OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

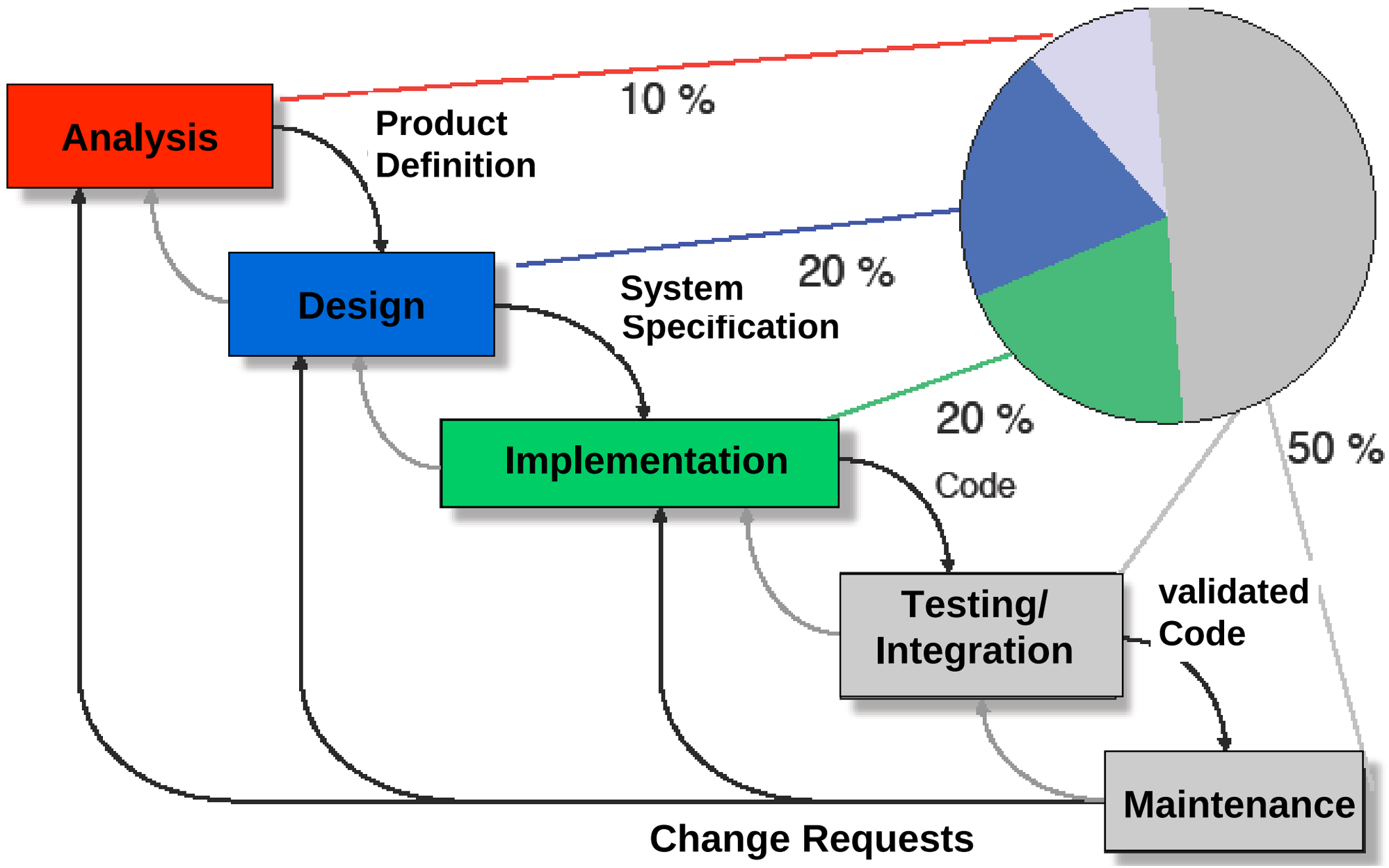
FACULTY OF COMPUTER SCIENCE

SwarmLab

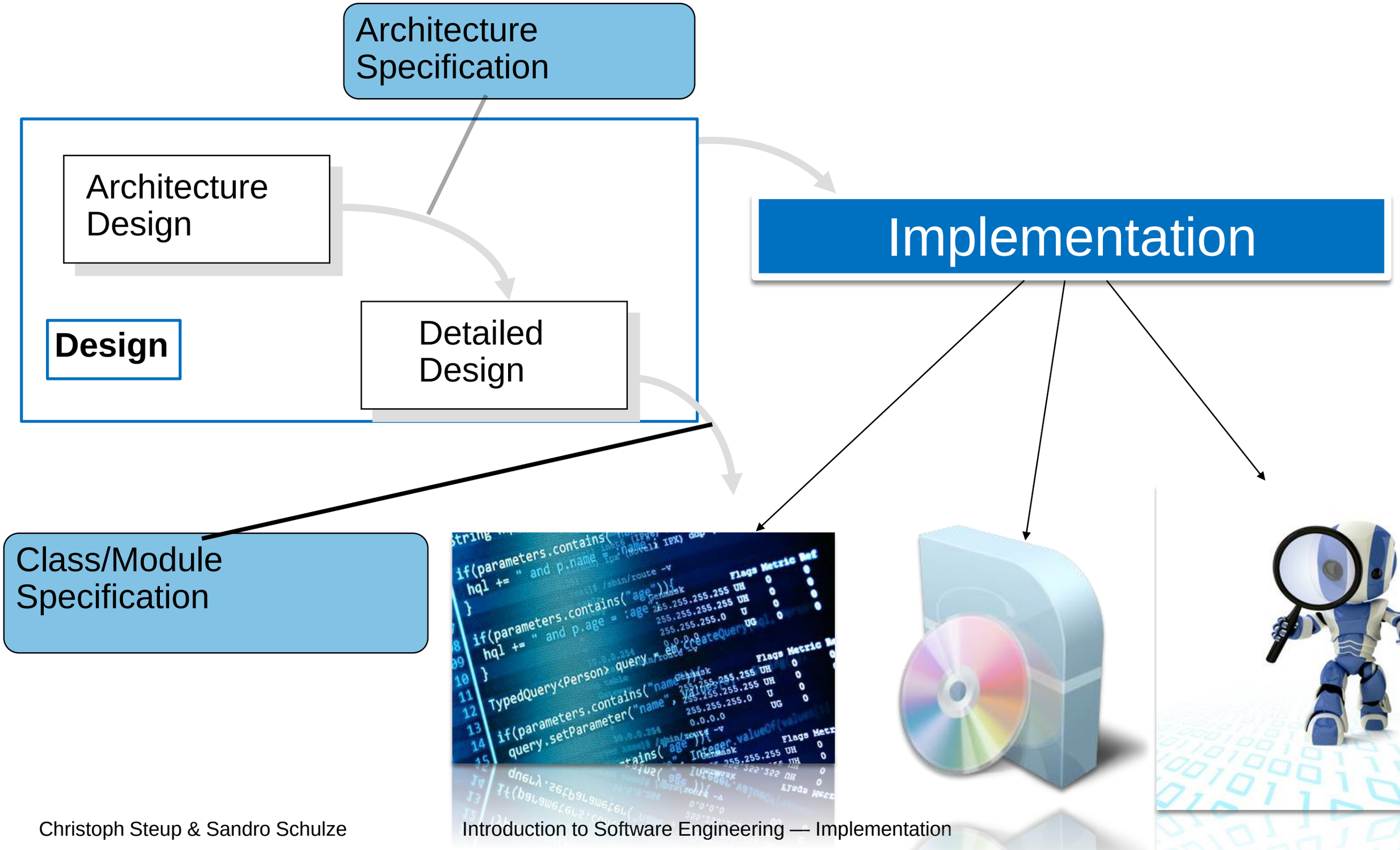
Introduction to Software Engineering for Engineers
Lecture-08: Implementation & UI Design
Part 1: Introduction and Programming Languages

Dr.-Ing. Christoph Steup

Implementation Phase

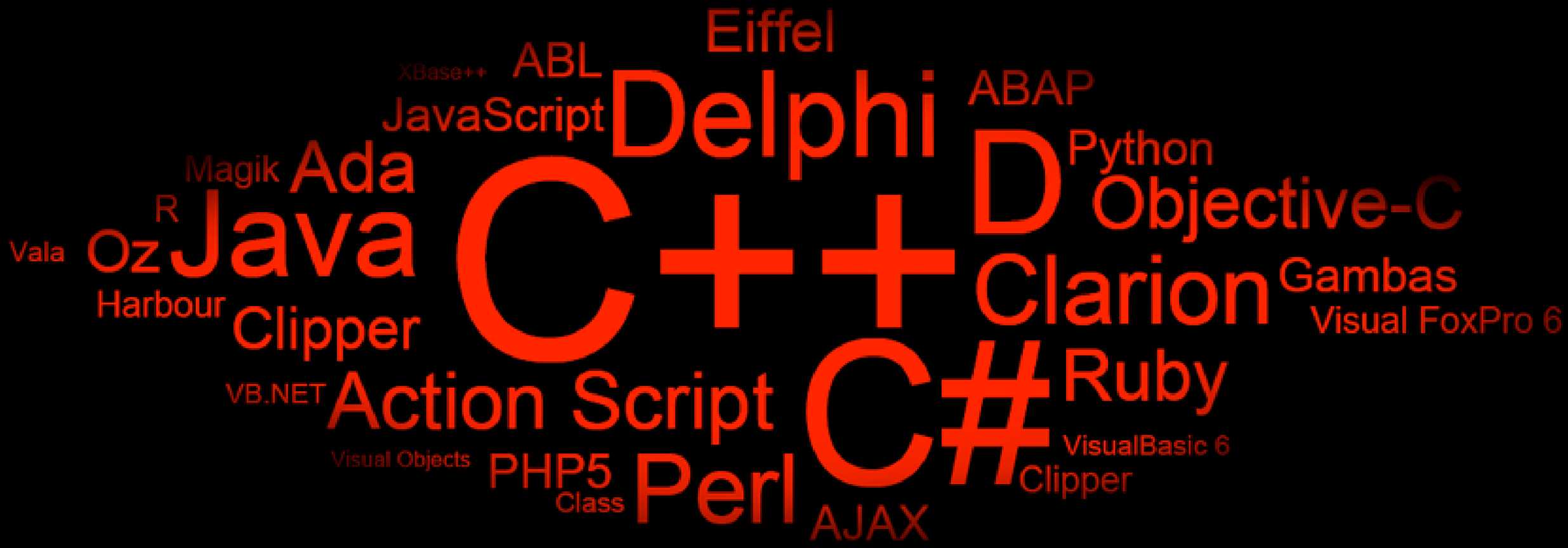


Definition: Implementation



Content

- Programming Languages
- Best practices in Implementation
  - Formatting
  - Choosing identifiers
  - Comments
  - Style guideline
- User Interface Design
- Summary



Programming Languages

Don’t underestimate the importance of beer

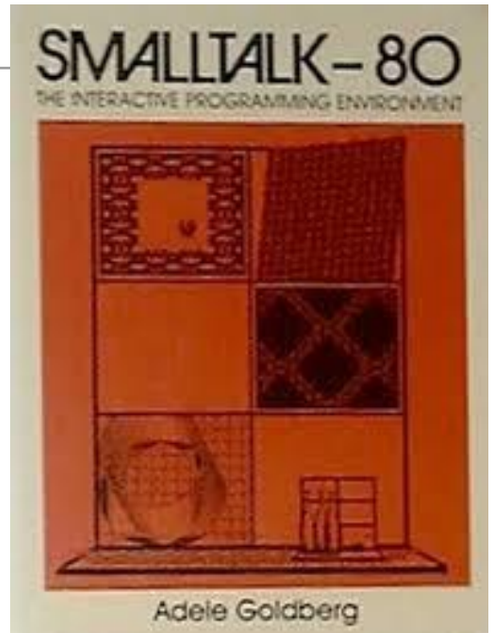
http://www.99-bottles-of-beer.net/

Procedural Programming

Characteristics	➢ imperative programming paradigm ➢ decompose programs & algorithms into subtasks ➢ ideas for a module concepts partially available ➢ comfortable definition of data structures in memory
Examples	*Fortran, COBOL, C, Pascal, Forth*
Conclusion	• appropriate for small and medium-sized systems • procedural programming paradigm is subsumed by object-oriented programming —> usually not used in its pure format. • side effects (e.g., by pointers, pointer arithmetic, …) • separation of data structure and functions impedes maintainability • Easy to learn – hard to master

Object-Oriented Programming

Characteristics	➢ Objects (classes) for encapsulation of data and functionality ➢ Objects are created dynamically: object identity
Concepts	✓ inheritance as mechanism for adaptation and improving the reuse ✓ „good design“, to support maintainability and extensibility ✓ coding conventions for readability
Examples	*Python, Ruby, Java, C#, C++, Kotlin*
Conclusion	• OOP is today the main paradigm for large projects • For generic problems usually more efficient than domain-specific languages • Needs more education then procedural



Functional Programming Languages

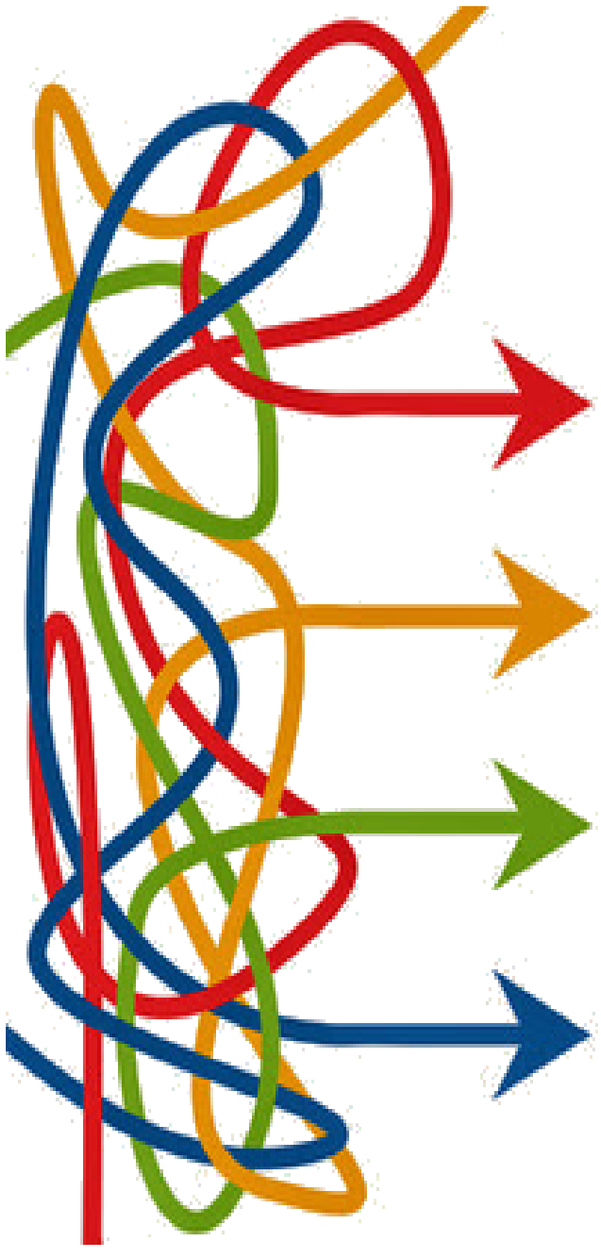
Characteristics	▪ very expressive type system (e.g., Lambda Calculus) ▪ pattern matching on arguments ▪ efficient definition of data structures and functions ▪ compact formulation ▪ Mostly free of side effects. and thus, easy to understand ▪ higher-order functions (functions on functions)
Examples	*Haskell, Makefiles, Scala, OCaml*
Conclusion	• effective programming, but slower execution times due to the interpreter (Make) • Compiled Ocaml is one of the fastest languages • difficulties with interactive systems (e.g., a GUI)

OCaml

Domain-Specific Languages (DSL)

Logical Programs	*Prolog* programs are logical statements as horn clauses
Visual Programming	*Matlab Simulink* a) program is composed by building blocks b) Modelling, e.g., with executable Statecharts
Parallel Programming	for massively distributed systems: *Erlang*
Script Languages	• Example: *R* • Mostly no static type system • Focus on specific problems: text processing (e.g., regular expressions) Examples: Perl, awk (gawk) • better and better integration with other languages (e.g., *Java + Prolog*) • Further DSLs: *HTML, JSP, XML, SQL, PHP* • Not all programming language, some are description languages

Selection Criteria



- Which technical environment is required?
- Legacy System? Any requirement given by the customer?
- Human Factor: Which knowledge/preferences to programmers have?
- Which kind of library support (e.g., which functions) are required?
- Tool support? Compiler, Debugger, Profiler, Linter, Code Completion, Testing

FACULTY OF
COMPUTER SCIENCE

Introduction to Software Engineering for Engineers
Lecture-08: Implementation & UI Design
Part 2: Coding Conventions

Dr.-Ing. Christoph Steup

Quality by means of Standards

Quality of Source Code:

- ✓ Functionality
- ✓ Readability
- ✓ Maintainability
- ✓ Efficiency
- ✓ Elegance

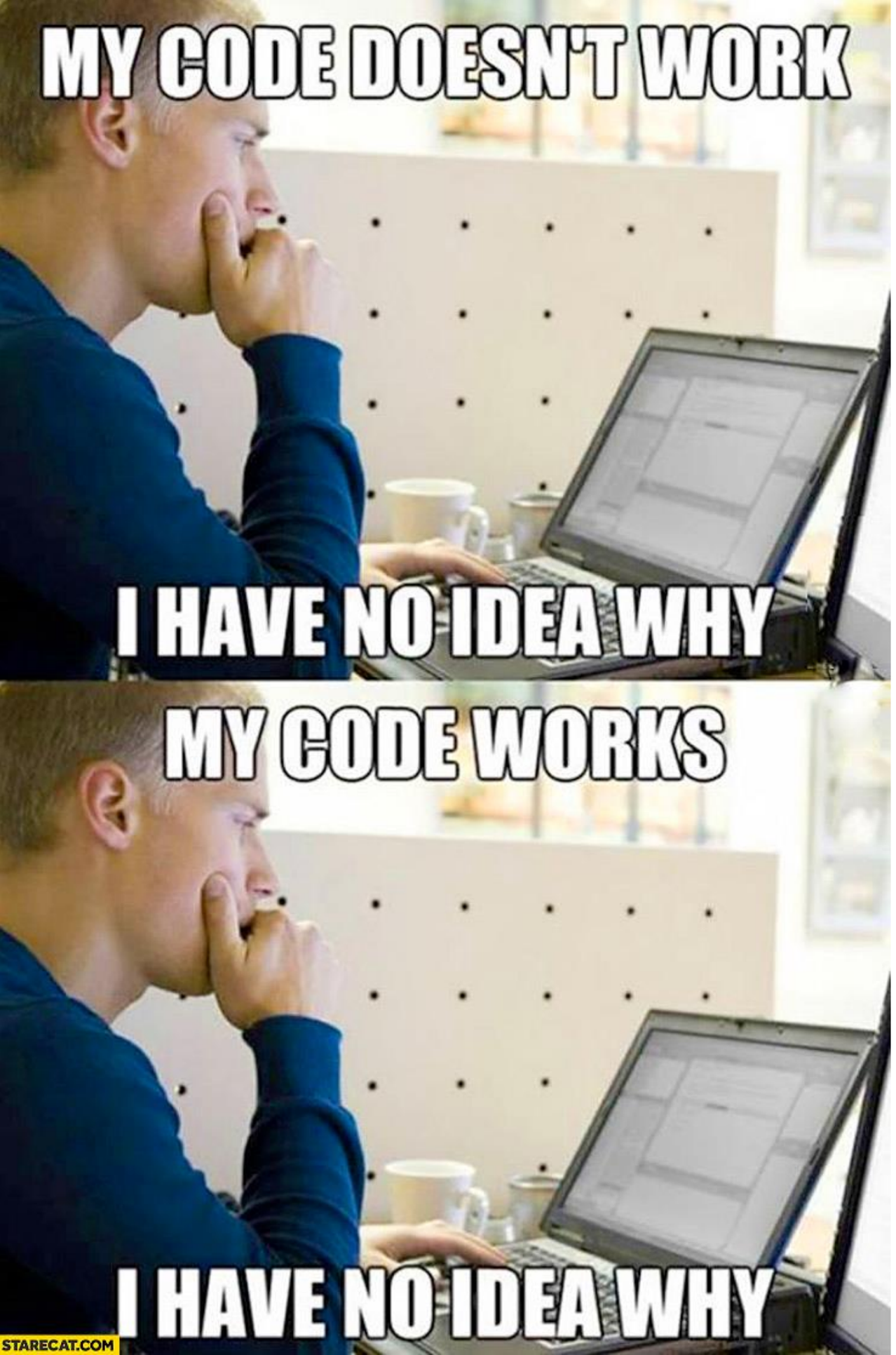
Highly depends on HOW you program → „goodness of code“

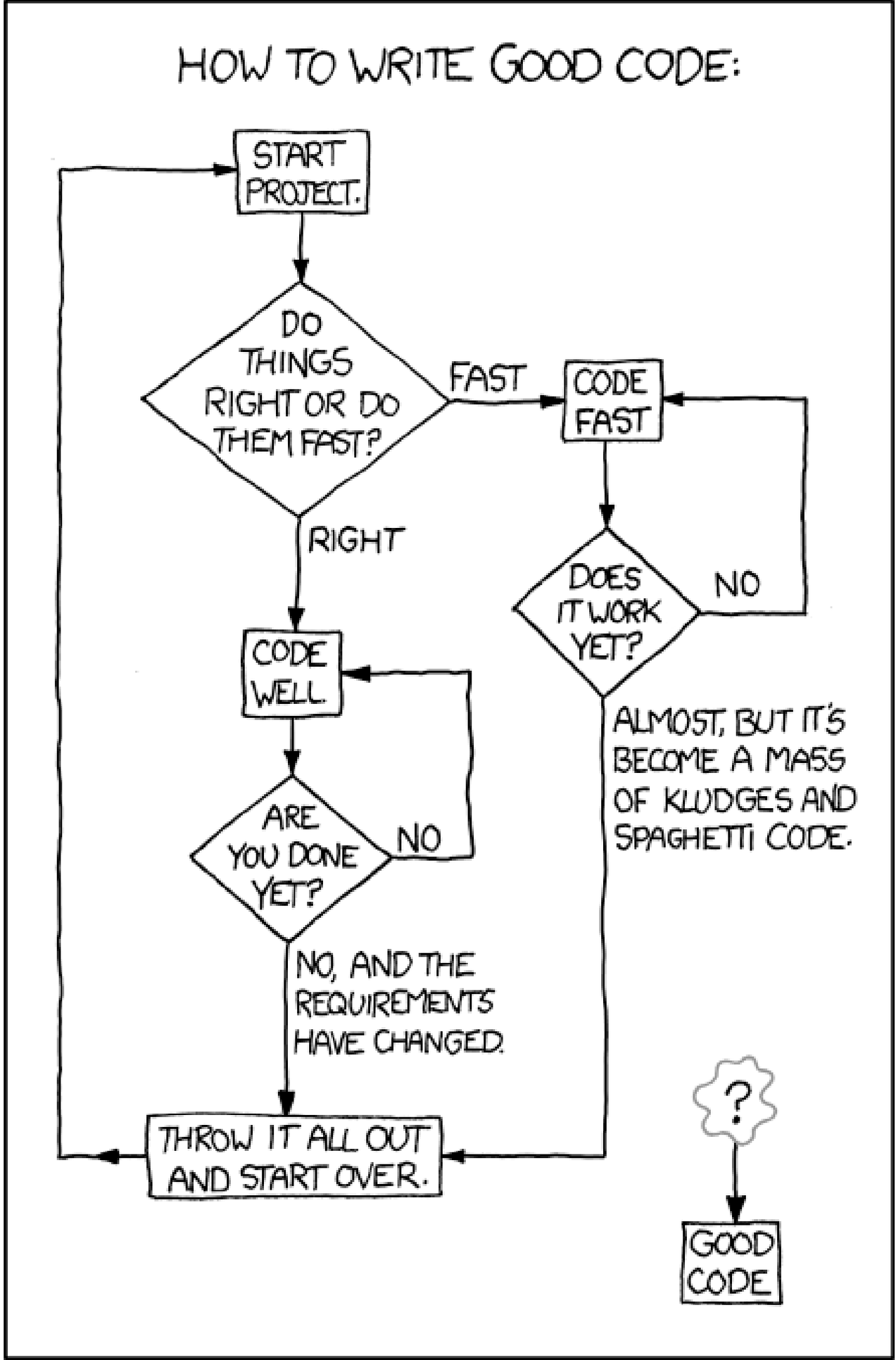
Obfuscated code

This is Akari:

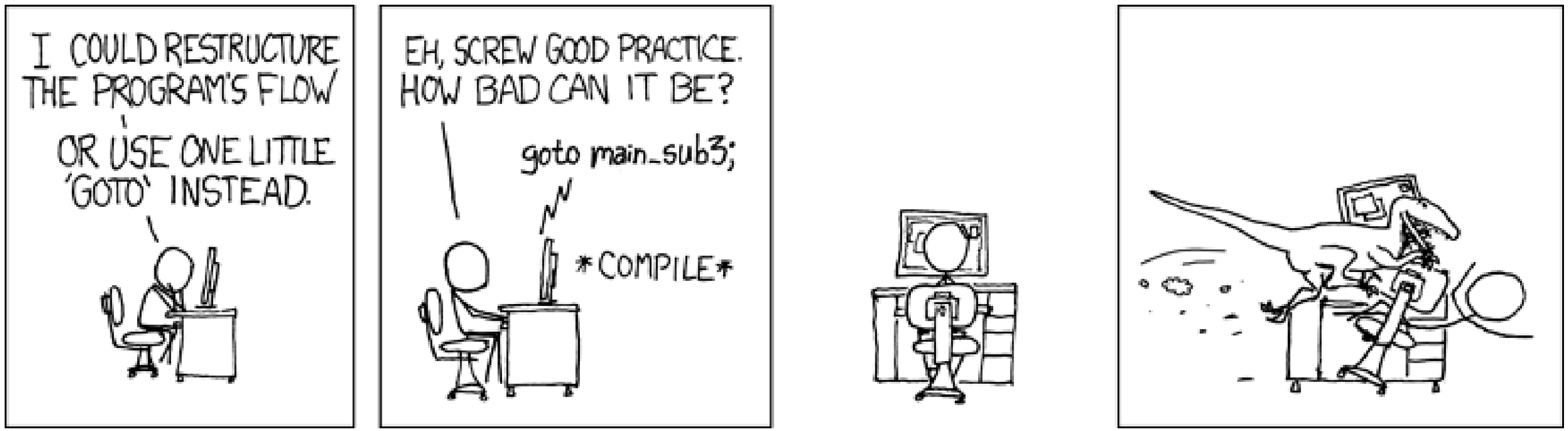
- Akari can scale down images
- She can even scale down herself
- How does she do it? No one knows …

http://www.geeks3d.com/20120418/20th-international-obfuscated-c-code-contest-results/


MY CODE DOESN'T WORK
I HAVE NO IDEA WHY
MY CODE WORKS
I HAVE NO IDEA WHY
STARECAT.COM


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Coding Conventions



“Go-To” Considered Harmful, E.W.Dijkstra, 1968

- Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Elements of Programming Style

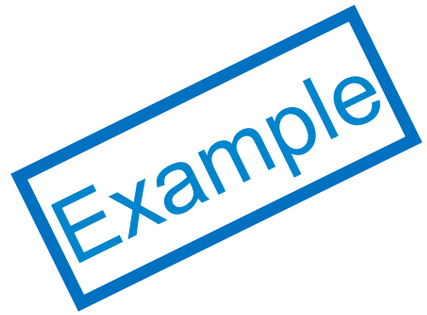
Write clearly - don't be too clever.
Don't sacrifice clarity for efficiency.
Don't patch bad code - rewrite it.
Don't comment bad code - rewrite it.

Make it right before you make it faster.
Make it clear before you make it faster.
Keep it right when you make it faster.
Let your compiler do the simple optimizations.

Excerpt from: Kernighan/Plauger,
The Elements of Programming Style, McGraw-Hill 1978 (!)

Formatting

Formatting Guidelines




```
public class Z {

  public static void main(String[] args) {
    double x;
    double z;
    int l;
    x = Console.readDouble("X:");
    z = Console.readDouble("Z:") / 100;
    l = Console.readInt("L:");
    double y;
    for (y = z - 0.01; y <= z + 0.01; y += 0.00125){
      double p = x * y /12 /
      (1 - (Math.pow(1/(1 + y / 12), l * 12)));
    System.out.println(100*y+" : "+p);
    }
  }

}
```

Indentation

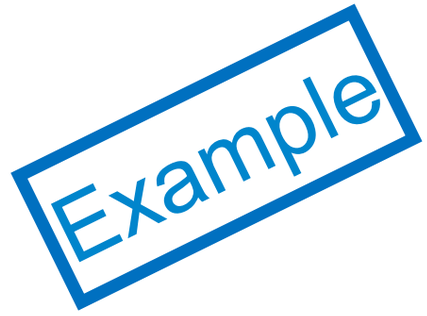
JavaSoft conventions for indentation:

- 4 characters for default indentation
- 8 characters indentation for special purposes

Important:

- **Code is written from the reader's perspective!**
- …because code is much more(!) often read than written.
- Also: Effort for reading bad code is enormous (can put you in really bad mood).

Examples for Indentation

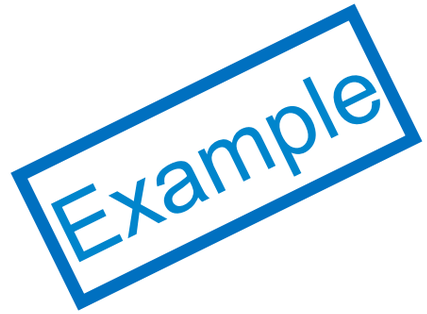


JavaSoft convention for long method signatures:

```
void method (int x, Object y, String z, Xyz v,
                    float p);  //conservative
```

```
private static synchronized void ratherLong
    (int x; Object y, String z, Xyz v,
     float p) {
        // better: 8 chars indentation
    x = ... // method body is indented only 4 chars
```

Examples for Indentation (2)

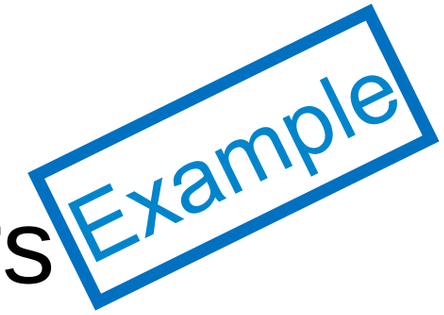


JavaSoft convention for conditional statements:

```
// Don’t use this indentation

if ((condition1 && condition2 && condition3)
   || condition4)
   doSomethingAboutIt(); // easy to miss this line
```

```
// better solution

if ((condition1 && condition2 && condition3)
       || condition4) {
     doSomethingAboutIt(); …

}
```

Examples for Parentheses and Separators



code that is relatively hard to maintain (Java):

```
if (condition)
   method();
```

```
if (condition)
   method();
   method2();
```

code that is easier to maintain:

```
if (condition) {
   method();
}
```

```
if (condition) {
   method();
   method2();

}
```

code that is relatively hard to maintain (Pascal):

```
if xyz then
   begin
     statement1;
     statement2
   end;
```

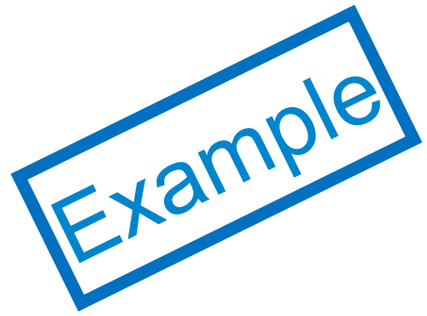
```
if xyz then
   begin
     statement1;
     statement2
     statement3;
   end;
```

code that is easier to maintain:

- semicolon after `statement2`

Identifiers

Bad Code?




```
1
2  int num[] = new int[5];
3  System.out.println("Ein:");
4  Scanner sc = new Scanner(System.in);
5  for(int p = 0; p<5; p++)
6      num[p] = sc.nextInt();
7  int j;
8  boolean flag = true; // set flag to true
9  int temp; // holding variable
10 while (flag) {
11     flag = false; // set flag to false awaiting a possible swap
12     for (j = 0; j < num.length - 1; j++) {
13         if (num[j] < num[j + 1]) // change to > for ascending sort
14         {
15             temp = num[j]; // swap elements
16             num[j] = num[j + 1];
17             num[j + 1] = temp;
18             flag = true; // shows a swap occurred
19 } }
20 }
21 int z=0;
22 for(int t = 0; t<5; t++)
23     z += num[t];
24 System.out.println("Erg:"+z/5.0);
```

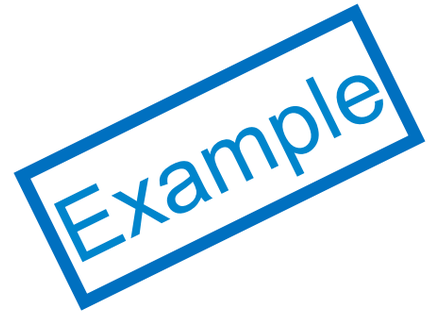
And Now?

Example

```
1
2  private static final int ARRAY_LENGTH = 5;
3  public static void main(String[] args) {
4      int arrayOfIntegerValues[] = new int[ARRAY_LENGTH];
5      askUserForInput(arrayOfIntegerValues);
6      sortWithBubbleSort(arrayOfIntegerValues);
7      double meanValue = calculateMeanValue(arrayOfIntegerValues);
8      printResultToConsole(meanValue);
9  }
10 private static void printResultToConsole(double result) {
11     System.out.println("The mean Value is:"+result);
12 }
13 private static double calculateMeanValue(int[] allValues) {
14     int sum = calculateSumOfValues(allValues);
15     return sum/(double)allValues.length;
16 }
```

Already better…

Notations




```
1 int nt = 5;
```

```
1 int neithersothiscannotbereadbyotherprogrammerhellya = 5;
```

```
1 int _ = 5;
2 int __ = 8;
3 int _$__ = _ - ___;
```

Disclaimer: These are NEGATIVE examples

CamelCaseNotation

```
String an_Example = “Test”;

int differentWordsSeparatedByASignAtTheBeginning;

StringWithExtensions = test = “Foo Bar”;
```

Usage	variables, functions, class names
CamelCaseNotation	

UPPER_CASE Notation

```
final int FIXED_ARRAY_LENGTH = 42;

final String RANDOM_HASH = “Foo Bar”;
```

Usage	Usually only for constants

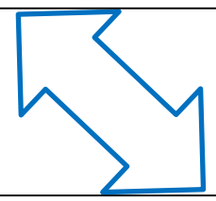
lower_case Notation

```
1 package de.ovgu.iti.software_engineering;
2
3 package de.ovgu.iti.softwareengineering;
```

Usage	usually for packages, imports…

Descriptive Variable Names

```
public static void main(String[] args {
    double x = Console.readDouble(“X:”);
    double z = Console.readDouble(“Z:”)/100;
    int l = Console.readInt(“L:”);
    double y;
    …
}
```



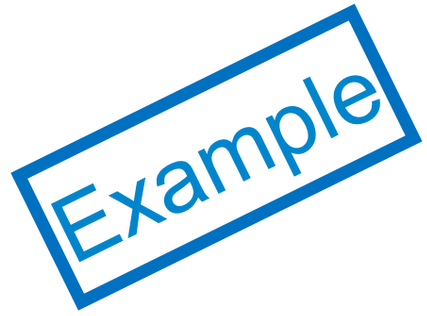
```
public static void main(String[] args {
  double amount = Console.readDouble(“Amount:”);
  double ratePerYear = Console.readDouble(“Interest rate:”)/100;
  int duration = Console.readInt(“Duration:”);
  double y;
}
```

- should always describe their content
- No “one-letter” variables
- no abbreviations

Examples for Naming Conventions

	Convention	Example
Class	• Noun, first letter upper case, remainder lower case • use entre words, composition of multiple words using CamelCase notation	`Account`, `StandardTemplate`
Method	• Verb, imperative, first letter in lower case • methods used to read/write attributes should have the prefix get/set respectively	`checkAvailability()`, `doMaintenance()`, `getDate()`
	Retrieval/Queries (i.e., methods that do not change anything, e.g., boolean result):	`isLarge()`, `hasFather()`
Constants	• Upper case only, composition of multiple words with "_" • Default prefixes: "MIN_", "MAX_", "DEFAULT_"	`NORTH`, `BLUE`, `MIN_WIDTH`, `MAX_WIDTH`, `DEFAULT_SIZE`,
Attributes	• Similar to methods • Should be meaningful yet not overly long; core should be mnemonic • Alternative: With leading underscore (e.g., often used in C	`_availability`, `_date`, `dateOfBirth`

Readability By Identifier Selection




```
public static void main(String[] args) {
  double amount;      //amount to get interest rate for
  double ratePerYear; //yearly interest rate
  int duration;       //duration in years

  amount = Console.readDouble("Amount:");
  ratePerYear = Console.readDouble(“Interest rate:") / 100;
  duration = Console.readInt("Duration:");

    double y;

    for (y = ratePerYear - 0.01;
        y <= ratePerYear + 0.01; y += 0.00125){
      double ratePerMonth = y/12;
      double payment = amount * ratePerMonth /
            (1 - (Math.pow(1/(1 + ratePerMonth),
                  duration * 12)));
      System.out.println(100*y+" : "+payment);
    }
  }

}
```

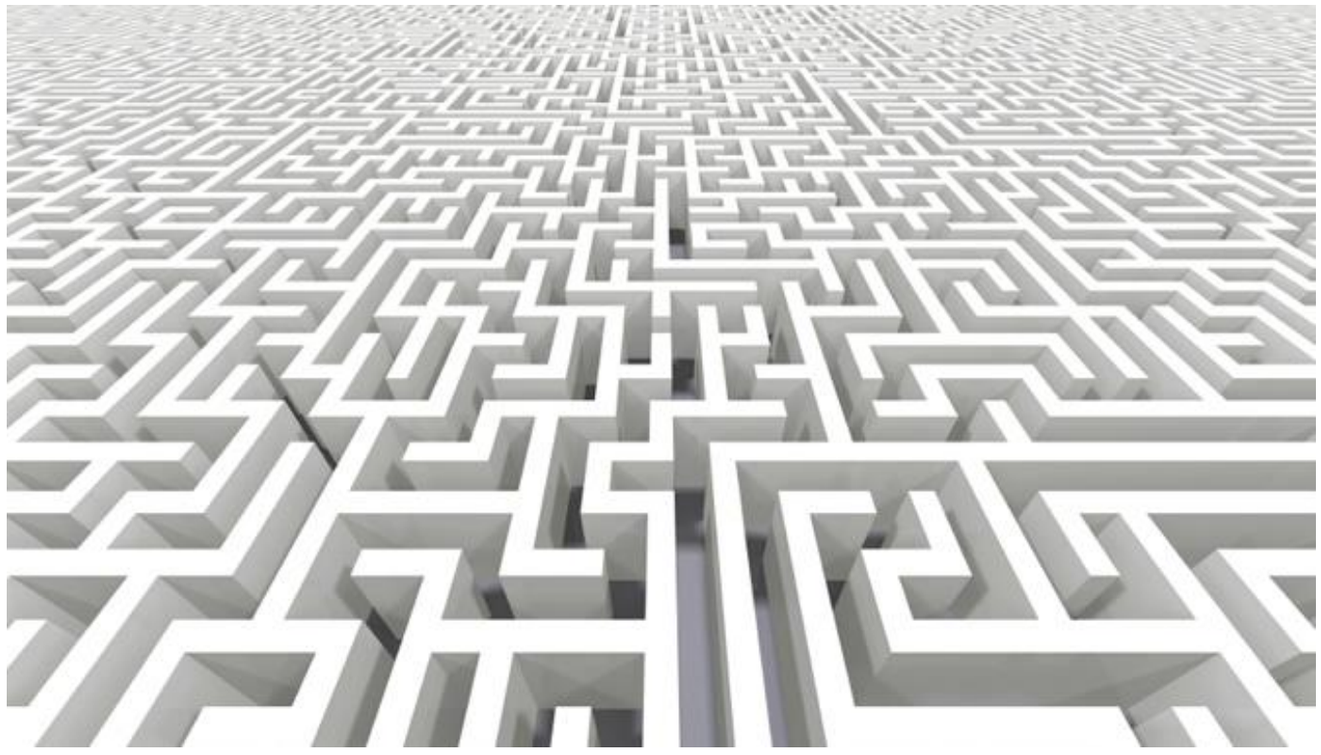
Complexity of Functions

But HOW to improve?

Splitting / Extraction of functions

Rule of thumb:

“When more then 7 lines, think about it.”



✓ Divide & Conqer → partial steps in separate function/method

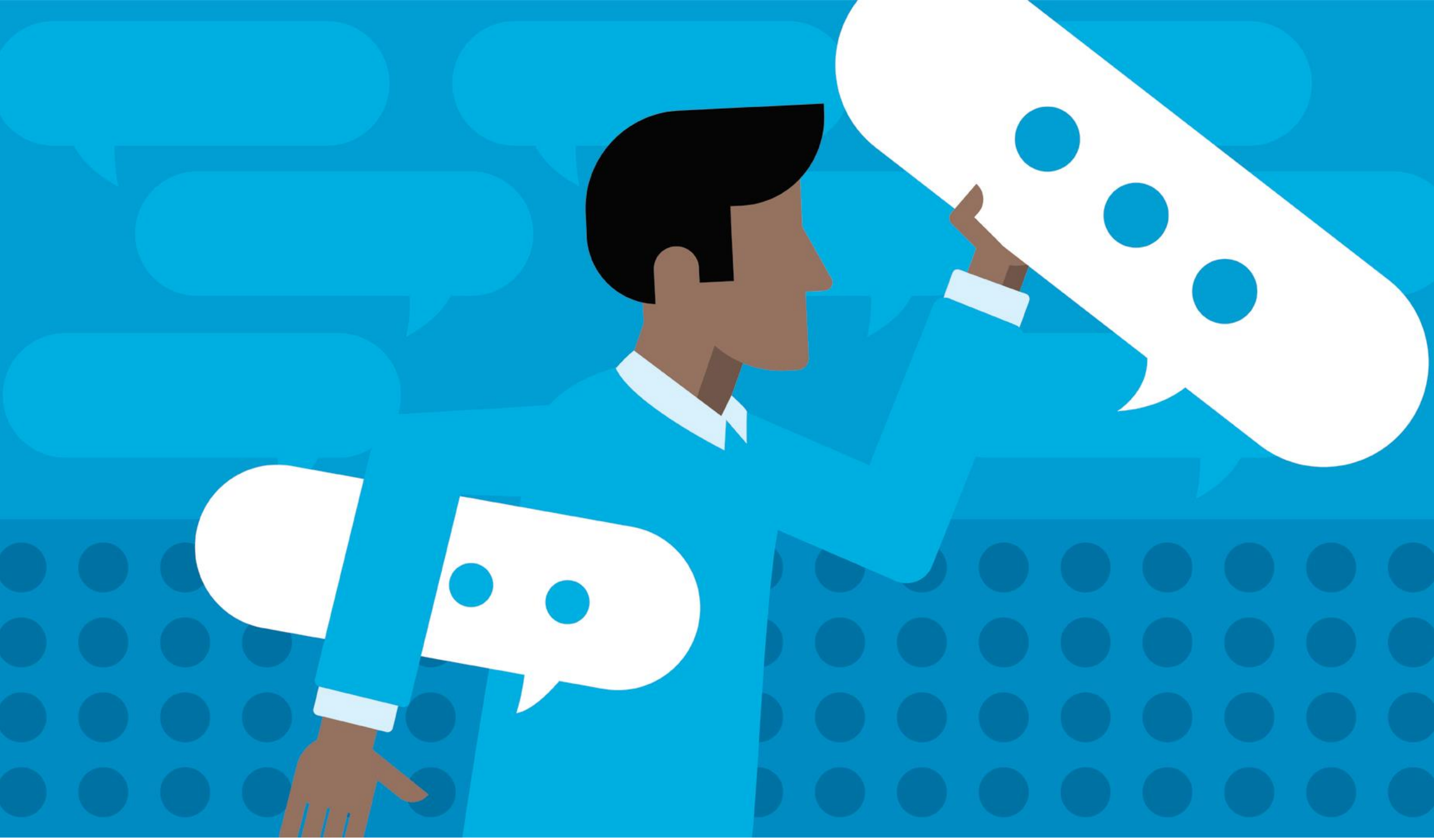
✓ descrMetNa → descriptive method names

Functions

Example

```
int num[] = new int[5];
System.out.println("Ein:");
Scanner sc = new Scanner(System.in);
for(int p = 0; p<5; p++)
        num[p] = sc.nextInt();
int j;
boolean flag = true; // set flag to true
int temp; // holding variable
while (flag) {
        flag = false; // set flag to false awaiting a possible swap
        for (j = 0; j < num.length - 1; j++) {
                if (num[j] < num[j + 1]) // change to > for ascending s
                {
                  temp = num[j]; // swap elements
                  num[j] = num[j + 1];
                  num[j + 1] = temp;
                  flag = true; // shows a swap occurred
                 }
        }
}
int z=0;
for(int t = 0; t<5; t++)
        z += num[t];
System.out.println("Erg:"+z/5.0);
```

```
public static void main(String[] args) {
    int arrayOfIntegerValues[] = new int[ARRAY_LENGTH];
    askUserForInput(arrayOfIntegerValues);
    sortWithBubbleSort(arrayOfIntegerValues);
    double meanValue = calculateMeanValue(arrayOfIntegerValues);
    printResultToConsole(meanValue);
}
```

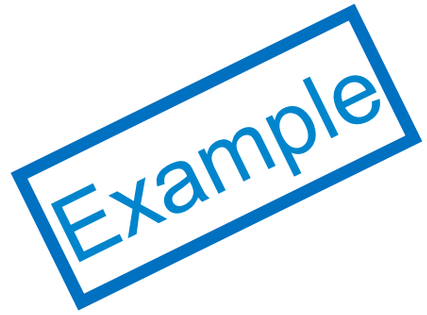


Comments

True of False?

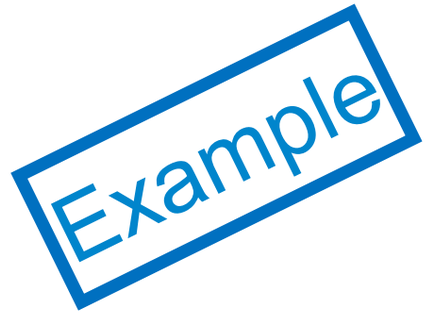
“Good code must not require any comments!!??”

Code Comments




```
1  // This method takes two integer values and adds them together via the built-in
2  // .NET functionality. It would be possible to code the arithmetic function
3  // by hand, but since .NET provides it, that would be a waste of time
4  private int Add(int i, int j) // i is the first value, j is the second value
5  {
6      // add the numbers together using the .NET "+" operator
7      int z = i + j;
8
9      // return the value to the calling function
10     // return z;
11
12     // this code was updated to simplify the return statement, eliminating the need
13     // for a separate variable.
14     // this statement performs the add functionality using the + operator on the two
15     // parameter values, and then returns the result to the calling function
16     return i + j;
17 }
```

Interface Comments



Well, how the cookies crumble … a small minority resists…

Good code should contain comments anyway:

```
1  /**
2    * @param front will be the first part
3    * @param middlepart will be the second part
4    * @param end will be the last part
5    * @return A concatenated String for the three params
6    */
7  public static String concatenate(String front,String middlepart,String
8     ...
9  }
10 /**
11   * @version 4.2 March 2013
12   * @author The Humble Programmer.
13     * This class provides some additional mathematical functions.
14   */
15 public class Mathematics {
16 ...
17 }
```

Comments

Commented Code	
Basic idea	▪ comments within the code are easier to maintain ▪ comments should arise in parallel with the corresponding code – “documenting afterwards" never works in practice! ▪ tools for generating the documentation (e.g., javadoc)
Perfection	✓ Comments for classes and methods constitute a consolidated specification of the source code
Comments should NOT	– make the code unreadable, e.g., by deforming the layout – contain redundant information compared to the corresponding code

NOTE: readable & comment-free source code is better than commented, yet unreadable source code.

Typical Application of Comments

Lead text of packages, classes, methods etc.	➢ purpose, parameter, result(s), e*xceptions* preconditions, dependencies (e.g., platform), side effects ➢ version, change history, status
Formal assertions	✓ pre- and post-conditions ✓ generally accepted assumptions (invariants)
Ease of reading	➢ summary of complex code fragments ➢ headings for structuring the code ➢ explanation of single peculiarities of the code, e.g., steps that are difficult to understand, side effects
Working notes	✓ restrictions, known problems ✓ open issues (“!!!“,“TODO“), proposals, placeholder

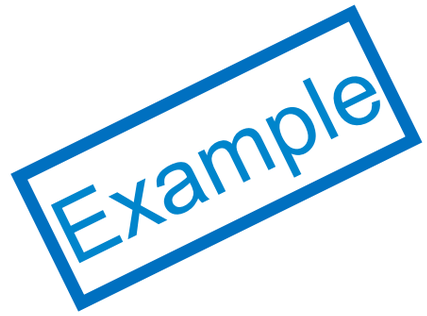
Documentation Tools

JavaDoc

- Tool that generates HTML documentation files out of Java source files
- like Java, developed by Sun Microsystems
- documentation can be enriched by specific comments and annotations in the source code
- usage of tags for detailed description of interfaces, classes, methods and fields

Alternative: Doxygen

Professionally commented Code

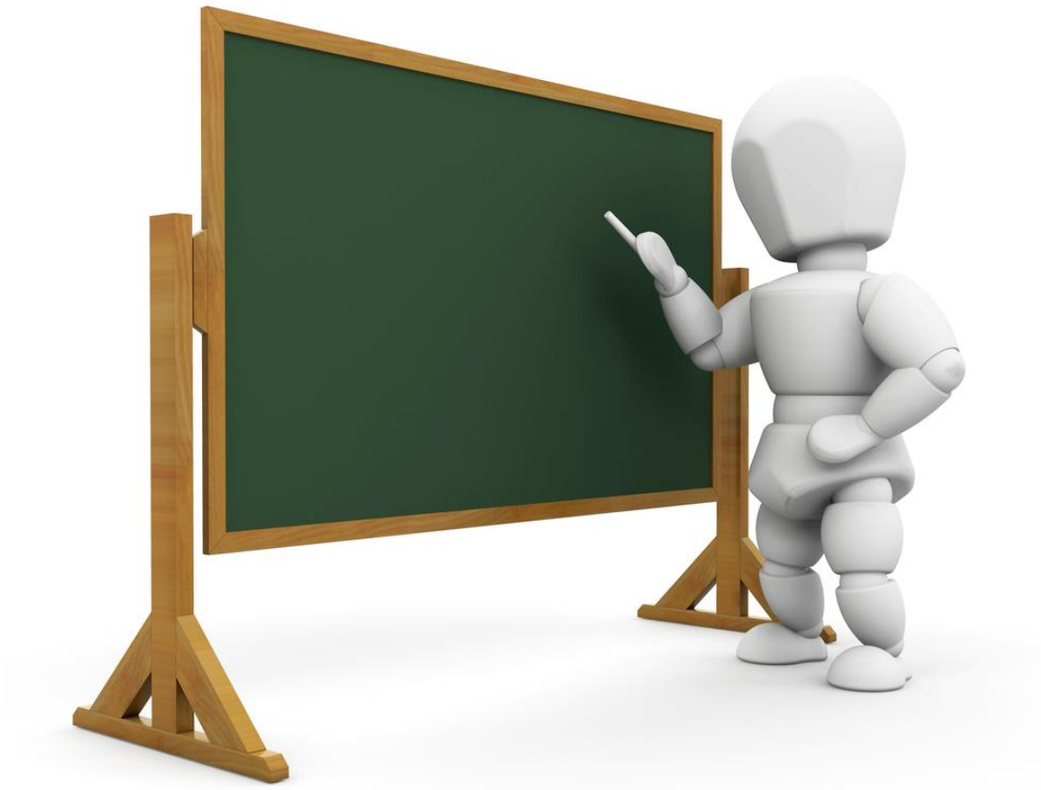



```
/*
 * @(#)Observer.java  1.14 98/06/29
 * Copyright 1994-1998 by Sun Microsystems, Inc., ...
 */
package java.util;

/**
 * A class can implement the <code>Observer</code> interface when it
 * wants to be informed of changes in observable objects.
 *
 * @author  Chris Warth
 * @version 1.14, 06/29/98
 * @see     java.util.Observable
 * @since   JDK1.0
 */
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the
     *                <code>notifyObservers</code> method.
     */
    void update(Observable o, Object arg);
}
```

Summary - Style of Coding/Programming

- Unfortunately, there is no common coding standard.
- Thus: Important to establish (and negotiate on) an own standard Within the team/company!

- Fundamental criteria:
  - meaningful and compact comments
  - choice of identifiers
  - indentation
  - size of code blocks (methods)
  - size of modules (classes)



- For Java, coding conventions can be found int the web (mostly compatible with those mentioned in the slides)

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FACULTY OF COMPUTER SCIENCE

SwarmLab

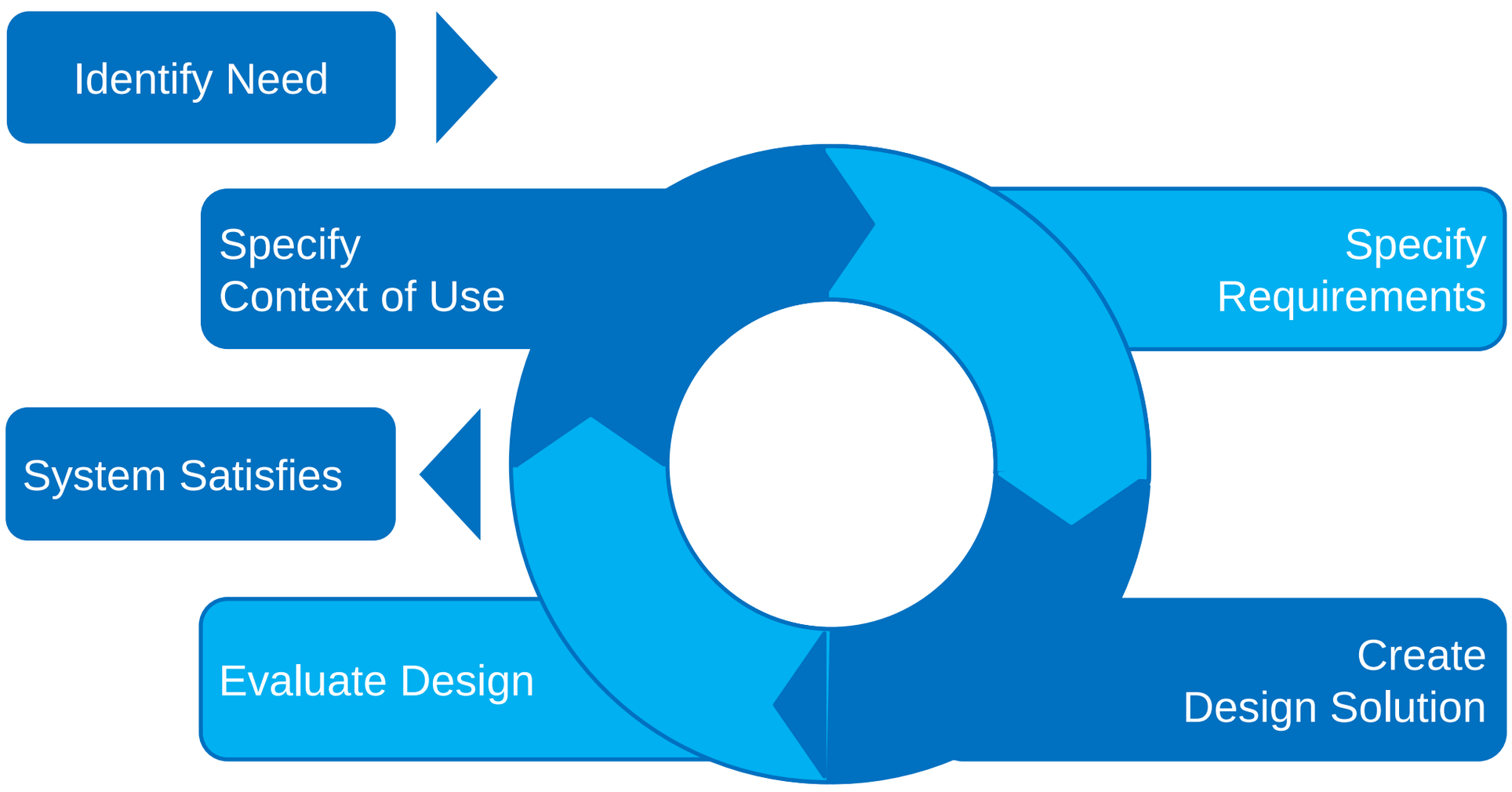
Introduction to Software Engineering for Engineers
Lecture-08: Implementation & UI Design
Part 3: UI Design - Specification

Dr.-Ing. Christoph Steup

User-Centered Design Process

Iterative Design

- to develop a design through repeated cycles (iterative) and in smaller portions at a time (incremental)



User-Centered Design Process

Iterative Design

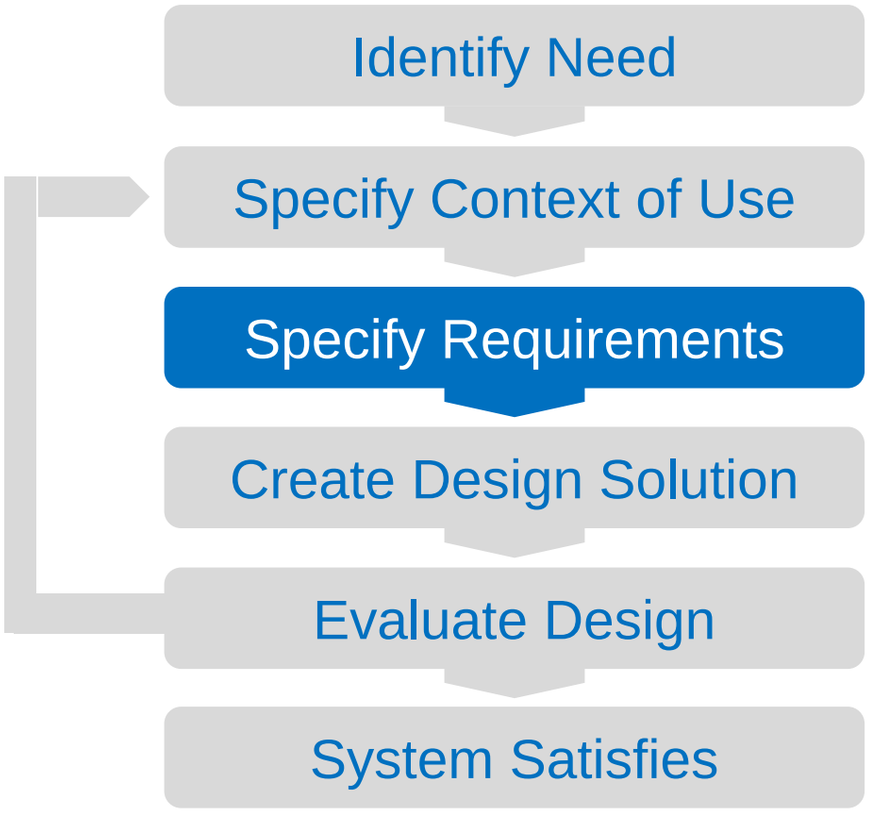
- to develop a design through repeated cycles (iterative) and in smaller portions at a time (incremental)

- Specify Context of Use
  - identify the user:
    - who will use the product
    - what they will use the product for
    - under what conditions they will use it

User-Centered Design Process

Iterative Design

- to develop a design through repeated cycles (iterative) and in smaller portions at a time (incremental)

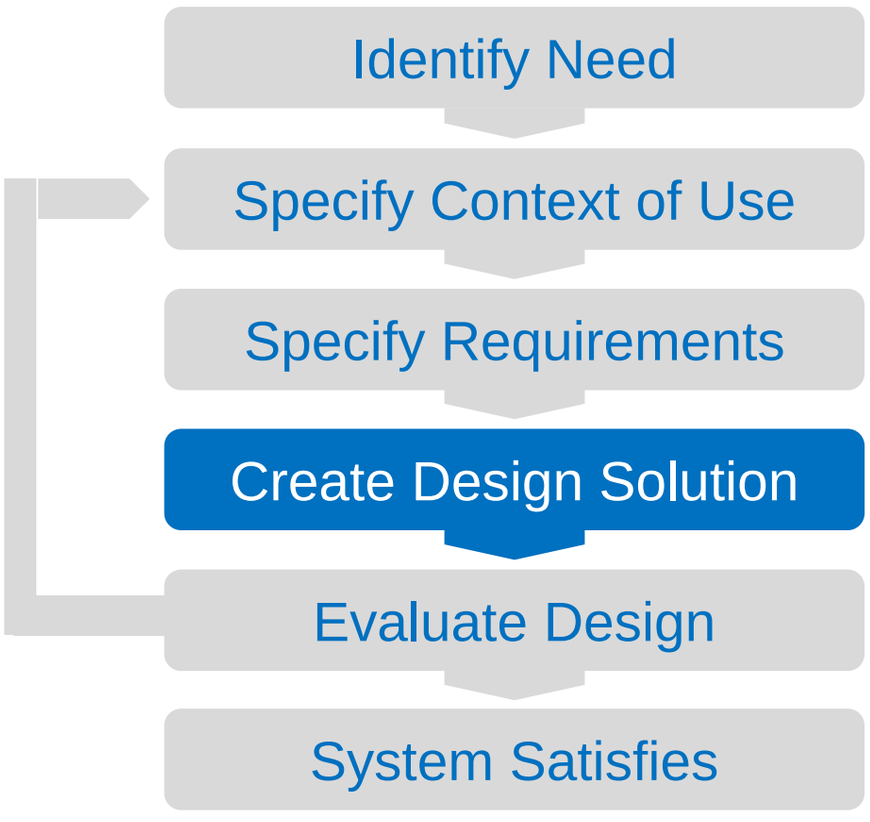


- Specify Requirements
  - identify any business requirements or user goals that must be met for the product to be successful

User-Centered Design Process

Iterative Design

- to develop a design through repeated cycles (iterative) and in smaller portions at a time (incremental)

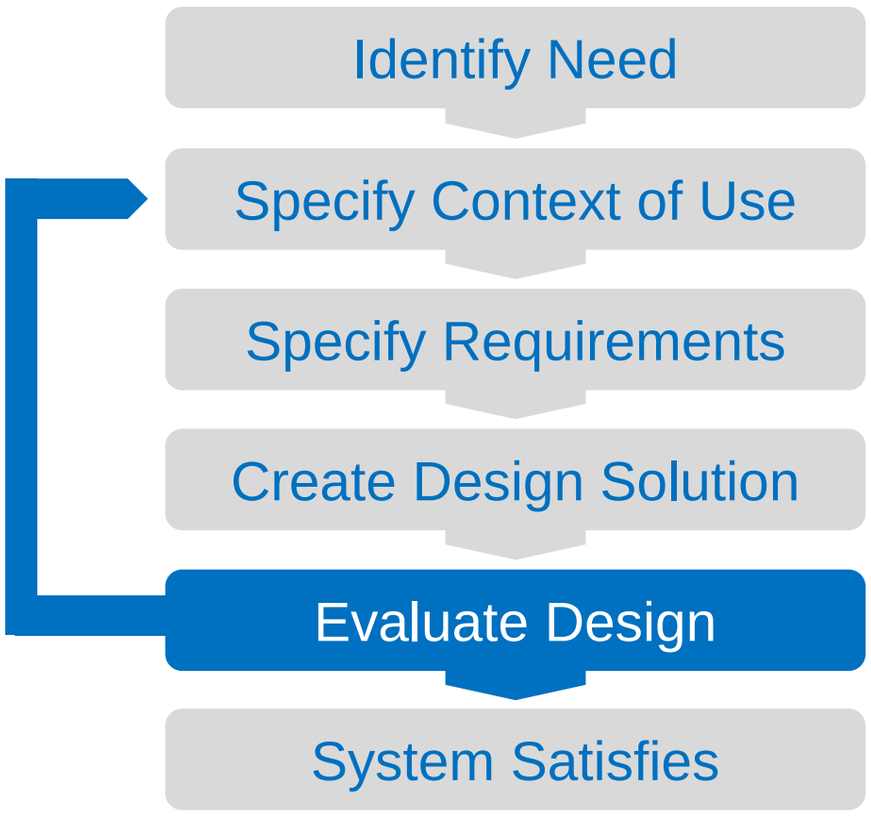


- **Create Design Solution**
  - building from a rough concept to a complete design

User-Centered Design Process

Iterative Design

- to develop a design through repeated cycles (iterative) and in smaller portions at a time (incremental)



- Evaluate Design Solution
  - ideally through usability testing with actual users

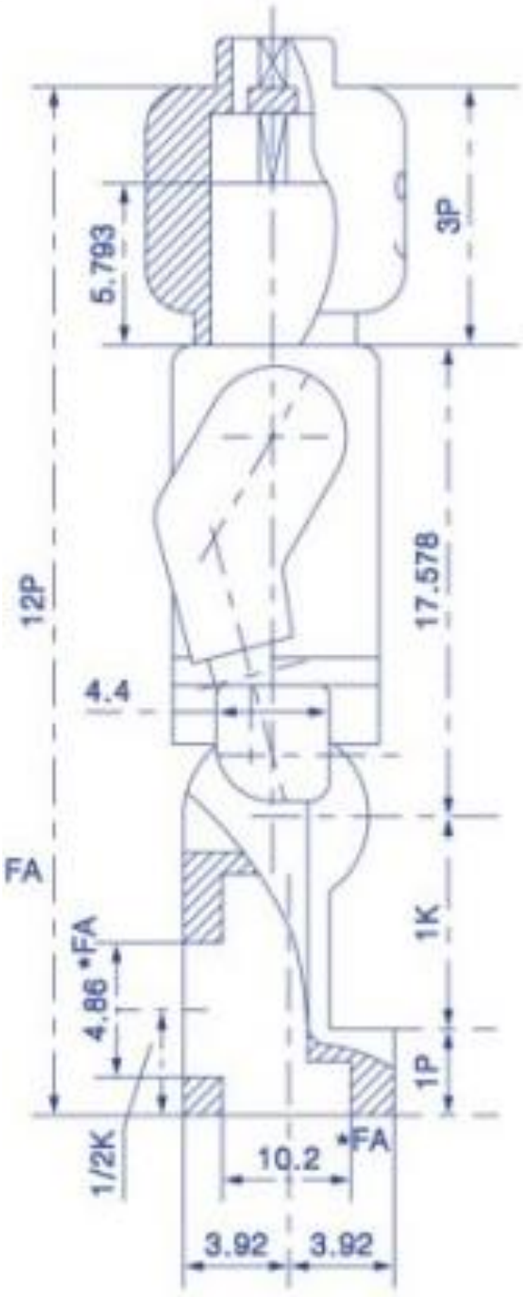
Specify Context of Use

Personas

Specify Context of Use: Personas

- ## Purpose of Personas
  - to create reliable and realistic representations of your key audience segments for reference

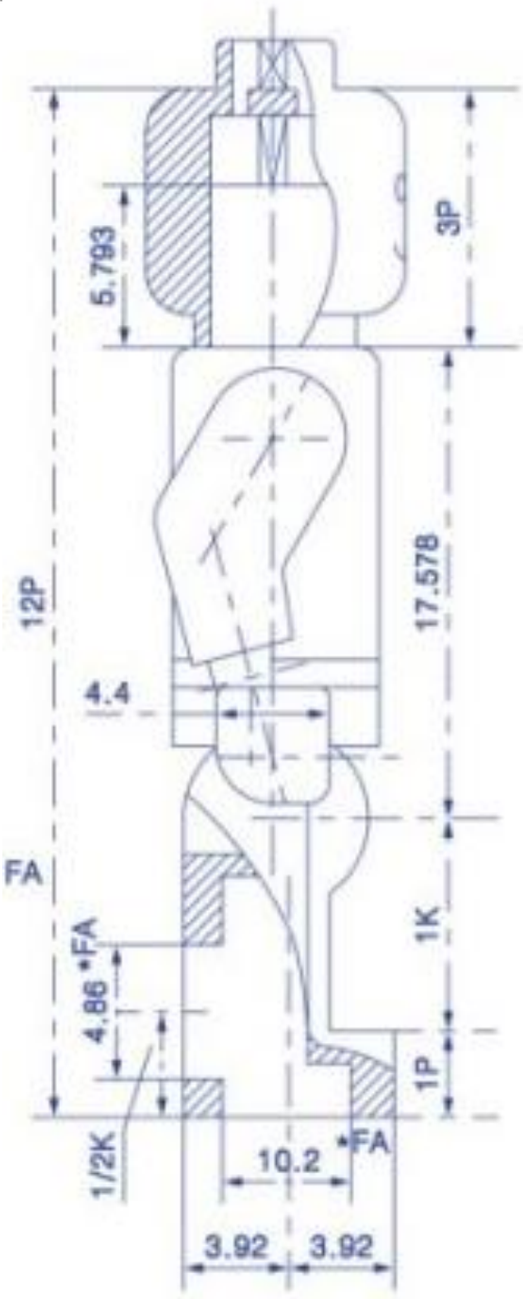
- ## Effective Personas
  - represent a major user group for the software application
  - express and focus on the major needs and expectations of the most important user groups
  - give a clear picture of the user's expectations and how they're likely to use the software application
  - aid in uncovering universal features and functionality
  - describe real people with backgrounds, goals, and values



Specify Context of Use: Personas

Questions to Ask During Persona Development:

- Define the Purpose
  - What is the purpose / goal of your application?



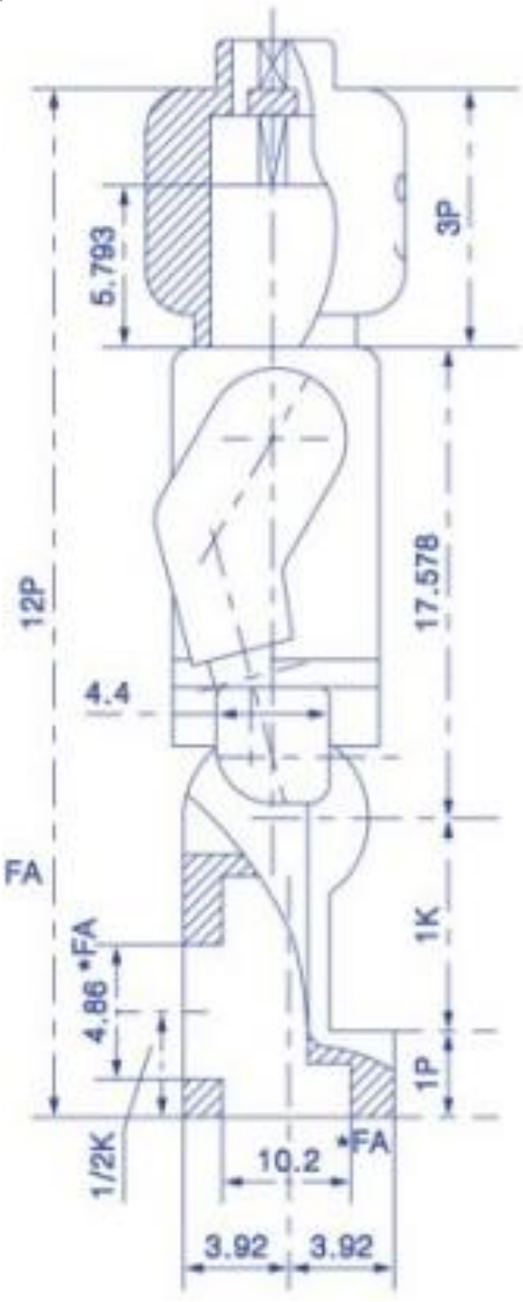
- Describe the User (personal)
  - What is the age of your person?
  - What is the gender of your person?
  - What is the highest level of education this person has received?

Specify Context of Use: Personas

Questions to Ask During Persona Development:

- Describe the User (professional)

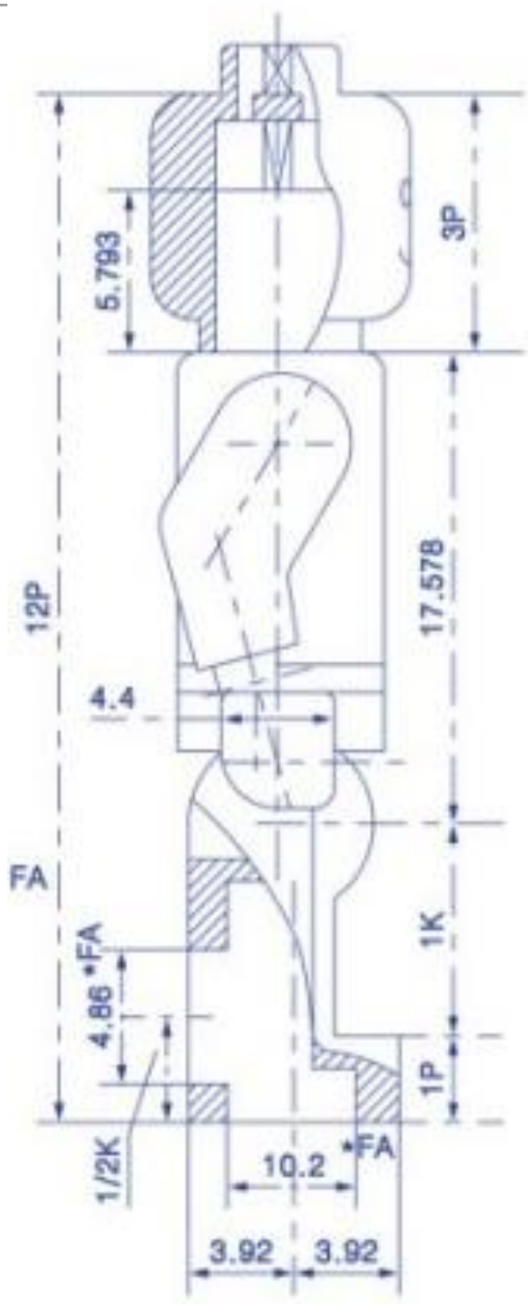
- How much work experience does your person have?
- What is your person’s professional background?
- Why will they use your application?
  - user needs, interests, and goals
- When and where will they use your application?
  - user environment and context



Specify Context of Use: Personas

Questions to Ask During Persona Development:

- Describe the User (technical)
  - What technological devices does your person use on a regular basis?
  - What applications does your person use on a regular basis?
- Describe User Motivation
  - What is your person motivated by?
  - What are they looking for?
  - What is your person looking to do?
  - What are his needs?



Specify Context of Use: Personas

Elements of a Persona:

Persona
- persona group (i.e. student, international student…)

Photo

Fictional name
- name of the persona

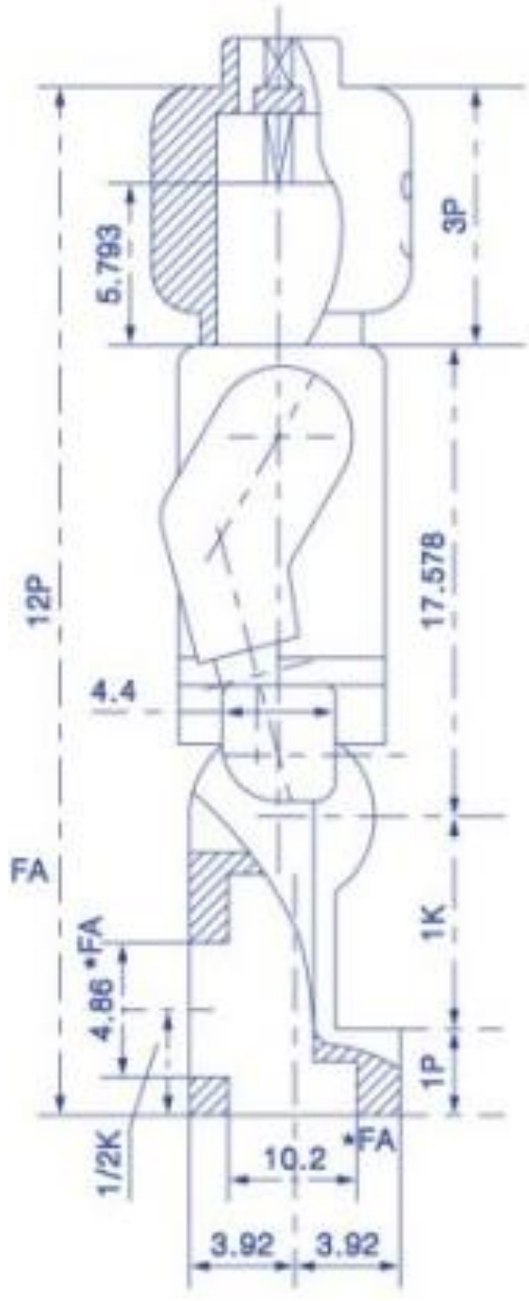
Major responsibilities
- such as job titles

Demographics
- such as age, education, ethnicity, and family status

Goals and tasks
- the persona is trying to complete using the application

Environment
- physical, social, and technological environment

Quote
- to sum up what matters most to the persona



OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FACULTY OF COMPUTER SCIENCE

SwarmLab

Introduction to Software Engineering for Engineers
Lecture-08: Implementation & UI Design
Part 4: UI Design – Principles and Components

Dr.-Ing. Christoph Steup

Design Principles

Dieter Rams' principles for good design (Part 1):

1. **Innovative** – The possibilities for progression are not, by any means, exhausted. Technological development is always offering new opportunities for original designs. But imaginative design always develops in tandem with improving technology, and can never be an end in itself.

2. **Useful** – A product is bought to be used. It has to satisfy not only functional, but also psychological and aesthetic criteria. Good design emphasizes the usefulness of a product whilst disregarding anything that could detract from it.

3. **Aesthetic** – The aesthetic quality of a product is integral to its usefulness because products are used every day and have an effect on people and their well-being. Only well-executed objects can be beautiful.

4. **Understandable** – It clarifies the product’s structure. Better still, it can make the product clearly express its function by making use of the user's intuition. At best, it is self-explanatory.

5. **Unobtrusive** – Products fulfilling a purpose are like tools. They are neither decorative objects nor works of art. Their design should therefore be both neutral and restrained, to leave room for the user's self-expression.

Design Principles

Dieter Rams' principles for good design (Part 2):

6. **Honest** – It does not make a product appear more innovative, powerful or valuable than it really is. It does not attempt to manipulate the consumer with promises that cannot be kept.

7. **Long-lasting** – It avoids being fashionable and therefore never appears antiquated. Unlike fashionable design, it lasts many years – even in today's throwaway society.

8. **Detailed** – Nothing must be arbitrary or left to chance. Care and accuracy in the design process show respect towards the consumer.

9. **Environmentally friendly** – Design makes an important contribution to the preservation of the environment. It conserves resources and minimizes physical and visual pollution throughout the lifecycle of the product.

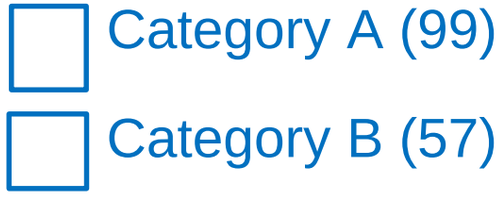
10. **Minimal** – Less is more. Simple as possible but not simpler. Good design elevates the essential functions of a product.

User Interface Elements

Input Controls

Checkboxes

- allow the user to select one or more options from a set



Radio buttons

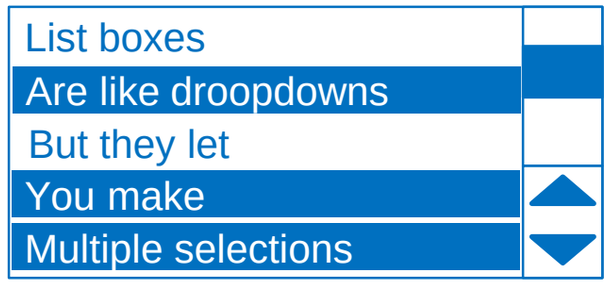
- allow the user to select one item at a time

Dropdown lists

- allow the user to select one item at a time
- more compact as radio buttons

List boxes

- allow the user to select multiple items at a time
- can support a longer list of options if needed



Buttons

- indicates an action upon touch
- typically labeled using text, an icon, or both

User Interface Elements

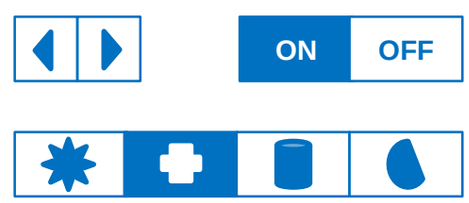
Input Controls

Dropdown Button

- consists of a button that when clicked displays a drop–down list of mutually exclusive items

Toggles

- allow the user to change a setting between two states
- most effective when the on/off states are visually distinct

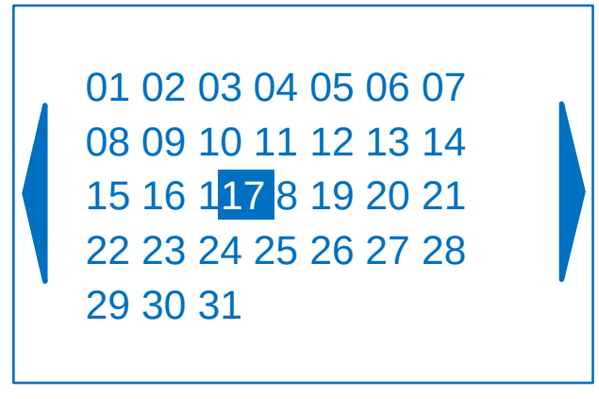


Text fields

- allow the user to enter text
- either a single line or multiple lines of text

Date pickers

- allow users to select a date and/or time



User Interface Elements

Navigational Components

Search Field

- allow user to enter a keyword or phrase (query) and submit it to search the index with the intention of getting back the most relevant results

Breadcrumb

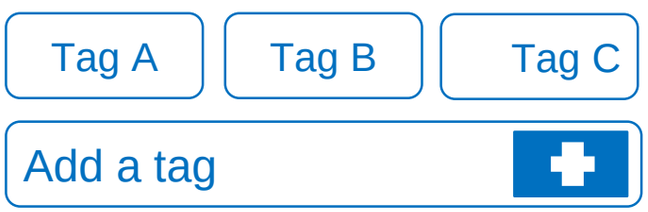
- allow users to identify their current location within the system by providing a clickable trail of proceeding pages to navigate by

Pagination

- divides content up between pages, and allows users to skip between pages or go in order through the content

Tags

- allow users to find content in the same category

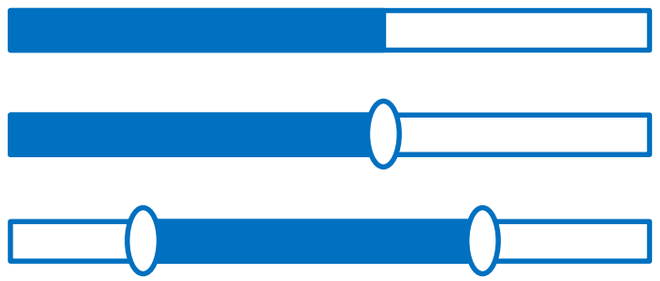


User Interface Elements

Navigational Components

Sliders

- allow users to set or adjust a value



Icons

- is a simplified image serving as an intuitive symbol that is used to help users to navigate the system

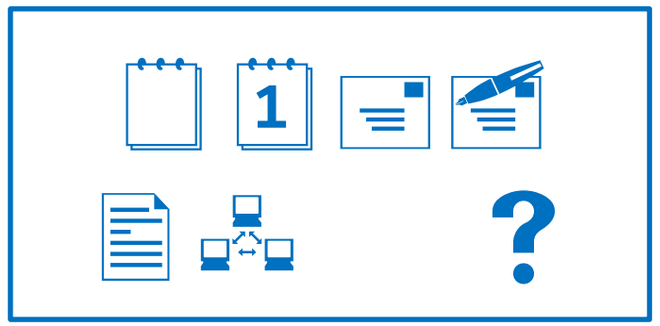
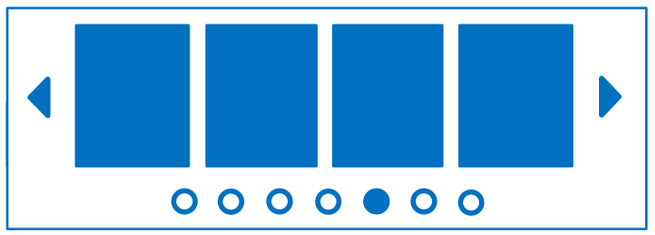


Image Carousel

- allow users to browse through a set of items and make a selection of one if they so choose



User Interface Elements

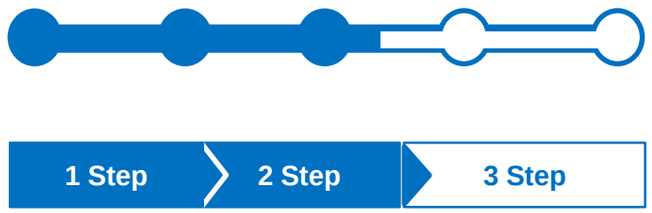
Information Components

Notifications

- an update message that announces something new for the user to see

Progress bars

- indicates where a user is as they advance through a series of steps in a process

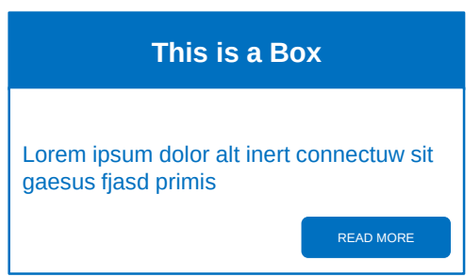


Tool Tips

- allow the user to see hints when they hover over an item indicating the name or purpose of the item

Message boxes

- a small window that provides information to users and requires them to take an action before they can move forward



Usability Principles

Nielson’s principles (Part 1):

1. **Visibility of system status:** The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

2. **Match between system and the real world:** The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

3. **User control and freedom:** Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

4. **Consistency and standards:** Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

5. **Error prevention:** Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

https://www.nngroup.com/articles/ten-usability-heuristics/

Usability Principles

Nielson’s principles (Part 2):

6. **Recognition rather than recall:** Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

7. **Flexibility and efficiency of use:** Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. **Aesthetic and minimalist design:** Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

9. **Help users recognize, diagnose, and recover from errors:** Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

10. **Help and documentation:** Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Colors

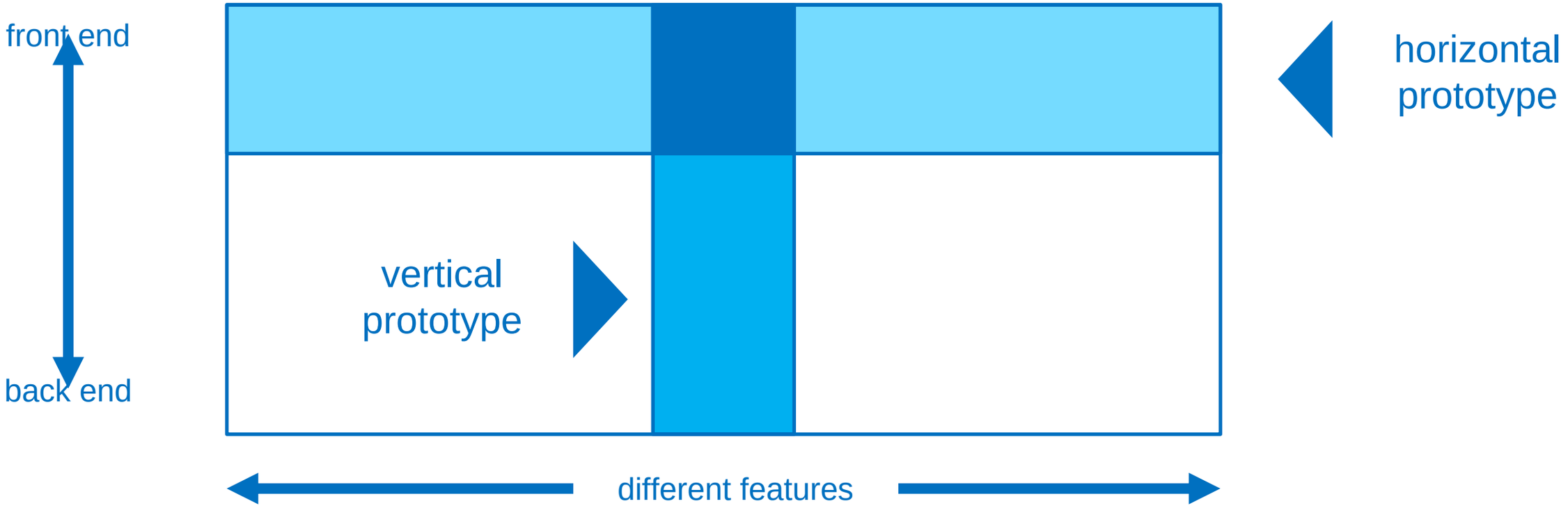
- Color Use Guidelines
  - don't use too many colors
  - use color coding to support use tasks
  - allow users to control color coding
  - design for monochrome then add color
  - use color coding consistently
  - avoid color pairings which clash
  - use color change to show status change
  - Keep color perception impaired people in mind

Design Evaluation: Prototyping

Prototyping:

- to use a draft version of a product to explore your ideas

- can be anything from paper drawings (low-fidelity) to something that allows click-through of a few pieces of content to a fully functioning site (high-fidelity)



Design Evaluation: Prototyping

Paper Prototyping

- interactive paper mockup
  - sketches of screen appearance
  - paper pieces show windows, menus, dialog boxes
- interaction is natural
  - pointing with a finger = mouse click
  - writing = typing
- a person simulates the computer’s operation
  - putting down & picking up pieces
  - writing responses on the “screen”
  - describing effects that are hard to show on paper

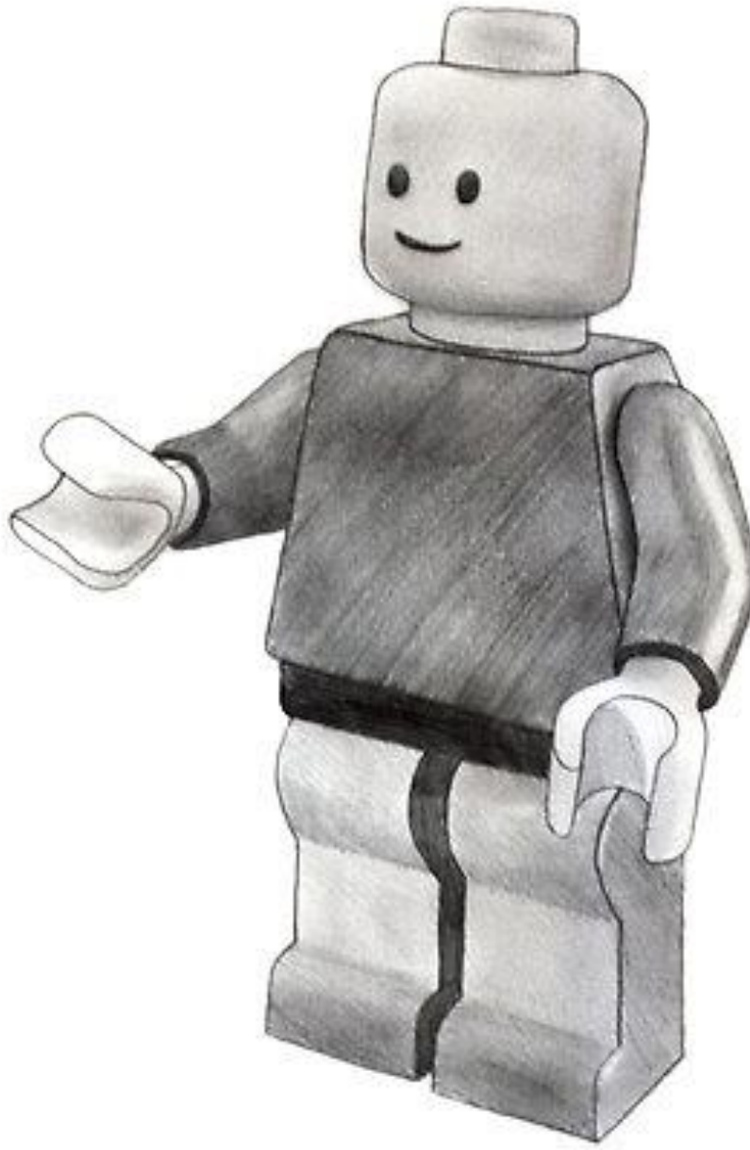
Design Evaluation: Prototyping

Paper Prototyping

Design Evaluation: Prototyping

Benefits of Paper Prototyping:

- faster to build
  - sketching is faster than programming
  - paper pieces show windows, menus, dialog boxes
- easier to change
  - easy to make changes between user tests, or even during a user test
  - no code investment– everything will be thrown away (except the design)
- focuses attention on big picture
  - designer doesn't waste time on details
  - customer makes more creative suggestions



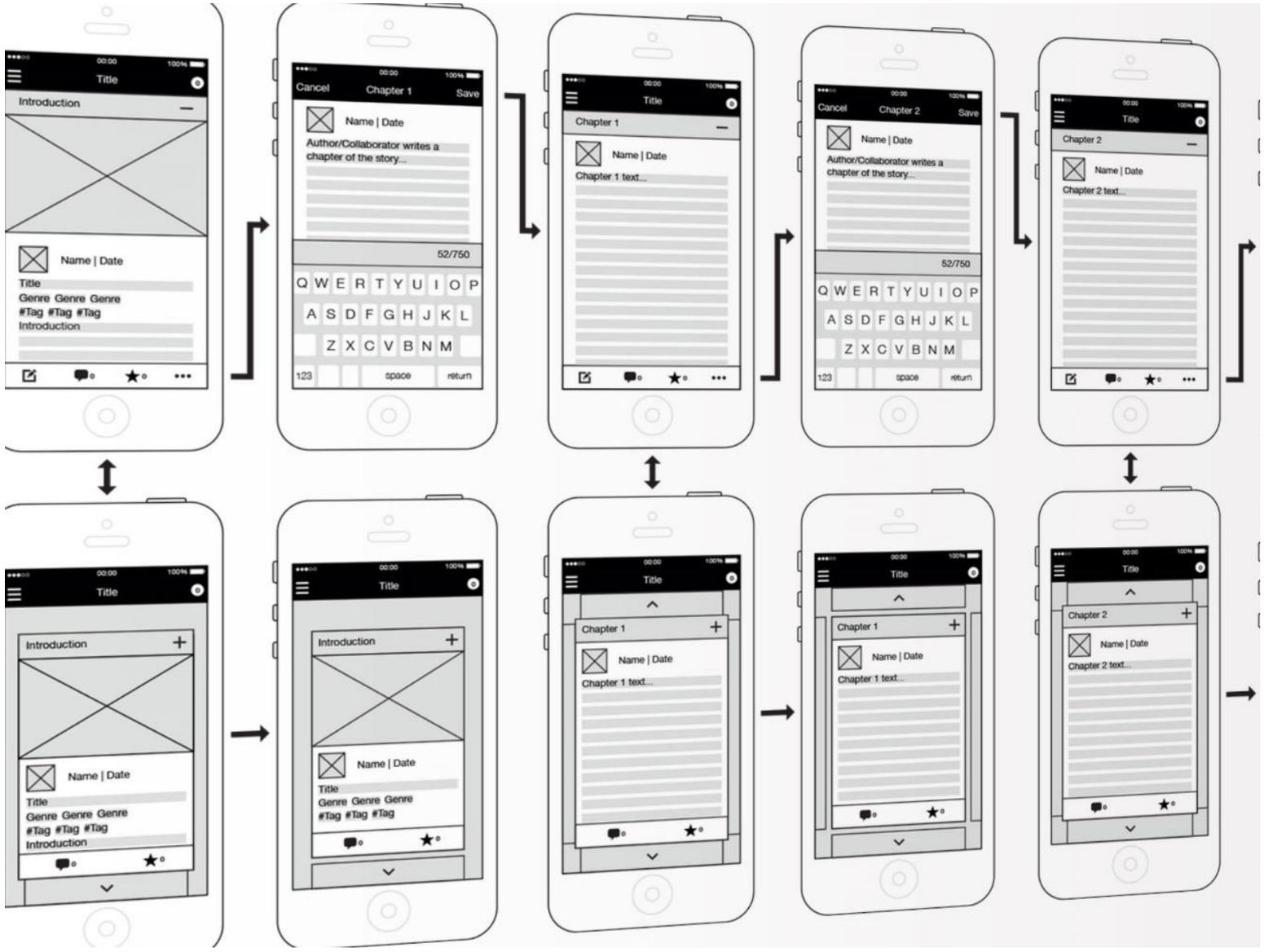
Design Evaluation: Prototyping

Benefits of Paper Prototyping

- Can be used to evaluate
  - conceptual model
    - the user's internal model of what the system provides
  - functionality
    - Does it do what's needed ? Missing features ?
  - navigation & task flow
    - Can users find their way around ?
    - Are information preconditions met ?
  - terminology
    - Do users understand labels ?
  - screen contents
    - What needs to go on the screen ?

Design Evaluation: Prototyping

Wireframe / Mockups



Summary

Summary - Implementation

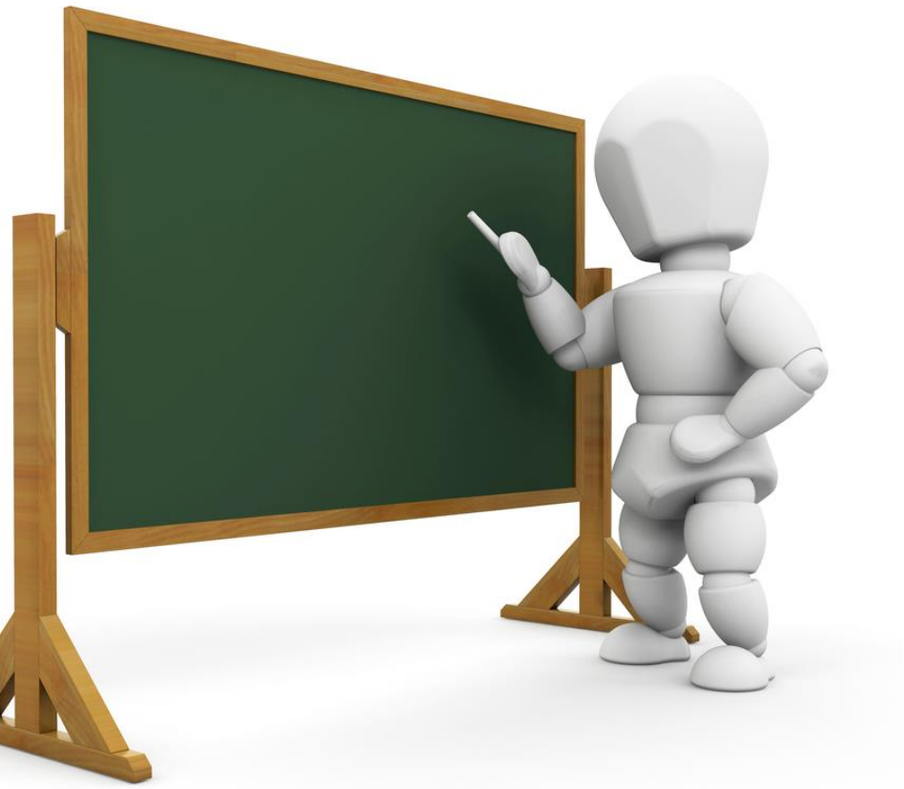
- Good code is no magic!

- But good code is important regarding:

  ✓ Understandability

  ✓ Maintainability

  ✓ Efficiency



- There is not the ultimate programming language

- Tools can simplify the implementation

  ➢ but they do not automatically realize good code!

Summary — User Interface Design

- Specify Context of Use
  - Personas
- Create Design Solution
  - Design Principles
  - User Interface Elements
  - Usability Principles
  - Colour Theory
- Evaluate Design Solution
  - Prototyping