Objektorientierte Analyse und Design

Design Patterns

DI Dr. Martin Stettinger
Institute for Software Technology
Inffeldgasse 16b/2

Agenda

- Introduction
- Programming guideline (GoF)
- Design Pattern examples
- Example solved with Design Pattern (GoF)

What are patterns?

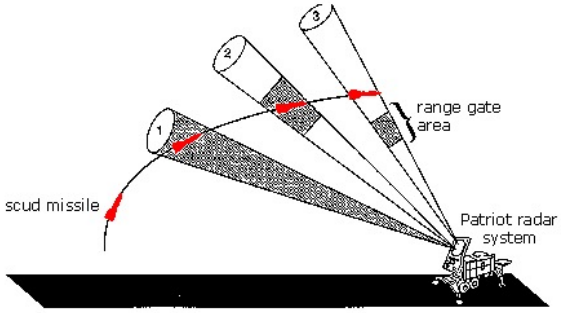
- General reusable solution to a commonly occurring problem
- Not a complete design, rather a template how to solve a specific problem
- The idea gained popularity in 1994:
  - Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four - GoF)

What are patterns for?

- Coupling and cohesion
  - Coupling of components
  - Cohesion within the components functions
- Isolates strategy from implementation
  - Strategy: context decision, interpretation
  - Implementation: algorithm
- Use of design pattern is no warrant for good design
  - Use of too much or unsuitable patterns = antipattern

Motivation for design patterns

- Airplane
- Patriot Missile Failure (25.02.1991)
- ABS / ESP / ASR / distance- and brake assist
- Train
- Database
- Network
- …



http://www-ti.informatik.uni-tuebingen.de/~ruf/seminar0102/Fehlertoleranz.pdf, Seite 3

http://www-ti.informatik.uni-tuebingen.de/~ruf/seminar0102/Fehlertoleranz.pdf, Seite 2

Pattern Collections

- GoF Design Pattern
  - Creational Design Patterns
  - Behavioural Design Patterns
  - Structural Design Patterns
- POSA1 Patterns
  - Architecture Pattern
  - Resource Management
- POSA2 Patterns
  - Concurrency Patterns
  - Synchronization Patterns

Pattern Collections

- GoF Design Pattern
- Purpose
  - Creational Design Patterns
    - Dedicated to the process of creating an object
  - Behavioural Design Patterns
    - Defines the interaction between objects and classes
  - Structural Design Patterns
    - Defines the aggregation of objects and classes
- Scope
  - Object based Pattern
    - Object relations can vary during runtime
  - Class based Pattern
    - Main focus on inheritance, static relations

Pattern Collection (GoF)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singelton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Inheritance vs. Composition

- Class inheritance
  - White-Box reuse
  - Internal of parent class visible
  - Easy to use (static)
  - No chance to change during runtime
  - Strong binding between parent class and inherited class
    - Any change in the parent changes the subclass

Inheritance vs. Composition

- Object Composition
  - Dynamic references during runtime
  - Black-Box reuse
    - No insight into reused object
  - Needs well defined interfaces between the objects
  - Bindings between objects only with interfaces
    - Support Encapsulation
  - Small classes and hierarchies
  - Use of delegate

Design Principles (GoF)

- Use interfaces, not implementations
- Use Object compositions not inheritance
  - What if there is no suitable object?
  - Still need inheritance!
- One Class, one purpose
- Classes should be open for extension, closed for changes!
- Object interaction with loose binding

Pattern (GoF)

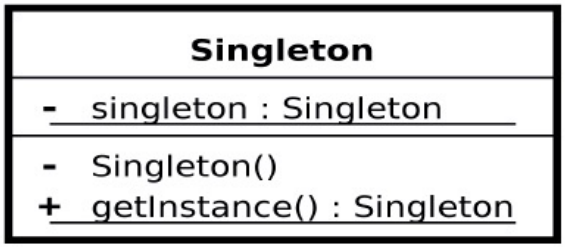
- Context
  - In what Situation
- Problem Forces
  - Define the most common problem in this context
- Solution
  - A well tested solution for the given problem
- Consequences
  - Pro / Cons / Borders / Trade-offs

Idiom (POSA)

"An Idiom is an expression peculiar to a certain programming language or application culture, representing a generally accepted convention for use of the language."

- Very low level pattern
  - Describes how different aspects of components or relations can be implemented in a certain programming language
  - Program language specific pattern (Buschmann - POSA)
- Example Incrementor
  - I = I + 1;
  - I++; // idiom

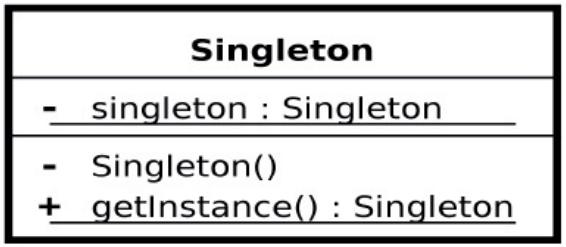
Singelton



- Context
  - Creation of exactly one instance
- Problem Forces
  - Ensure a class has one instance, provide a global point of access
- Caution
  - Global Vars

- Prevent multiple instantiation of objects

Singelton



- Define a static variable

```
public class RecommenderAlgorithmus {
    private static RecommenderAlgorithmus _instance = null;
}
```

- Make constructor private

```
private RecommenderAlgorithmus() {
    _similarityObject = new DefaultSimilarity();
}
```

- Define static access via special function

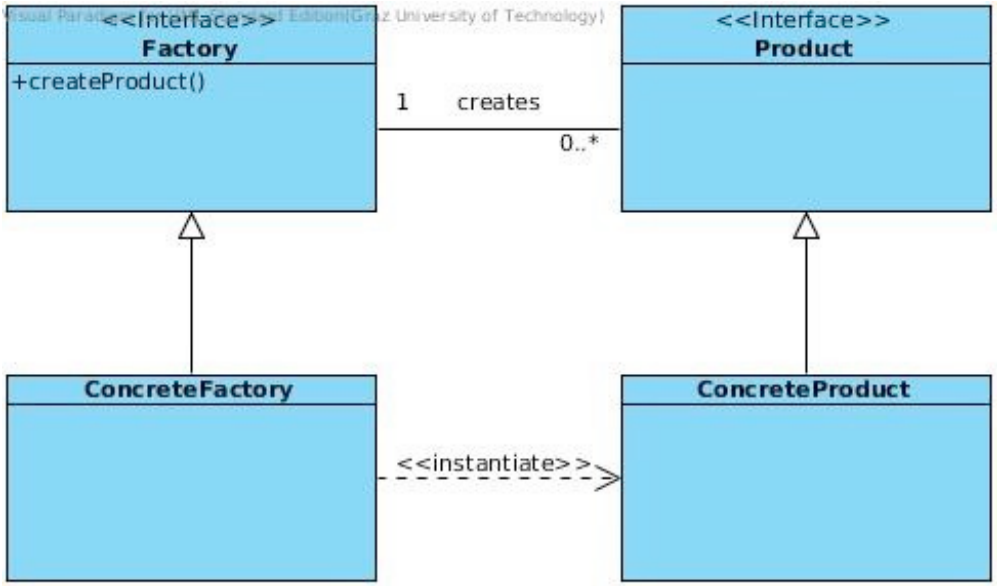
```
public static RecommenderAlgorithmus getInstance() {
    if(_instance == null) {
        _instance = new RecommenderAlgorithmus();
    }

    return instance;
}
```

Factory Pattern

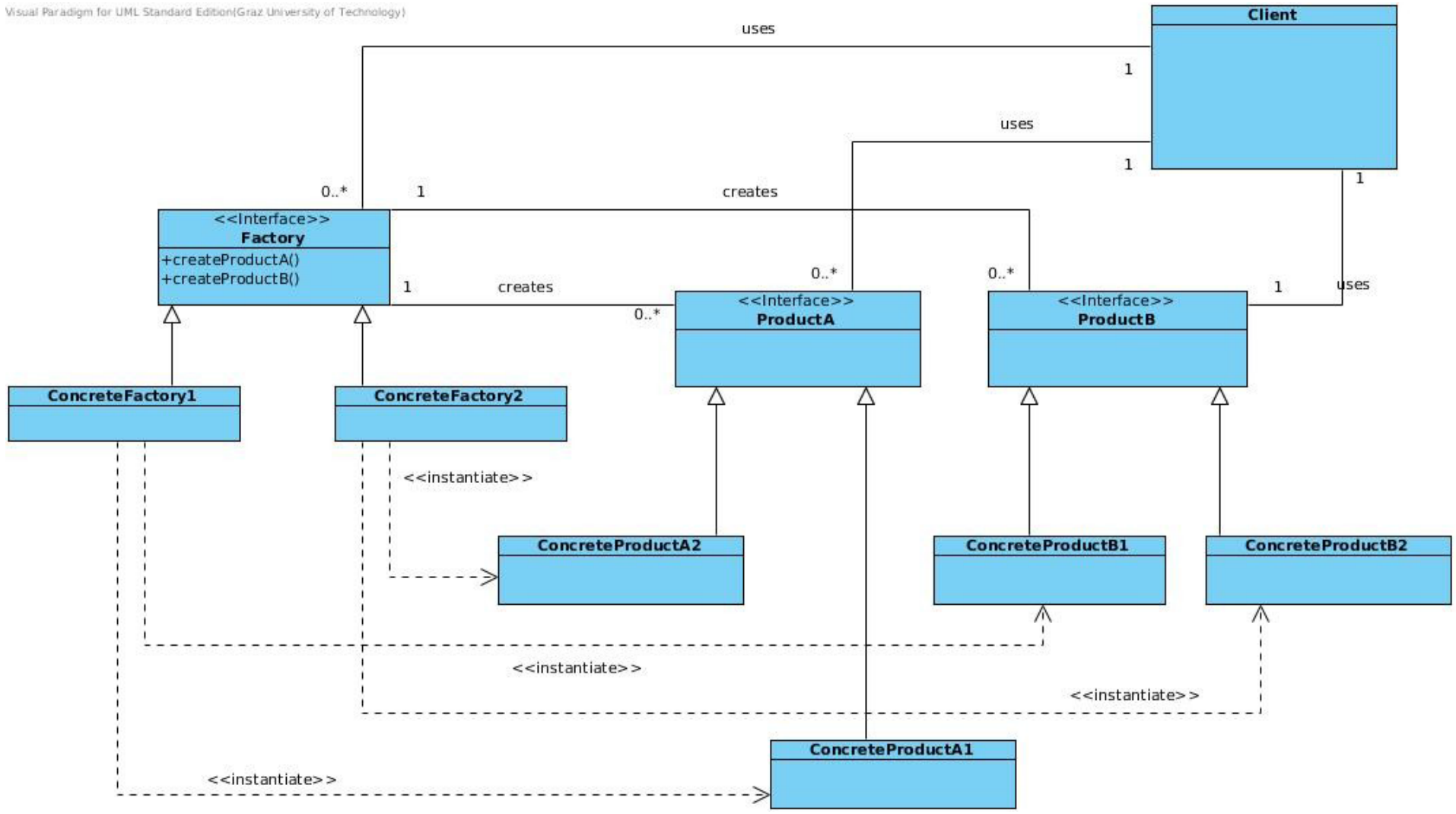
- Context
  - Creation of an object, whose class is not known until run-time
- Problem Forces
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
  - A class can‘t anticipate the class of objects it must create
  - Class delegates responsibility

Factory Pattern



Only concrete factory knows which implementation of „ConcreteProduct“ should be used

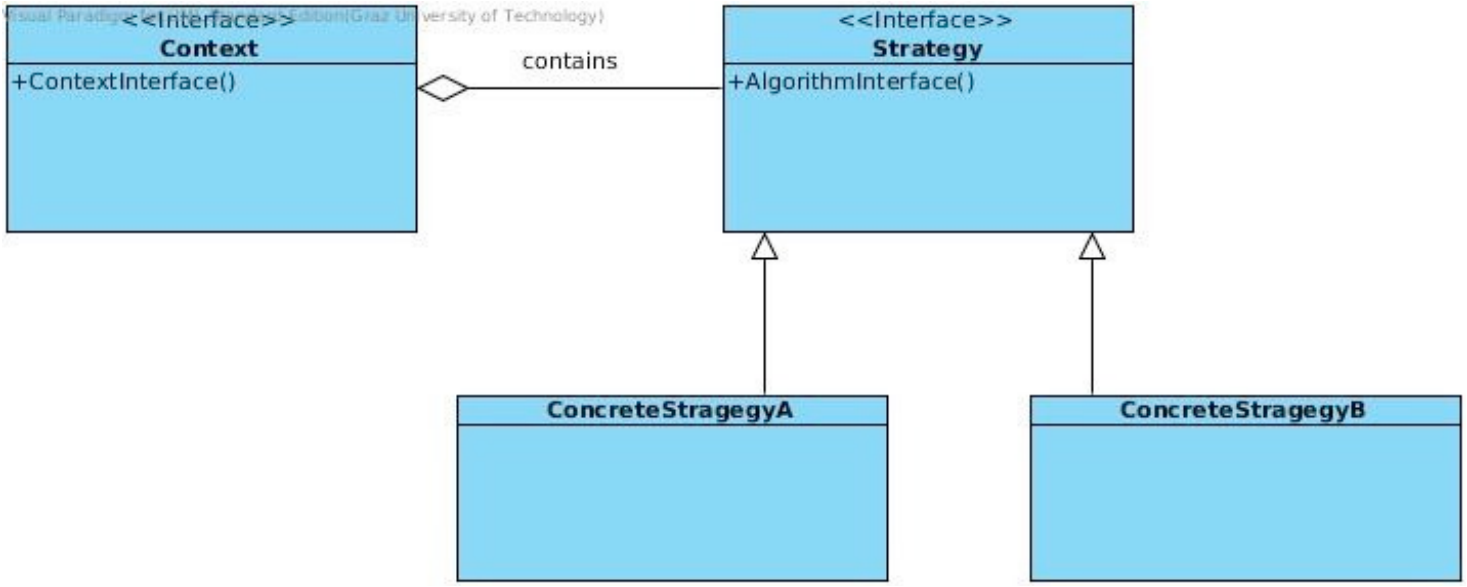
Factory Pattern



Strategy Pattern

- Context
  - Interchangeable algorithm
- Problem Forces
  - Many related classes differ only in their behaviour
  - Different algorithm needed
  - Work on same data structure
  - Different conditionals not needed

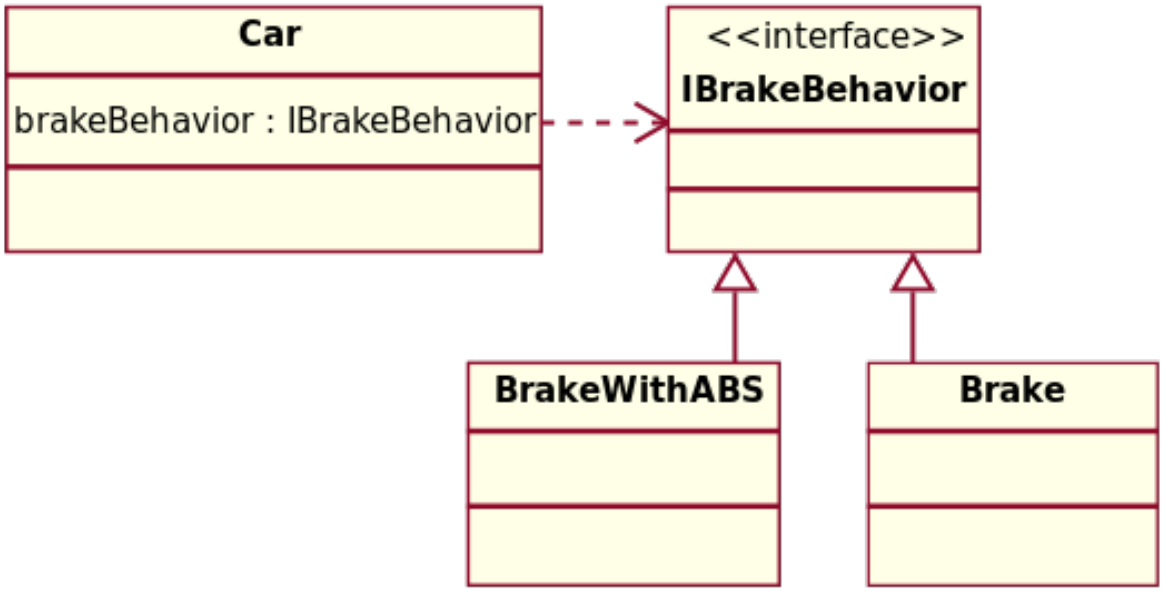
Strategy Pattern



- Advantages compared to switch / if – else ?
- Do we really need an own class ?
- Open for extension, closed for change

Strategy Pattern

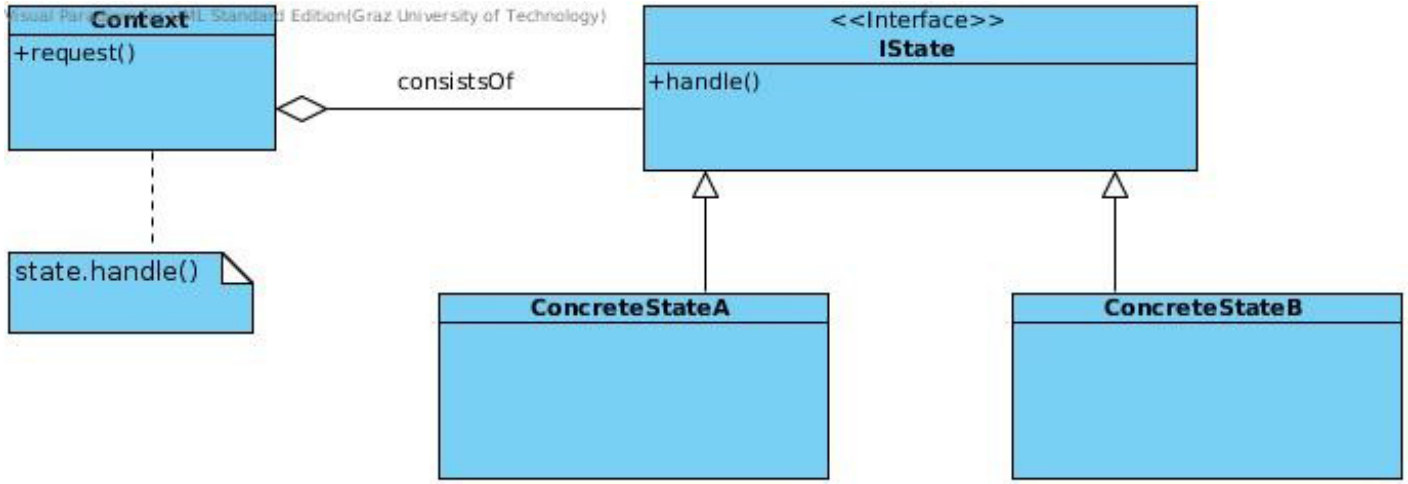
Example: Car



State Pattern

- Context
  - Allow object to change behaviour
- Problem
  - Object‘s behaviour depends on its state and it must change its behaviour at run-time depending on that state
  - Operations have large, multi part conditional statements that depend on the object‘s state
- Difference to Strategy?
  - When do we use State and not a Strategy pattern?

State Pattern



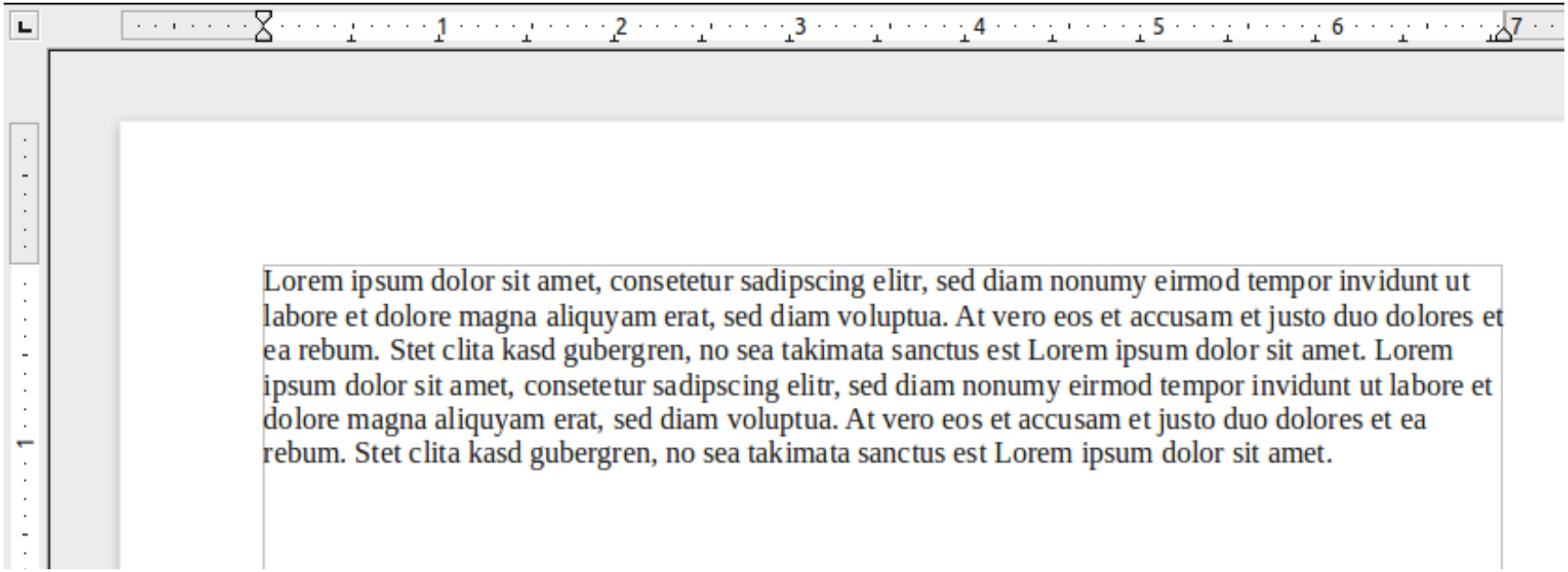
- State holds a reference on Context
- State decides on itself if a transition is necessary

State Pattern vs. Strategy

- Best method depends on the desired design
- State Pattern
  - Each State has a reference to „parent“
  - Each State can change this reference to the new State
  - Each State defines it‘s possible transitions
- Normally one big switch to change the current strategy
  - If there is only need for methods, not for members

Example

- Program a text editor



What must be done?

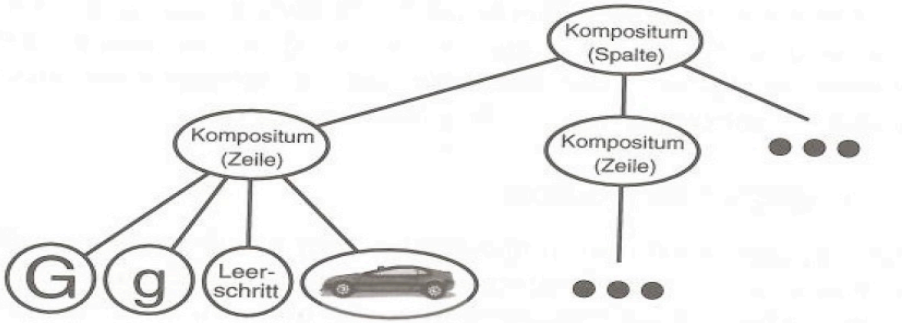
- Document structure?
- Arrange Text?
- Design the GUI?
- Support different look and feel standards
- Different window systems
- Different user interaction components
- Spell checking

Document Structure

- Specific Positioning of simple graphic elements
  - Chars, lines polygons aso.
  - Represents the whole information of a document
  - Only useful in the context of rows, columns, with pictures, tables aso.
- Manipulation of text and graphic should be done with the same functionality
  - No redundancy of functionality
  - Drawing of the object should be done on itself
  - But: How useful is spell checking on a picture?

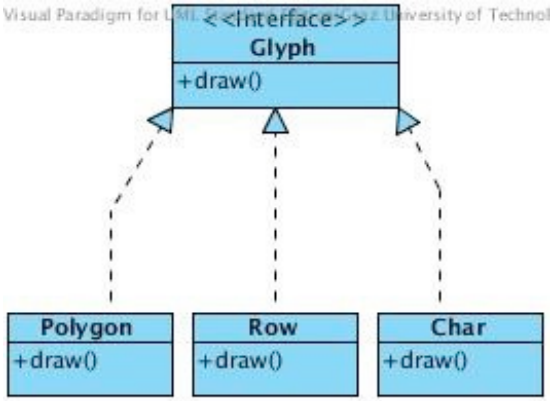
Recursive composition Pattern

- Build as composition of small elements
- Complex elements are aggregation of simple objects
- A row is an object that holds char objects
- A page is the collection object of rows
- There can also be picture objects in the page object
- Each simple object can draw itself



Recursive composition Pattern

- Two important implications
  - Each object needs a class
  - Each class must implement the same interface
    - We want to use them with the same procedures
      - Draw function (execute draw on children)
      - Know parents / children
      - Add child
      - Remove child ...



Problem?

- Is this implementation memory efficient?
- How many objects do we have in a book?
  - Example
    - 2000 chars on a page
    - 500 pages in a book
    - Each object implements drawing function?
    - Too much memory for a book?
- Are there differences between the objects?
  - What are the differences?

Flyweight Pattern

- Context
  - Large numbers of objects
- Problem Forces
  - All must be true
    - Application uses large number of objects
    - Storage costs are high because of the quantity
    - Most object state can be made extrinsic
    - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
    - Application doesn't depend on object identity

Flyweight Pattern

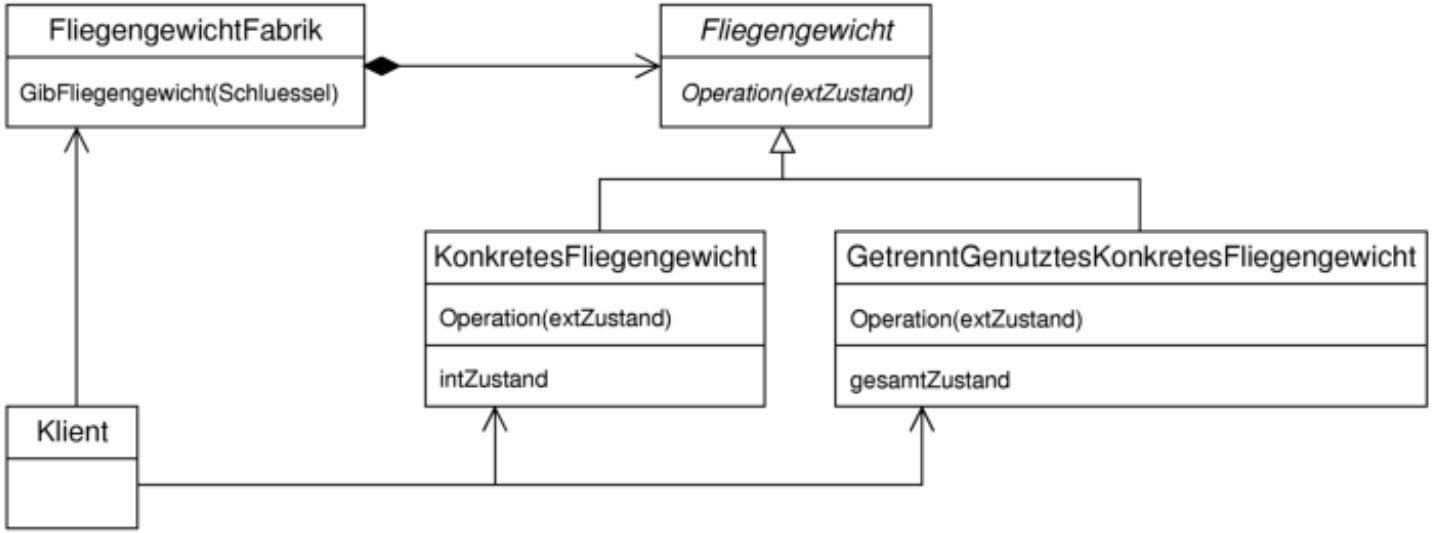
- T H I S I S A T E X T

- Char “T” has a draw function (intrinsic)
  - Contains font size?
  - Contains font type?
- Char “T” has a position (extrinsic)
  - Depends on the context
  - Object is only reusable if this is stored outside

Flyweight Pattern

- Shared objects are not instantiated directly
- Use a Factory Pattern for flyweight objects
- Create only a new object if it does not exists
  - Else return a reference
- Drawback
  - May create additional run-time costs
    - e.g. computing extrinsic state
- Save a lot space

Flyweight Pattern



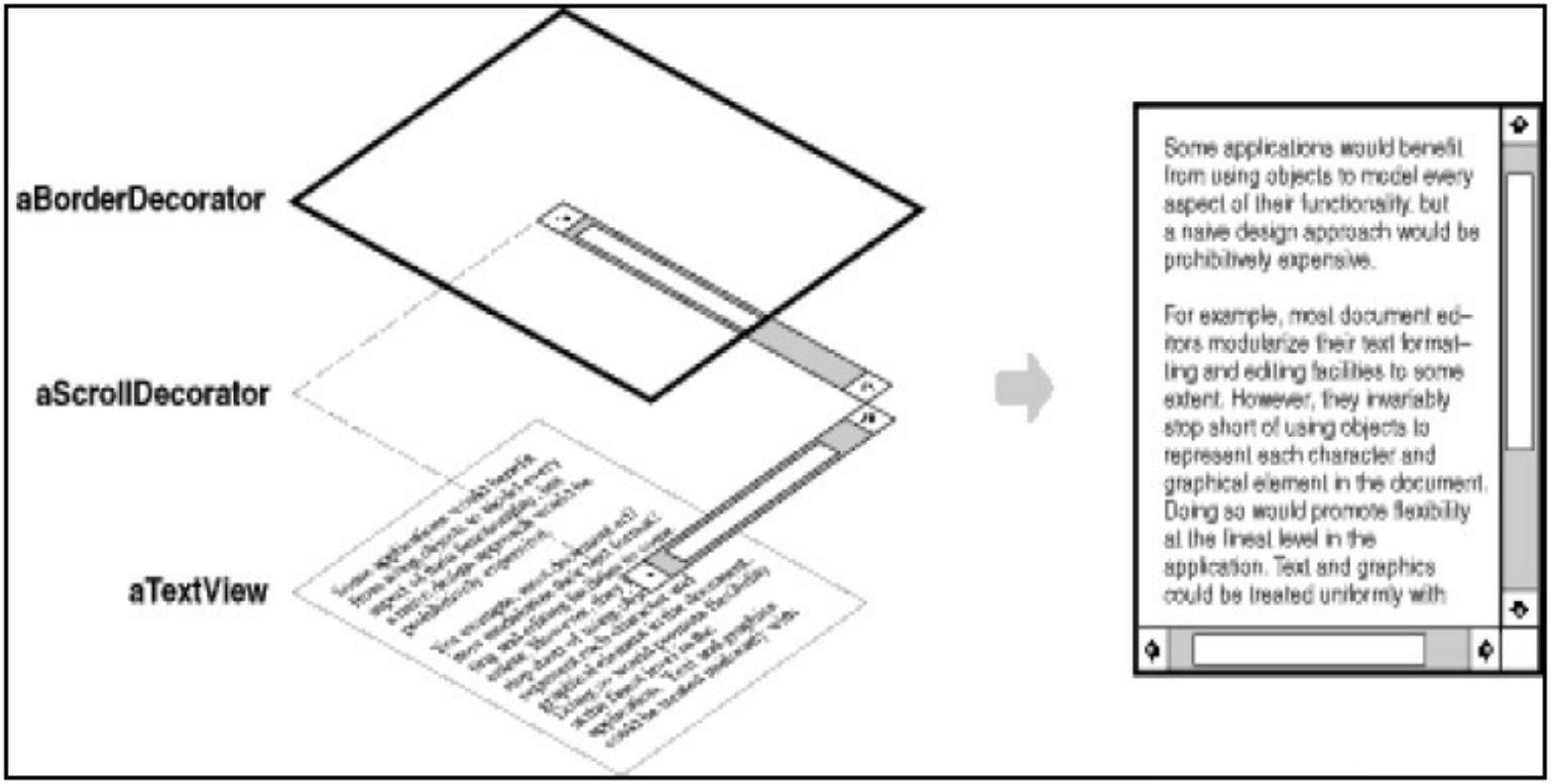
- Objects with no shared operations are also possible

GUI Design

- We want to support different GUI Modes
- Let the user switch on / off different modes
- Keep the core the same (and all functions)
- Perhaps there should be a menu bar?
- Enable / disable scroll bar?
- If there are different windows objects, what is the best way to handle this?

Decorator Pattern

- Extend Objects?



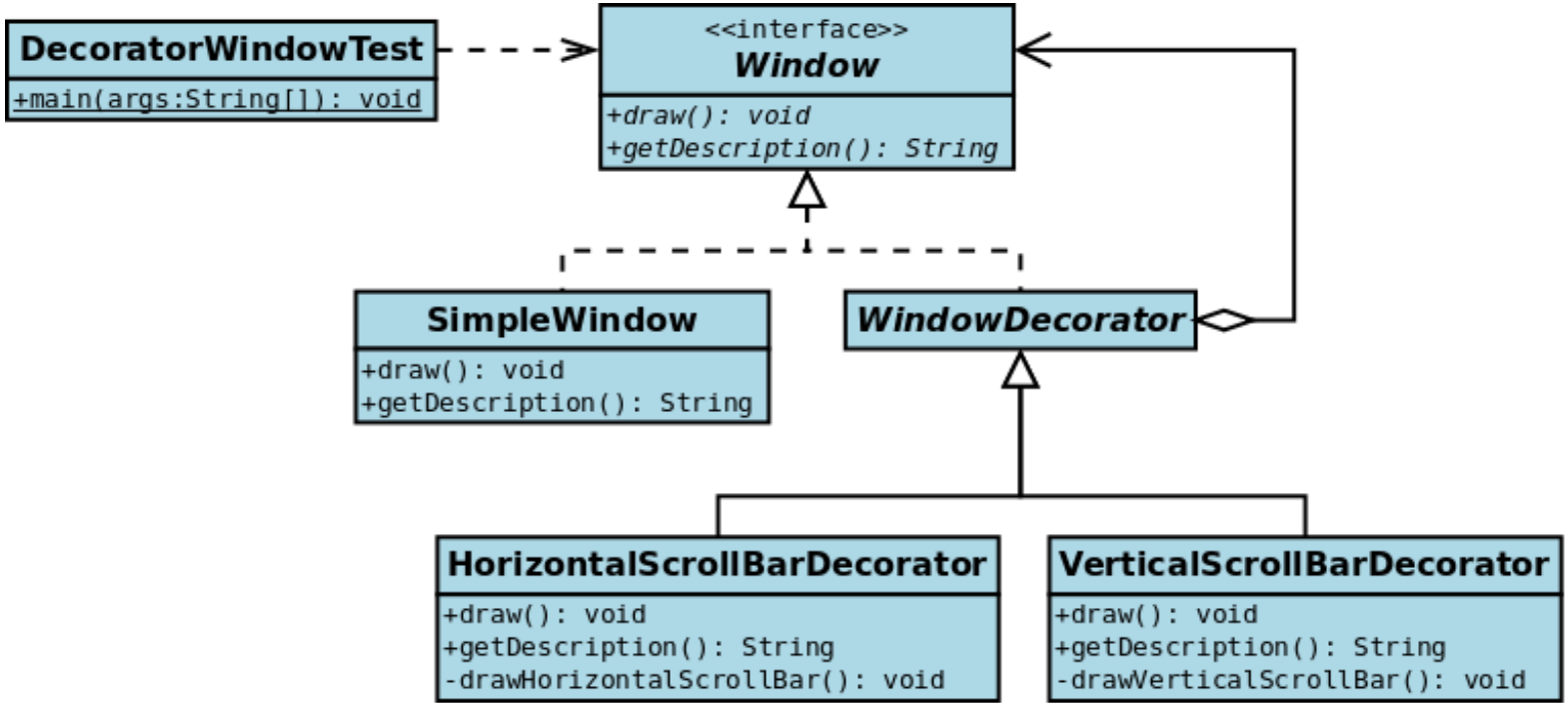
Decorator Pattern

- Context
  - Function extension of objects
- Problem Forces
  - Add and withdraw responsibilities to individual objects dynamically
  - Extension by sub classing is impractical
    - Large number of independent possible extensions

Decorator Pattern

- Also known as Wrapper (GoF)
- Not only for GUI Design but also for normal object extension
- Normal inheritance would not be flexible
  - We could add a frame inside an object
  - Changes must be done inside
- Decorator needs to implement the same interface like the original object

Decorator Pattern



Model-View-Controller

- Strict separation of:
  - Model
  - View
  - Controller
- Enables reuse of single components in different application contexts
  - i.e.: Web UI and Swing UI

Model-View-Controller

Contains:

- Representable data

- If necessary business logic (Recommender)

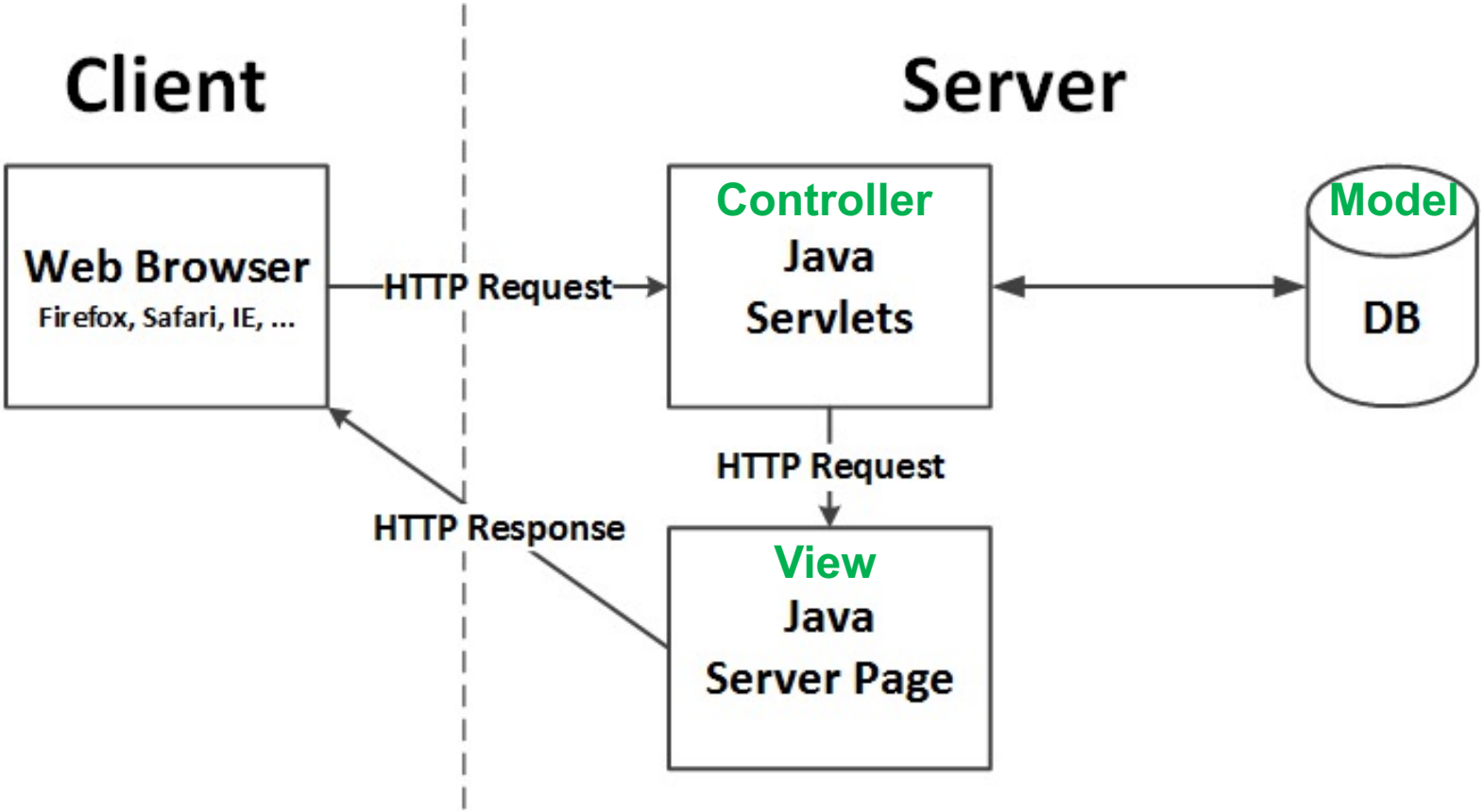
Model-**View**-Controller

- Contains no logic
- Sends events to the controller (e.g. Button Click)
- Includes information for display data from the model
- Navigation through the UI is controlled by the controller

Model-View-Controller

- Saves / loads data into / from the model
- Includes among other things the logic for the navigation of the view
  - e.g. via a state machine

Model-View-Controller



Voting

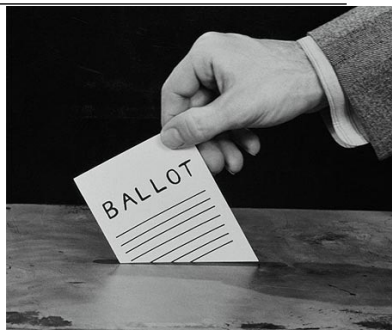
Context

- Redundant system with multiple results

Problem Forces

- Find the „correct“ result out of multiple results
- Results are calculated in parallel

Voting



http://goo.gl/NP1W9

Solution

- New component selects out of several results the „correct“ one.
- Selection is determined by algorithm

Modification

- Components, which are overruled can be treated as faulty components

Voting

Consequences

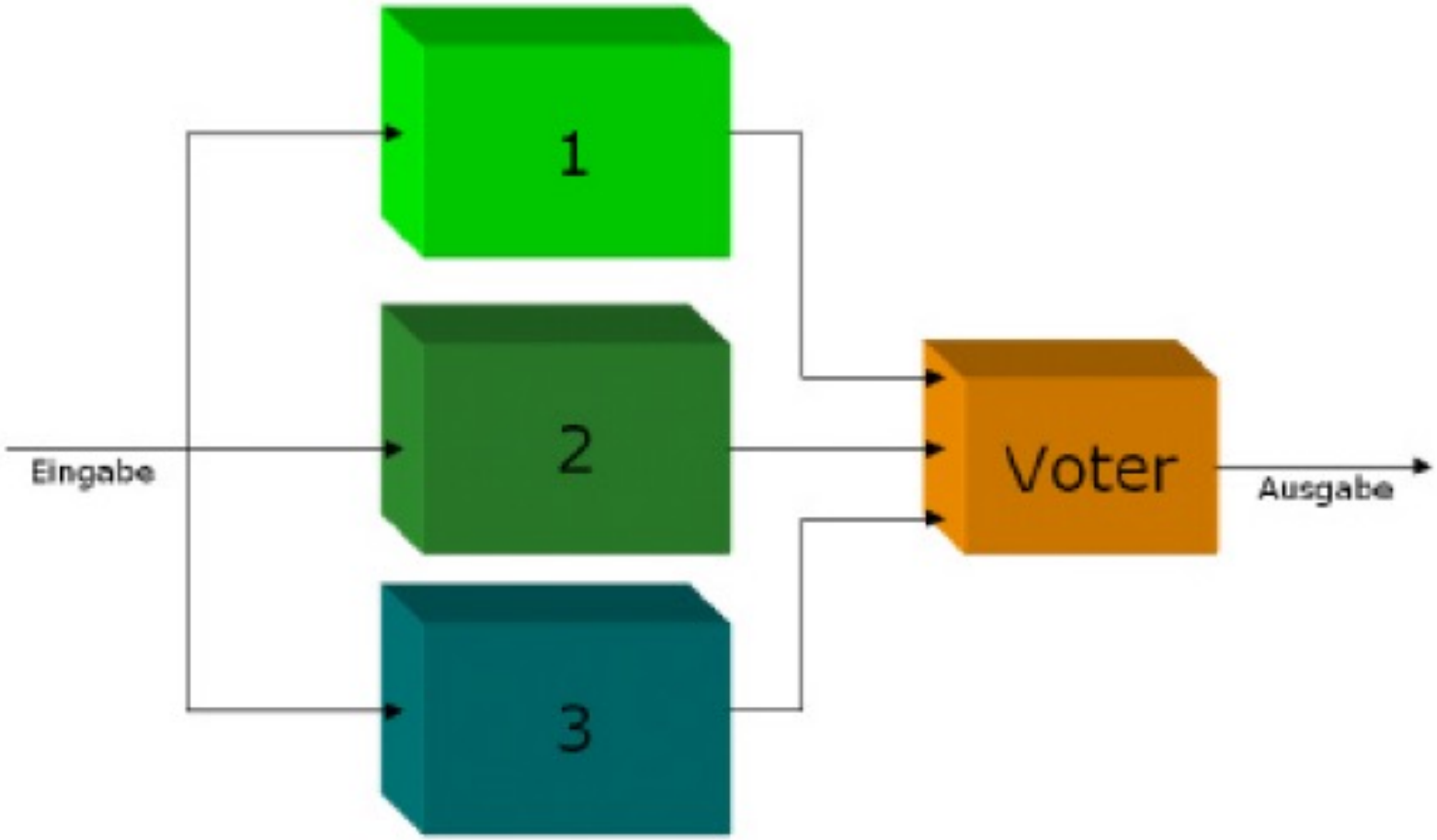
+ Redundancy provides a single result
+ Allows to detect faulty components

- Voter is new critical component
- Decision for „correct“ voting algorithm is very complicated

N-Version Programming

- Used Design Patterns: Redundancy and Voting
- Problem is solved by N-different algorithms
- Voter selects „correct“ result

N-Version Programming



http://www-ti.informatik.uni-tuebingen.de/~ruf/seminar0102/Fehlertoleranz.pdf, Seite 9

Thank you

References

- E. Gamma: Design Patterns: Elements of Reusable Object-Oriented Software
- F. Buschmann: Pattern-Oriented Software Architecture – On Patterns and Pattern Languages
- Pictures from Wikipedia, http://www.wikipedia.com