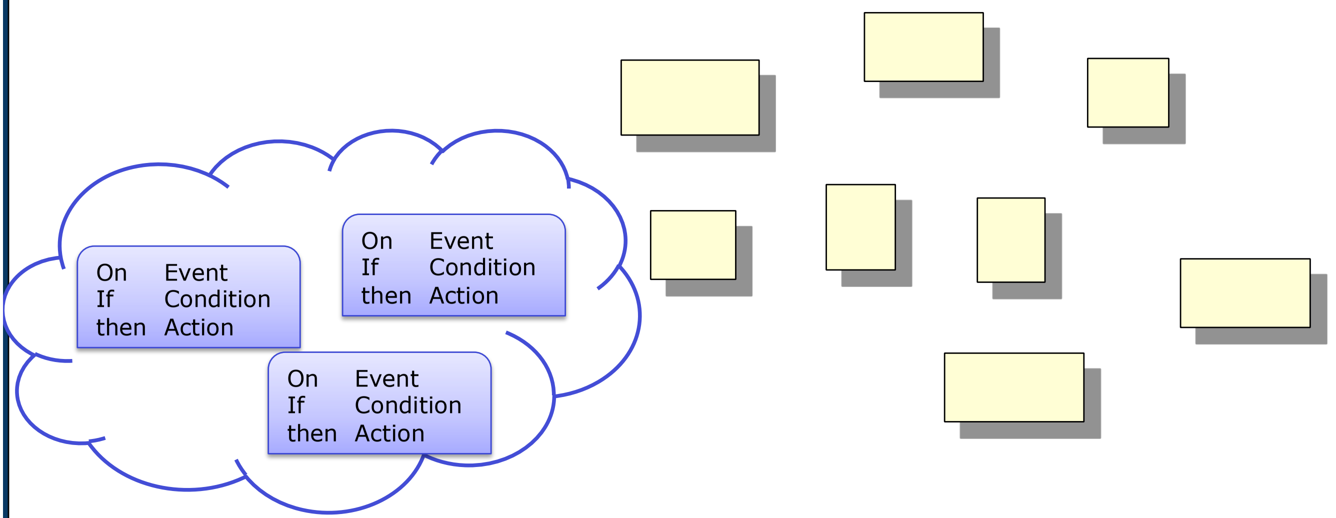
20.3.2.2) Event-Condition-Action-Oriented Design

- 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 (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?

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



SBVR Example (OMG Business Rule Language)

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

---

JBoss Rules

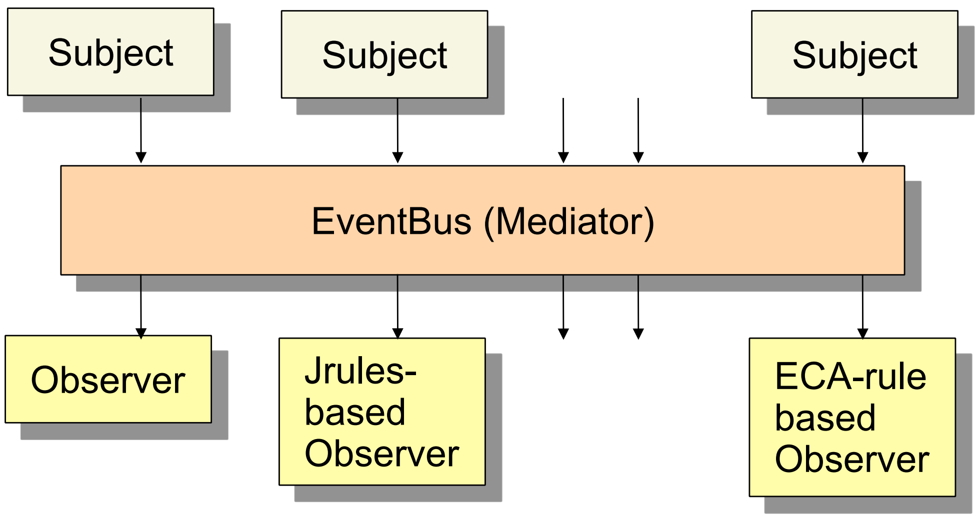
```
<rule name="Free Fish Food Sample">
   <parameter identifier="cart">
      <java:class>org.drools.examples.java.petstore.ShoppingCart</java:class>
   </parameter>
   <parameter identifier="item">
      <java:class>org.drools.examples.java.petstore.CartItem</java:class>
   </parameter>

   <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>
```

Event-Bus

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



Subject

Subject

Subject

EventBus (Mediator)

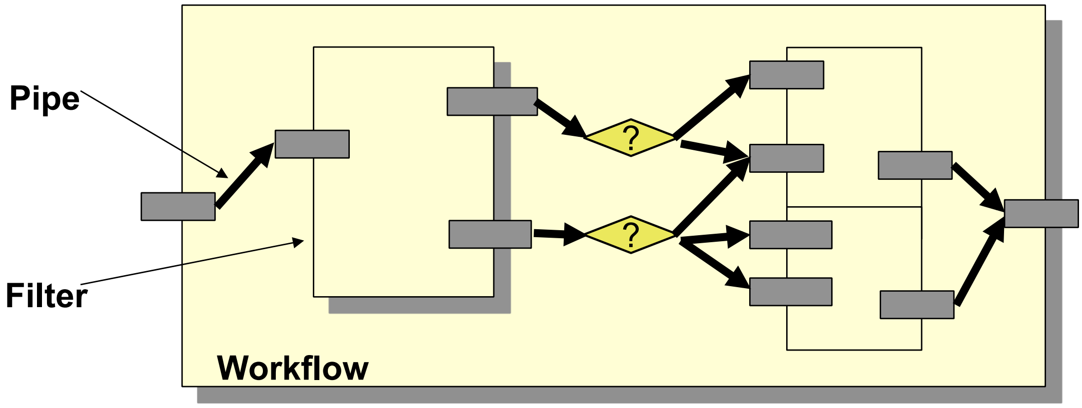
Observer

Jrules-based Observer

ECA-rule based Observer

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"

Arch. Style: Architectural Style of Communicating State Machines

- 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

Applications

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

20.3.3) Data-Oriented Design

- 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

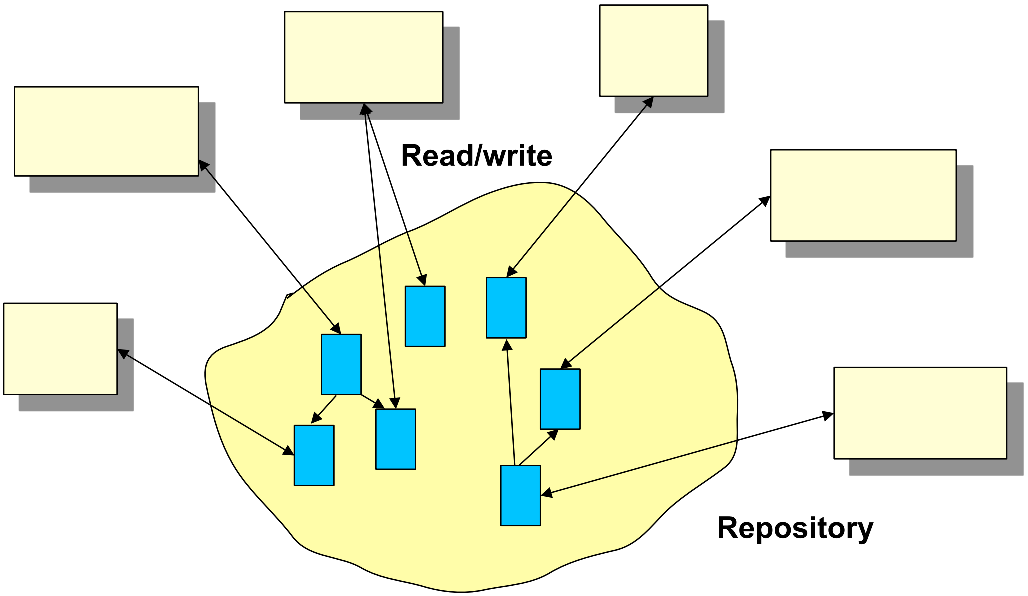
What does the data look like?

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)

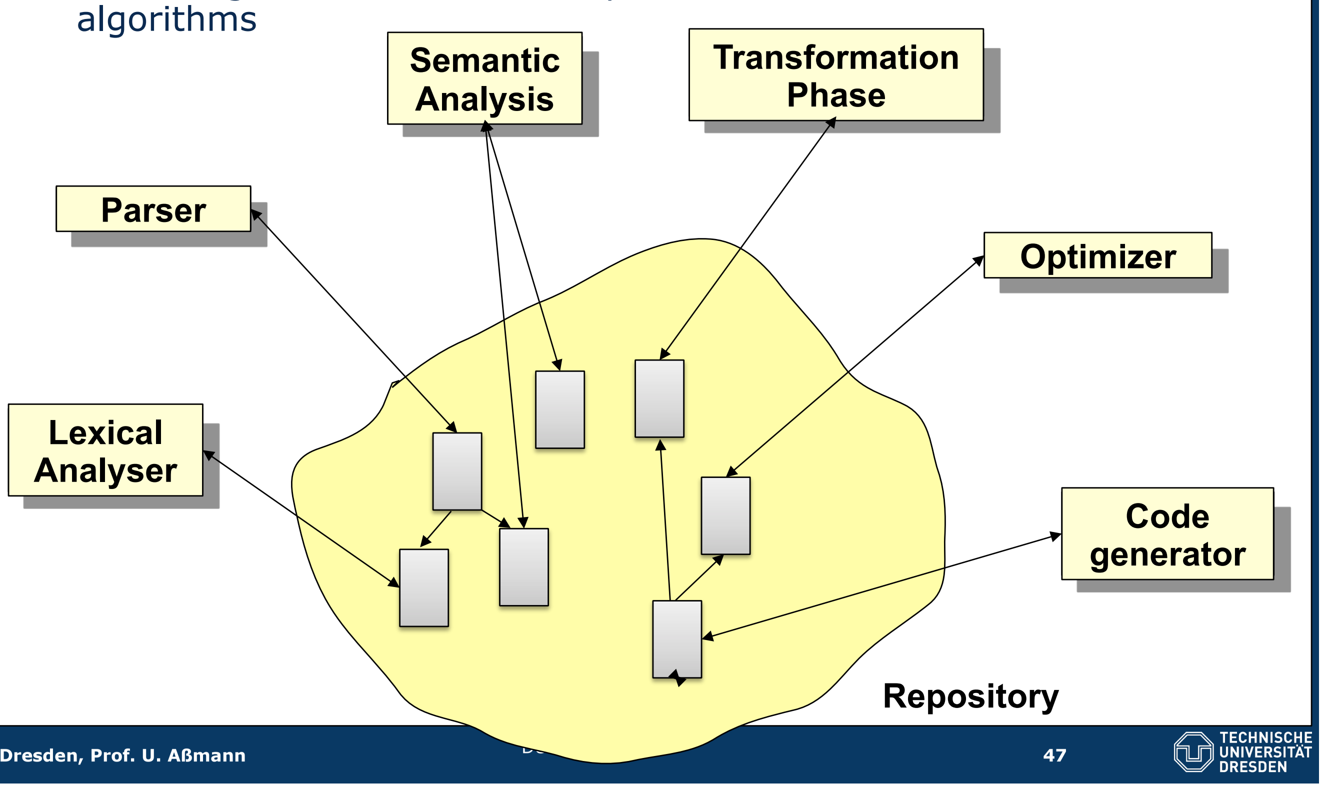
Arch. Style: Repository Systems (Data Base Systems)

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



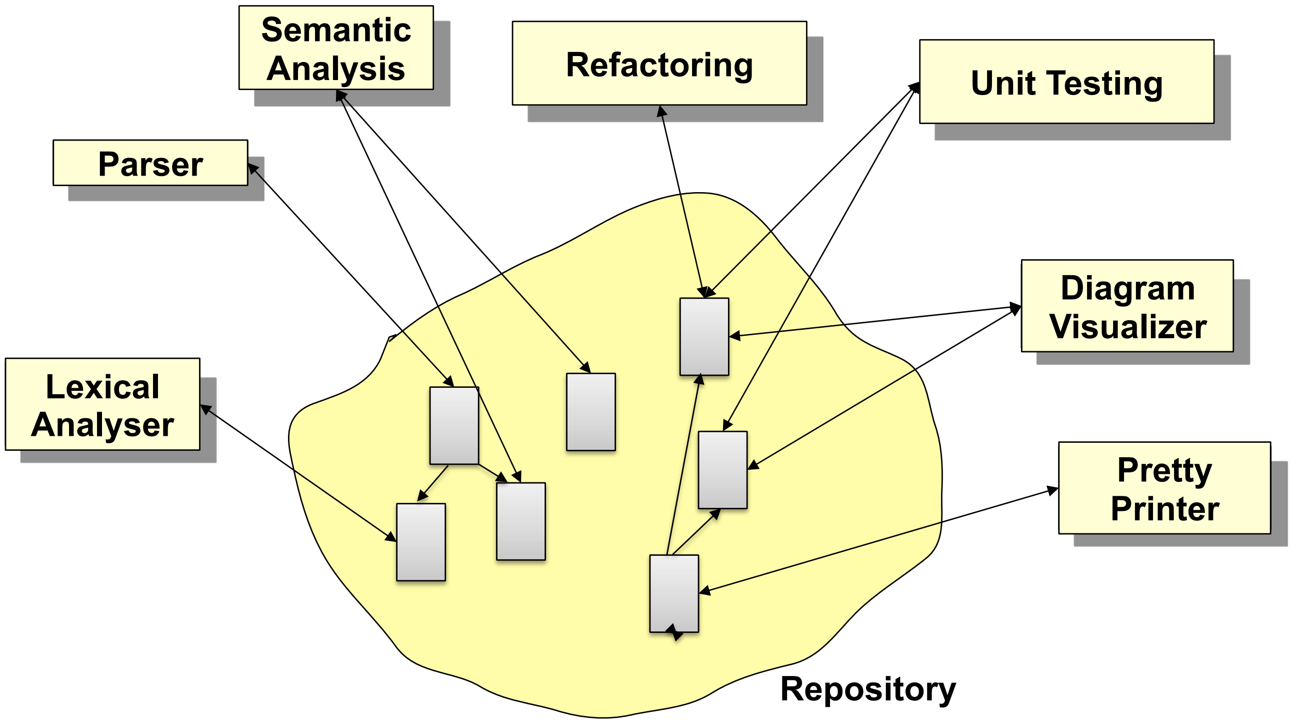
Example: Repository Style in a Compiler

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



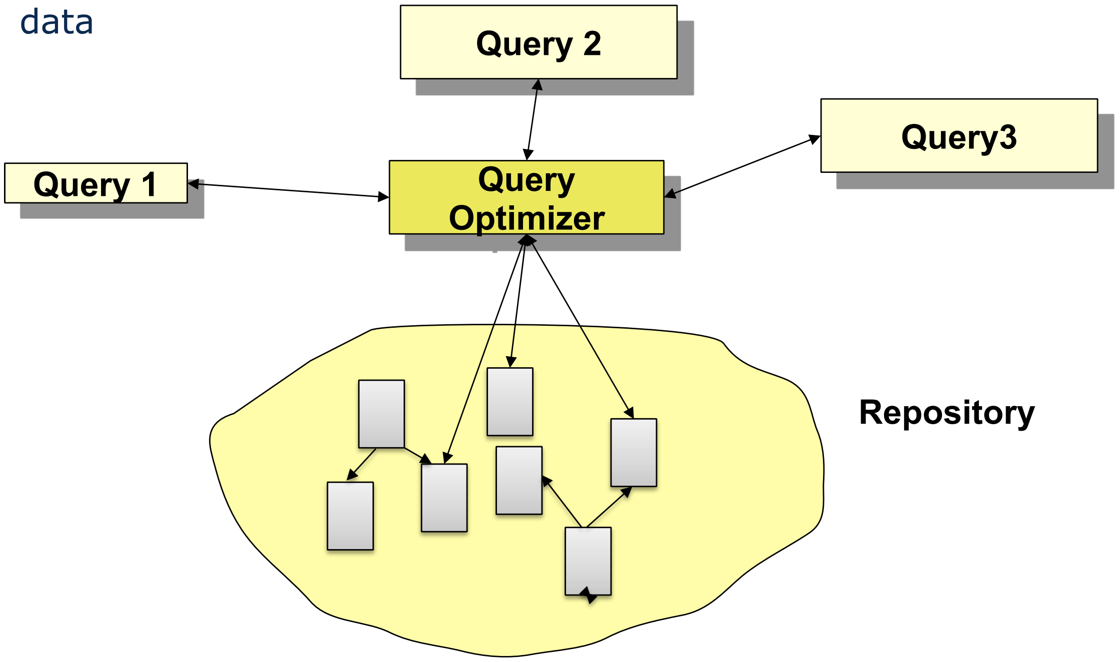
Repository Style in a Integrated Development Environment

- IDE store programs, models, tests in their repository



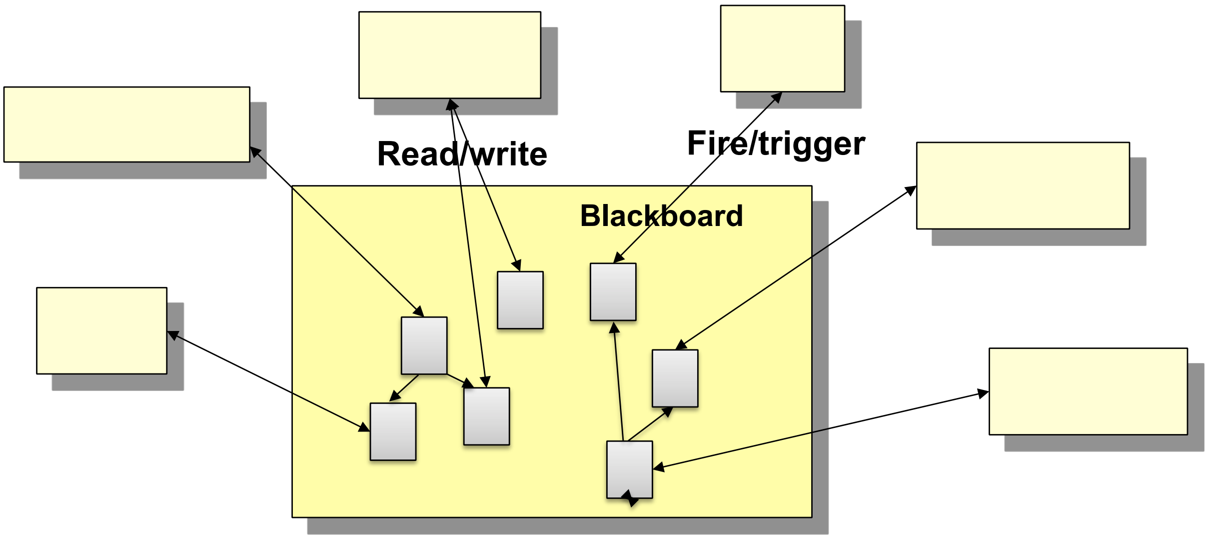
Information Systems – Queries on a Repository

- 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



20.3.4) Component-Based Design (Structure-Oriented Design)

- 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 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?

20.3.5) Object-Oriented Design

- 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?

Object-Oriented Design Methods

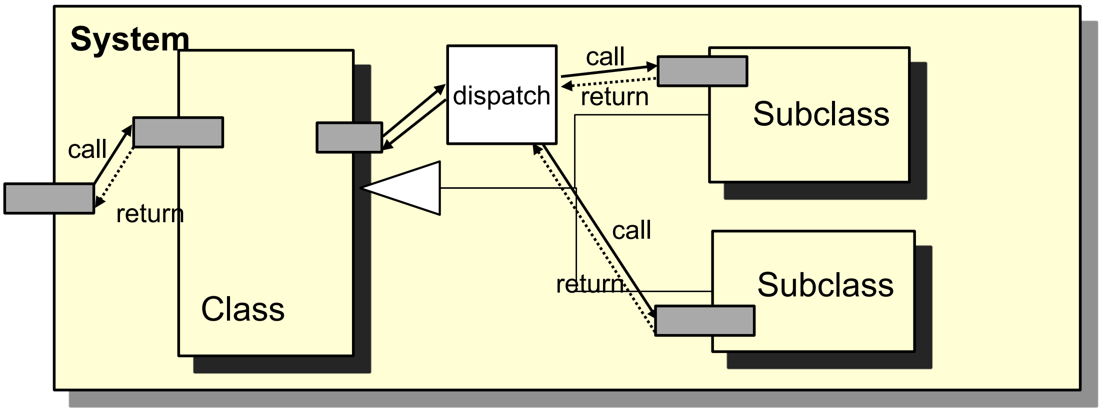
- CRC cards (ST-1)
- Verb substantive analysis (ST-1)
- Collaboration-based design and CRRC (ST-1)
- Use-Case Realization Analysis

- 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)

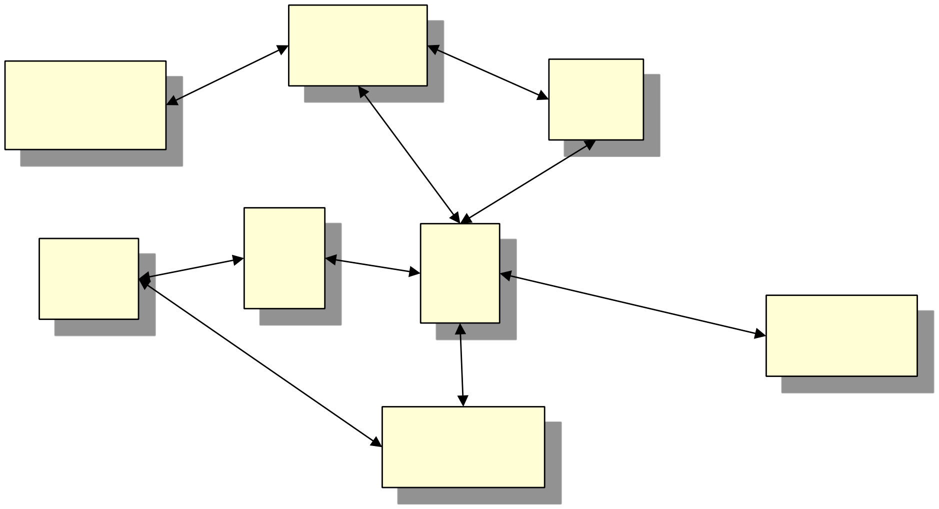
Arch. Style: Object-Oriented Call-Based Architectural Style

- 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*)



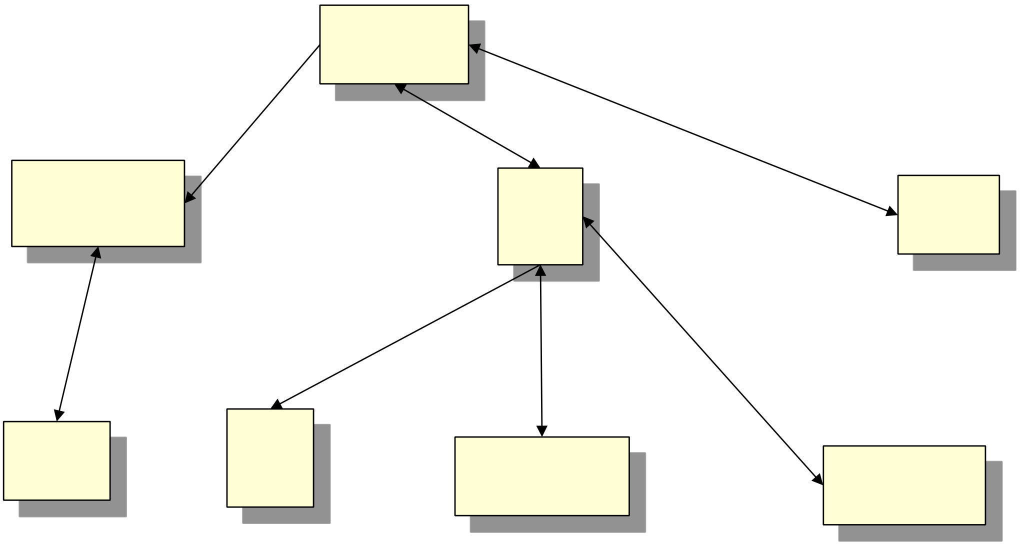
Arch. Style: Object-Oriented Process Systems (Actor Systems)

- 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



20.3.6) Transformational Design

- 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 an better version and finally, the implementation?

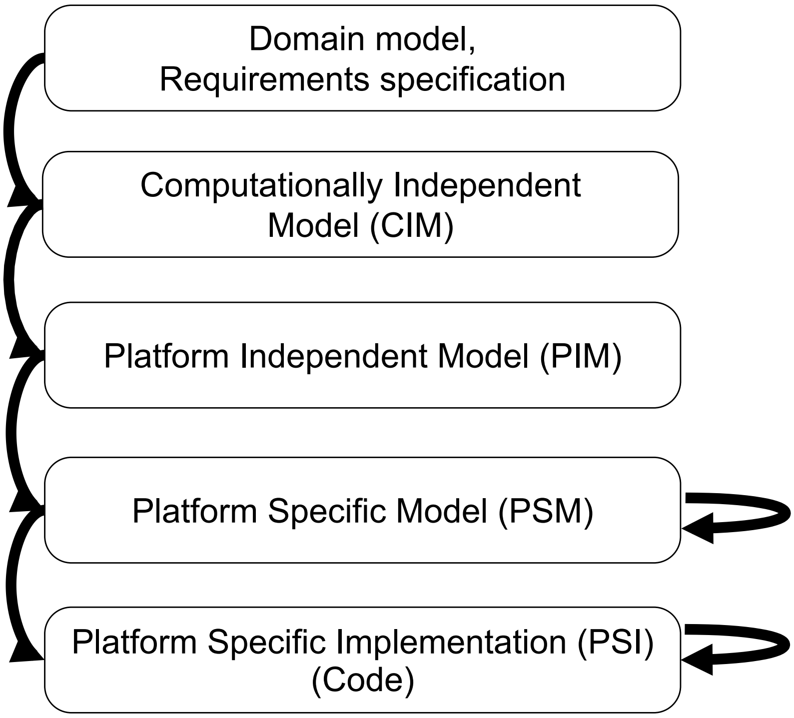
Transformational Refinement-Based Design

- **Wide spectrum languages** uses rule-based transformation systems and transformation planners
  - This starts at the requirement specification and refines (under proofs of correctness) expressive expressions to executable programs (**semantic-preserving refinement**)
  - The **semantic refinements** are refactorings which **lower** expressive expressions to low-level
  - Semantics can be proven in different forms, e.g., with Hoare logic, Dynamic logic, or denotational semantics
- Semantic-preserving refinement does not need testing, because all derived programs are correct by construction. The method is also a formal method.
- 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), VDM, Z, B, Event-B

Refactoring-Based Extreme Programming (XP)

- 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

Domain model,
Requirements specification

Computationally Independent
Model (CIM)

Platform Independent Model (PIM)

Platform Specific Model (PSM)

Platform Specific Implementation (PSI)
(Code)

20.3.7) Generative Design

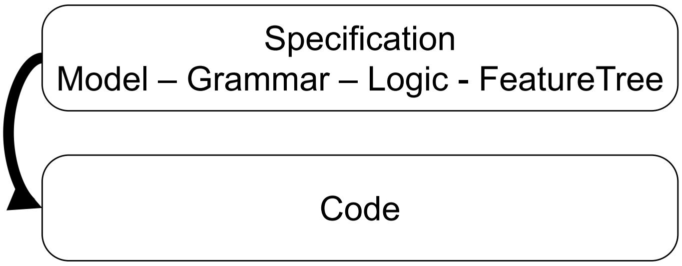
- (aka Generative Programming)
- Specify the solution in
  - a "formal method", a specification language
  - a template which is expanded (generic programming)
  - In UML, which is generated into code by a CASE tool
- Generate a solution with a generator tool that plans the solution
  - Planning the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

Generative Specifications

- Developing a specfication 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
    - Optimizers from graph rewrite systems (see chapter on GRS)
  - Feature-oriented development (FODA): specify *feature trees* and derive the components from them


Specification
Model – Grammar – Logic - FeatureTree

Code

Automatic Programming

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

1.3.8 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”
- MDA needs Aspect-Oriented Modeling (model weaving)

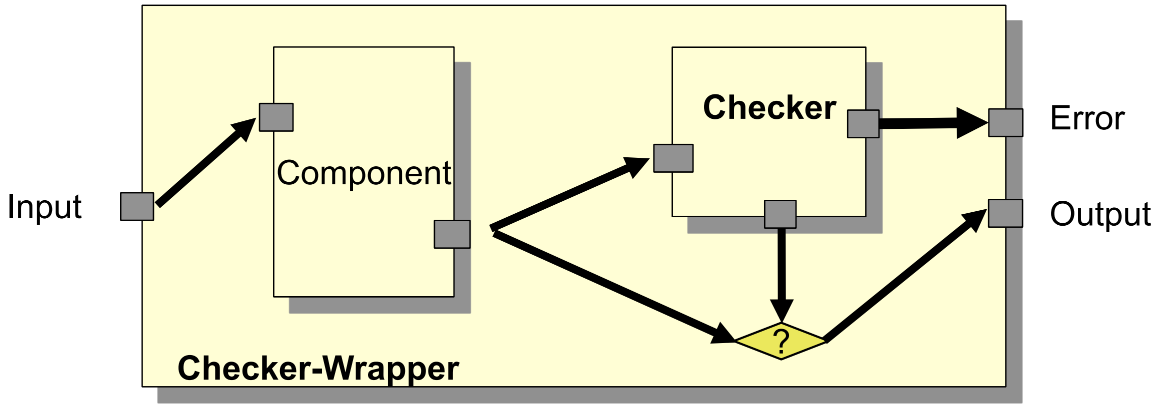
20.3.9) Formal Methods

- 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, ...

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

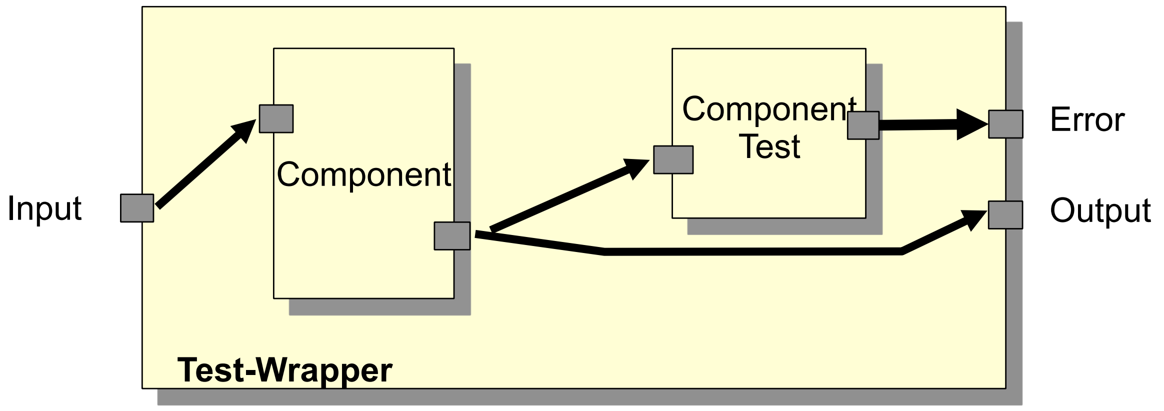
Checker-Based Systems

- 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



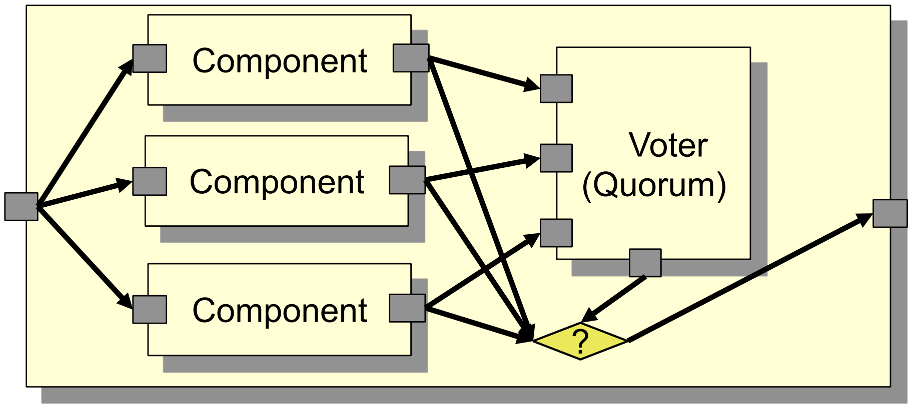
Test-Driven Architecture

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



Voting Architectures

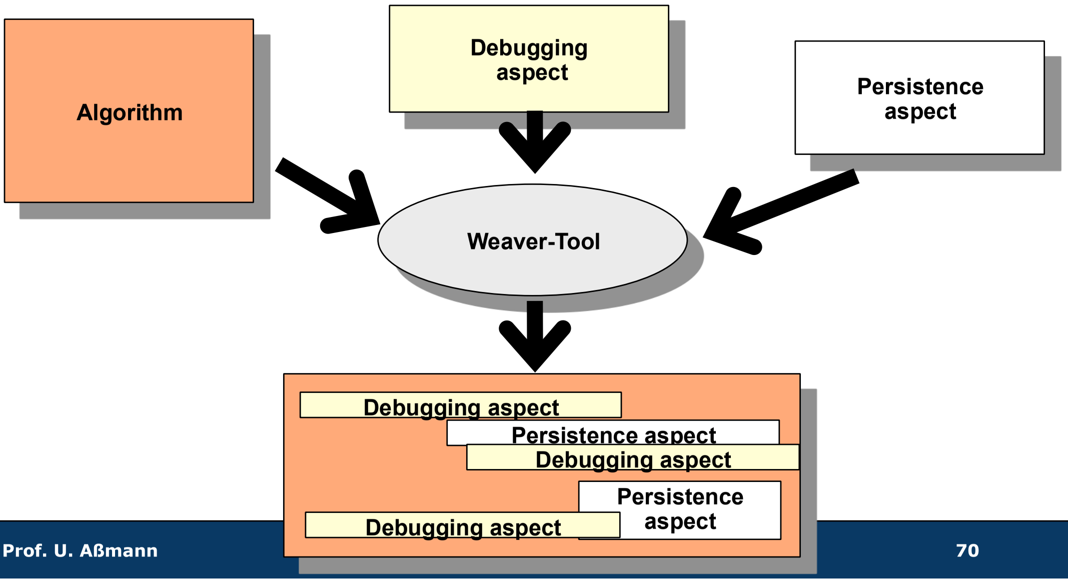
- In a *voting fault-tolerant architecture,* the run-time checker is a majority voter (quorum) that compares the results of several instances of the component
- Example: Space Shuttle



20.3.10 ASPECT-ORIENTED SOFTWARE DESIGN

Arch. Style: Aspect-Oriented Software Design

- Usual design methods have *one* aspect of development in focus ("tyranny of decomposition")
- 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 chapters "Aspect-orientation", "Feature-based product lines" and course CBSE



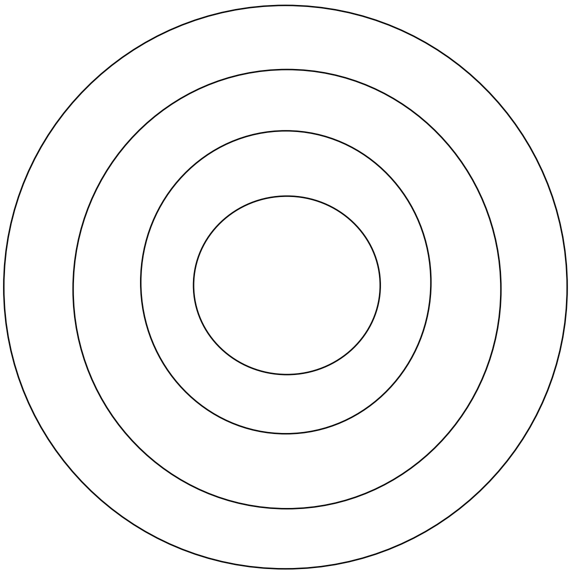
20.3.11 ADAPTIVE ARCHITECTURES

To be filled...

20.4 ARCHITECTURAL STYLES SPECIFIC TO LAYERS

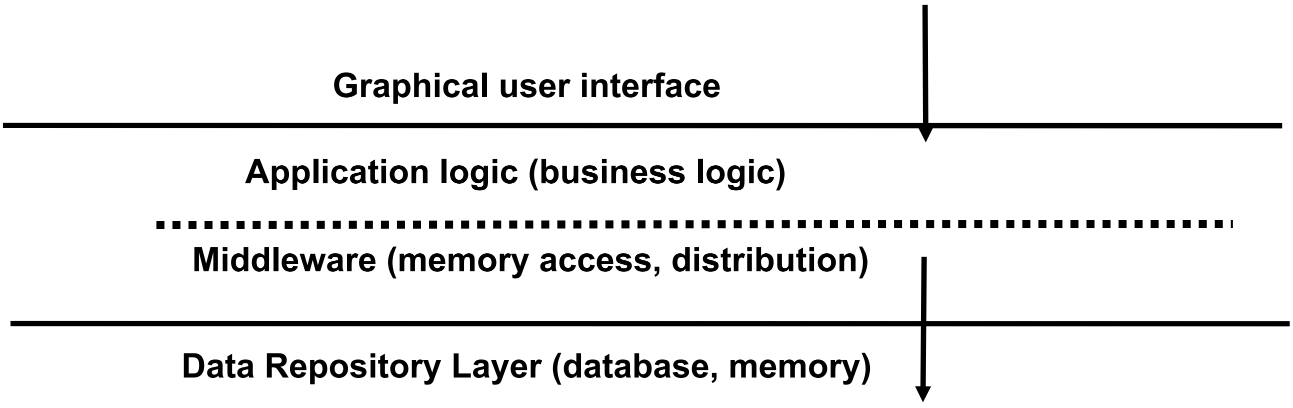
Layered Architecture

- The most general architectural style, which can be combined with all others are **layers**
- Layers can be combined with many other styles
- Ingredients:
  - Connectors: procedure calls or streams
  - Ports: component 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



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)



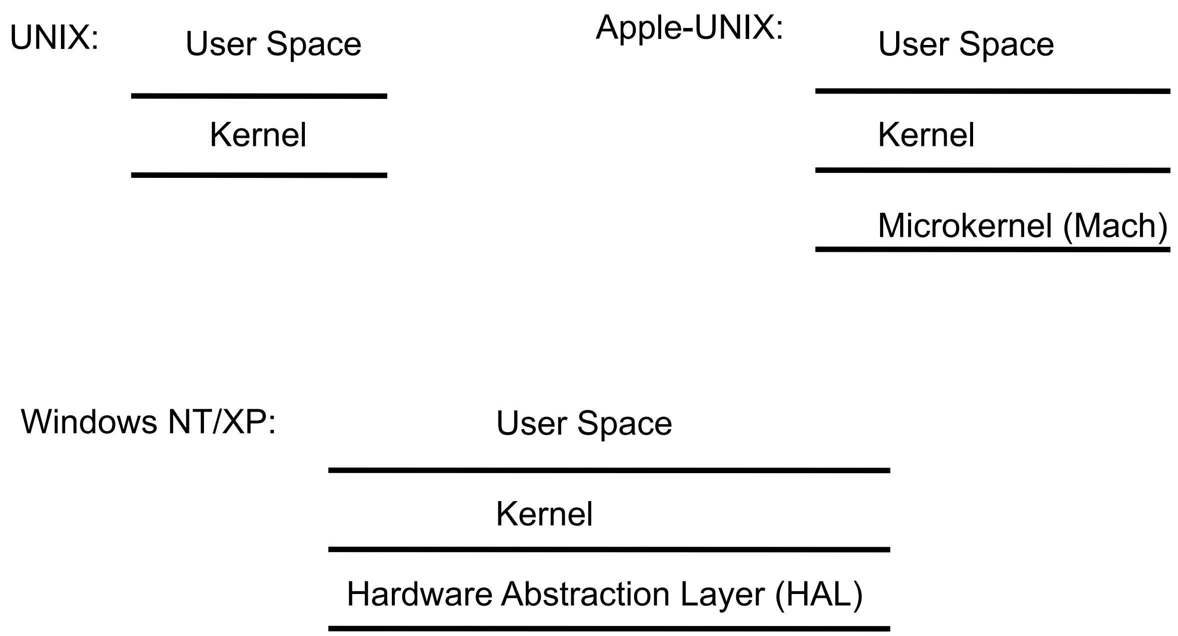
Graphical user interface

Application logic (business logic)

Middleware (memory access, distribution)

Data Repository Layer (database, memory)

Example: Operating Systems



UNIX:

User Space

Kernel

Apple-UNIX:

User Space

Kernel

Microkernel (Mach)

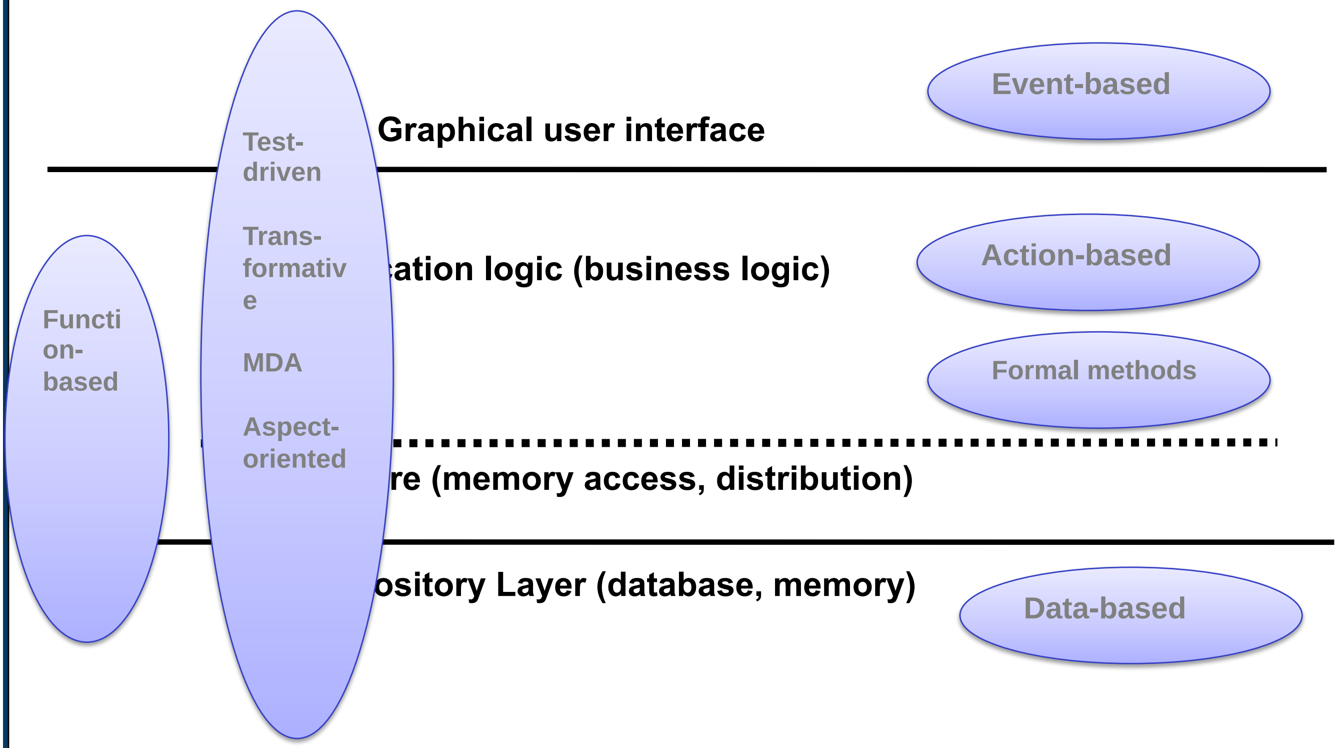
Windows NT/XP:

User Space

Kernel

Hardware Abstraction Layer (HAL)

Architectural Styles Can Be Layer Specific



Graphical user interface

ation logic (business logic)

re (memory access, distribution)

ository Layer (database, memory)

Event-based

Action-based

Formal methods

Data-based

Test-driven

Trans-formative

MDA

Aspect-oriented

Function-based

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

Law of Method and Style

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

A specific design method leads to a specific architectural style

- A specific application domain needs a specific architectural style, and due to that, a specific design method, e.g.,
  - Embedded software needs formal methods
  - Enterprise software needs workflow-based style
  - Information systems need repository style

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!

What Have We Learned?

- 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

What is running in Part III – Design?

- Presentation of Design Methods with Notations, Processes, Heuristics
- Presentation of the Development Focus
- Presentation of resulting Architectural Styles
- Presentation of Variability and Extensibility mechanims, to prepare product line engineering

Why Is This Important? (The Engineer's Parkinson Disease)

- 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

General Strategies in Design Processes

20.5 DESIGN HEURISTICS AND BEST PRACTICES

Literature

- Obligatory Reading
  - [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.

- Other Literature
  - [Budgen] David Budgen. Software Design. Addison-Wesley. Expands on the Budgen paper. Pretty systematic.
  - [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!

Brook's Paradox on Software Beauty

Exciting
- Unix
- OS/2
- APL
- Pascal
- Modula
- Algol 68
- Smalltalk

Useful, but unexciting
- MVS/370
- MS-DOS
- Cobol
- PL/1
- Fortran
- Algol 60
- php

Nice systems are often too late in the market

... be the first or the second bird!

Heuristic: Lazy or Eager Design

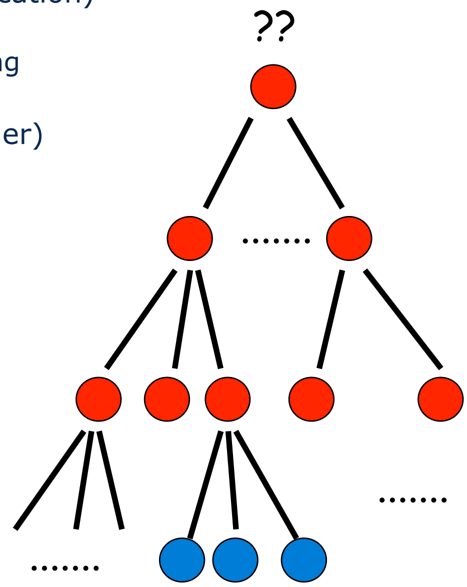
- In case of a difficult design decision
  - (when elaborating, refining, refactoring or changing representation)
  - ...**defer** it (lazy design)
    - Iterative Software development methods such as Extreme Programming
  - ...**decide** it (eager design)
  - ...**anticipate** further developments in 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

Prepare for Evolution: Grow Living Software

- Build development: “build, not write” [Brooks]
- Software is a living thing
  - Lehman's first law of software evolution: *“A system that is used will be changed”*
- Incremental development
  - “grow, not build software” [Brooks]
  - Refactorings and refinement should always be possible

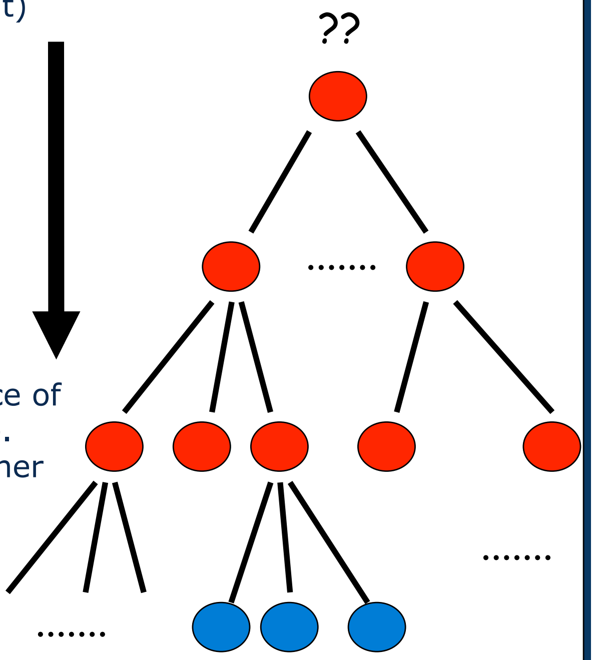
Heuristic: Divide and Conquer Strategy

- 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)



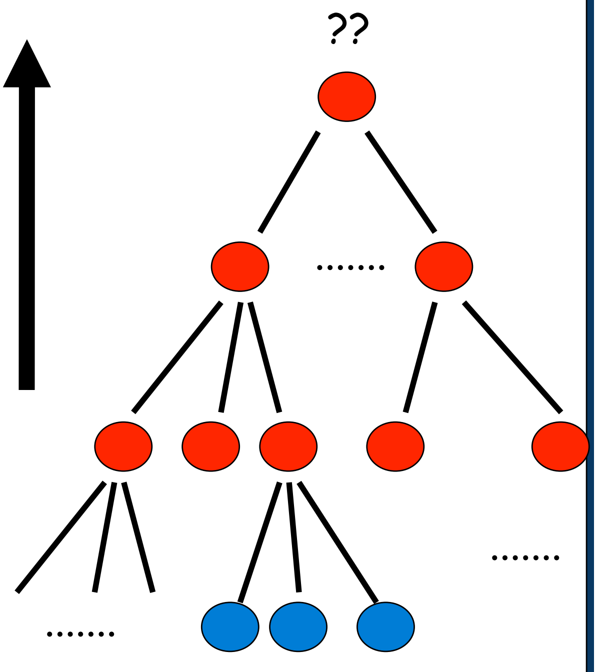
Stepwise Refinement (Top-Down, Classic Divide-and-Conquer)

- **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



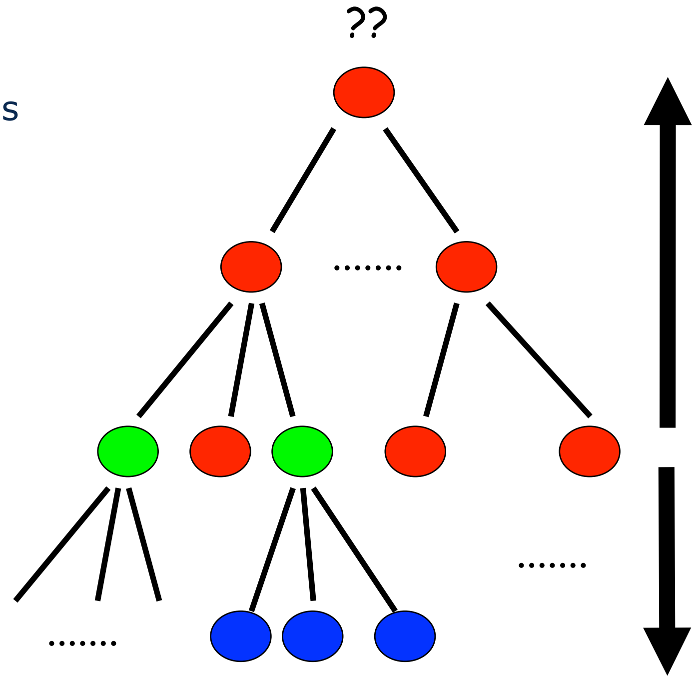
Stepwise Construction (Bottom-up)

- 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



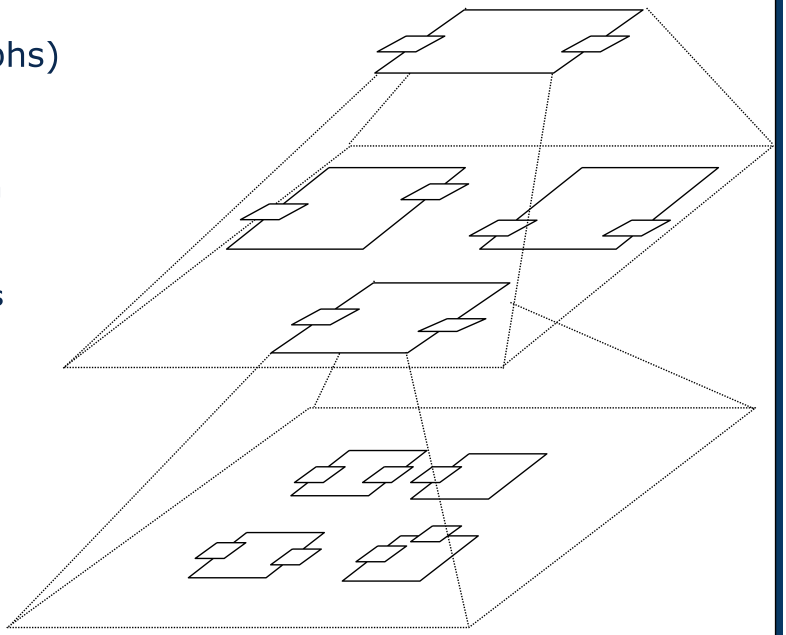
Middle out

- 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
- Finding lemmas in a mathematical proof is similar



Heuristic: Use Hierarchies and Reducible Graphs

- Trees, trees, 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 a page
    - on a slide
    - 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 it as simple as possible, but not simpler!”
  - Stephen King: “When I think, I am ready, I usually have to reduce about 30% fat from my book."
- *Abstraction* is *neglection of unnecessary detail*
  - Focus at one problem at a time and to forget about others
  - Display only essential information
  - Change representation if development strategy changes
  - This leads to design methods or decomposition methods

Heuristics on Abstraction

- 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

---

But Consider Brooks Law..

The central question in design is how to improve on the software art centers - as it always has be - on

people.

[Brooks]

Reflections on Brooks' Law

- Education of people is very important!
  - However, the differences are not minor - they are rather like the differences between Salieri and Mozart.
  - Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort.
  - Great designers and great managers are both very rare
- However, Farkas' Law: Fighting helps!
  - Farkas, a prominent trombone teacher, noticed that the most talented pupils didn't make it
  - Instead, the middle-class survived that learned how to work hard

---

Other Literature

- Simon Singh. Fermats letzer Satz. Die abenteuerliche Geschichte eines mathematischen Rätsels. dtv.
  - Gute-Nacht-Geschichte über Fermat's jahrhundertealtes Rätsel. Erklärt den komplizierten Beweis Andrew Wiles' für Nicht-Experten. Zum Verschenken! (Galois inklusive..)
  - Uhrenarithmetik. Elliptische Gleichungen. Modulformen.
- Merke: Genie entsteht aus viel, viel Fleiss (man beachte das Erlebnis Wiles' bei der Korrektur des Beweises!)
  - Wenn selbst solch grosse Mathematiker Fehler in ihren Beweisen produzieren..... keine Angst vor grossen Aufgaben...
- Excellence is the result of enormous correction..

The End

- 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