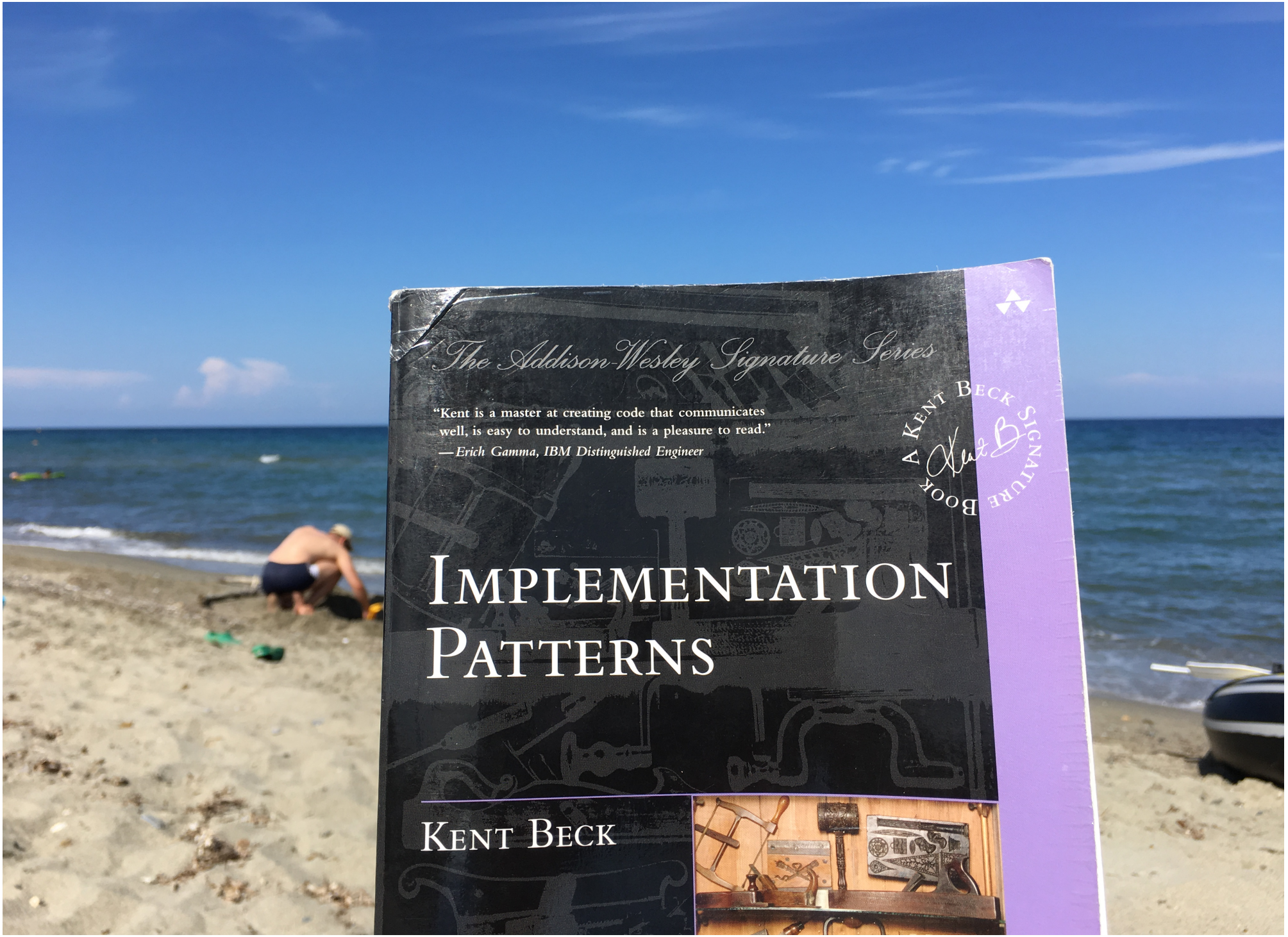
# IMPLEMENTATION PATTERNS

**Dr. Frank Gerhardt**

03.07.2019


The Addison-Wesley Signature Series
"Kent is a master at creating code that communicates well, is easy to understand, and is a pleasure to read."
—Erich Gamma, IBM Distinguished Engineer
A Kent Beck Signature Book
Implementation Patterns
Kent Beck

# DR. FRANK GERHARDT

- Smalltalker, LISP-Lover, Emacs
  - auch Java, Eclipse, neuerdings JS, Python
- Gründer und 1. Erster Vorsitzender der JUGS
  - erster JFS-Organisator
- Software Experts Network Stuttgart e.V.
  - **nachher am Stand**
- Gerhardt Informatik, 10 Mitarbeiter
- we're not hiring, kein Gewinnspiel ;-)
- freie Kapazitäten ab September

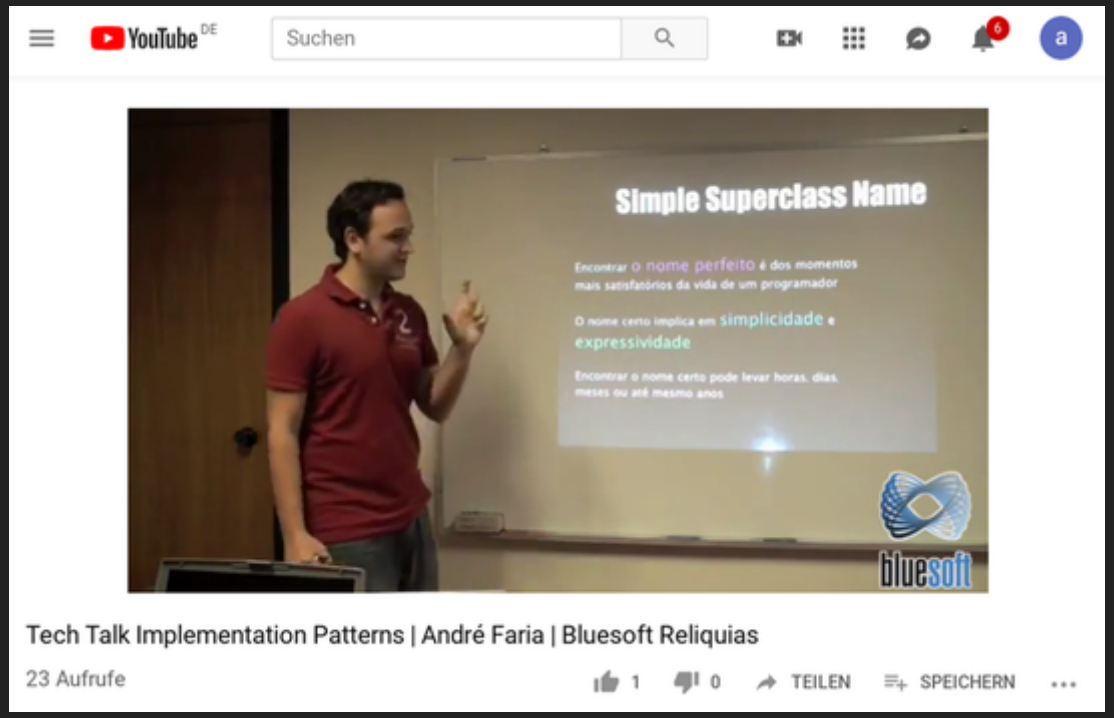
# ABSTRACT

Implementation Patterns ist ein leider in Vergessenheit geratenes Buch von Kent Beck aus dem Jahr 2007. Diese Patterns, auch Idiome genannt, sind unterhalb von Design Patterns aber oberhalb von Code-Formatierungsregeln angesiedelt. Bei den Implementation Patterns geht es darum wie man einzelne Klassen und Methoden gestaltet. In Code Reviews bin ich immer wieder auf Fälle gestoßen, bei denen schlechter Code auf Unkenntnis der Implemenation Patterns zurück geführt werden kann. Anfänger und Fortgeschrittene können die Qualität ihres Codes wesentlich verbessern, wenn sie die Implementation Pattern kennen und anwenden.

# WHY?

- I have two copies of "Implementation Patterns"
  - people don't read books
- I read a lot more code nowadays
  - reviewing is coaching
  - I explain these patterns again and again
- I need this presentation for my team
  - there was no talk on YouTube

# SPANISH TALK




https://www.youtube.com/watch?v=y82xz547zs8

# KENT BECK

- SUnit -> JUnit -> xUnit
- Ward and Kent, c2.com Wiki
- talk at JUGS 1998
  - going out to Cannstatter Wasen

G

er

java.net MEMBER

JUGS

java user group stuff

ent

en und Vor-

# Vergangene JUGS-Veranstaltungen 1

JUGS Talklet

10.12.1998

Ansgar Schurek / IBM Deutschland

Christian Wege / Uni Tübingen

JUGS Talklet

12.11.1998

Christian Jänsch / Cortex Brainware

Thomas Prause / Object International

JUGS Talklet

02.10.1998

Kent Beck (Sonderveranstaltung mit Gemstone)

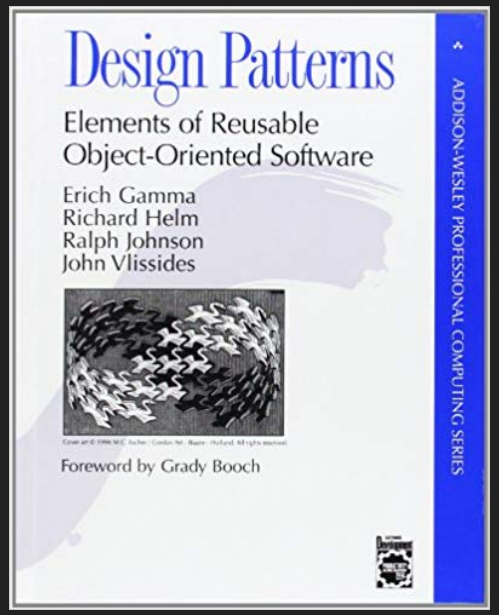
**Java Testing Framework / JUnit**

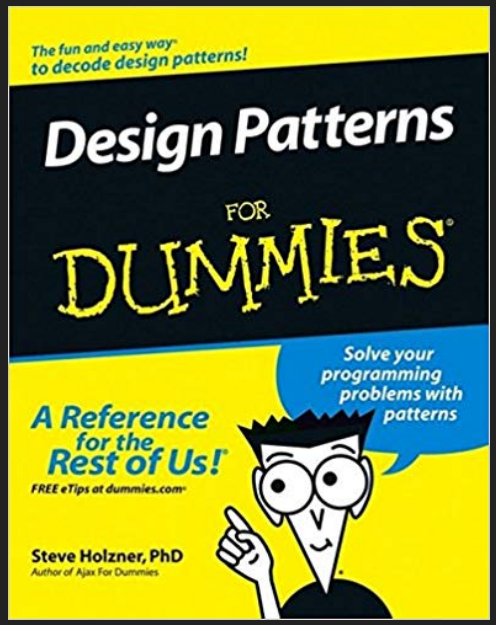
Frank Gerhardt: **JUGS Wiki Wiki Web**

Kent Beck: **Extreme Programming**

# DESIGN PATTERNS

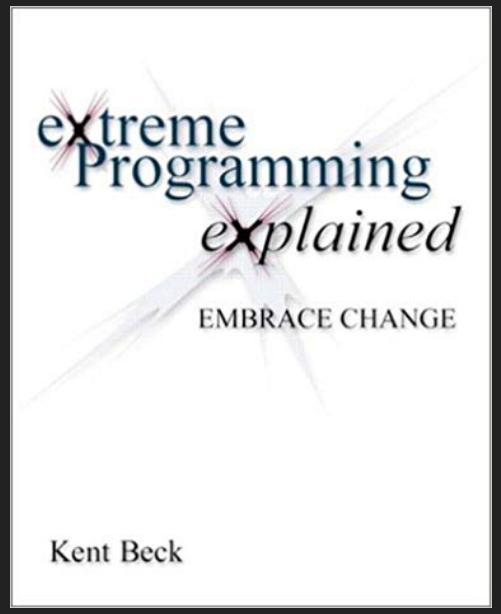
Gang of Four, 1994, using C++



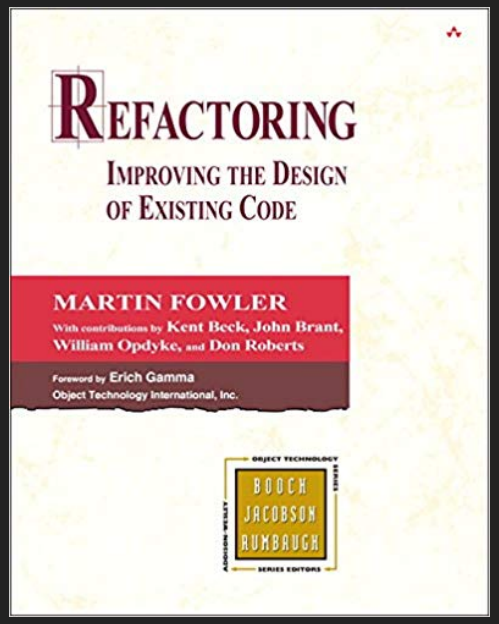
# XP

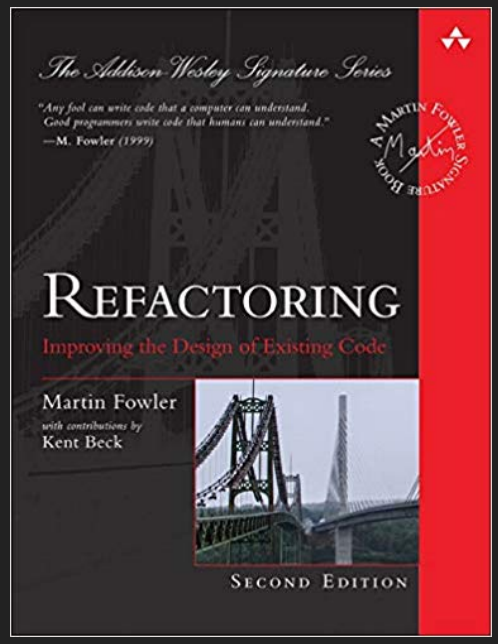
eXtreme Programming, 1999, 2nd ed. 2004



# REFACTORING

1999, 2nd ed. 2019



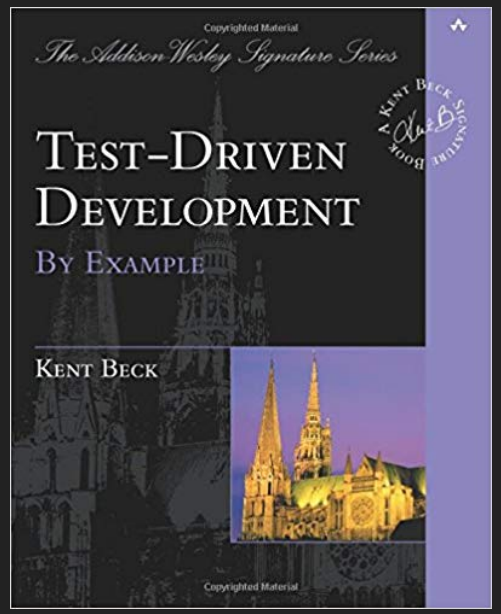



# AGILE MANIFESTO

- 2001
- **values** and **principles**
- Kent **B**eck is first signer
  - thanks to alphabetical order

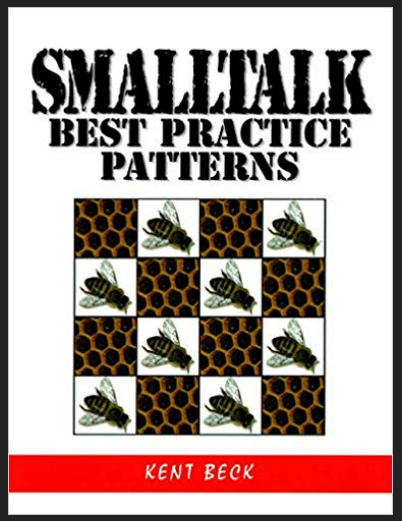
# TDD

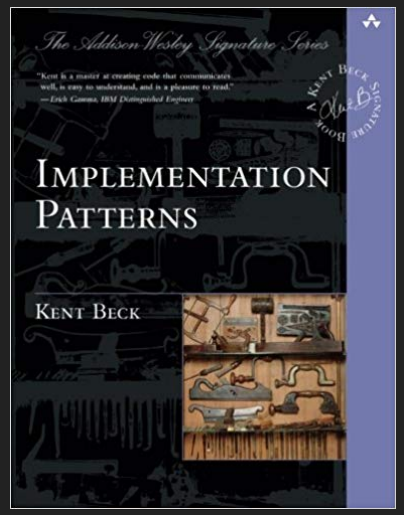
Test-Driven Development, 2002



# ABOUT THE BOOK

- first as Smalltalk book, 1996
- then Java, when JUnit 4 was new, 2008
- 77 patterns, 157 pages
  - in 45 minutes!? Nope.



# TOUR GUIDE

- How to read the Book?
- Kent writes
  - read just in time
  - use patterns every few seconds
- First, browse content to know what's in there
- Read ch. "A Theory of Programming" first, it's great

# MOTIVATION

# ASSUMPTION, PREMISE

*Good code matters.*

Most of the time, not always.

# GOAL

- don't program by instinct
  - explainable, why?
- importance of other people
  - the other's perspective
  - programming so that other people can understand your code
  - consciously for others as well as for yourself

# JEOPARDY

# JEOPARDY

- Answer: A word describing being thrown out of a window

# JEOPARDY

- Answer: A word describing being thrown out of a window
- Question: What is defenestration?

# JEOPARDY

- Answer: A word describing being thrown out of a window
- Question: What is defenestration?
- Coding is like Jeopardy

# JEOPARDY

- Answer: A word describing being thrown out of a window
- Question: What is defenestration?
- Coding is like Jeopardy
- Answer: your Java code

# JEOPARDY

- Answer: A word describing being thrown out of a window
- Question: What is defenestration?
- Coding is like Jeopardy
- Answer: your Java code
- Question: what was the question?

# JEOPARDY

- Answer: A word describing being thrown out of a window
- Question: What is defenestration?
- Coding is like Jeopardy
- Answer: your Java code
- Question: what was the question?
- E.g. a field declared as a Set means...

"THEORY"

# "LAWS"

- code is more often read than written
- there is no "done"
- basic set of control flow concepts (sequence, branch, loop, call)
- path of understanding
  - detail-to-concept
  - concept-to-detail
- the patterns are not general
  - adapt to personal style

# VALUES AND PRINCIPLES

- 3 values
- 6 principles

# VALUES

- communication
- simplicity
- flexibility

# 1. COMMUNICATION

- with other people
- literate programming
  - nice: Jupyter notebooks
- the other's perspective

# 2. SIMPLICITY

- remove excess complexity
- in the eye of the beholder
- have audience in mind
- waves of complexity and simplification

# 3. FLEXIBILITY

- keep options open
- misused? speculative?

# PRINCIPLES

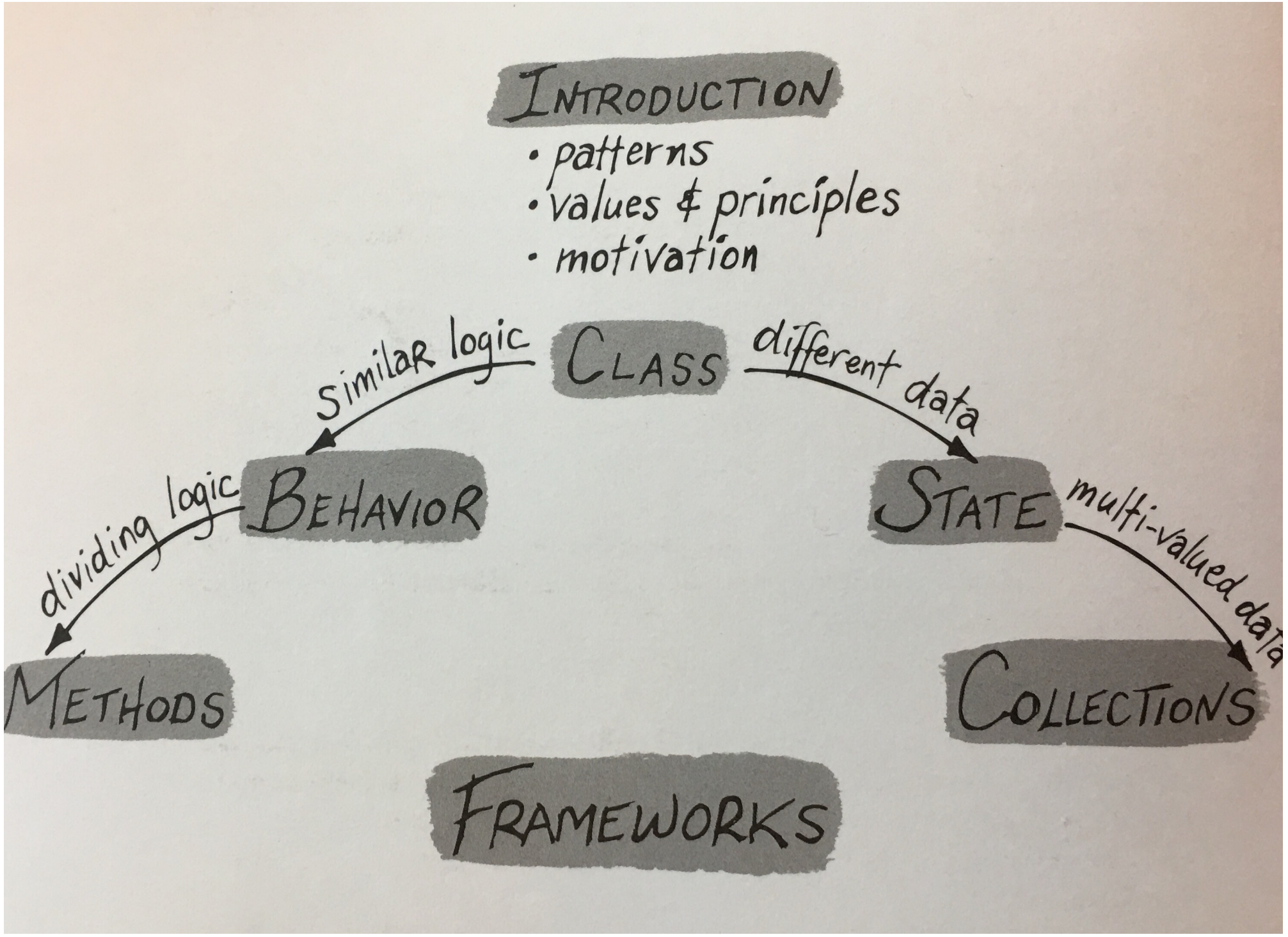
- local consequences
- minimize repetition
- logic and data together
- symmetry
- declarative expression
- rate of change (temporal symmetry)

# COST

- initial
- maintenance
  - understanding
  - changing
  - testing
  - deploying

# PATTERNS OVERVIEW

- Class (18): data and logic together
- State (19): different data
- Behavior (14): similar logic
- Methods (24): dividing logic
- Collections: multi-valued data
- Frameworks


INTRODUCTION
• patterns
• values & principles
• motivation
CLASS
similar logic
different data
BEHAVIOR
STATE
dividing logic
multi-valued data
METHODS
COLLECTIONS
FRAMEWORKS

# CLASS PATTERNS

**Class** Simple Superclass Name **Qualified Subclass Name** Abstract Interface **Interface** Abstract Class **Versioned Interface** Value Object **Specialization** Subclass **Implementor** Inner Class **Instance-Specific Behavior** Conditional **Delegation** Pluggable Selector **Anonymous Inner Class** Library Class

# STATE PATTERNS

State **Access** Direct Access **Indirect Access** Common State **Variable State** Extrinsic State **Variable** Local Variable **Field** Parameter **Collecting Parameter** Optional Parameter **Var Args** Parameter Object **Constant** Role-Suggesting Name **Declared Type** Initialization **Eager Initialization** Lazy Initialization

# BEHAVIOR PATTERNS

Control Flow **Main Flow** Message **Choosing Message** Double Dispatch **Decomposing (Sequencing) Message** Reversing Message **Inviting Message** Explaining Message **Exceptional Flow** Guard Clause **Exception** Checked Exceptions **Exception Propagation**

# METHODS PATTERNS

**Composed Method** Intention-Revealing Name **Method Visibility** Method Object **Overridden Method** Overloaded Method **Method Return Type** Method Comment **Helper Method** Debug Print Method **Conversion** Conversion Method **Conversion Constructor** Creation **Complete Constructor** Factory Method **Internal Factory** Collection Accessor Method **Boolean Setting Method** Query Method **Equality Method** Getting Method **Setting Method** Safe Copy

# COLLECTIONS PATTERNS

- Metaphors, Issues
- Interfaces
  - **Array** Iterable **Collection** List **Set** SortedSet **Map**
- Implementations
  - **Collection** List **Set** Map **Collections** Searching **Sorting** Unmodifiable Collections **Single-Element Collections** Empty Collections **Extending Collections**

# EVOLVING FRAMEWORKS

- Changing Frameworks without Changing Applications
- Incompatible Upgrades
- Encouraging Compatible Change
- Library Class
- Objects

# CLASS PATTERNS (18)

# CLASS

- data changes for often than logic
- expensive
- reduce number

# SIMPLE SUPERCLASS NAME

- the super class name is the most important name
- operation follow after the class name, not reverse
- tension long or short?
- pick strong metaphor
- a few single words
- use a thesaurus
- go for a walk

# QUALIFIED SUBCLASS NAME

- one word would be best
- express how is the subclass
  - similar
  - different
- multi-level hierarchies are generally delegation waiting to happen
- too short names burden the developers' short term memory

# ABSTRACT INTERFACE

- each interface has a cost
  - one more thing to learn, understand, document, debug, organize, browse, and name
- can cause inflexibility
- there are limits to the value of "future-proofing" software through speculation
- Putting all these factors together - the need for flexibility, the cost of flexibility, the unpredictability of where flexibility is needed - leads me to the belief that the time to introduce flexibility is when it is definitely needed.

# INTERFACE

- named like classes
- names already used up by classes
  - ISomething is fine, see Eclipse
  - SomeImpl is a bit ugly
- changes to interfaces are discouraged

# ABSTRACT CLASS

- a mix of interface and class

# VERSIONED INTERFACE

- OK if not too many
- see Eclipse
  - ISomething, ISomething2, ISomething3
- `instanceOf`

# VALUE OBJECT

- set all fields in constructor
- always return new objects
- whenever possible create microworlds of math

# SPECIALIZATION

- extremes
  - same logic, different data
  - different logic, same data

# SUBCLASS

- can play this card only once :-/
- static, can not change at runtime
  - delegation can
- methods should do one job
  - to not have to copy from super class
- avoid parallel hierarchies

# IMPLEMENTOR

- polymorphic messages
  - to different receivers
  - for parameters see *overloading*
- to express choice
- open a system to variation

# INNER CLASS

- when nobody outside needs to know

# INSTANCE-SPECIFIC BEHAVIOR

- static
- dynamic

# CONDITIONAL

- or maybe better
  - subclass
  - delegation

# DELEGATION

- pass delegator as a parameter
- store in a field
- or calculated

# PLUGGABLE SELECTOR

- call with reflection
- store name in field

# ANONYMOUS INNER CLASS

- alternative way to implement *instance-specific behavior*
- override one or more methods for specific behavior
- more static that delegation
- difficult to test

# LIBRARY CLASS

- start with static methods
- try instances
- maybe evolve to a real class

# STATE PATTERNS (19)

# STATE

- state is not only bad
  - functional programming not yet popular
  - single assignment
  - variable-less programming
- state is valuable

# ACCESS

- accessing values
- invoking computations

# DIRECT ACCESS

- "most of my thoughts have nothing to do with storage"

# INDIRECT ACCESS

- cache
- listeners

# COMMON STATE

- used in one operation only?
- used at one time only?
- what operations have to work on the state?
- what is the lifetime of the state?

# VARIABLE STATE

- use a map

# EXTRINSIC STATE

- e.g. persistence
- identity map
  - the state is stored elsewhere

# VARIABLE

- not much to say

# LOCAL VARIABLE

- use simple names
- should have same lifetime and scope
  - "siblings"

# FIELD

- consider final
- roles
  - helper
  - flag
  - strategy
  - state
  - components

# PARAMETER

- preferable to static fields
  - coupling
- weaker coupling than permanent reference

# COLLECTING PARAMETER

- use one parameter as a "basket" to collect results

# OPTIONAL PARAMETER

- e.g. creating a `Socket`
  - with few or more parameters
  - "telescoping"

# VAR ARGS

- use varargs to pass a variable number of arguments instead of a collection

# PARAMETER OBJECT

- group many parameters in its own object

# CONSTANT

- store state that doesn't vary as a constant

# ROLE-SUGGESTING NAME

- short names
- no type info in name
- no "l" for local, like lCount, or "f" for field
- reuse common names
  - result
  - each
  - count
- saving letters with 1-letter names is false economy

# DECLARED TYPE

- List or Collection, think about promises you make
- be pessimistic ;-)
- allow as litte information as possible to spread as narrowly as possible
- communicate the use
  - not in variable name, e.g. `uniqueBooks`
  - rather `Set<Book>`

# INITIALIZATION

- initialize variables declaratively as much as possible

# EAGER INITIALIZATION

- initialize fields at instance creation time

# LAZY INITIALIZATION

- initialize fields whose values are expensive to calculate just before they are first used

# BEHAVIOR PATTERNS (18)

- history: all based on the concept of the **instruction**
  - by Neumann János Lajos, from Budapest, aka John von Neumann
- later: the **expression**
  - the LISP school

# CONTROL FLOW

- sequence
- conditional
- loops
- groupings
  - in class
  - delgating
- levels of abstraction

# MAIN FLOW

- happy day, or hostile environment?
- focus on main flow, make seldom executed corner cases less prominent
- use exceptions to keep main flow clean
  - main exception handler
  - consider coupling and dependencies

# MESSAGE

- all about **change**
  - receiver will be changed
  - sender will not be changed
- ```
  compute() {
    input();
    process();
    output();
  }
  ```
- no functions, no return values
- all in receiver, or side-effects
  ```

# CHOOSING MESSAGE

- vary the implementors of a message to express choices
- like a case statement
- 
```
public void displayShape(Shape subject, Brush brush) {
  brush.display(subject);
}
```
- implementation chosen by runtime type of brush

# DOUBLE DISPATCH

- vary the implementors of messages along two axes to express cascading choices

# DECOMPOSING (SEQUENCING) MESSAGE

- break complicated calculations into cohesive chunks

# REVERSING MESSAGE

- Make control flow symmetric by sending a message that can be implemented in different ways

# INVITING MESSAGE

- invite future varation by seindin a message that can eb implemented in different ways
- aka hook

# EXPLAINING MESSAGE

- send a message to explain the purpose of a clump of logic

# EXCEPTIONAL FLOW

- express the unusual flow of control as clearly as possible without interfering with the expression of the main flow

# GUARD CLAUSE

- express local exceptional flows by an early return
- IF/ELSE is equally important
  - do you want that?
- single entry, single exit
  - not so relevant any more today
- **nested conditionals breed defects**

# EXCEPTION

- express non-local exceptional flow with exceptions

# CHECKED EXCEPTIONS

- makes refactoring more difficult
- more cost
- risk of termination

# EXCEPTION PROPAGATION

- wrap
- add detail

# METHODS PATTERNS (24)

- the gigantic procedure, why not?
  - independent reuse
  - independent reading
- issues
  - size
  - purpose
  - naming

# COMPOSED METHOD

- compose methods our of calls to other mehods
- levels of abstraction

```
compute() {
  input();
  flags |= 0x0080;
  output();
}
```

# INTENTION-REVEALING NAME

- name methods after what they are inteded to do
- think about the calling side
- not: implementation detail

# METHOD VISIBILITY

- make methods as private as possible
- private, final
- thik about responsibility

# METHOD OBJECT

- turn complex methods into their own objects
- Kent's favourite

# OVERRIDDEN METHOD

- override methods to express specializaiton
- abstract?
- invitaton
- final

# OVERLOADED METHOD

- provide alternative interfaces to the same computation
- different parameters
- number of parameters, carefully
- don't mess with return type

# METHOD RETURN TYPE

- declare the most general possible return type
- `void` implies side effects
  - in the receiver, or elsewhere
- distinguis procedures and functions
- use most abstract type
  - consider interface
  - expect changes

# METHOD COMMENT

- comment methods to communicate information not easily read from the code
- better: use names
- redundancy is waste
- even consider tests
  - e.g. for calling order of different methods
  - this after that

# HELPER METHOD

- create small, private methods to express the main computation more succintly
- hide details in composed method

# DEBUG PRINT METHOD

- use `toString()` to print useful debugging information
- this is valuable
- for different audiences
- abuse: parse `toString()` output

# CONVERSION

- express the conversion of one type of object to another cleanly

# CONVERSION METHOD

- for simple, limited conversions, provide a method on the source object that returns the converted object
- use sparingly, not 20 times
- better: constructor, see next
- beware of dependency

# CONVERSION CONSTRUCTOR

- for most conversions, provide a method on the converted object's class that takes the source object as parameter
- don't pile up conversions in source object

# CREATION

- express object creation clearly

# COMPLETE CONSTRUCTOR

- write constructors that return fully formed objects

# FACTORY METHOD

- express more complex creation as a static method on a class rather than a constructor
- advantages
  - return different type
    - even interface
  - **can name after intention**

# INTERNAL FACTORY

- encapsulate in a helper method object creation that may need explanation or later refinement

# COLLECTION ACCESSOR METHOD

- provide methods that allow limited access to collections
- an `iterator` is safe except for `remove()`
- example, next slide

```
Interator<Book> getBooks() {
  final Iterator<Book> reader = books.iterator();
  return new Iterator<Book>() {

    public boolean hasNext() {
      reaturn reader.hasNext();
    }

    public Book next() {
      return reader.next();
    }

    public void remove() {
      throw new UnsupportedOperationException();
    }
  };
```

# BOOLEAN SETTING METHOD

- if it helps communication, provide two methods to set boolean values, one for each state

```
void setState(boolean newState);

void valid() { ...
void invalid() { ...
```

# QUERY METHOD

- return boolean values with methods as `asXXX`
- prefix with a form of "be" or "has"
  - `isValid()`, `hasNext()`

# EQUALITY METHOD

- define `equals()` and `hashCode()` together

# GETTING METHOD

- occasionally provide access to fields with a method returning that field

# SETTING METHOD

- even less frequently provide the ability to set fields with a method

# SAFE COPY

- avoid aliasing errors by copying objects passed in or out of accessor methods
- better: immutable objects

# WRAP UP

# UPDATE NEEDED?

- style, fashion
  - "he" not "she"
  - functional is underrated
- recent additions to Java not covered
  - interfaces, defaults
  - annotations
- patterns explained in "Hello, world!" context
  - not in real-world context
  - authorization, persistence, logging, concurrency

# CONCLUSION

- everything in the book is valid today
- worth reading also for non-Java developers
- pattern language to talk about your code
  - table of contents will be your most valuable pages
  - lists all the patterns conscisely

# NEXT

- buy the book, read it
- keep it on your desk
- don't expect colleagues to read it
  - it's a conversation
- establish patterns language in your team
  - when reviewing code, pull requests

# THE END.

- https://github.com/gerhardt-io/slides
- Software Experts Network Stuttgart e.V.
  - **nachher am Stand**
- we're not hiring, kein Gewinnspiel ;-)
- freie Kapazitäten ab September