andrena OBJECTS

Experts in agile software engineering

# Readable Code

## Wie funktioniert Code-Verstehen?

andrena objects

Stefan Mandel

stefan.mandel@andrena.de

07.07.2022

Java Forum Stuttgart 2022

# Wer sind wir?

## Stefan Mandel

- Diplom-Informatiker
  - Compiler
  - Software-Technik
  - Biologie
- Seit 2012 bei andrena
- Full-Stack-Entwickler
  - Refactoring
  - Clean Code

# Wer hat diesen Code geschrieben?

```
public List<Item> computeEntriesUptodateAndVisibleSortingBy(
    List<RawItem> rawEntries,
    Instant now,
    boolean v,
    String field,
    boolean order) {

    return rawEntries.stream()
    //...
    .collect(toList());
}
```



# Verständlichkeit und Lesbarkeit von Code!

- Es gibt zahlreiche Merkmale eines schlecht verständlichen Codes → „Code Smells“

- Es gibt zahlreiche Faustregeln für gut verständlichen Code
  - sprechende Namen
  - Methoden mit maximal 2-3 Argumenten
  - kurze Methoden
  - Klassen mit maximal 4 Dependencies
  - Kleine Vererbungshierarchien

- Ist das jetzt eine Glaubenssache?

# Verständlichkeit ist messbar!

Gehirn wird anders bedient als es bedient werden will

erzeugt hohe Last beim Verstehen

Code ist schlecht verständlich

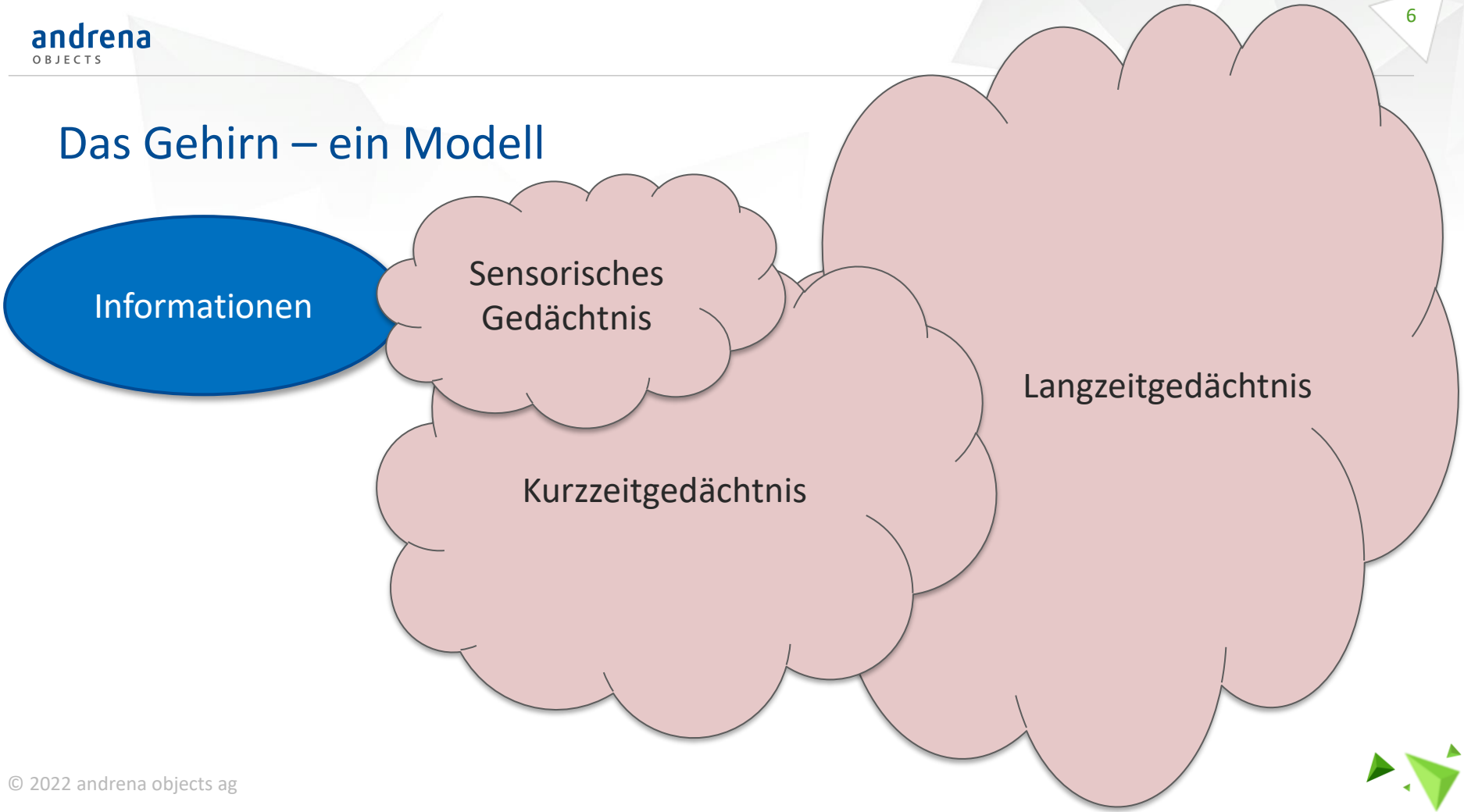
Wenn wir wissen wie es bedient werden will …

können wir Metriken für die Last finden …

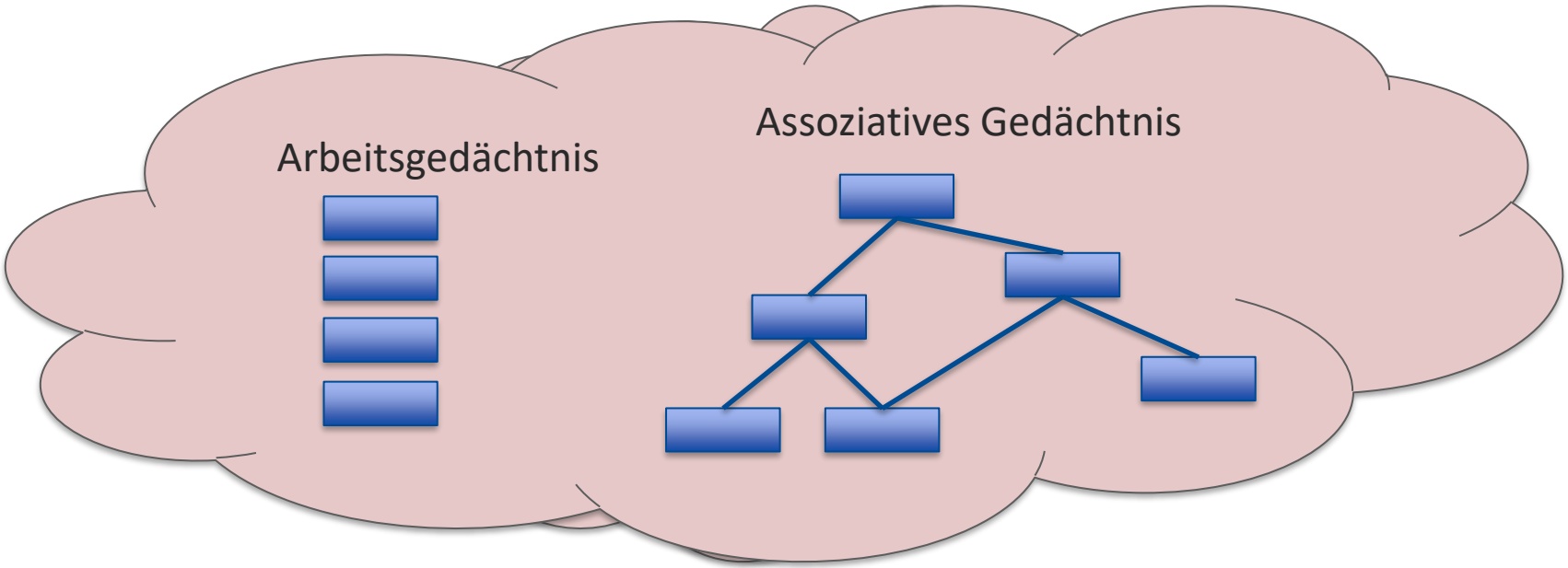
um die Verständlichkeit zu bewerten/messen



# Das Gehirn – ein Modell



# Das Kurzzeitgedächtnis – ein Modell



# Verständnisprozess - Herausforderungen

- Das **Arbeitsgedächtnis (Working Memory)**
  - ist zuständig für die Operationen
  - ist limitiert bezüglich der Anzahl Operanden

- Das **Assoziative Gedächtnis (**im weiteren Kurzzeitgedächtnis)
  - speichert Informationen als „Graph“
  - ist limitiert in der Kapazität (Cache des Langzeitgedächtnisses)

# Verständnisprozess - Optimierungen

- Optimierungen des **Arbeitsgedächtnisses**
  - das Operanden-Limit einhalten

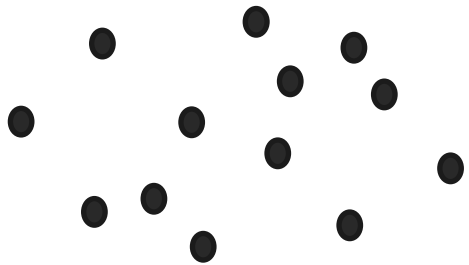
- Optimierung des **Assoziative Gedächtnisses**
  - möglichst kompakte, nichtredundante Informationen speichern
  - möglichst eindeutige Wege im Graph speichern
    - Lesezugriffe sollten genau eine Information auffinden
    - Schreibzugriff sollten keine Informationen überschreiben

# Optimierung des Arbeitsgedächtnisses

- Das Arbeitsgedächtnis ist limitiert bezüglich der Operanden *(George A. Miller, 1956)*
- Experiment zum Mitmachen: Ab wann könnt ihr die Anzahl der Punkte erfassen?

# Optimierung des Arbeitsgedächtnisses

- Das Arbeitsgedächtnis ist limitiert bezüglich der Operanden *(George A. Miller, 1956)*
- Ab wann könnt ihr die Anzahl der Punkte erfassen?

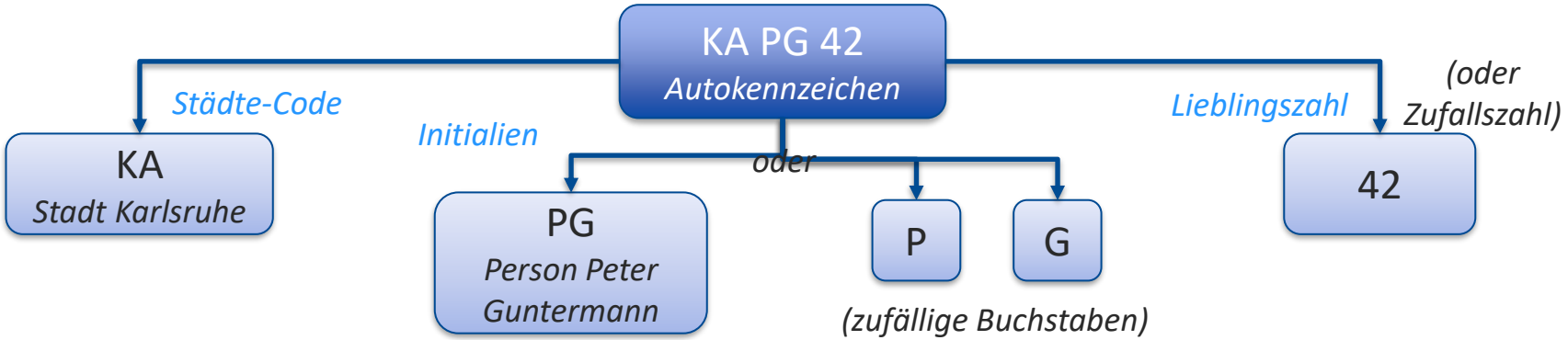


Übliche Werte sind hier 7

# Operanden und Chunks

- Im Arbeitsgedächtnis hat also ein Operanden-Limit
  - 7±2 nach *George A. Miller*
  - Jeder Operand kann ein sog. **Chunk** sein
- Ein **Chunk** ist eine Information, der eine Bedeutung zugemessen wird
  - Es gibt keine genaue Limitierung bezüglich Chunkgröße

# Operanden und Chunks - Beispiel



# Millers Number – Magic 7


- Wikipedia erwähnt z.B.:
  - Wenn mehr als 7 Zie… …t der Überblick verloren.
  - Bei Scrum, einem Vorgeh… …twareentwicklung wurde 7 ± 2 als ideale Größe für ein Tea… …en höherer Flexibilität wurde diese allerdings auf 6 ± 3 ange…
  - In der Programmier… …r als 7 Parameter haben.
  - Bei der objektorient… …klung sol… …e nicht mehr Attribute haben, als in das Kurzz… …achtnis des Entw… …passen. […]
  - …
- Eine Folie soll max. 7 Spiegelstriche haben

# Das wirkliche Limit ist kleiner – nämlich 4

- Miller hatte vor allem ein Limit vermutet
  - die 7 war ein Beispiel
    - und wurde hinterher nur wenig hinterfragt
  - die 7 funktioniert tatsächlich mit einigen Gedächtnisaufgaben
    - mehr dazu später
- Spätere Experimente (Cowan, Parker) bestätigten eher:
  - unser **Arbeitsgedächtnis** hat eine Kapazität von **4 Chunks** (±1)

# Chunks - Ein kleines Experiment …

- Merkt euch bitte die folgenden Zahlen/Buchstaben:

| | |
|---|---|
| 3378981510 | 10 Chunks |
| 0151 89 87 33 | 4 Chunks |
| GNUHCSARREBÜ | 12 Chunks |
| ÜBERRASCHUNG | 1 Chunk |

# Für verständlichen Code …

- Benutze nie mehr als 4
  - Parameter
  - Dependencies
  - Zeilen
  - ….

- Zu verkürzt!
  - Wir dürfen die Probleme nie isoliert sehen
  - Manchmal benutzen wir Operanden, derer wir uns nicht direkt bewusst sind

# Naming

```
public interface FromSpringData {

  Entry findByIdAndLang(long id, String language);

  Entry findByIdNotAndLangAndCollectionAndCategoryInAndDateBefore(
    long id,
    String language,
    String collection,
    List<String> categories,
    long page,
    long size);
}
```

3 Chunks

10 Chunks

Namen kurz halten



# Klassen und Attribute

```
public record Product1(
    int id,
    String title,
    String description,
    BigDecimal price,
    int available,
    List<String> comments,
    List<Integer> stars) {
}
```

7 Chunks

```
public record Product2(
    int id,
    Label label,
    Supply supply,
    UserFeedback userFeedback) {
}
```

4 Chunks
(durch Abstraktion)

Also maximal 3-4 Attribute für Structs/Records
(Klassen belegen einen zusätzlichen Operanden mit dem Kontrakt)



# Ausdrücke

```
System.out.println(a);

System.out.println(a & b);

System.out.println(a & !b);
System.out.println(!a | b);

System.out.println(a | b & !c);
```

1 Chunk

3 Chunks

4 Chunks

4 Chunks

Demo

Binäre Ausdrücke können wir gerade noch verstehen, alles andere ist schwierig

# Optimierung über Operanden hinaus

Was sind typische Chunks im Quellcode?

- Zeilen
- aber auch Ausdrücke, Variablen, Zahlen
- und Schlüsselwörter, Buchstaben, Formatierungen
- ….

Aber das sind doch zu viele …

- auf jeden Fall mehr als 4!
- Wie soll unser Gehirn damit zurechtkommen?

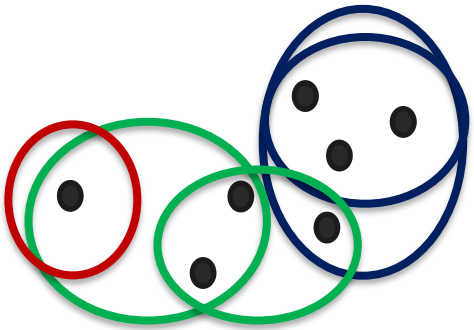
# Umgang mit mehr als 4 Operanden

Die Fülle aller Informationen wird reduziert durch:

- **Verdrängung**
  - das Gehirn bewertet die Chunks
  - und ignoriert die unwichtigen
- **Chunking**
  - das Gehirn kombiniert kleine Chunks zu größeren
  - die dann gemeinsam einen Operanden belegen

# Chunking – geht fast automatisch

- Manche Regelmäßigkeiten chunkt das Gehirn selbstständig
  - z.B. das Punkt-Experiment
  - oft ist diese Fähigkeit nie aktiv erlernt worden



# Chunking – kann man lernen

wiederkehrende Zusammenhänge bekommen eine Bedeutung, z.B.

- im Alltag
  - Wörter
  - Eselsbrücken
  - Verkehrsschilder

→ Bremsen: Fuß aufs Pedal bewegen
→ Lenken: Hände bewegen, rechts/links
→ Physik: Geschwindigkeit, Trägheit
→ …

- in der Mathematik
  - Definitionen

„Ein **Baum** ist ein zusammenhängender, azyklischer, ungerichteter Graph.“

- in der Informatik
  - Abstraktionen (Klassen, Methoden, …)
  - Design Patterns

# Chunking – im Kurzzeitgedächtnis

Chunks im Kurzzeitgedächtnis sollten sein:

- möglichst **präzise erreichbar**
  - dadurch Vermeidung von fehlerhaften Vorstellungen
  - dadurch Vermeidung von falschen Einprägungen
- möglichst **abstrakt**
  - dadurch geringer Speicherverbrauch

# Chunking – Beispiel – Sprechende Namen

Sprechende Namen helfen bei der Organisation im Kurzzeitgedächtnis (KZG)

- Sie sind **präzise**
  - jeder Lesezugriff erreicht die korrekte Information
  - jeder Schreibzugriff überschreibt nichts anderes
- Sie sind **kompakt**
  - umfassen genau das, was enthalten ist und nicht jedes Detail
  - schonen die Kapazität des KZG

# Automatische Vereinfachung

```
public class AutoChunking {

    public void autoChunking(String[] parameters) {
        System.out.println(parameters[0]);
        System.out.println(parameters[1]);
        System.out.println(parameters[2]);
        System.out.println(parameters[3]);
        System.out.println(parameters[4]);
        System.out.println(parameters[5]);
        System.out.println(parameters[6]);
        System.out.println(parameters[7]);
    }
}
```

8 Chunks?
Unverständlich?

Nein!
3 Chunks:

- `System.out.println`
- `parameters[i]`
- 8

# Aufruf reiner Funktionen

```
int result = this.add(1, 2);
```

3 Chunks

```
public class Scaler {

    private int factor;

    private int add(int a, int b) {
        return a + b;
    }

}
```



# Aufruf einfacher Methoden

```
int result = scaler.addUnits(1, 2);
```

4 Chunks

```
public class Scaler {

    private int factor;

    public int addUnits(int a, int b) {
        return add(a, b) * this.factor;
    }
}
```

2-3 Argumente
entsprechen 4 Operanden
(je nach Funktionstyp)

# Aufruf mutierender Methoden

```
scaler.setFactor(3); // scaler changed
```

4 Chunks

```
public class Scaler {

    private int factor;

    public void setFactor(int factor) {
        this.factor = factor;
    }
}
```

Mutierende Funktionen
belegen mehr Operanden



# Aufruf unkonventioneller Methoden

```
public class Scaler {

    private int factor;

    public Scaler plus(Scaler that) {
        that.factor += this.factor;
        return that;
    }
```

```
public static void main(String[] args) {
    final Scaler x = new Scaler(1);
    final Scaler y = new Scaler(2);
    final Scaler z = new Scaler(4);

    Scaler sumXY = x.plus(y);
    Scaler sumXYZ = z.plus(y);//change
}
```

Demo

5 Chunks
(und de

Nicht gegen die Konventionen verstoßen

# Code-Konventionen

- Code Konventionen helfen dem Gehirn
  - indem sie **Unruhe eliminieren**
  - alles, was ruhig ist, bewertet das Gehirn als nebensächlich
  - Folge: das Gehirn hat eine bessere Quote beim Verdrängen der unwichtigen Information

- Störinformationen sind z.B.
  - Unpassende Namen
  - Unkonventionelle Scopes (Klammerungen)
  - Reihenfolgen von Schlüsselworten
  - Individuelle Teamkonventionen



# Gruppierung/Alignment von Statements

```
public class MyObject {

    private int value;

    public MyObject mult(MyObject that) {
        int value1 = this.value();
        int value2 = that.value();

        int product = value1 * value2;

        return new MyObject(product);
    }
}
```

- Gleichartiges gruppieren
- Wichtiges freistellen
- Neues absetzen

Jede Gruppe sollte selbst wieder gut verständlich sein (z.B. 4 Zeilen)

# Nochmal zusammengefasst

- **Verständlichkeit** entsteht 
  - indem wir unseren Code/Text auf unseren Denkprozess anpassen
  - nie mehr als **4** wichtige Informationen (**Chunks**) gleichzeitig verarbeiten
  - komplexere Informationen als **zusammengesetzte Chunks** kapseln

- Das Gehirn kann nur **ideal performen** 
  - wenn es die Informationen auch **ideal präsentiert** bekommt

# Lasst uns mal beim Programmieren darauf achten!

- Früher oder später werden wir auf Code stoßen
  - oder selbst schreiben 😉
  - den wir intuitiv als **unverständlich** ansehen

- Ab jetzt haben wir ein **neues Werkzeug** zur Bewertung von **Verständlichkeit**:
  - Wie viele Chunks muss mein Gehirn beim Lesen verarbeiten?
  - Werden vielleicht wichtige Informationen aus Versehen verdrängt?
  - Kann ich manche Informationen zu neuen Chunks zusammenfassen?
  - Kann ich eine irrelevante Information weglassen?

# Vielen Dank!

Github-Repository:

https://github.com/andrena/readable-code

Fragen und Anmerkungen an:

- stefan.mandel@andrena.de
- peter.guntermann@andrena.de