Clean Code

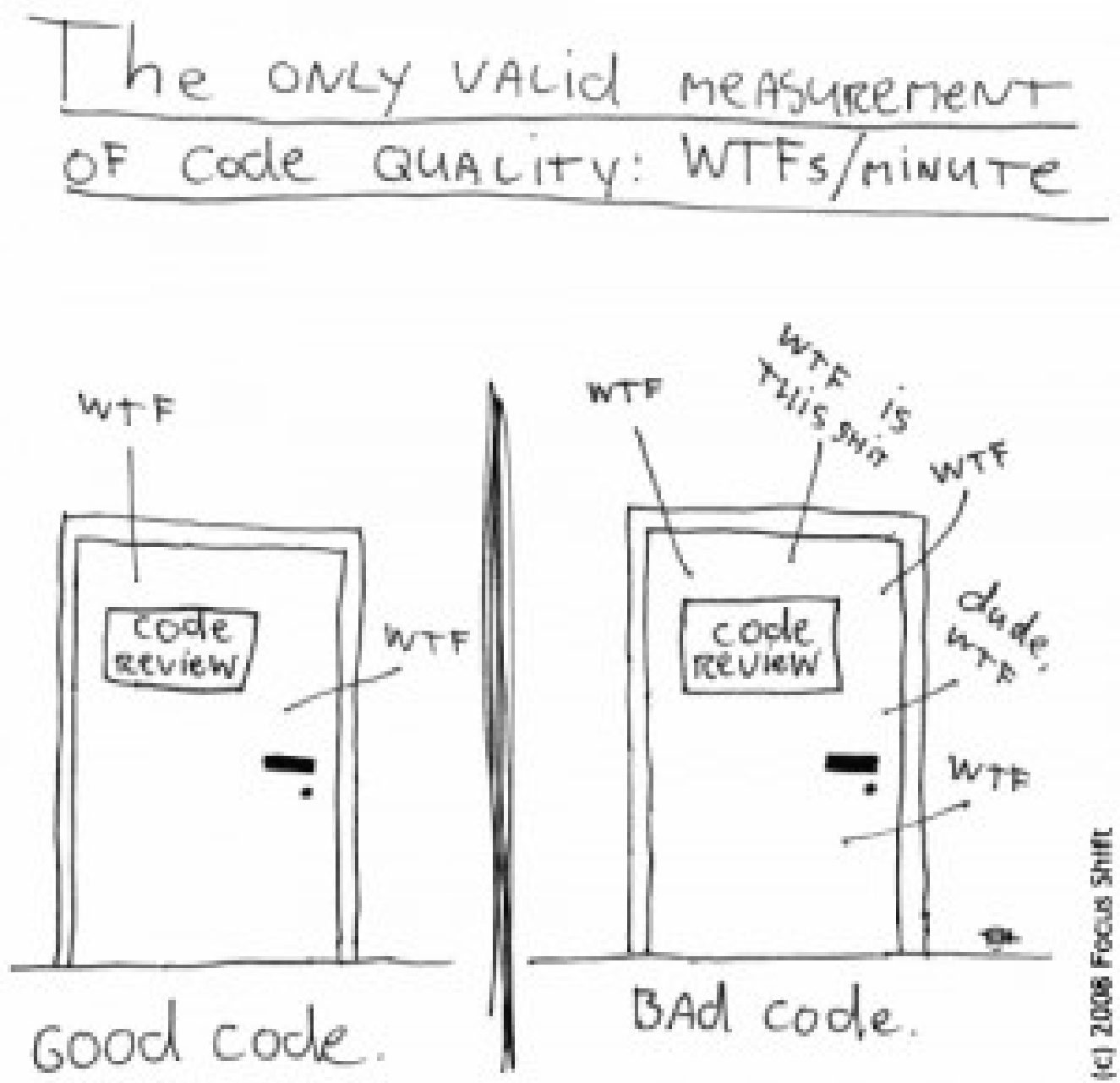
How to write comprehensible Code regarding cognitive abilities of human mind

24.02.2010

Java User Group Mannheim

Mario Gleichmann

Clean Code


The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this
WTF
code review
dude wtf
WTF
Bad code.
(c) 2008 Focus Shift

Mario Gleichmann

Books ... Not

Speaker at well known conferences ... Not

Just a

Humble Programmer

... who wants to improve ...

blog: http://gleichmann.wordpress.com

Motivation

Economics is the primary driver of software development

- Software should be designed to **reduce** a companies overall **cost**

-> delivering working software, raising the companies **efficiency**

-> keep costs of software development small over whole lifecycle

Motivation

The Cost of Software Development

COST software = COST initialDevelopment + COST maintenance

"The Cost of maintenance is usually much higher than the cost of initial development"

(Kent Beck)

Motivation

Maintainability Requirements

Code should be easy to

- **change**
- **adapt**
- **extend**
- **reuse**

The cost of Maintenance

The cost of Maintenance

"Making changes is generally easy

if you exactly know what needs to be changed"

Maintenance is expensive, because unterstanding existing code is time consuming and error prone

Challenges of Maintenance

- Preserve correctness / consistency
- Not break or otherwise destabilize production system
- Protect production data
- Improve the design or code that already exists
- Deployable into existing system with least disruption possible

Challenges of Maintenance

"Maintenance is intellectually complex"

- Requires innovation while obeying severe constraints
- Work with a concept, a design and its code all at the same time
- Work with fragmentary Knowledge about the concept / code

Values

"There is a difference between
getting something to work and getting something right.
It is about the value we place in the structure of our code"

Universal overaching themes of programming

- supporting some goals
- provide 'motivation' for achieving the goals

Values

Flexibility

- Extendable – statisfying new requirements
- Adaptable to new usage contexts
- Recombinable with initially unforseen collaborators

maintain potential to respond to change

keeping options open

Values

Simplicity

"Any fool can make things bigger, more complex and more violent.

It takes a touch of genius to move in the opposite direction"

(John Dryden)

No unnessecary complexity

Values

Simplicity

"The competent programmer is fully aware
of the strictly limited size of his own skull;

therefore he approaches the programming task in full humility,
and among other things he avoids clever tricks like the plague"

(The humble programmer)

Intellectually manageable programs

Values

Communication

"existing code is much more read than new code is written"
(Kent Beck)

Modules easy to change, firstly have to be easy to understand

Principles

Bridge between values and

patterns

practices

idioms

in order to finally achieve desired values

Principles give some explanation for the motivation of applied patterns or provide a guide in novel siuations

(Kent Beck)

Programming as a cognitive process

"The programmer, like the poet, works only slightly removed from pure thought-stuff.

He builds his castles in the air, from air, creating by exertion of the imagination"

(Fred Brooks)

Programming as a cognitive process

Computer programming potentially involves

- **understanding** (application context and possibility)
- **planning** (design)
- **imaging** (imagination and visualization)
- **logic** (conceptualization, language use, and knowledge)
- **creativity** (artistry)
- **work** (persistance, exploration, purpose and committment).

Programming as a cognitive process

Cognition

information processing

development of concepts

result of perception, learning and reasoning

Concepts

Concepts

Basic Structures of 'Knowledge'

- class of items, words, or ideas that are known by a common name
- includes multiple specific examples
- shares common features + rules
- embedded within a net of concepts

"Stack"

"Stack"

Store for Collecting Elements

"Stack"

Store for Collecting Elements

Pushing Elements to the Stack

Accessing Elements from the Stack

Removing Elements from the Stack

"Stack"

Store for Collecting Elements

Pushing Elements to the Stack

Accessing Elements from the Stack

Removing Elements from the Stack

LIFO Access Principle

Assimilation

Assimilation

Acquisition and Integration of new concepts

- Learning by examples
- Infering core Features + Rules
- Consistent integration into existing knowledge base

Assimilation

*"Look, a **Monoid** ..."*

M := <T extends String>

op := public T concat(T s1, T s2){ return s1 + s2;}

unit := ""

```
assertEquals(    concat( "a", concat( "b", "c" ) )  ,
                 concat( concat( "a", "b" ), "c" )  );

assertEquals(    concat( "a", "" )  ,    "a" );

assertEquals(    concat( "", "a" )  ,    "a" );
```

Assimilation

"Look, another ***Monoid*** *..."*

M := <T extends Integer>

op := public T add(T i1, T i2){ return i1.add(i2); }

unit := 0

```
assertEquals(    add( 1, add( 2, 3 ) )  ,  add( add( 1, 2 ), 3 )  );

assertEquals(    add( 23, 0 )   ,  23 );

assertEquals(    add( 0, 23 )   , 23 );
```

Assimilation

*"Yet another **Monoid** ..."*

```
M       := enum Hour{ ZERO( 0 ), ... ELEVEN( 11 ) }

op      := public Hour rotate( Hour h1, Hour h2 ){
                return Hour.from( ( h1.value + h2.value ) % 12 ); }

unit    := Hour.ZERO


assertEquals(    rotate( THREE, rotate( FIVE, SEVEN ) )  ,
                 rotate( rotate( THREE, FIVE ), SEVEN )  );

assertEquals(    rotate( EIGHT, ZERO ) ,    EIGHT );

assertEquals(    rotate( ZERO, EIGHT ) ,    EIGHT );
```

Assimilation

"Monoid"

Features

M := *Set of Elements*

op := *'Composition' - Operation*

M x M -> M

unit := *'Neutral' Element*

Assimilation

"Monoid"

Rules

Composition' – Operation ***o*** *preserves Associativity :*

for all (a, b, c) in M : $(a \circ b) \circ c = a \circ (b \circ c)$

'Neutral' Element ***e*** *doesn't 'hurt' during composition :*

for all a in M : $a \circ e = e \circ a = a$

Abstraction & Categorization

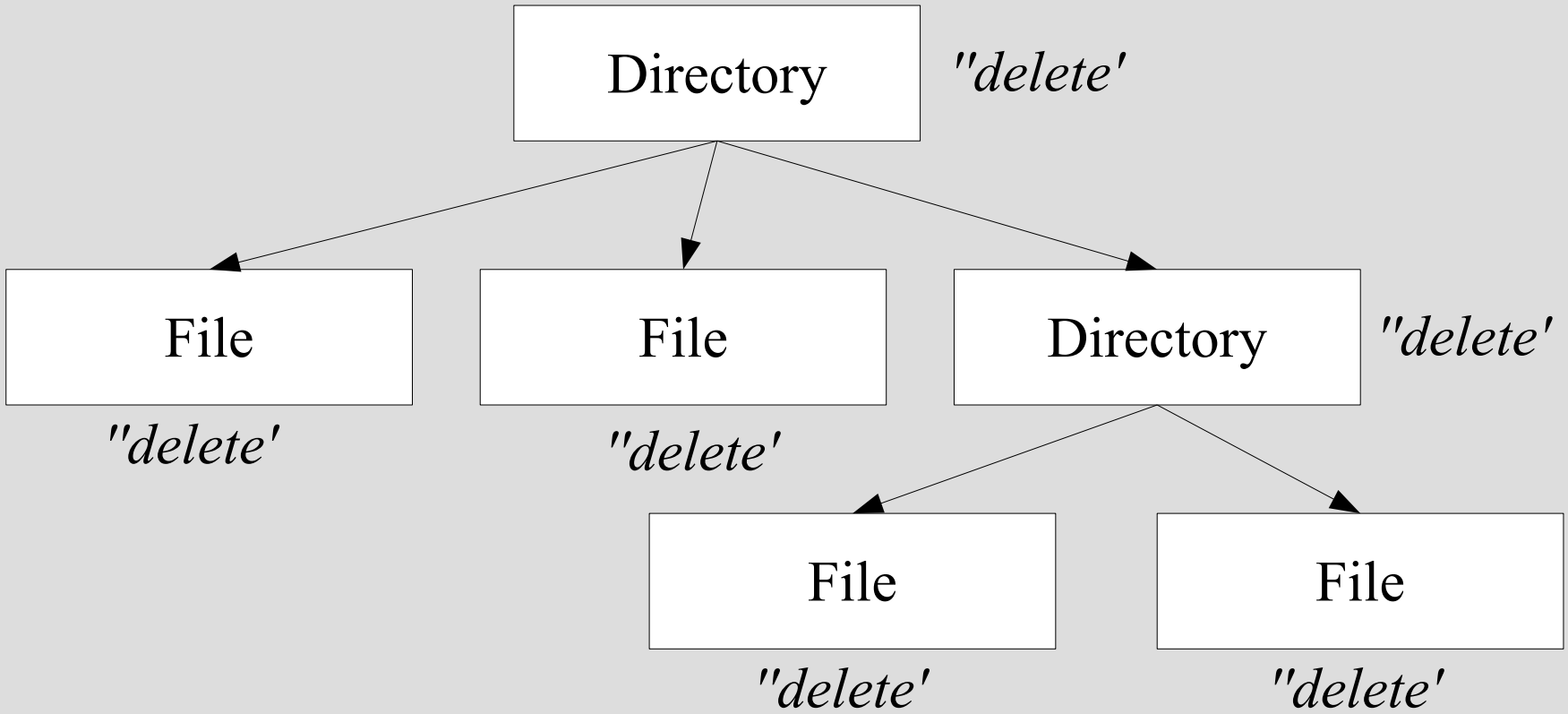
Abstraction & Categorization

Forming an impression of the general category from which the examples were drawn

- Comparison of examples
- Extraction of common Features and Rules
- Generalization from examples

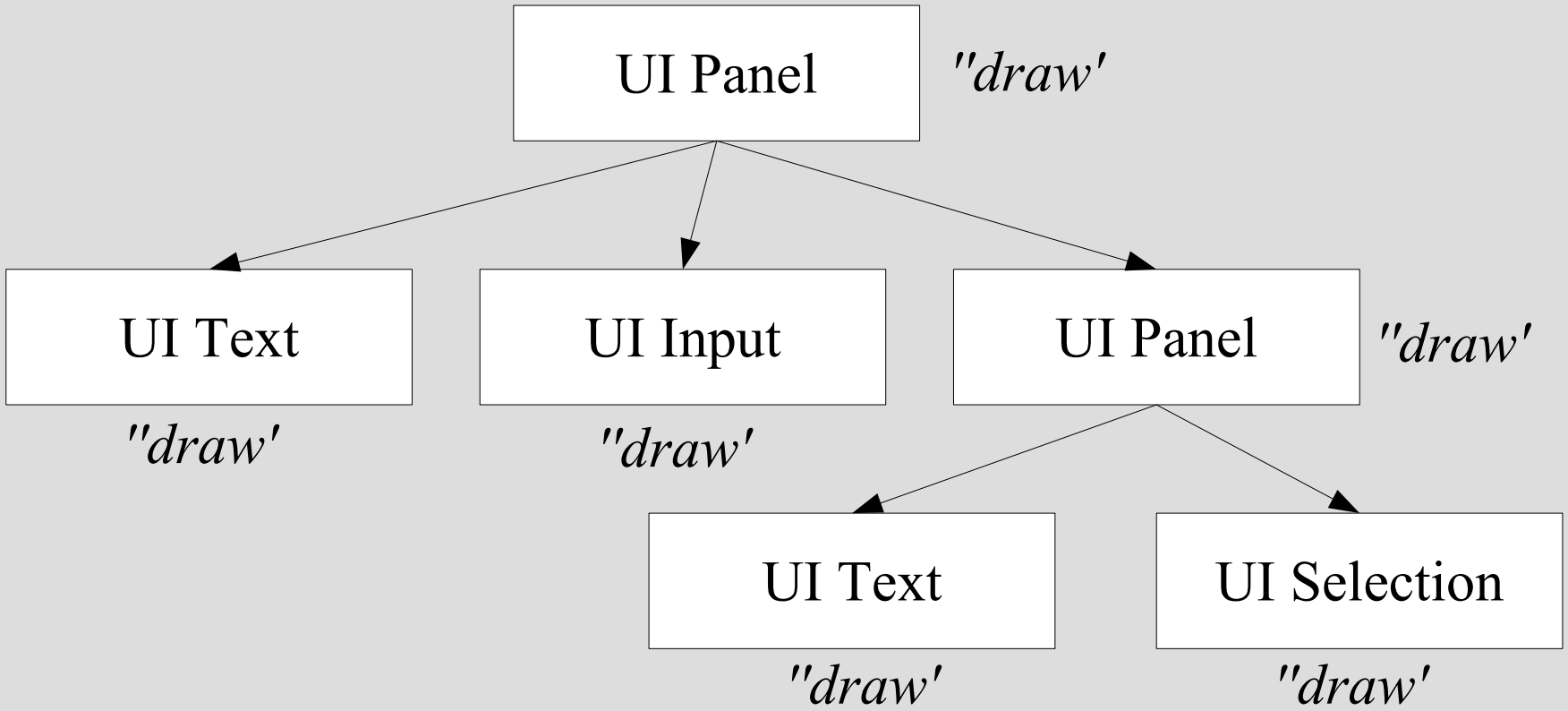
Abstraction & Categorization

*"Look, a **Composite** ..."*

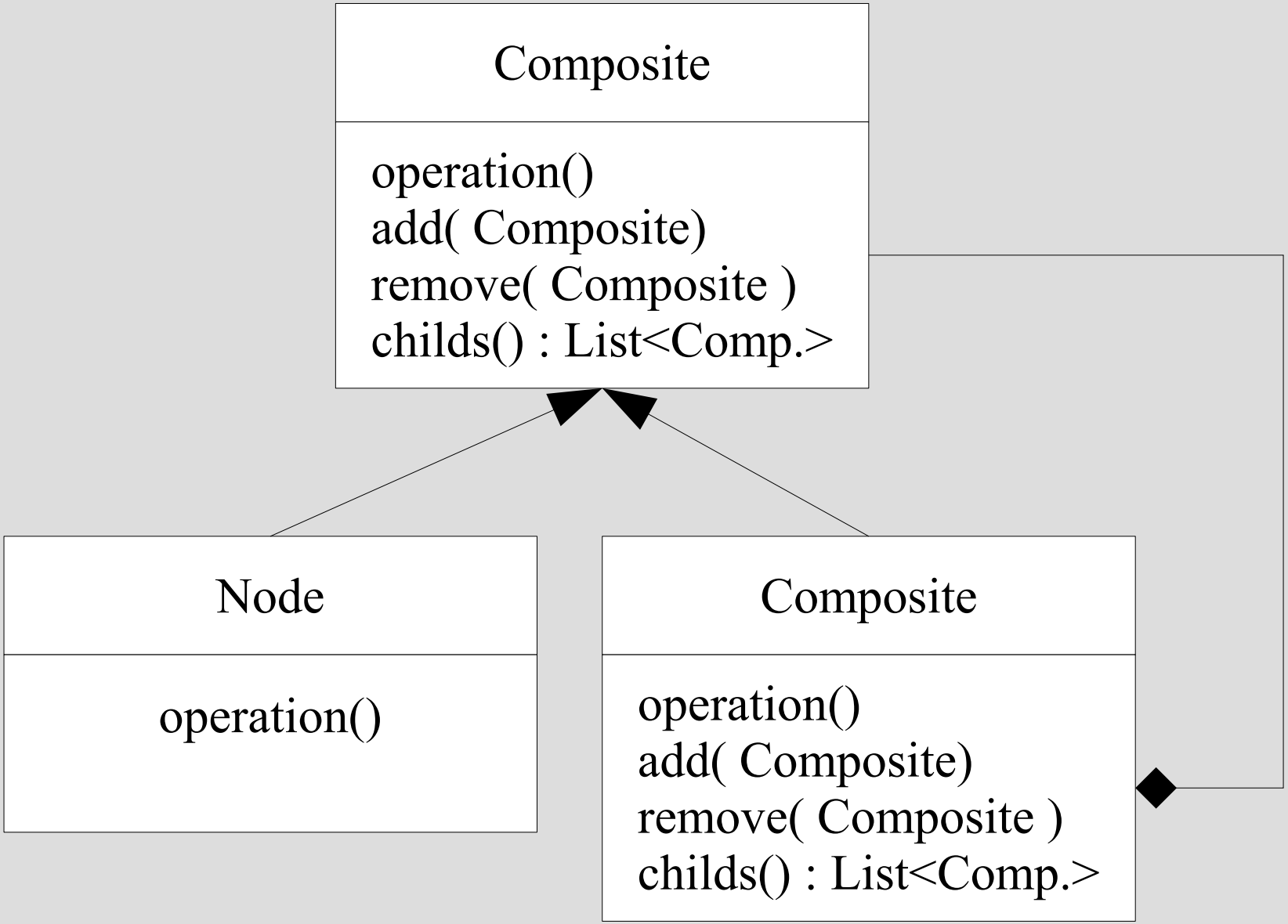


Abstraction & Categorization

*"Look, another **Composite** ..."*

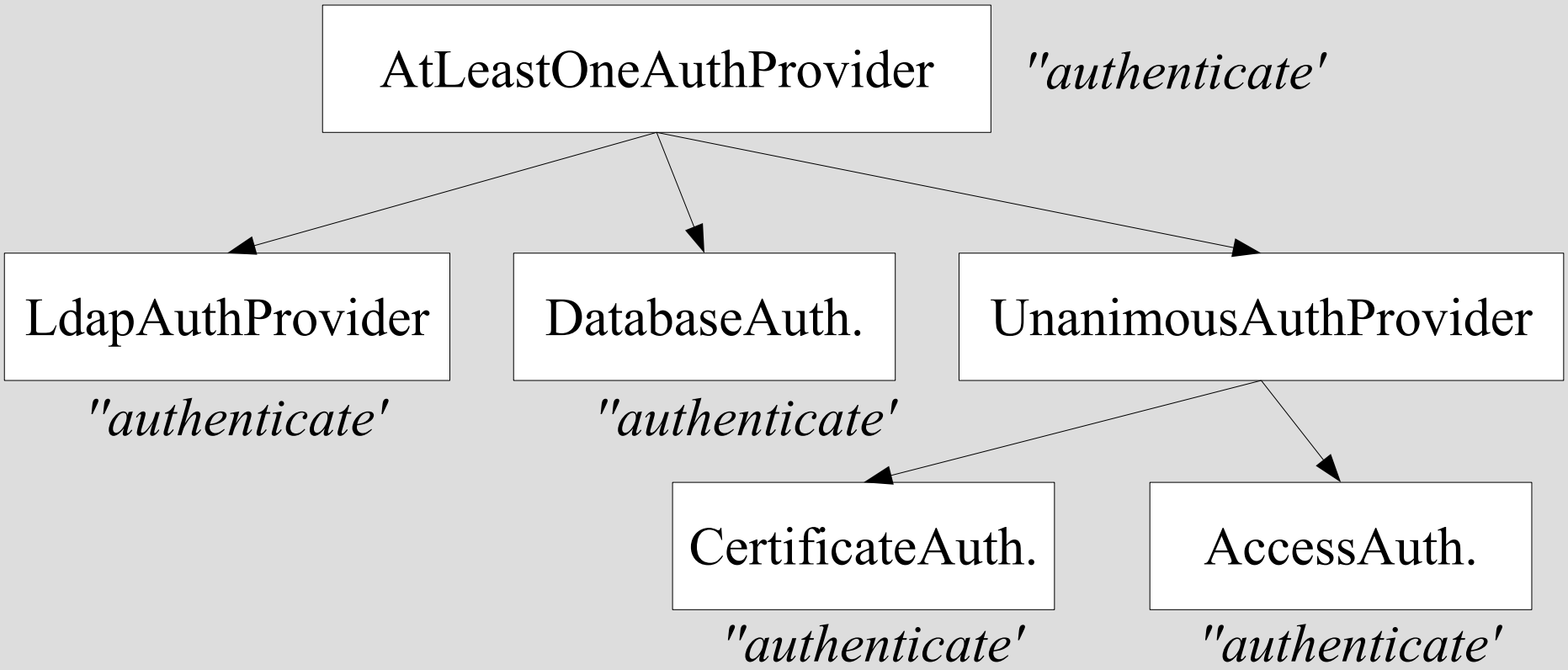


Abstraction & Categorization



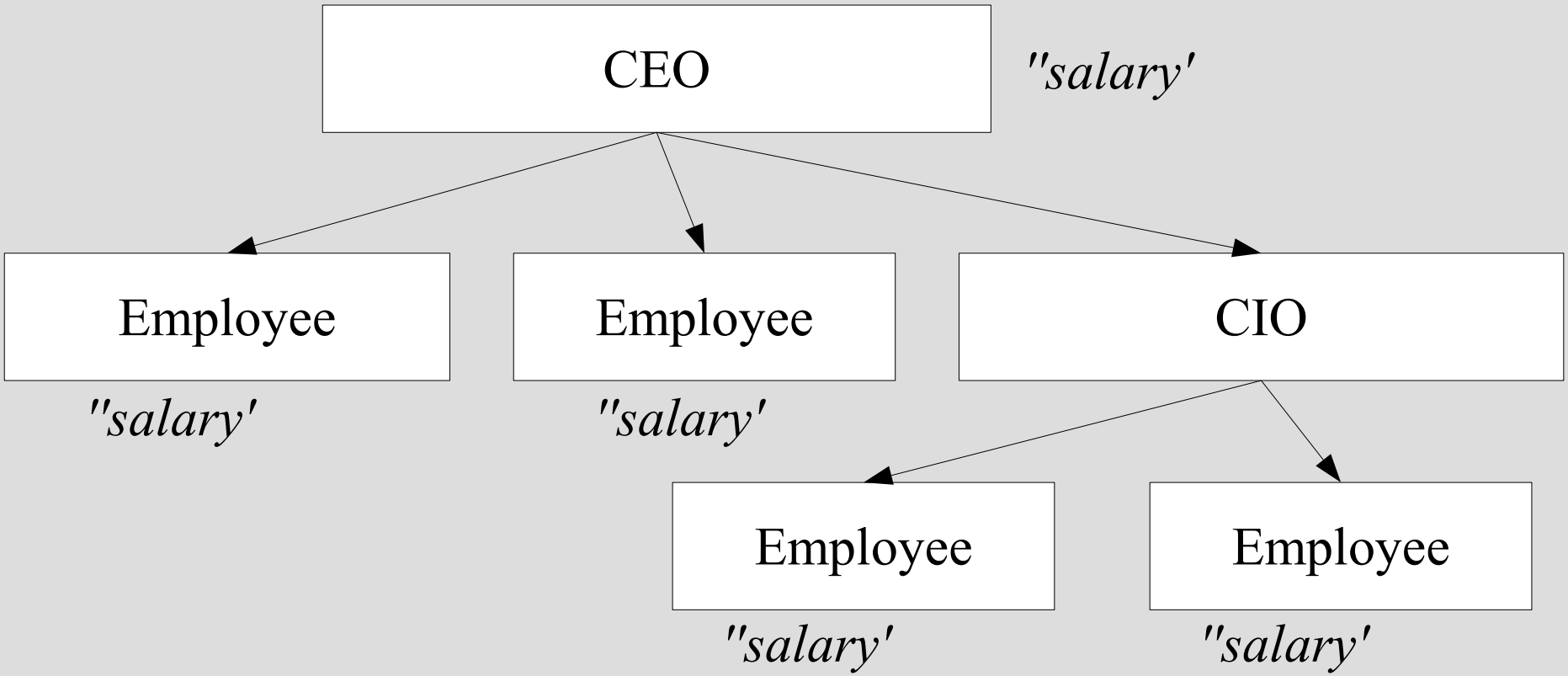
Abstraction & Categorization

Is this a Composite ?



Abstraction & Categorization

... and this ?



Adaption

Adaption

Is this a Stack ?

```
Class S{

    public void push( Element element ){ ... }

    public Element peek(){ ... }

    public Element pop(){ ... }

    pulic int size(){ ... }

    public boolean isEmpty(){ ... }
    ...
}
```

Adaption

... here are some 'rules' ...

```
S s = new S();

s.push( element_1 );
s.push( element_2 );
s.push( element_3 );

assertEquals( element_1, s.top() );

s.pop();
assertEquals( element_2, s.top() );

s.pop();
assertEquals( element_3, s.top() );
```

Adaption

... here are some 'rules' ...

```
S s = new S();

s.push( element_1 );
s.push( element_2 );
s.push( element_3 );

assertEquals( element_1, s.top() );

s.pop();
assertEquals( element_2, s.top() );

s.pop();
assertEquals( element_3, s.top() );
```

FIFO Access Principle

Adaption

... here are some 'rules' ...

Queue s = new **Queu**

```
s.push( element_1 );
s.push( element_2 );
s.push( element_3 );

assertEquals( element_1, s.top() );

s.pop();
assertEquals( element_2, s.top() );

s.pop();
assertEquals( element_3, s.top() );
```

FIFO Access Principle

Adaption

Accomodation

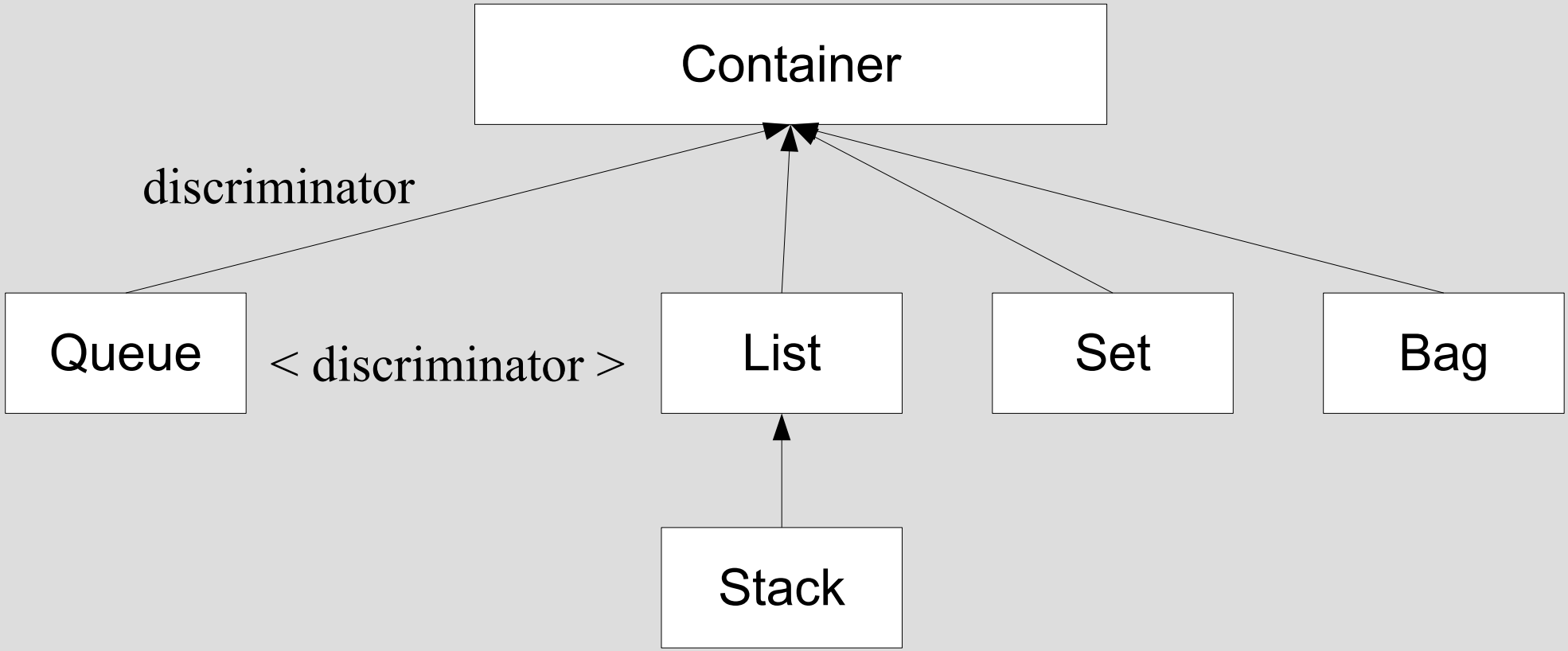
Restoration of consistent Knowledge base

Discrimination

Differentiation

Specialization

Hierarchies of Abstraction



Compression

Compression

Principle of complexity minimization

"one choose the simplest hypothesis consistent with the given data"

(Occams Razor)

"Organisms seek to understand their environment by reducing incoming information to a simpler, more coherent, and more useful form"

(Neisser and Weene)

Compression

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isKlingon( Keyword keyword ){

    if( keyword.equals( 'a' ) return true;
    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'abc' ) return true;

    return false;
}
```

Compression

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isKlingon( Keyword keyword ){

    if( keyword.equals( 'a' ) return true;
    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'abc' ) return true;

    return false;
}
```

Compression:

Klingon = 'starts with an 'a'

Compression

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isVolcan( Keyword keyword ){

    if( keyword.equals( 'a' ) return true;
    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'abc' ) return true;
    if( keyword.equals( 'cab' ) return true;

    return false;
}
```

Compression

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isVolcan( Keyword keyword ){

    if( keyword.equals( 'a' ) return true;
    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'abc' ) return true;
    if( keyword.equals( 'cab' ) return true;

    return false;
}
```

Compression:

Volcan = 'starts with an 'a' or is 'cab'

Compression

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isVolcan( Keyword keyword ){

    if( keyword.equals( 'a' ) return true;
    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'abc' ) return true;
    if( keyword.equals( 'cab' ) return true;

    return false;
}
```

main rule 'exception'

Volcan = 'starts with an 'a' or is 'cab'

Compression & Complexity

"we attempt to encode them in a manner as compact as possible. The more effectively the examples can be compressed, the lower the complexity. Complexity is, in essence, incompressibility"

(Kolmogorov)

induce the simplest category consistent with the observed examples

the most parsimonious generalization available 'wins'
(emphasizing the extraction of useful regularities)

Compression & Complexity

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isRomulan( Keyword keyword ){

    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'cab' ) return true;
    if( keyword.equals( 'b' ) return true;
    if( keyword.equals( 'ba' ) return true;

    return false;
}
```

Compression & Complexity

```
enum Keyword =     "a" , "ab", "abc",
                   "b", "ba", "bac",
                   "c", "ca", "cab",


boolean isRomulan( Keyword keyword ){

    if( keyword.equals( 'ab' ) return true;
    if( keyword.equals( 'cab' ) return true;
    if( keyword.equals( 'b' ) return true;
    if( keyword.equals( 'ba' ) return true;
```

"The most complex case is a 'string' so lacking in pattern or order that there is no better way to encode it than simply to quote it verbatim, so that the shortest encoding is about as long as the representation itself"

Re - Cognition

Re - Cognition

Pattern Matching

Identifying features and / or rules

Detecting connection-points

Stimuli

... by form / structure / function

Activating associations

Recognizing concepts

... do you recognize the underlying concept ?

```
interface Stepper<T>{

    public boolean isExhausted();

    public void step();

    public T getCurrent()
}
```

Recognizing concepts

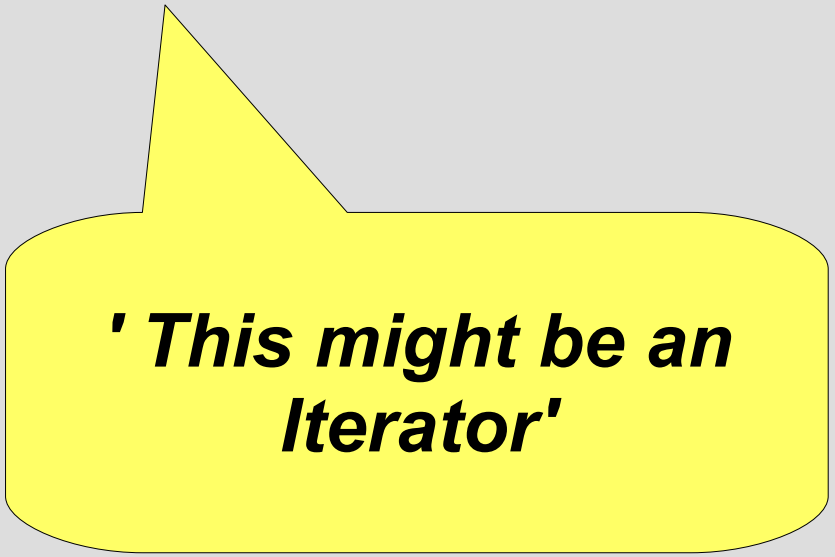
... do you recognize the underlying concept ?

```
interface ElementSupplier<T>{

    public boolean hasMoreElements();

    T nextElement();
}
```

Recognizing concepts

... do you recognize the underlying concept ?

```
interface ElementSupplier<T>{

    public boolean hasMoreElements();

    T nextElement();
}
```



Mental Distance

Mental Distance

... do you recognize the underlying concept ?

```
interface ElementConsumer{

    public void observe( BigElement e );

    public void calculate( AnotherElement e );

    public void extract( YetAnotherElement );
}
```

Mental Distance

... do you recognize the underlying concept ?

```
interface ElementInspector{

    public void inspect( BigElement e );

    public void inspect( AnotherElement e );

    public void inspect( YetAnotherElement );
}
```

Mental Distance

... do you recognize the underlying concept ?

```
interface ElementVisitor{

    public void visit( BigElement e );

    public void visit( AnotherElement e );

    public void visit( YetAnotherElement );
}
```

Mental Distance

"Similarity refers to the psychological nearness or proximity of two mental representations"

Featural

comparing the feature lists / rules that represent the concepts (commonalities and differences)

Structural

determining the differences requires finding the commonalities

Communication

Communication

```
date.compareTo( otherDate ) < 0
```

vs.

```
date.isBefore( otherDate )
```

Communication

"Language isn't private, but social. Shared Experience is shared Construction and Understanding of the 'world'. Therefore, using a Language is participation in a certain Language Game"

(Pelle Ehn, citing Wittgenstein)

Intersubjective consensus

Common agreement of Definitions and Opinions

Shared 'background knowledge'

Language Games

*"A **proxy** of the **service facade** is given to the **controller** via **Dependency Injection**"*

*"The **Builder** encapsulates certain **Strategies** for easy composement of miscellaneous document **composites**"*

*"Unfortunately, this **Factory** is a **Singleton**"*

*"You could **decorate** this **Mediator** in order to log the communication between the participants"*

Language Games

```
val projects = Map(        "Jan"  -> "IKT",
                           "Joe"  -> "TensE",
                           "Luca" -> "InTA" )

val customers = Map(       "IKT"  -> "Hanso GmbH",
                           "InTA" -> "RAIA Duo" )

val cities = Map(          "Hanso GmbH" -> "Stuttgart",
                           "RAIA Duo" -> "Mailand" )
```

*"i could use the **Option Monad** directly or rely on **list comprehensions** in order to traverse the maps"*

Language Games

```
def whereIs( person: String ) = {

    projects.get( person ).flatMap( customers get ).flatMap( cities get )
}
```

or

```
def whereIs( person: String ) = {

    for( project     <- projects get person;
         customer    <- customers get project;
         city        <- cities get customer
    )
    yield city;
}
```

Language Games

Domain Driven Design

*"the greatest value of a domain model is that it provides a **ubiquitous language** that ties together developers and domain experts"*

*"the vocabulary of a **ubiquitous languages** have been made* **explicit** ***in the model** (enriched with the names of patterns the team commonly applies to the domain model)"*

*"tightly relating the code to an underlying model gives the **code meaning** and makes the model relevant"*

*"in domain modelling you **shouldn't separate** the **concepts** from* the **implementation**"

Language Games

```
...

assertEquals(     "should adopt startJahr",
                  2000, haltephase.startJahr() );

assertEquals(     "should adopt endJahr",
                  2000, haltephase.endJahr() );

assertEquals(     "should calculate duration in days as difference between start and endDate",
                  276, haltephase.durationInDays() );

assertTrue(       "should detect Haltephase as historical",
                  haltephase.isHistorisch() );

assertFalse(      "should NOT detect Haltephase as currently running",
                  haltephase.isAktuell() );

assertEquals(     "should determine startValue equals to amount of first (buy-)transaction",
                  Betrag.parse( "10" ), haltephase.startValue() );

assertEquals(     "should determine endValue equals to sum of amount of all transactions",
                  new Betrag( new BigDecimal( "-5" ) ), haltephase.endValue() );

assertEquals(     "should detect no change of year within Haltephase (since Haltephase is within one single year)",
                  0, haltephase.getYearEndsWithin().size() );

assertTrue(       "should be active at arbitrary day within Haltephase)",
                  haltephase.wasActiveAt( new Datum( "2000", APR, "01" ) ) );

assertTrue(       "should be active at last day of Haltephase",
                  haltephase.wasActiveAt( new Datum( "2000", NOV, "05" ) ) );

assertFalse(      "should NOT be active at arbitrary day after Haltephase)",
                  haltephase.wasActiveAt( new Datum( "2000", DEZ, "01" ) ) );

...
```

Considerations

Consideration: Visibility

make Types / domain concepts visible

make attributes / rules visible

Consideration: Visibility

```
public reorderBook( String isbn ){
    ...

    if( isbn.substring( 0, 2 ).equals( "978" ){

        pubNr = isbn.substring( 6, 10 );
    }
    else{

        pubNr = isbn.substring( 8, 12 );
    }
    ...
}
```

Consideration: Visibility

```
public reorderBook( String isbn ){
    ...

    if( isbn.substring( 0, 2 ).equals( "978" ){

        pubNr = isbn.substring( 6, 10 );
    }
    else{

        pubNr = isbn.substring( 8, 12 );
    }
    ...
}
```

ISBN is NOT a String !!!

Consideration: Visibility

```
class Isbn{
    public Region getRegion(){ ... }
    public Integer getPublisherNumber(){ ... }
    public Integer getTitelNumber(){ ... }
    public Integer getChecksum(){ ... }
    public boolean isValid(){ ... }
}
```

```
public reorderBook( Isbn isbn ){
    ...
    isbn.getPublisherNumber();
    ...
}
```

Consideration: Visibility

```
class Isbn{
    public Region getRegion(){ ... }
    public Integer getPublisherNumber(){ ... }
    public Integer getTitelNumber(){ ... }
    public Integer getChecksum(){ ... }
    public boolean isValid(){ ... }
}
```

ISBN is a concept in its own right !!!

```
public reorderBook( Isbn isbn ){
    ...
    isbn.getPublisherNumber();
    ...
}
```

Consideration: Visibility

Give concepts a 'contoure'

Make the features visible on the contoures surface

Allow for Re-Cognition within your code

"implied concepts or collocated capabilities can be made more visible by recognizing these as objects of distinct and explicit types"

(Kevlin Henney)

Consideration: Consistence

Consideration: Consistence

Behaviour / Concept Rules are mostly not expressed at interface level

Infering expected Behaviour by (concept -) name resolution

Trusting the 'Language Game'

Principle of least Astonishment

Consideration: Consistence

```
class SimpleStack<T> implements Stack<T>{

    private List<T> elements = ...;

    public void push( T element ){ elements.add( element ); }

    public void top(){ return elements.get( 0 ); }

    public void pop(){ return elements.get( elements.getSize() ); }
    ...
}
```

Consideration: Consistence

```
class SimpleStack<T> implements Stack<T>{

    private List<T> elements = ...;

    public void push( T element ){ elements.add( element ); }

    public void top(){ return elements.get( 0 ); }

    public void pop(){ return elements.get( ...ents.getSize() ); }
    ...
}
```

Expectation broken

should return 'last' element

Consideration: Consistence

```
class SimpleStack<T> implements Stack<T>{

    private List<T> elements = ...;

    public void push( T element ){ elements.add( element ); }

    public void top(){ return elements.get( 0 ); }

    public void pop(){ return elements.get( elements.getSize() ); }
    ...
}
```

Expectation broken

should also remove 'last' element

Verifiable Specifications

Verifiable Specifications

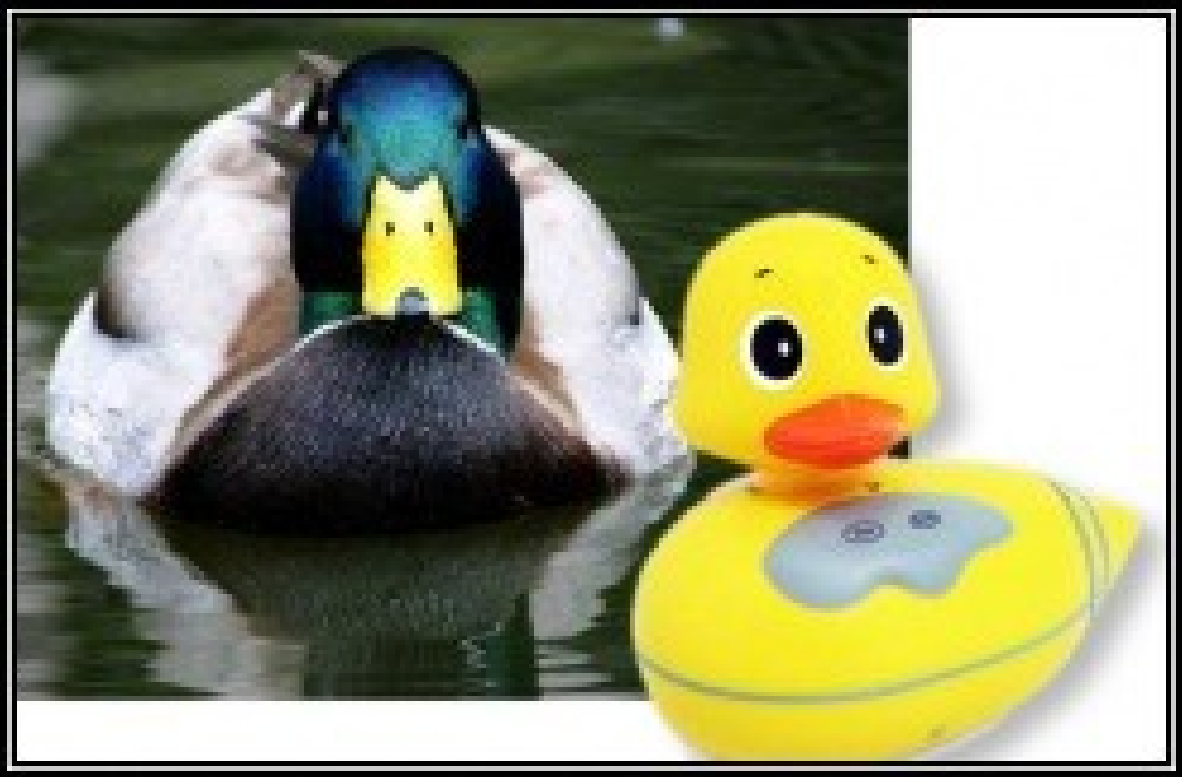
```
Stack stack = is(
        new DescriptionOf <Stack>(){
            public Stack isDescribedAs(){
                Stack<String> stack = new Stack<String>();
                stack.push( foo );
                stack.push( bar );
                return stack; } } );

    it( "should contain foo" );
    state( stack ).should( contain( foo ) );

    it( "should contain bar" );
    state( stack ).should( contain( bar ) );

    it( "should return bar when calling pop the first time" );
    state( stack.pop() ).should( returning( bar ) );

    it( "should return foo when calling pop the second time" );
    stack.pop();
    state( stack.pop() ).should( returning( foo ) );

    it( "should be empty after popping the two elements" );
    stack.pop();
    stack.pop();
    state( stack ).should( be ( empty() ) );
```

Design by Contract

Design by Contract

```
@Invariant( "this.size >= 0 and this.size <= this.capazity" )
public interface Stack { ... }

    @Postcondition( "return > 0" )
    public int getCapacity();

    public int getSize();

    @Precondition( "elem not empty and this.size < this.capacity" )
    @Postcondition( "elem == this.top and this.size == old:this.size + 1" )
    public void push( Object elem );

    @Postcondition( "this.top == old:this.top ) " )
    public Object getTop();

    @Postcondition( "(old:this.size>0) ==>
                                (return == old:this.top and this.size == old:this.size - 1) " )
    public Object pop();
}
```

Building Trust



LISKOV SUBSTITUTION PRINCIPLE

"if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties"

(strong behavioral subtyping)

Consideration: Locality

Consideration: Locality

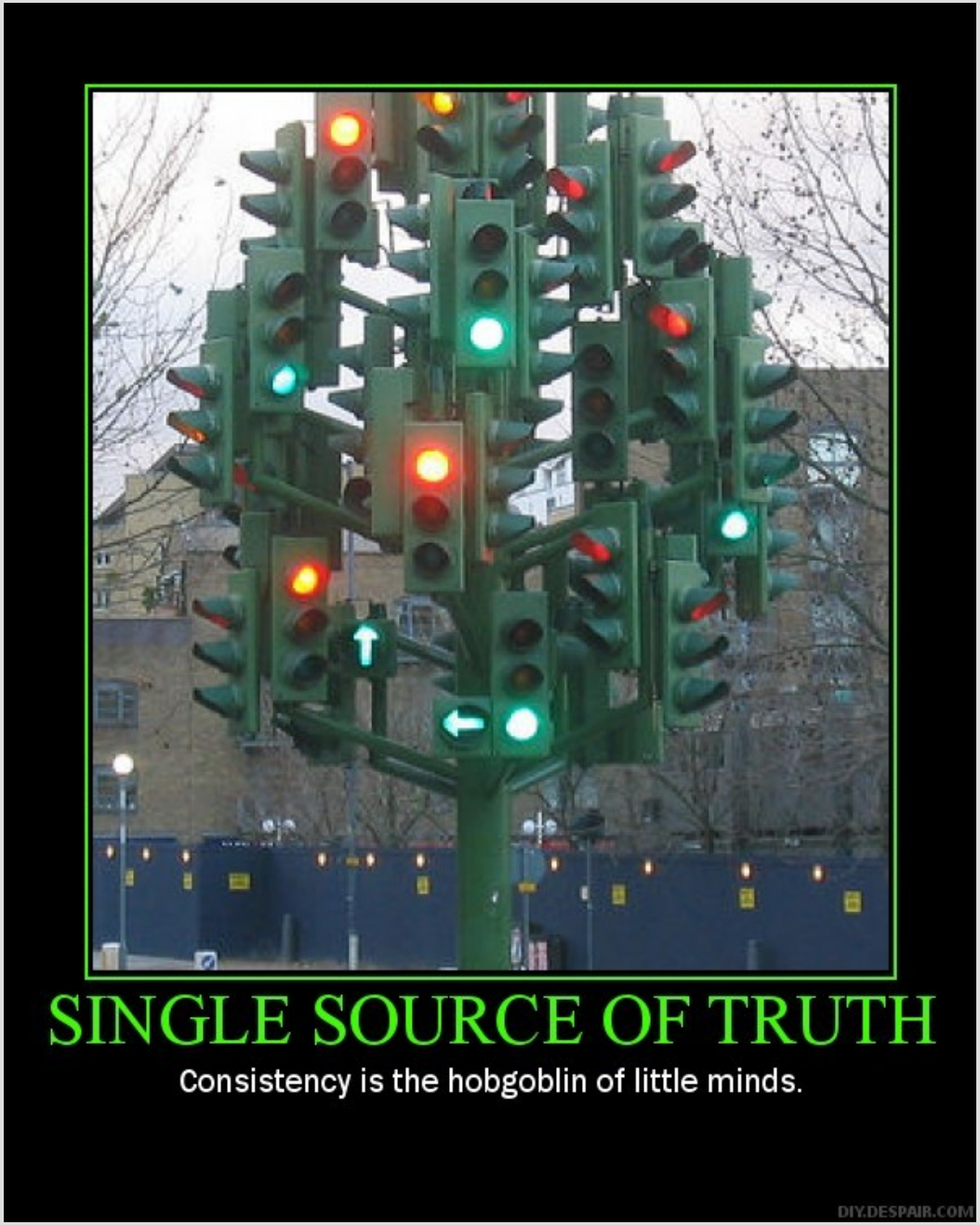
```
public void calculateFee( String depotStammNr ){

    if( depotStammNr.endsWith( "A" ) ||
        depotStammNr.endsWith( "P" ) ) {

        // some complex calculation
    }
    else{

        // some other complex calculation
    }
    ...
}
```

Don't Repeat Yourself

Don't Repeat Yourself

```
public void calculatePerformance( String depotStammNr ){

    if( depotStammNr.endsWith( "A" ) ||
        depotStammNr.endsWith( "P" ) ) {

        // some complex calculation
    }
    else{

        // some other complex calculation
    }
    ...
}
```

Single Source of Truth



Single Source of Truth

```
class DepotStammNrUtils{
    ...
    public static boolean isAktivDepot( String depotStammNr ){
        return depotStammNr.endsWith( "A" );
    }
    public static boolean isPassivDepot( String depotStammNr ){ ... }
}

...
public void calculatePerformance( String depotStammNr ){

    if( DepotStammNrUtils.isAktivDepot( depotStammNr ) ||
        DepotStammNrUtils.isPassivDepot( depotStammNr ) ) {

        // some complex calculation
    }
    ...
```

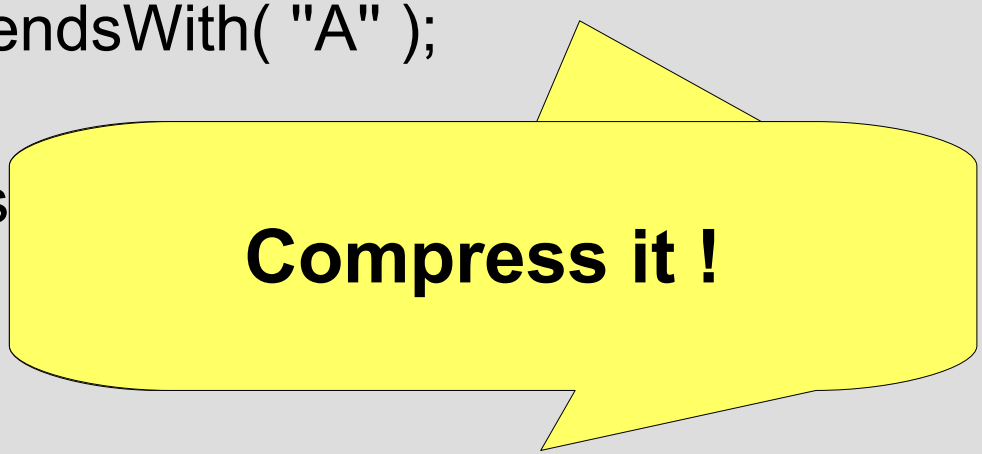
Single Source of Truth

```
class DepotStammNrUtils{
    ...
    public static boolean isAktivDepot( String depotStammNr ){
        return depotStammNr.endsWith( "A" );
    }
    public static boolean isPassiv                                    }
}

...
public void calculatePerformance( String depotStammNr ){

    if( DepotStammNrUtils.isAktivDepot( depotStammNr ) ||
        DepotStammNrUtils.isPassivDepot( depotStammNr ) ) {

        // some complex calculation
    }
    ...
```

Cohesion ?

Single Source of Truth

```
class DepotStammNrUtils{
    ...
    public static boolean isAktivDepot( String depotStammNr ){
        return depotStammNr.endsWith( "A" );
    }
    public static boolean isPas                                    ... }
}

...
public void calculatePerformance( String depotStammNr ){

    if( DepotStammNrUtils.isAktivDepot( depotStammNr ) ||
        DepotStammNrUtils.isPassivDepot( depotStammNr ) ) {

        // some complex calculation
    }
    ...
```



Encapsulation

"separate the contractual interface of an abstraction and its implementation"

(Grady Booch)

Encapsulation

```
class DepotStammNr{

    private String depotNr;

    public boolean isPartOfKombiDepot(){ ... }
    ...
}


public void calculateFee( DepotStammNr depotStammNr ){

    if( depotStammNr.isPartOfKombiDepot() ) {

        // some complex calculation
    }
    ...
}
```

Encapsulation

```
class DepotStammNr{

    private String depotNr;

    public boolean isPartOfKombiDepot(){ ... }
    ...
}


public void c

    if( depotS

        // some complex calculation
    }
    ...
}
```

Again, 'Depot-Stamm-Nr' is a concept in it's own right, holding discrete features and rules

Encapsulation

make it easy to build autonomic mental representations of a concept

make it easy to build an abstraction which hides the details

keep the details out of the 'business'

Encapsulation

```
class KaufAuftrag implements Auftrag{

    private Fonds fondsToBuy;
    private Stueckzahl anteileToBuy;
    ...

    public boolean isKauf(){
        return true;
    }
    ...
}
```

Encapsulation

```
public Betrag calculateSaldoFor( Collection<Auftrag> aufraege ){

    Betrag saldo = Betrag.ZERO;

    for( Auftrag auftrag : auftraege ){

        if( auftrag.isKauf() )
            saldo.add(    auftrag.getFonds().getPreis()
                                .multiply( auftrag.getAnteile() ) );

        if( auftrag.isVerkauf )
            saldo.add(    auftrag.getFonds().getPreis()
                                .multiply( auftrag.getAnteile() )
                                    .negate() );
        ...
}
```

Encapsulation

```
public Betrag calculateSaldoFor( Collection<Auftrag> aufraege ){

    Betrag saldo = Betrag.ZERO;

    for( Auftrag auftrag : auftraege ){

        if( auftrag.isKauf() )
            saldo.add(    auftrag.getFonds().getPreis()
                                .multiply( auftrag.getAnteile() ) );

        if( auftrag.is
            saldo.a

        ...
}
```

Why do i have to know about Fonds in order to calculate an Orders effective amount ?

(Law of Demeter)

Encapsulation

```
public Betrag calculateSaldoFor( Collection<Auftrag> aufraege ){

    Betrag saldo = Betrag.ZERO;

    for( Auftrag auftrag : auftraege ){

        if( auftrag.isKauf() )
            saldo.add(    auftrag.getFonds().getPreis()
                                .multiply( auftrag.getAnteile() ) );

        if( auftrag.i
            saldo.a

        ...
}
```

Why do i have to know about calculating an Orders effective amount at all?

Tell, don't ask

"Procedural code gets information then makes decisions. Object-oriented code tells objects to do things"

(Alec Sharp)

Tell, don't ask

```
VerkaufAuftrag extends Auftrag{
    ...
    public Betrag getEffectiveBetrag(){
        return fonds.getPreis().multiply( anteile ).negate();
    }
}
```

```
public Betrag calculateSaldoFor( Collection<Auftrag> aufraege ){

    Betrag saldo = Betrag.ZERO;

    for( Auftrag auftrag : auftraege ){
        saldo.add( auftrag.getEffectiveBetrag() );
        ...
}
```

Tell, don't ask

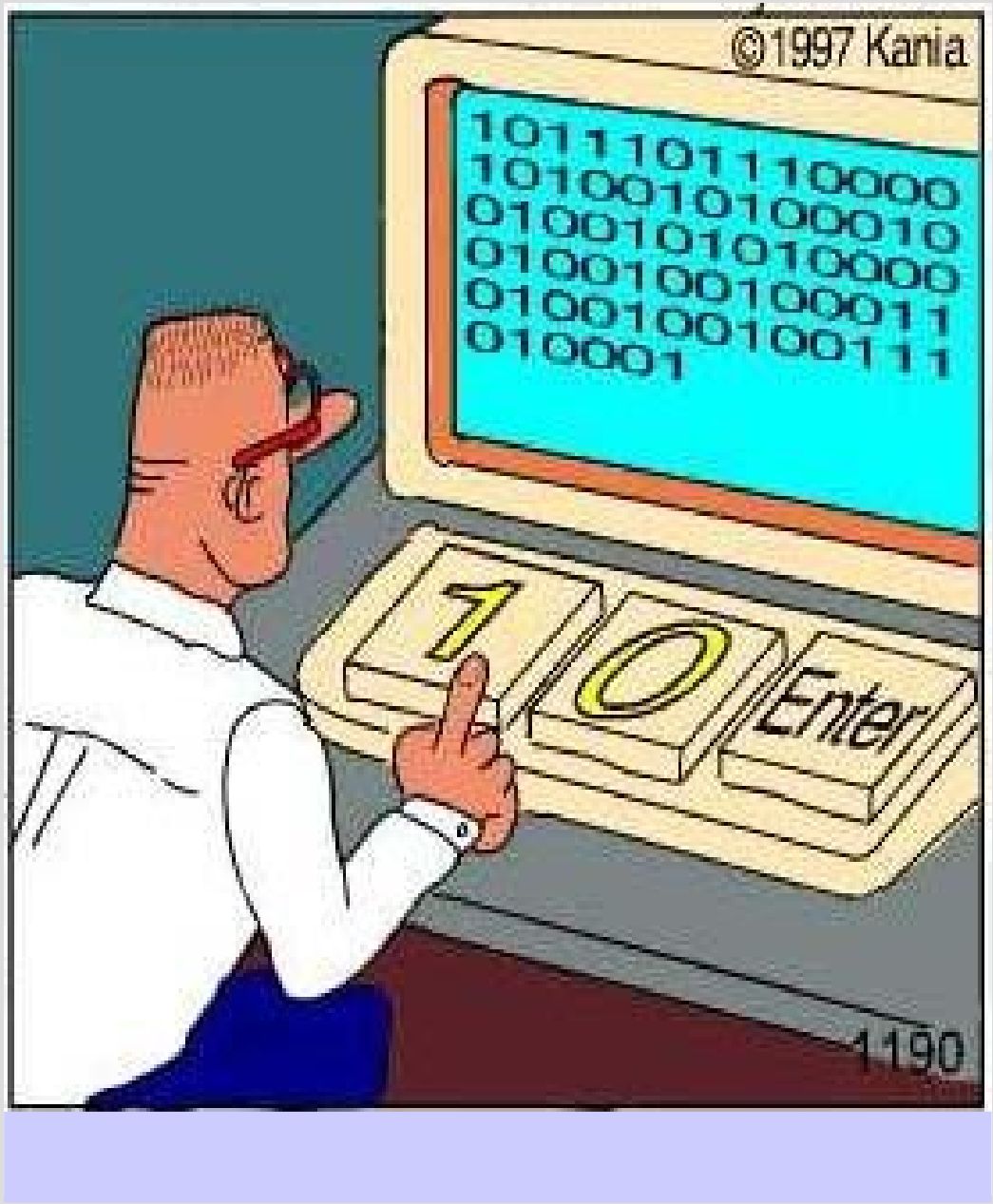
```
VerkaufAuftrag extends Auftrag{
    ...
    public Betrag getEffectiveBetrag(){
        return fonds.getPreis().multiply( anteile ).negate();
    }
}
```

'Compress it'

Replace conditionals with Polymorphism

```
public Betrag calculateS                                    {

    Betrag saldo = Betrag.ZERO;

    for( Auftrag auftrag : auftraege ){
        saldo.add( auftrag.getEffectiveBetrag() );
        ...
}
```

Consideration: Abstraction



Consideration: Abstraction

```
String separator =" ";
String query = "select ";
if( ... ) query += " name "; separator = ", ";
if( ... ) query = query + separator + " alter ", separator = ", ";
query += " from person "
if( existFilter ){
    query += " where  "
    if( ... ) query += " stadt = " + stadt; separator = " and ";
    else{
        query += " stadt in ( ";
        for( String stadt in staedte )
            query += stadt + inSeparator; inSeparator = ", ";
        query += " ); separator = " and "
    }
...
```

Consideration: Abstraction

```
String separator =" ";
String query = "select ";
if( ... ) query += " name ";
if( ... ) query = query + sep
query += " from person "
if( existFilter ){
    query += " where  "
    if( ... ) query += " stadt = " + stadt; separator = " and ";
    else{
        query += " stadt in ( ";
        for( String stadt in staedte )
            query += stadt + inSeparator; inSeparator = ", ";
        query += " ); separator = " and "
    }
...
```

Do you see the core idea ?

(Hint: it's NOT about String Handling)

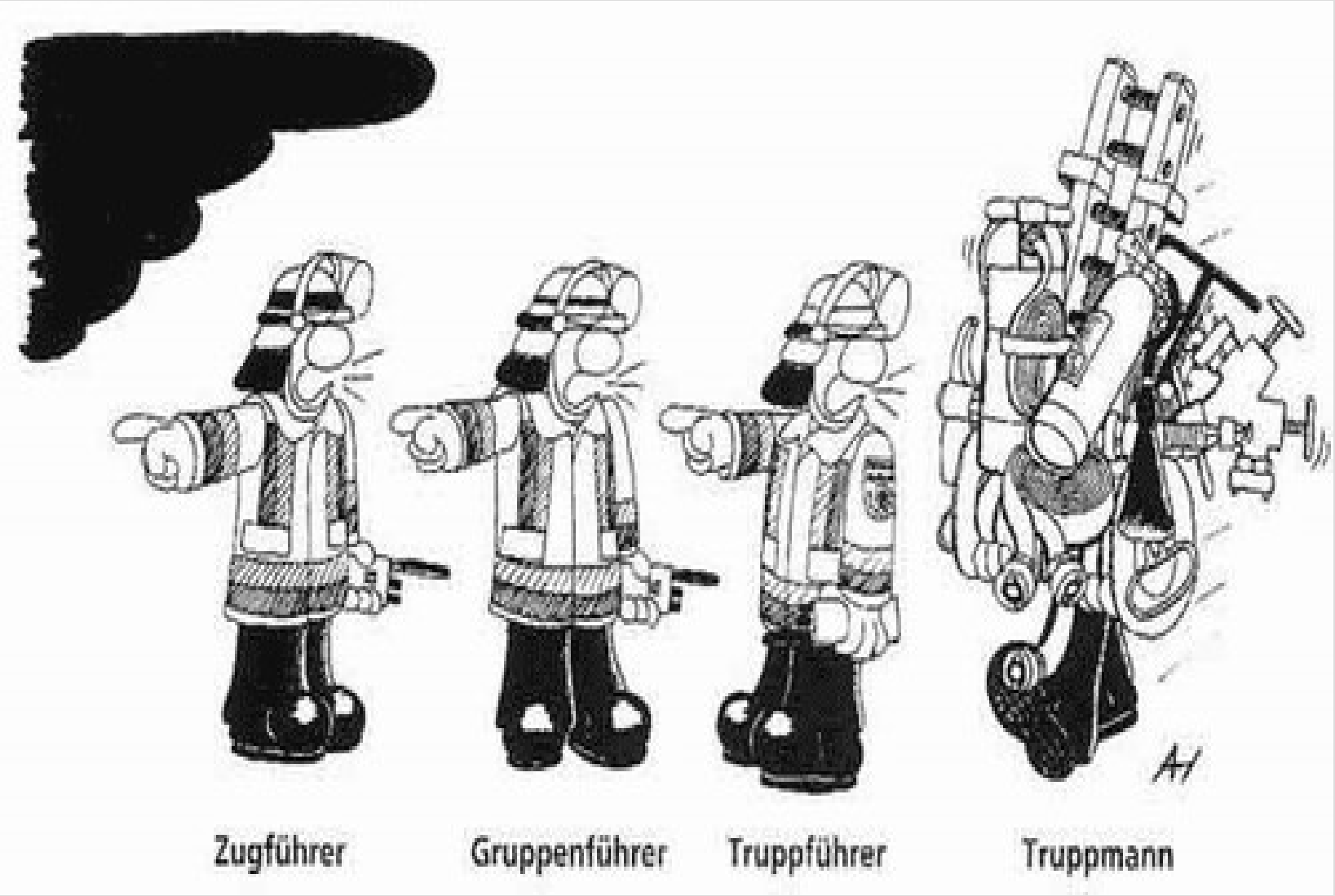
Consideration: Abstraction

```
String separator =" ";
String query = "select ";
if( ... ) query += " name ";
if( ... ) query = query + sep
query += " from person "
if( existFilter ){
    query += " where  "
    if( ... ) query += " stadt = " + stadt; separator = " and ";
    else{
        query += " stadt in ( ";
        for( String stadt in staedte )
            query += stadt + inSeparator; inSeparator = ", ";
        query += " ); separator = " and "
    }
...
```

Building SQL-Statements ...

... is NOT about String-Manipulation

SEP Principle



Abstraction

"the only way the brain can keep up with the complexity of reality is to reduce large complex systems into simple, easily manipulated symbols"

(Kevlin Henney)

abstract the stuff away that is not relevant
for expressing the core idea

make the details *someone else' problem*

Abstraction

```
Query personQuery = Query.onTable( "Person" )

personQuery.projectTo( "name" )
personQuery.projectTo( "alter " )

personQuery.add( criteria( "stadt" ).equals( stadt ) );
personQuery.add( criteria( "stadt" ).in( staedte ) );

...
```

Abstraction

```
Query personQuery = Query.onTable( "Person" )

personQuery.projectTo( "name" )
personQuery.projectTo( "alter " )

personQuery.add( criteria( "stadt" ).equals( stadt ) );
personQuery.add( criteria( "stadt" ).in( staedte ) );

...
```

"i tell you what i want -
you'll generate the query-string"

Abstraction

```
Query personQuery = Query.onTable( "Person" )

personQuery.projectTo( "name" )
personQuery.projectTo( "alter " )

personQuery.add( criteria( "stadt" ).equals( stadt ) );
personQuery.add( criteria( "stadt" ).in( staedte ) );

...
```

"...the detail is somebodies problem"

("i dont care how the string is generated, or tuned for a specific database")

Abstraction

```
Query personQuery = Query.onTable( "Person" )

personQuery.projectTo( "name" )
personQuery.projectTo( "alter " )

personQuery.add( criteria( "stadt" ).equals( stadt ) );
personQuery.add( criteria( "stadt" ).in( staedte ) );

...
```

'Separation of concerns'

("i dont care how the string is generated, or tuned for a specific database")

Abstraction

```
Query personQuery = Query.onTable( "Person" )

personQuery.projectTo( "name" )
personQuery.projectTo( "alter " )

personQuery.add( criteria( "stadt" ).equals( stadt ) );
personQuery.add( criteria( "stadt" ).in( staedte ) );

...
```

make the concept of a Query

'compressable'

to it's core intention

Consideration: Abstraction

```
public void validate( Order order ) throws ValidationException{

    orderDateValidator.validateOrderDate( order );

    orderAmountValidator.checkAmount( order );
    ...
    if( order.isBuy() ){
        bankAccountValidator.assertEnoughMoneyFor( order )
    }
    ...
}
```

Consideration: Abstraction

```
public void validate( Order order ) throws ValidationException{

    orderDateValidator.validateOrderDate( order );

    orderAmountValidator.checkAmount( order );
    ...
    if( order.isBuy() ){
        bankAccountValidator.assertEnoughMoneyFor( order )
    }
    ...
}
```

Compress

quantitative Associations

to

qualitative Association

Consideration: Abstraction

```
public void validate( Order order ) throws ValidationException{

    orderDateValidator.validate( order );

    orderAmountValidator.validate( order );
    ...
    if( order.isBuy() ){
        bankAccountValidator.validate( order )
    }
    ...
}
```

Compress

quantitative Associations

to

qualitative Association

Open Closed Principle

software entities
(classes, modules, functions, etc.)
should be open for extension,
but closed for modification

Open Closed Principle

```
Interface OrderValidator{
    public void validate( Order order ) throws ValidationException
}
```

```
private List<OrderValidator> validators = ... // e.g. injected

public void validate( Order order ) throws ValidationException{

        for( OrderValidator validator : validators ){
            validator.validate( order );
        }
}
```

Consideration: Abstraction

abstraction is a matter of choice. The quality of abstraction relates to compression and balance of forces.

(Kevlin Henney)

of itself, abstraction is neither good nor bad

encapsulation is a vehicle for abstraction

Consideration: Minimalism

Consideration: Minimalism

```
class HibernateCustomerDAO implements CustomerDAO{
    public Customer findById( Integer custId ){
        Logger.info( ... );
        accessCounter++;
        if( cache.containsKey( custId ){
            return cache.get( custId );
        }
        else{
            String query = ... // build Query String
            Customer customer = ... // executeQuery
            cache.put( custId, customer );
        }
        Logger.info( ... );
        return customer;
    }
    ...
}
```

Consideration: Minimalism

```
class HibernateCustomerDAO implements CustomerDAO{
    public Customer findById( Integer custId ){
        Logger.info( ... );
        accessCounter++;
        if( cache.containsKey( custId ){
            return cache.get( custId );
        }
        else{
            String query = ... // build Query String
            Customer customer = ... // executeQuery
            cache.put( custId, customer );
        }
        Logger.info( ... );
        return customer;
    }
    ...
}
```

Where's the core idea ?

Consideration: Minimalism

```
class HibernateCustomerDAO implements CustomerDAO{
    public Customer findById( Integer custId ){
        Logger.info( ... );
        accessCounter++;
        if( cache.containsKey( custI
            return cache.get( custId )
        }
        else{
            String query = ... // build Query String
            Customer customer = ... // executeQuery
            cache.put( custId, customer );
        }
        Logger.info( ... );
        return customer;
    }
    ...
}
```

What's the core idea ?

(and what's the 'clutter' – in respect to data access)

Single Responsibility Principle

Every class should have only
one reason to change

Single Responsibility Principle

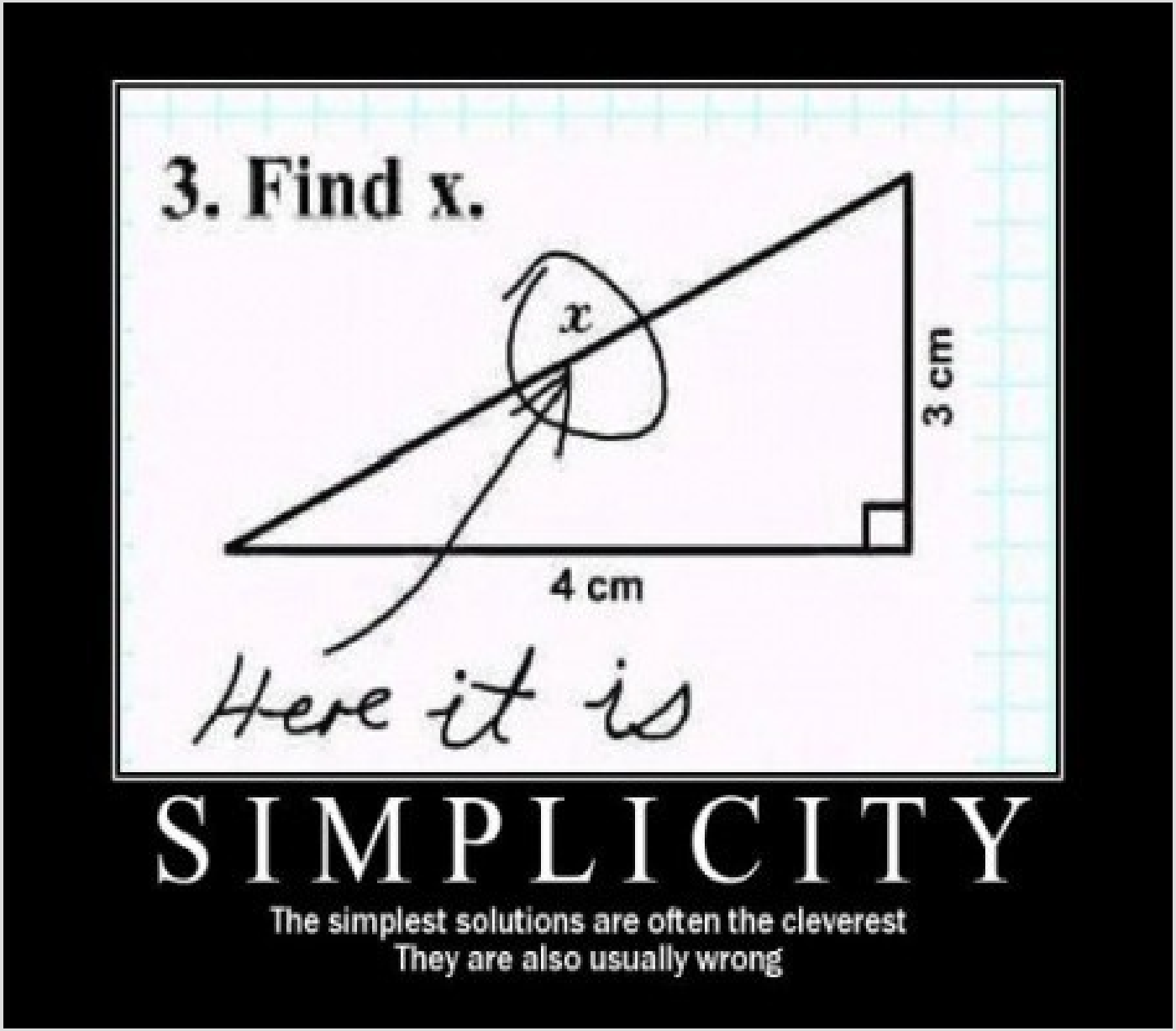
```
class CachingCustomerProxy implements CustomerDAO{

    CustomerDAO target = ... // injected

    public Customer findById( Integer custId ){
        if( cache.containsKey( custId ){
            return cache.get( custId );
        }
        else{
            Customer customer = target.findById( custId );
            cache.put( custId, customer );
        }
        return customer;
    }
    ...
}
```

Single Responsibility Principle

```
class CachingCustomerProxy implements CustomerDAO{
    CustomerDAO target = ... // injected
    public Customer findById( Integer custId ){
        if( cache.containsKey( custId ){
            return cache.get( custId );
        }
        else{
            Customer customer = target.findById( custId );
            cache.put( custId, customer );
        }
        return customer;
    }
    ...
}
```

Keep the concept of
accessing data
and
caching data
separated

Consideration: Minimalism



Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new BigDecimal( 100 ) );

    BigDecimal wka = wk.divide( oneP, 2, DOWN );

    return ! wka.compareTo( new BigDecimal( 50 ) ) > 1;
}
```

Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new BigDecimal( 100 ) );

    BigDecimal wka = wk.divide( oneP, 2, DOWN );

    return ! wka.compareTo( new BigDecimal( 50 ) ) > 1;
}
```

simplicity isn't simplistic !

Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new

    BigDecimal wka = wk.divide( oneP, 2, DOWN );

    return ! wka.compareTo( new BigDecimal( 50 ) ) > 1;
}
```

BigDecimal as Money

Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new BigDecimal( 100 ) );

    BigDecimal wka = wk.divide( on

    return ! wka.compareTo( new Big
}
```

BigDecimal as Percent

Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new BigDecimal( 100 ) );

    BigDecimal wka = wk.divide( oneP, 2, DOWN );

    return ! wka.compareTo( new BigDecimal( 50 ) ) > 1;
}
```

What's the result 'type' of combining 'Money' with 'Percent'

Simplicity

```
public boolean isGoodDeal( BigDecimal ik, BigDecimal wk ){

    BigDecimal gk = ik.add( wk );

    BigDecimal oneP = gk.divide( new BigDecimal( 100 ) );

    BigDecimal wka = wk.divide( oneP, 2, DOWN );

    return ! wka.compareTo( new BigDecimal( 50 ) ) > 1;
}
```

Which combinations are allowed -
which not ?

LiM Principle

give reader a clear distinction between concepts / types

restrict the way you can combine types

offer all necessary features at the types interface
-
but NOT more

LiM Principle

```
public boolean isGoodDeal( Money ik, Money wk ){

    Money gk = ik.add( wk );       // e.g. no 'division' allowed on Money

    return gk.proportionOf( wk ).isGreaterThan( Percent.FIFTY )
}
```

LiM Principle

```
public Bill createBill( Collection<Order> orders ){
        ...
        Money overallPrize = sum( order.getPrize() ).forAll( orders );

        rabattService.calcRabatt( overallPrize )
        ...
}
```

LiM Principle

```
public Bill createBill( Collection<Order> orders ){
        ...
        Money overallPrize = sum( order.      tPrize() ).forAll( orders );

        rabattService.calcRabatt( overallP
        ...
}
```

The idea of Billing
is coupled to Orders

LiM Principle

```
public Bill createBill( Collection<Order> orders ){
        ...
        Money overallPrize = sum( order.      tPrize() ).forAll( orders );

        rabattService.calcRabatt( overallP
        ...
}
```

... serving a specific client

LiM Principle

```
public Bill createBill( Collection<Order> orders ){
        ...
        Money overallPrize = sum( order.    tPrize() ).forAll( orders );

        rabattService.calcRabatt( overallP
        ...
}
```

... prevents Generalization

LiM Principle

```
public Bill createBill( Collection<Order> orders ){
        ...
        Money overallPrize = sum( order.      tPrize() ).forAll( orders );

        rabattService.calcRabatt( overallP
        ...
}
```

decouple it

keep dependencies to 'outer world' to a minimum

Dependency Inversion Principle

Abstractions should not depend upon Details – Details should depend upon Abstractions

Dependency Inversion Principle

```
public interface Prizable{
    public Money getPrize();
}

...


public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}
```

Dependency Inversion Principle

```
public interface Prizable{
    public Money getPrize();
}

...
```

Belongs to the 'server'

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}
```

Dependency Inversion Principle

```
public interface Prizable{
    public Money getPrize();
}

...
```

Re-Usable by any client, which may provide prizable items

(but i don't care any longer about which ones)

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}
```

Dependency Inversion Principle

Remove unnecessary Dependencies from a core concept

Shield a concepts interface from outside influences

Provide loose coupling between loosely coupled concepts

LiM Principle

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}

public interface RabattService{
    public Money calcRabatt( Money prize );
    public Money calcRabatt( Customer customer );
    public Money calcRabatt( Date date );
    ...
}
```

LiM Principle

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}

public interface RabattService{
    public Money calcRabatt( Money prize );
    public Money calcRabatt( Customer customer );
    public Money calcRabatt( Date date );
    ...
}
```

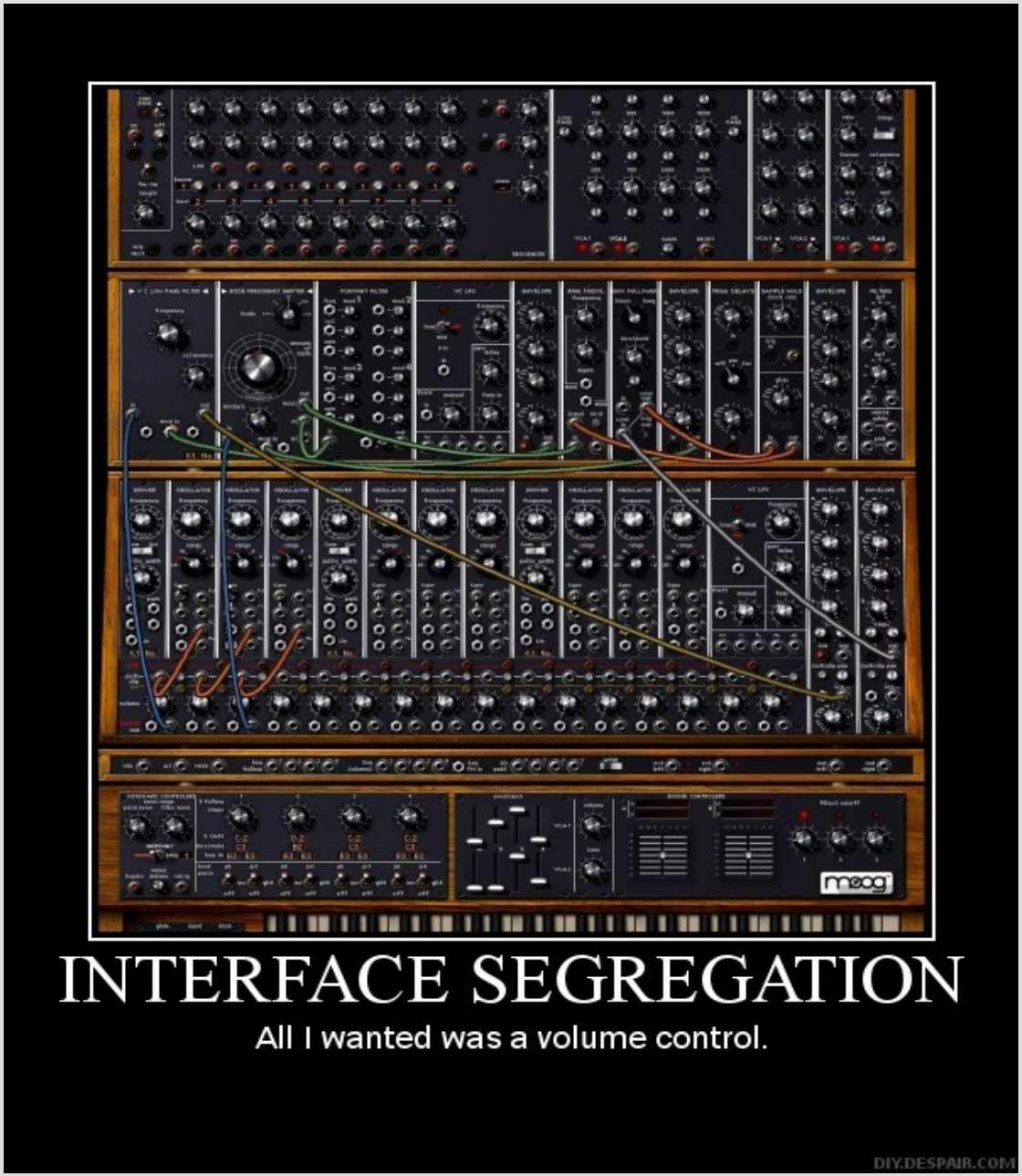
Depending on an 'external' idea of fee calculation

LiM Principle

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        rabattService.calcRabatt( overallPrize )
        ...
}

public interface RabattService{
    public Money calcRabatt( Money prize );
    public Money calcRabatt( Customer customer );
    public Money calcRabatt( Date date );
    ...
}
```

Depending on some more (unneeded) ideas of fee calculation

Interface Segregation Principle



Interface Segregation Principle

Interface Segregation Principle

```
public interface PrizeDiscounter{
    public Money calcRabatt( Money prize );
}



public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        prizeDiscounter.calcRabatt( overallPrize )
        ...
}
```

Interface Segregation Principle

```
public interface PrizeDiscounter{
    public Money calcRabatt( Money prize );
}
```

Belongs to the 'client'

```
public Bill createBill( Collection<Prizable> prizables ){
        ...
        Money overallPrize =
                        sum( prizables.getPrize() ).forAll( prizables );

        prizeDiscounter.calcRabatt( overallPrize )
        ...
}
```

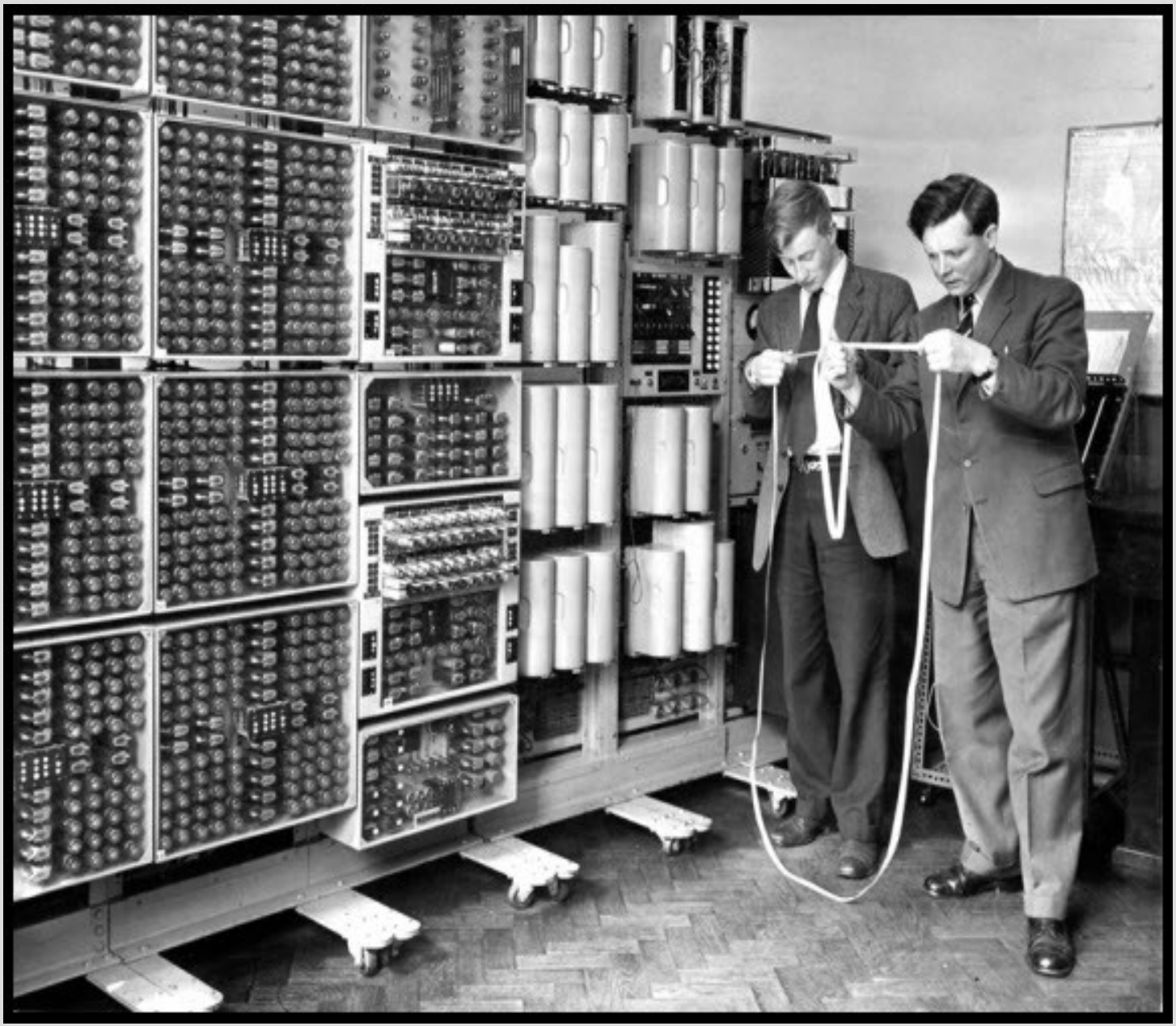
Interface Segregation Principle

```
public interface PrizeDiscounter{
    public Money calcRabatt( Money prize );
}

public Bill createBill( Collectio
        ...
        Money overallPrize =
                            sum( prizab                    ).forAll( prizables );

        prizeDiscounter.calcRabatt( overallPrize )
        ...
}
```

May be offered by any server, which provide fee calculation based on Prizes

(but i don't care any longer about which ones)

Program comprehension



Mental Model

3

Mental Model

3 5

Mental Model

3 5 7

Mental Model

3 5 7

Mental Model

3 5 7 13

Mental Model

3 5 7 13 11

Mental Model

3 5 7 13 11 9

Mental Model

3 5 7 13 11 9

Involved concepts :

numbers

prime <-> odd numbers

math. series

Mental Model

3 5 7 13 11 9

keeping series

removing prime

Mental Model

3 5 7 13 11 9

removing series

restoring prime

Mental Model

3 5 7 13 11 9

removing prime

refocus odd numbers

Mental Model

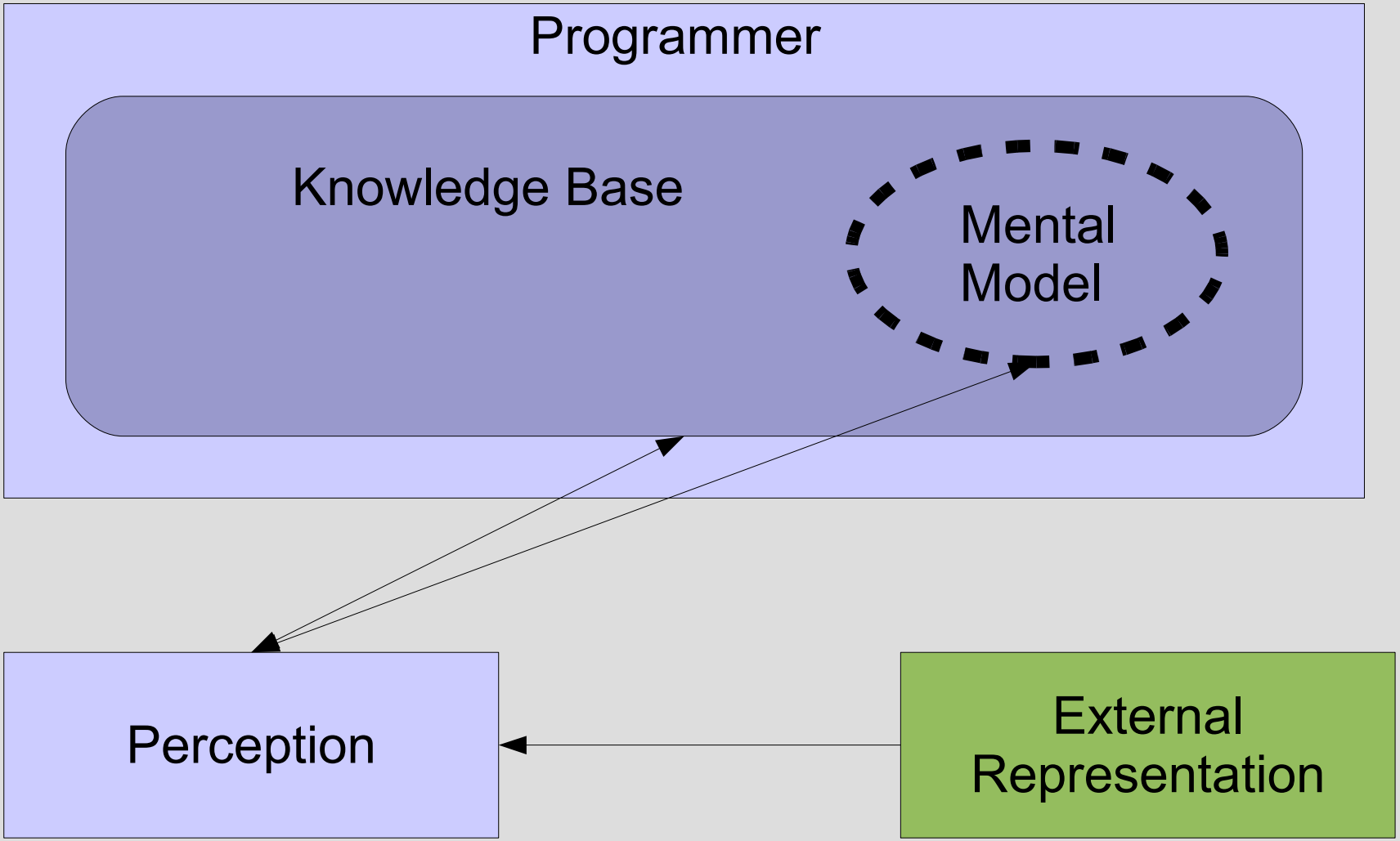
Constructed Representation of the perceived 'world'
(under a certain context)

(concurrent) simulation models of reality, used to reason,
to anticipate events and to underlie explanation

"Integration of new 'information' into an existing context in order to understand and judge that information"

(Michael O'Brien)

Mental Model



Program Model

```
Program Check_Group
  use crystallographic_symmetry, only: Space_Group_Type, set_spacegroup
  use reflections_utilities, only: Hkl_Absent
  use Symmetry_Tables, only: spgr_info, Set_Spgr_Info

  ....... ! Read reflections, apply criterion of "goodness" for checking,
          ! set indices i1,i2 for search in space group tables ...
  ....... ! omitted for simplicity
  call Set_Spgr_Info()
  m=0
  do_group: do i=i1,i2
    hms=adjustl(spgr_info(i)%HM)
    hall=spgr_info(i)%hall
    if( hms(1:1) /= "P" .and. .not. check_cent ) cycle do_group ! Skip centred groups
    call set_spacegroup(hall,Spacegroup,Force_Hall="y")
    do j=1,nhkl
       if(good(j) == 0) cycle  !Skip reflections that are not good (overlap) for checking
       absent=Hkl_Absent(hkl(:,j), Spacegroup)
       if(absent .and. intensity(j) > threshold) cycle do_group  !Group not allowed
    end do
    ! Passing here means that all reflections are allowed in the group -> Possible group!
    m=m+1
    num_group(m)=i
  end do  do_group
  write(unit=*,fmt=*) " => LIST OF POSSIBLE SPACE GROUPS, a total of ",m," groups are possible"
  write(unit=*,fmt=*) "     ---------------------------------------------------"
  write(unit=*,fmt=*) "     Number(IT)      Hermann-Mauguin Symbol     Hall Symbol"
  write(unit=*,fmt=*) "     ---------------------------------------------------"
  do i=1,m
    j=num_group(i)
    hms=adjustl(spgr_info(j)%HM)
    hall=spgr_info(j)%hall
    numg=spgr_info(j)%N
    write(unit=*,fmt="(i10,4a)")  numg,"                    ",hms,"          ",hall
  end do
  ...............
```

Program model

code representation

what is said in the source code and how it is said

text structure knowledge

elementary operations

single lines of code up to methods

control flow between those operations

Situation Model

Situation Model

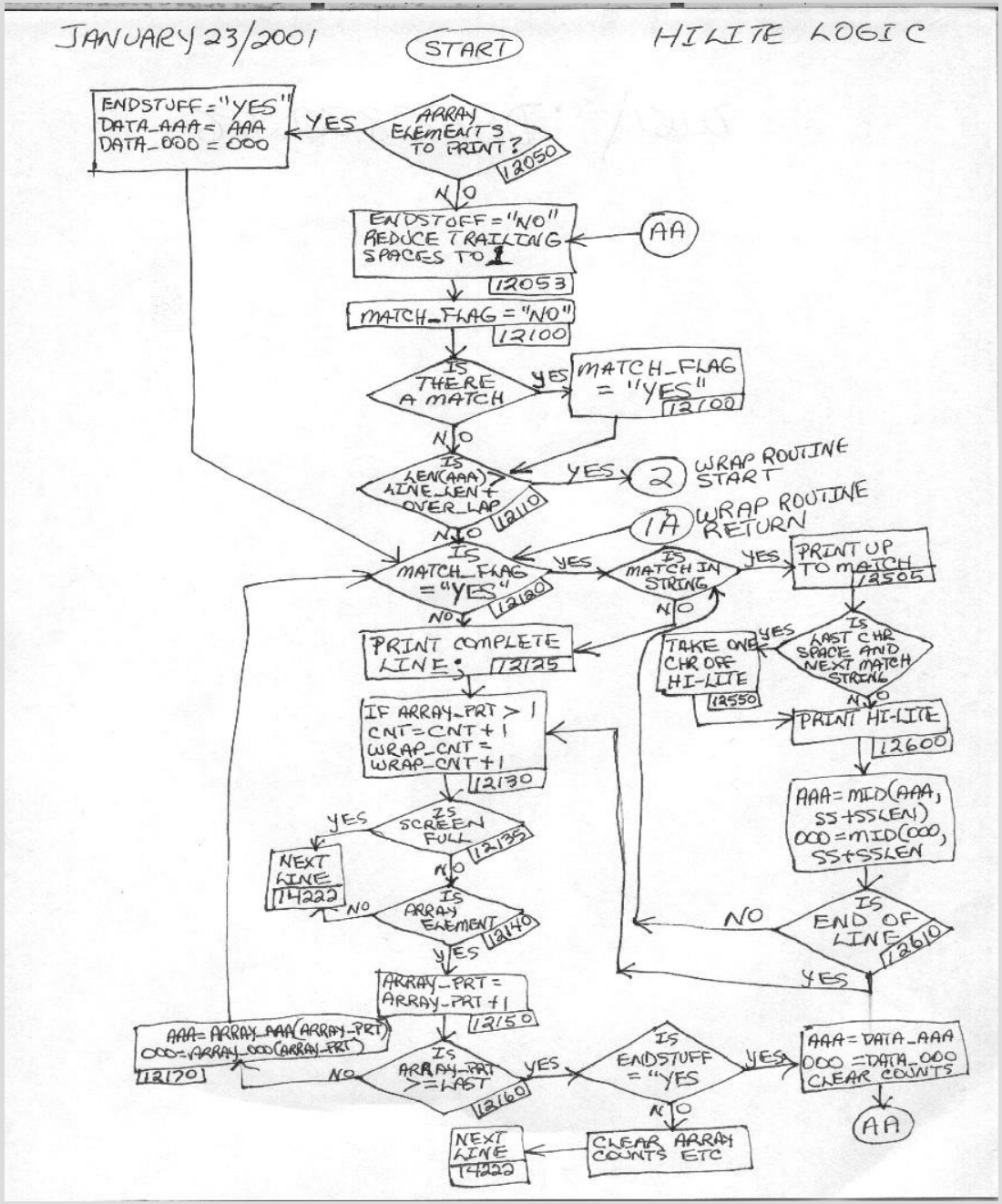
represents the (world) situation referred to by the code

contains problem knowledge and plan knowledge

reflects elements of the referred domain and their relationships

makes extensive use of the reader's existing domain knowledge

Plans


JANUARY 23/2001
START
HILITE LOGIC
ENDSTUFF = "YES"
DATA_AAA = AAA
DATA_000 = 000
YES
ARRAY ELEMENTS TO PRINT?
12050
NO
ENDSTOFF = "NO"
REDUCE TRAILING SPACES TO 1
AA
12053
MATCH_FLAG = "NO"
12100
IS THERE A MATCH
YES
MATCH_FLAG = "YES"
12700
NO
IS LEN(AAA) > LINE_LEN + OVER_LAP
12110
YES
2
WRAP ROUTINE START
NO
1A
WRAP ROUTINE RETURN
IS MATCH_FLAG = "YES"
12120
YES
IS MATCH IN STRING
YES
PRINT UP TO MATCH
12505
NO
NO
PRINT COMPLETE LINE
12125
TAKE ONE CHR OFF HI-LITE
12550
YES
IS LAST CHR SPACE AND NEXT MATCH STRING
NO
PRINT HI-LITE
12600
IF ARRAY_PRT > 1
CNT = CNT + 1
WRAP_CNT = WRAP_CNT + 1
12130
AAA = MID(AAA, SS+SSLEN)
000 = MID(000, SS+SSLEN
IS SCREEN FULL
12135
YES
NEXT LINE
14222
NO
IS ARRAY ELEMENT
12140
NO
IS END OF LINE
12610
YES
YES
ARRAY_PRT = ARRAY_PRT + 1
12150
AAA = ARRAY_AAA(ARRAY_PRT)
000 = ARRAY_000(ARRAY_PRT)
12170
IS ARRAY_PRT >= LAST
12160
NO
YES
IS ENDSTUFF = "YES"
YES
AAA = DATA_AAA
000 = DATA_000
CLEAR COUNTS
AA
NO
NEXT LINE
14222
CLEAR ARRAY COUNTS ETC

Plans

structuring programming actions to achieve a desired goal

knowledge of main program goals corresponds to
knowledge about the goals of plans

main goals are not explicitly represented by the programm text

Plans

structuring programming actions to achieve a desired goal

knowledge of main program goals corresponds to
knowledge about the goals of plans

main goals are not explicitly represented by the programm text

to be inferred / recognized within a program

Program Comprehension

Mapping the program Model to the situation Model

Bridging the semantic gap between both Models

Identifying Plans and mapping them to the program goals

Bottom-up Comprehension

```
public class GP {
	static int[] gen( int max ){

		int j;

		if( max >= 2 ){

			int s = max + 1;
			boolean[] f = new boolean[s];
			int i;

			for( i=0; i<s; i++){
				f[i] = true;
			}

			for( i=2; i < Math.sqrt(s) + 1; i++){
				if( f[i] ){
					for( j=2*i; j<s; j+=i ){
						f[j] = false;
					}
				}
			}

			int cnt = 0;
			f[0] = f[1] = false;

			for( i=0; i<s; i++ ){
				if( f[i] ) cnt++;
			}

			int[] ret = new int[cnt];

			for(i=0, j=0; i<s; i++ ){
				if( f[i] ) ret[j++] = i;
			}

			return ret;
		}
		else{
			return new int[0];
		}
	}

}
```

Bottom-up Comprehension

Starts by reading elementary operations (building a program model)

grouping them into larger text structure units

"Program goals and data flow knowledge, which make up the situation model, emerged with continued processing of the code"

"Reconstructs a high level of abstraction that can be derived through 'reverse engineering' of source code"

Top-down Comprehension

```
private boolean[] crossedOut;
private int[] result;

public int[] generatePrimesUpTo( int maxValue ){

    if( maxValue < 2 ) return EMPTY_PRIME_ARRAY;

    uncrossIntegersUpTo( maxValue );

    crossOutMultiples();

    putUncrossedIntegersIntoResult();

    return result;
}
```

Top-down Comprehension

```
...

public void crossOutMultiples(){

    int limit = determineIterationLimit();

    for( int i = 2; i <= limit; i++ ){

        if( notCrossed( i ){

            crossOutMultiplesOf( i );
        }
    }
}
```

Top-down Comprehension

starting with an pre-existing domain modell (situation Model)

building hypothesis

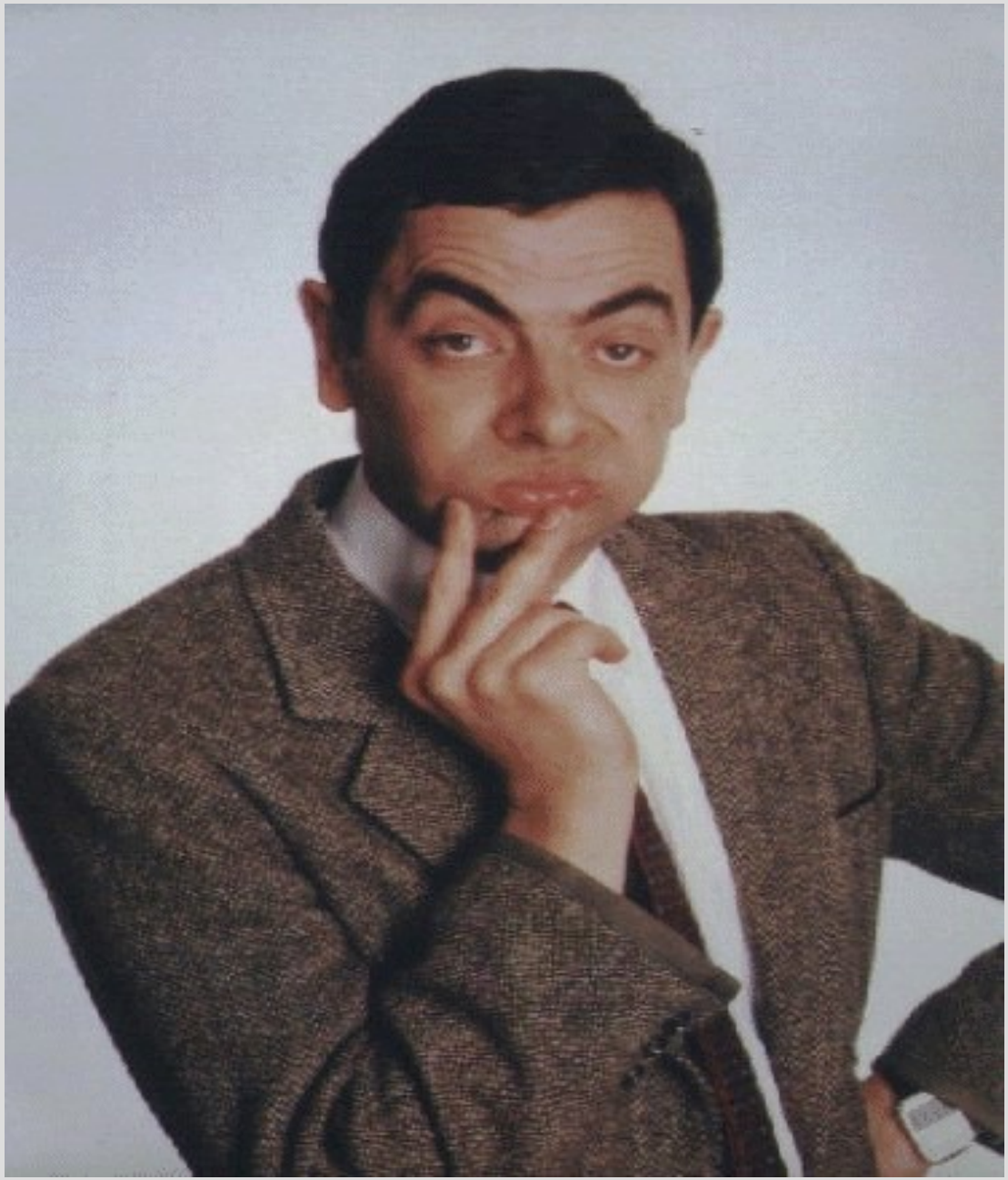
how is the domain modell (plans / goals) represented in the code

testing hypothesis

regognize expected plans & participating concepts in the code

Asimillatiion & Adaption

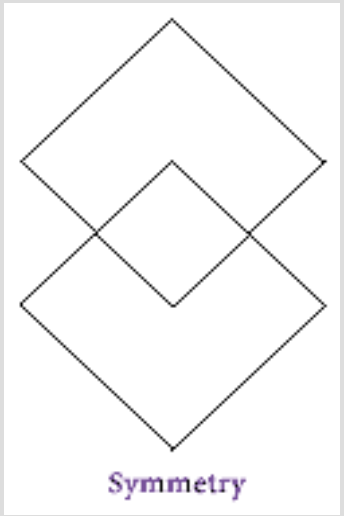
Considerations

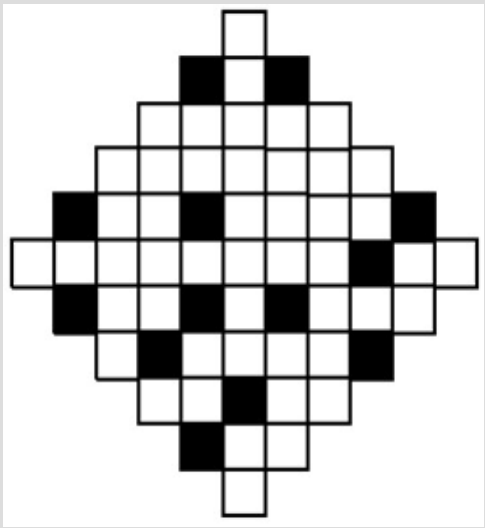


Consideration: Symmetry

support Plan 'completion'

eliminate 'disharmony' in Plan tracing





Consideration: Symmetry

```
...
void writeToFile( Output outout ){
    openFile();
    writeToFile( output );
}
```

Consideration: Symmetry

```
...
void writeToFile( Output outout ){
    openFile();
    writeToFile( output );
}
```

Violating another kind of 'Open – Closed' Principle'

Keep the Balance

Keep the Balance

```
...
void writeToFile( Output outout ){
    openFile();
    writeToFile( output );
    closeFile();
}
```

One Level of Abstraction

One Level of Abstraction

```
void process{
    input();
    count++;
    output();
}
```

One Level of Abstraction

```
void process{
    input();
    count++;
    output();
}
```

Implementation Detail

One Level of Abstraction

```
void process{
    input();
    incrementCount();
    output();
}
```

One Level of Abstraction

```
void process{
    input();
    incrementCount();
    output();
}
```

Intention Driven

Implementation Driven

One Level of Abstraction

```
void process{
    input();
    tally();
    output();
}
```

One Level of Abstraction

Newspaper Metaphor

Make it easy to recognize and follow a plan

Separate the high level concept from the details

Give the chance to follow a plan on different levels of abstraction

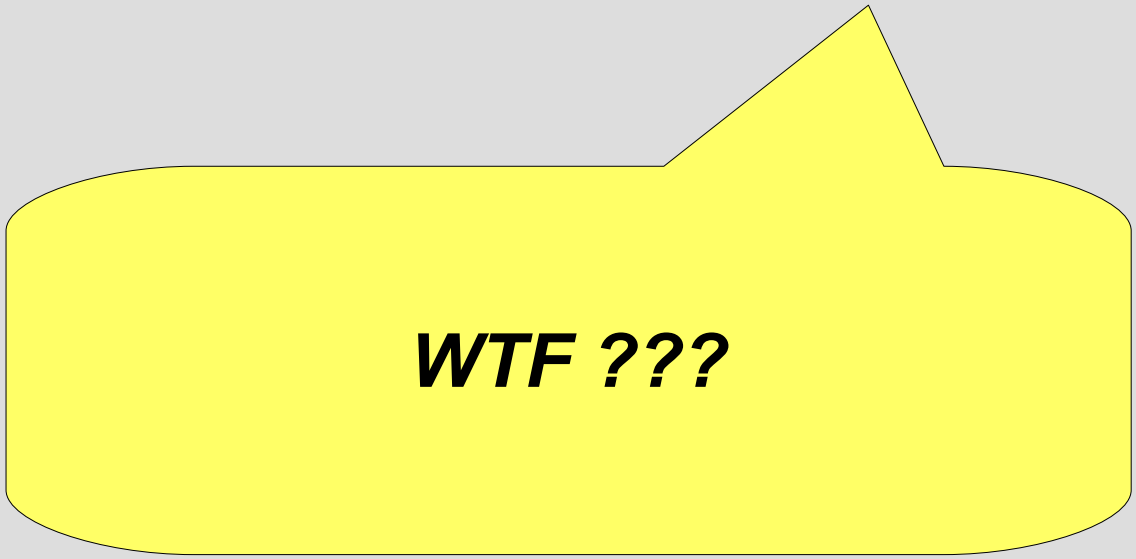
Consideration: Expressiveness

Consideration: Expressiveness

```
...
rechnung.setBetrag( rechnung.getBetrag() * 1.02f );
...
```

Consideration: Expressiveness

```
...
rechnung.setBetrag( rechnung.getBetrag() * 1.02f );
...
```



Consideration: Expressiveness

```
...
addSkontoTo( rechnung, Prozent.ZWEI );
...


public void addSkontoTo( Rechnung rechnung, Prozent prozent ){
    ...
}
```

Consideration: Expressiveness

```
...
rechnung.addSkonto( Prozent.ZWEI );
...
```

PIE Principle

Program intently and expressively

PIE Principle

"Explicit design helps understanding the nature of a rule as a distinct and important business rule - not just an obscure calculation"

code that produces the effect of a rule without explicitly stating the rule, forces us to think of step-by-step procedures

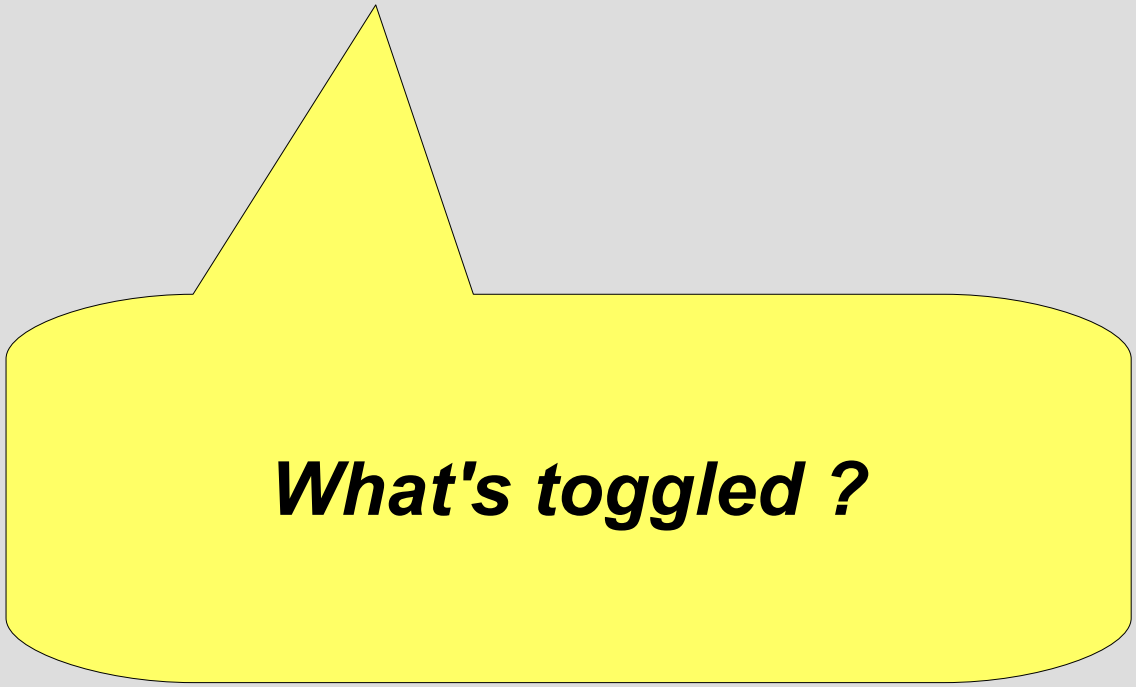
(Eric Evans – Domain Driven Design)

PIE Principle

```
class OrderController{

    public void toggleSelection(){
        ...
    }

}
```

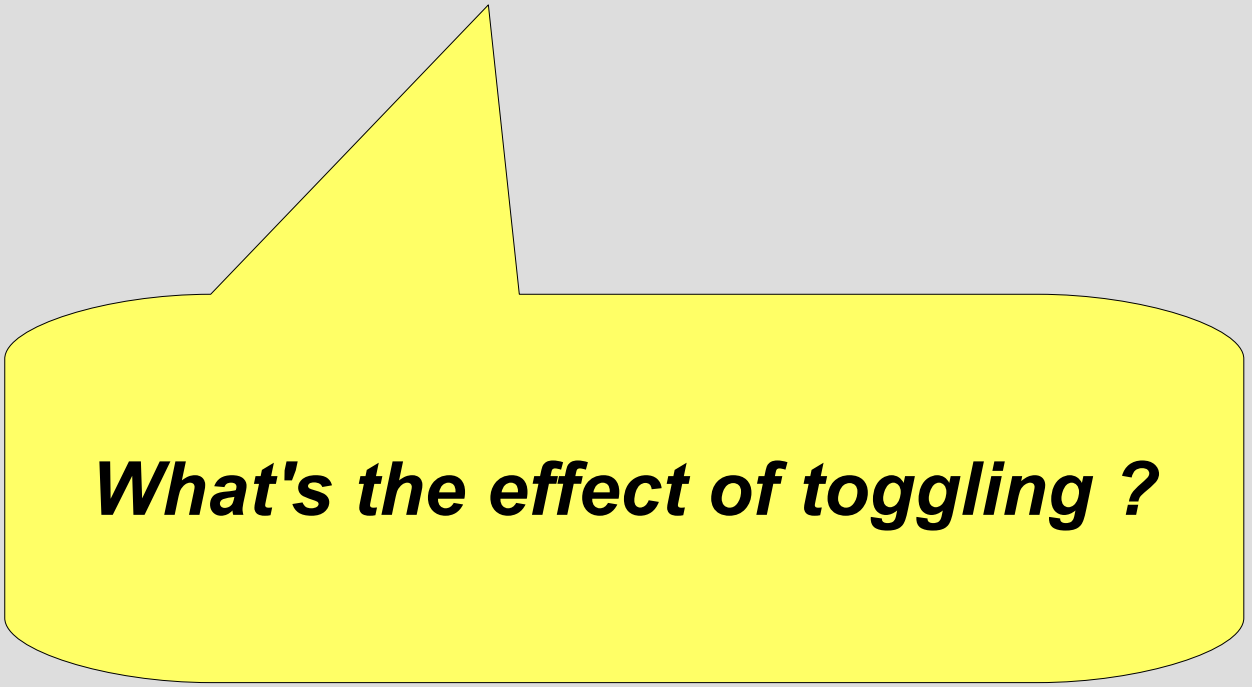
PIE Principle

```
class OrderController{

    public void toggleSelection(){
        ...
    }

}
```



PIE Principle

```
class OrderController{

    public void toggleSelection(){
        ...
    }

}
```



PIE Principle

Intention Revealing Interfaces

if a developer must consider the implementation of a component in order to use it, than the value of encapsulation is lost

(Eric Evans – Domain Driven Design)

Intention Revealing Interface

```
class OrderController{

    public void showOrderDetails(){
        ...
    }

    puplic void cancelOrderDetails(){
        ...
    }

}
```

Consideration: Reduce Variety

Consideration: Reduce Variety

```
class OrderService{
    private int accessCounter = 1;

    public Order getOrder( int orderId ){
        accessCounter++
        ...
    }

    public void order( Order order ){
        accessCounter++;
        if( counter > MAX_ORDERS_PER_DAY ) throw ...

        if( isAfterDailyExport() ) throw ...
        ...
    }
}
```

Consideration: Reduce Variety

```
class OrderService{
    private int accessCounter = 1;

    public Order getOrder( int orderId ){
        accessCounter++
        ...
    }

    public void order( Order order ){
        accessCounter++;
        if( counter > MAX_ORDERS_PER_DAY ) throw ...

        if( isAfterDailyExport() ) throw ...
        ...
    }
}
```

Huh ?
Sometimes it works ...

Consideration: Reduce Variety

avoid side-effects

if a developer using a high level command must understand the consequences of each underlying command, then the value of encapsulation is lost

(Eric Evans – Domain Driven Design)

Consideration: Reduce Variety

```
class Betrag{

    public static final Betrag ZERO = new Betrag( 0 );

    private int value = 0;


    public Betrag( int val ){ this.value = val; }


    public Betrag add( Betrag other ){

        this.value = this.value + other.value;
    }

}
```

Consideration: Reduce Variety

```
public Betrag calcPrize( Collection<Auftrag> auftraege ){
    Betrag sum = Betrag.ZERO
    for( Auftrag auftrag : aufgtraege )
        sum.add( auftrag.effectiveBetrag )
    ...
}

public Betrag calcRabatt( ... ){

    Betrag rabatt = Betrag.ZERO;
    ...
    rabatt.add( ... );
    ...
}
```

Consideration: Reduce Variety

```
public Betrag calcPrize( Collection<Auftrag> auftraege ){
    Betrag sum = Betrag.ZERO
    for( Auftrag auftrag : aufgtraege )
        sum.add( auftrag.effective
    ...
}

public Betrag calcRabatt( ... ){

    Betrag rabatt = Betrag.ZERO;
    ...
    rabatt.add( ... );
    ...
}
```

Wow !
What a big Fee

:o)

Consideration: Reduce Variety

```
public Betrag calcPrize( Collection<Auftrag> auftraege ){
    Betrag sum = Betrag.ZERO
    for( Auftrag auftrag : aufgtraege )
        sum.add( auftrag.effectiveBetra
    ...
}

public Betrag calcRabatt( ... ){
    Betrag rabatt = Betrag.ZERO;
    ...
    rabatt.add( ... );
    ...
}
```

ooh !
Is it that expensive ?

:o(

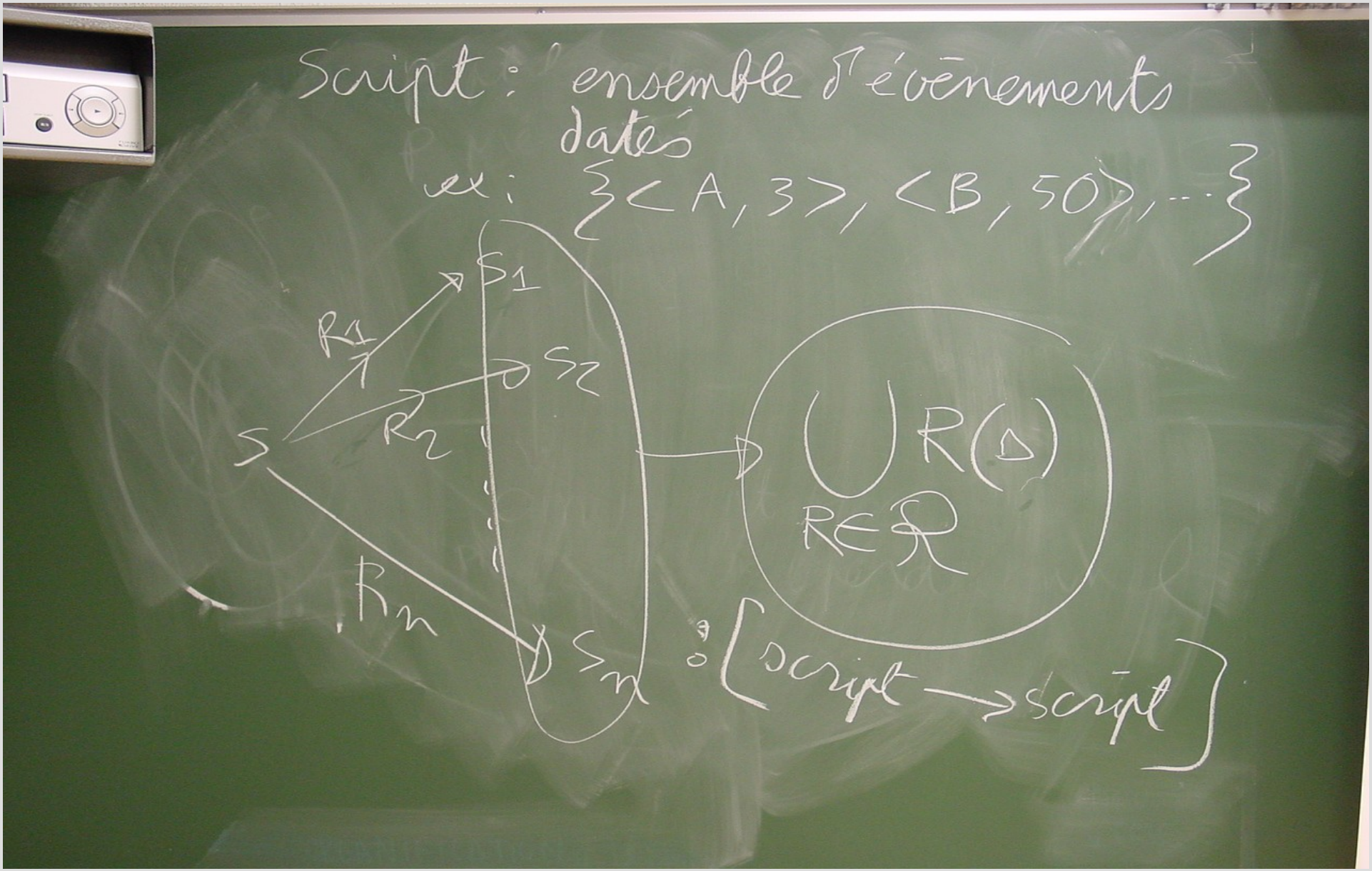
Consideration: Reduce Variety

Prefer Immutability

Consideration: Reduce Variety

```
class Betrag{

    public static final Betrag ZERO = new Betrag( 0 );

    private int value = 0;


    public Betrag( int val ){ this.value = val; }


    public Betrag add( Betrag other ){

        return new Betrag( this.value + other.value );
    }

}
```

Consideration: Declarative vs Imperative



Consideration: Declarative vs Imperative

```
public void validate( Collection transaktionen ){
    ...
    BigDecimal sum = new BigDecimal( "0" );
    boolean future = false;

    Iterator trans = transaktionen.iterator();
    while( trans.hasNext() ){

        WriteTransaktionDTO dto = (WriteTransaktionDTO) trans.next();

        if(   dto.getAuftrtyp() == 10 || dto.getAuftrtyp() == 11 || dto.getAuftrtyp() == 15 ||
              dto.getAuftrtyp() == 30 || dto.getAuftrtyp() == 31 || dto.getAuftrtyp() == 32 ||
              dto.getAuftrtyp() == 39 ){

              BigDecimal betr = dto.getTrabetrag();

              if( betr == null ){

                    betr =
                          fpreis( dto.getAuftrtyp() < 30 ? dto.getFonwknext() : dto.getZfonwknext() )
                          .multiply( dto.getTrastueck() );
              }

              sum = sum.add( betr );

              if( dto.getTraausterm().compareTo( new Date() ) > 1 ) future = true;
        }
    }
    if( new BigDecimal( "100000" ).compareTo( sum ) < 0 && future ) throw ...
}
```

Consideration: Declarative vs Imperative

Separate the 'WHAT' from the 'HOW'

Declarative Specification provides a concise way of expressing certain kinds of 'rules', extracting them from conditional logic and making them explicit in the model.

Otherwise you can get lost in the sheer mass of rule evaluation code

(Eric Evans – Domain Driven Design)

Consideration: Declarative vs Imperative

```
public void validate( Auftraege auftraege ){
      ...

if(
      auftraege.filter( isKauf().or( isTausch() ) ).effectiveBetrag().isGreaterThan( MAXIMAL_KAUF_BETRAG )
      &&
      auftraege.exist( isTerminAuftrag() ){

      throw ...
}
```

Considerations

... and some more to come ...

Considerations

... but that is another story ...

The end

Thanks !!!

Reference

Robert C. Martin	Clean Code: A Handbook of Agile Software Craftsmanship Prentice Hall
Robert C. Martin	Agile Software Development, Principles, Patterns, and Practices Prentice Hall
Kent Beck	Implementation Patterns Addison-Wesley
Kevlin Henney	Five Considerations for Software Developers http://wiki.parleys.com
Eric Evans	Domain-Driven Design: Tackling Complexity in the Heart of Software Addison-Wesley
Andrew Hunt Dave Thomas	The Pragmatic Programmer: From Journeyman to Master Addison-Wesley
Andy Hunt	Pragmatic Thinking and Learning: Refactor Your Wetware Pragmatic Bookshelf
Venkat Subramaniam	Practices of an Agile Developer: Working in the Real World Pragmatic Bookshelf
Ben Liblit[1], Andrew Begel[2], Eve Sweetser[3]	Cognitive Perspectives on the Role of Naming in Computer Programs http://pages.cs.wisc.edu/~liblit/ppig-2006/ppig-2006.pdf
Vaclav Rajlich, James Doran, Reddi.T.S.Gudla	Layered Explanations of Software: A Methodology for Program Comprehension http://www.cs.wayne.edu/~vip/publications/Rajlich.IWPC.1994.LayeredExplanation.pdf
Jean-Marie Burkhardt, Françoise Détienne, Susan Wiedenbeck	Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase http://arxiv.org/ftp/cs/papers/0612/0612004.pdf

Reference

Jean-Marie Burkhardt, Françoise Détienne, Susan Wiedenbeck	Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension http://arxiv1.library.cornell.edu/ftp/cs/papers/0612/0612018.pdf
C. Kothari	Overview of Program Comprehension http://class.ece.iastate.edu/kothari/cpre556/Lectures/W6/Lecture%2012-%20Program%20Comprehension.pdf
Michael P. O'Brien	Software Comprehension – A Review & Research Direction http://www.st.cs.uni-saarland.de/edu/empirical-se/2006/PDFs/brien03.pdf
Juergen Rilling Tuomas Klemola	Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics http://www-sst.informatik.tu-cottbus.de/~db/doc/People/IEEE/papers/18830115.pdf
Václav Rajlich	Program Comprehension as a Learning Process http://www.cs.wayne.edu/~severe/publications/Rajlich.ICCI.2002.Program.Comprehension.Learning.Process.pdf
Tim Tiemens	Cognitive Models of Program Comprehension
Robert L. Glass	The Cognitive View: A Different Look at Software Design http://www.developerdotstar.com/mag/articles/glass_cognitive_view.html
Jacob Feldman	The Simplicity Principle in Human Concept Learning http://ruccs.rutgers.edu/~jacob/Papers/feldman_CDPS.pdf