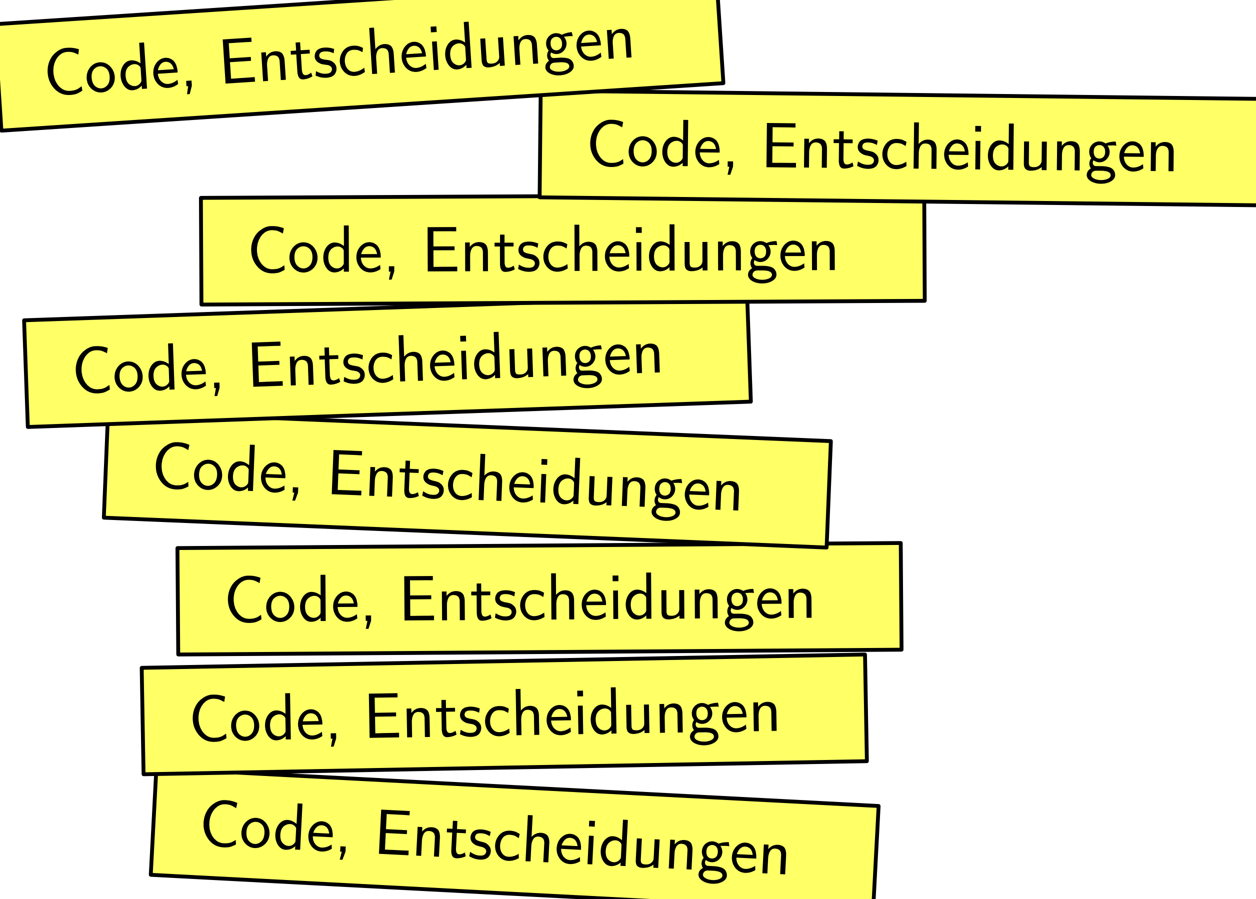
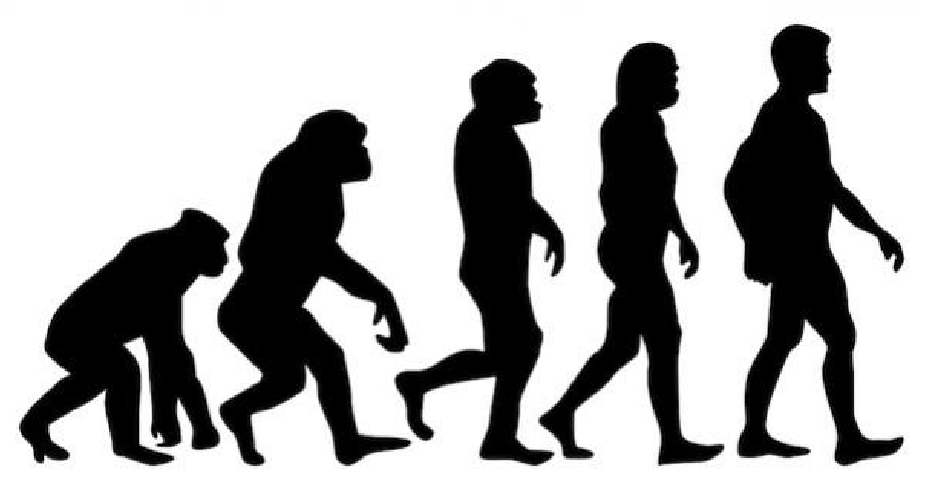
Evolution von Entwurfsentscheidungen in der Praxis

Sphenon GmbH, Hamburg
Andreas Leue
www.leue.net

Software-Entwicklung





Evolution eines Stücks Code

Story

Warenkorb

Artikel	Anzahl	Verfügbar
DVD, Star Wars	1	prüfen
iPhone	2	prüfen
Eier	6	prüfen

Jetzt bestellen Abbrechen

Bei Click soll die Verfügbarkeit aktualisiert werden

Quick & Dirty

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    verfuegbar = queryDatabase("select anzahl from artikel where name = :currentArtikel");
    if (verfuegbar >= bestellt) {
      verfuegbarFeld.setText("ok").setColor(GRUEN);
    } else {
      verfuegbarFeld.setText("ausverkauft").setColor(ROT);
    }
  }
}
```

Bunt gemischter Code

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    verfuegbar = queryDatabase("select anzahl from artikel where name = :currentArtikel");
    if (verfuegbar >= bestellt) {
      verfuegbarFeld.setText("ok").setColor(GRUEN);
    } else {
      verfuegbarFeld.setText("ausverkauft").setColor(ROT);
    }
  }
}
```

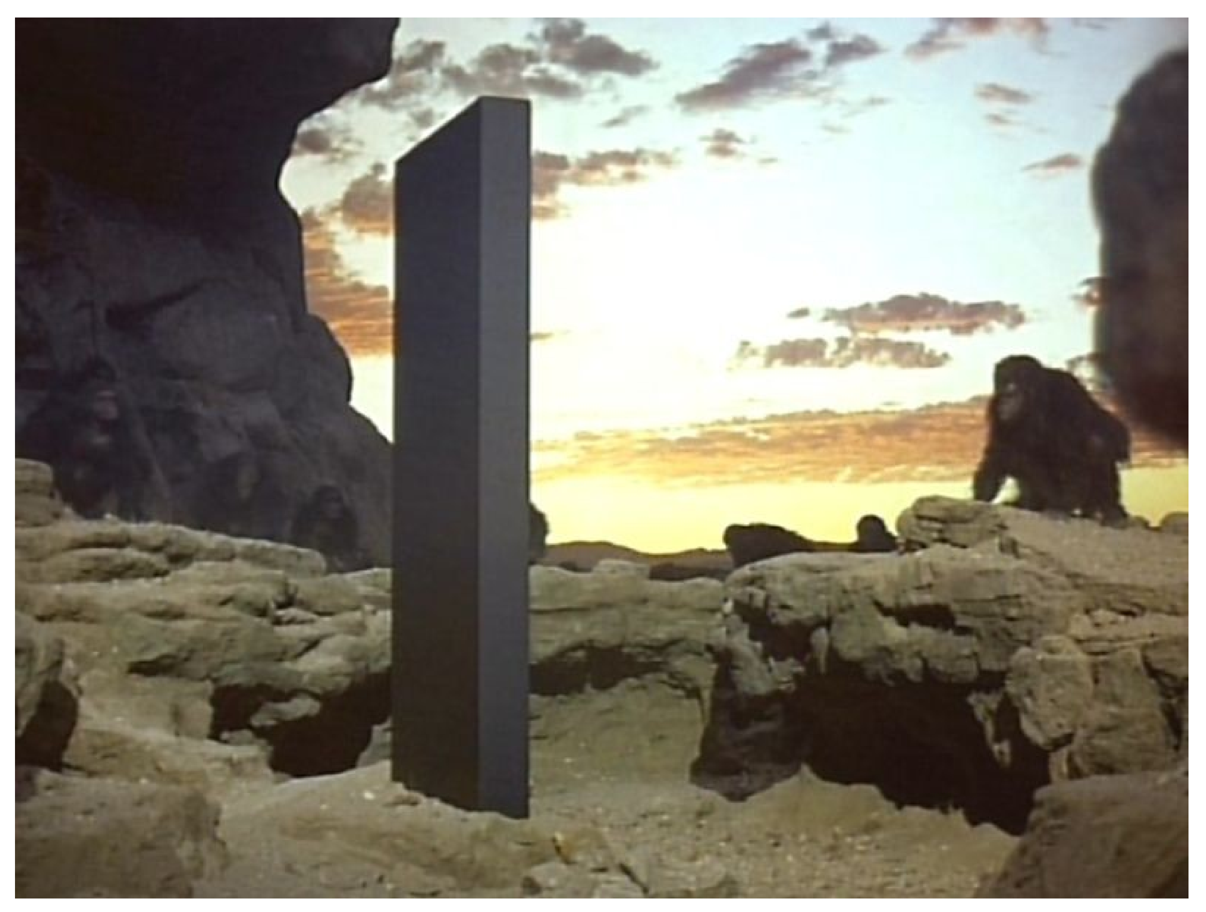
Stets gleicher Glue-Code

User Interface Code

Datenbank-Code

Geschäftslogik

MONO
Der Monolith



Eine Schicht

Sinlge Tier

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    verfuegbar = queryDatabase("select anzahl from artikel where name = :currentArtikel");
    if (verfuegbar >= bestellt) {
      verfuegbarFeld.setText("ok").setColor(GRUEN);
    } else {
      verfuegbarFeld.setText("ausverkauft").setColor(ROT);
    }
  }
}
```

MONO
Der Monolith

Es kommen Stories hinzu, nicht nur eine, sondern hunderte, oder mehr.

Im Laufe der Zeit entstehen Probleme:

- Mit der Zeit vermengen sich die Themen immer mehr (z.B. Main-Loop)
- Der Code wird immer unübersichtlicher
- Dinge werden zwar implizit festgelegt (“zementiert”), aber nicht als explizite Entscheidungen; sie bleiben undokumentiert

MONO
Der Monolith

Das wäre eine Architekturentscheidung

Je länger das Projekt läuft, desto schwieriger werden Änderungen.

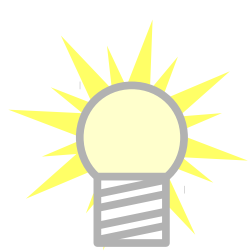
Daher wird (für das nächste Projekt) folgende grundlegende Entscheidung gefällt:

> Alle Anforderungen müssen zu Beginn bekannt sein!
>
> Sie müssen dokumentiert werden!

Das sind dann auch alles Architekturentscheidungen

MONO
Der Monolith

Natürlich hält sich niemand an diese Regel!
(schon gar nicht der Kunde)



So geht es nicht weiter! --> Code muß refactored werden

Wir teilen den Code nach Themen auf.

Wir trennen Themen

(bessere Übersicht, weniger Impact)

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

Immer der gleiche Code (Main-Loop)

```
void callback() {
  if (isLieferbar(currentArtikel, bestellt)) {
    verfuegbarFeld.setText("ok").setColor(GRUEN);
  } else {
    verfuegbarFeld.setText("ausverkauft").setColor(ROT);
  }
}
```

User Interface

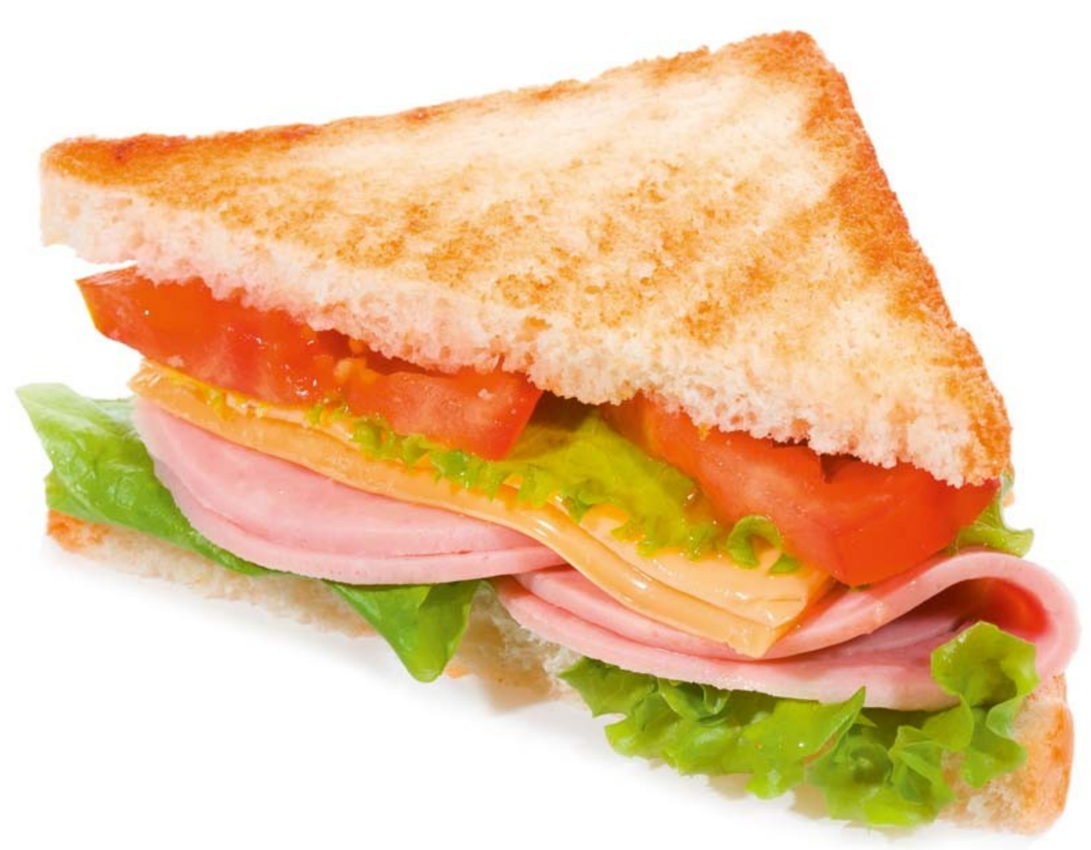
Geschäftslogik

```
boolean isLieferbar(artikel, bestellt) {
   return getVerfuegbar(artikel) >= bestellt);
}
```

Datenbank

```
int getVerfuegbar(name) {
   return queryDatabase("select anzahl from artikel where name = :name");
}
```

3 Tiers / MVC
Model View Controller



3 Tiers / MVC

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

Standard

Controller

zählt zum UI

```
void callback() {
  if (isLieferbar(currentArtikel, bestellt)) {
    verfuegbarFeld.setText("ok").setColor(GRUEN);
  } else {
    verfuegbarFeld.setText("ausverkauft").setColor(ROT);
  }
}
```

View

UI

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) >= bestellt);
}
```

Model

BL

```
int getVerfuegbar(name) {
  return queryDatabase("select anzahl from artikel where name = :name");
}
```

Persistency

DB

Wir haben getrennt - jetzt ist alles gut. Oder?

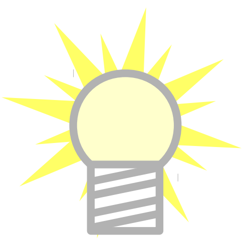
Die Oberfläche läßt sich leicht und unabhängig verändern.

Änderungen im Domain-Modell ziehen aber immer Änderungen in der Oberfläche nach sich.

Grundlegende Entscheidung:

> Das Domain-Modell muß zu Beginn bekannt sein!
>
> Es muß dokumentiert werden!

Wie können wir das Problem weiter verringern?



Noch loser koppeln: Zwischenschicht einziehen!

Loser gekoppelt

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

```
void notifyViewModelChange() {
  verfuegbarFeld.setText(viewModel.getStatusText());
  if (viewModel.getStatus() == OK) {
    verfuegbarFeld.setColor(GRUEN);
  } else {
    verfuegbarFeld.setColor(ROT);
  }
}
```

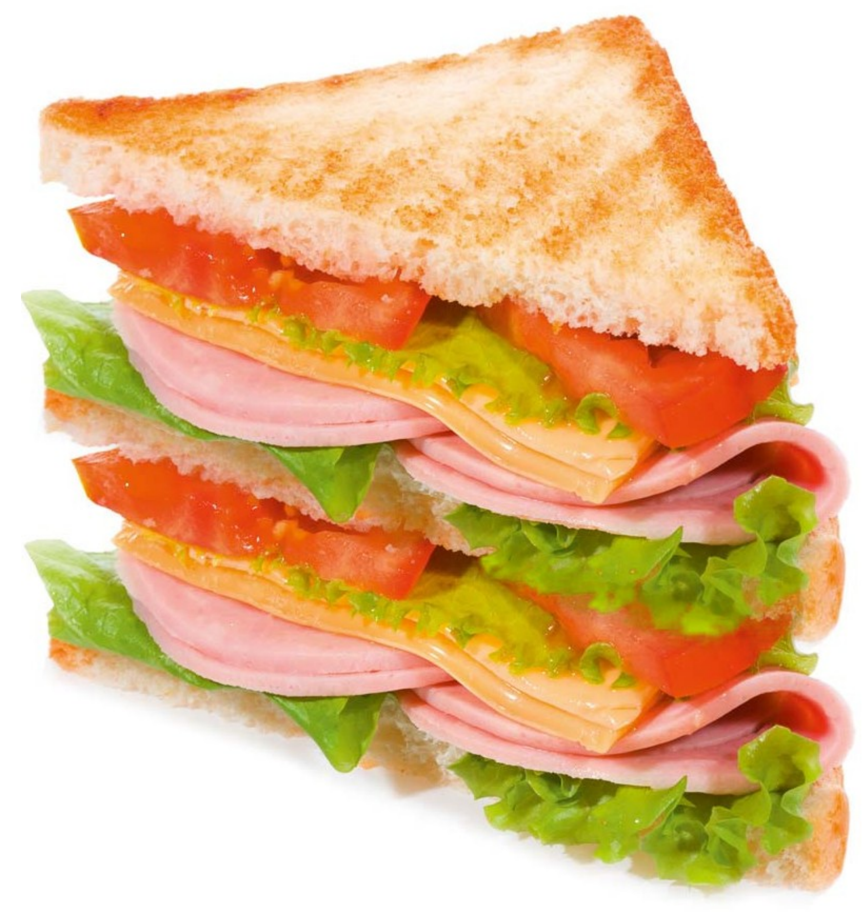
Enthält UI-Logik:
Abläufe, Aggregation

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt())) {
      setStatus(OK); setStatusText("ok");
    } else { ... }
    notifyViewModelChange();
  }
}
```

```
boolean isLieferbar(artikel, bestellt) {
   return getVerfuegbar(artikel) >= bestellt);
}
```

```
int getVerfuegbar(name) {
   return queryDatabase("select anzahl from artikel where name = :name");
}
```

4 Tiers / MVVM
Model View ViewModel



4 Tiers / MVVM

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

Standard Framework

```
void notifyViewModelChange() {
  verfuegbarFeld.setText(viewModel.getStatusText());
  if (viewModel.getStatus() == OK) {
    verfuegbarFeld.setColor(GRUEN);
  } else {
    verfuegbarFeld.setColor(ROT);
  }
}
```

View

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt())) {
      setStatus(OK);  setStatusText("ok");
    } else { ... }
    notifyViewModelChange();
  }
}
```

ViewModel

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) >= bestellt);
}
```

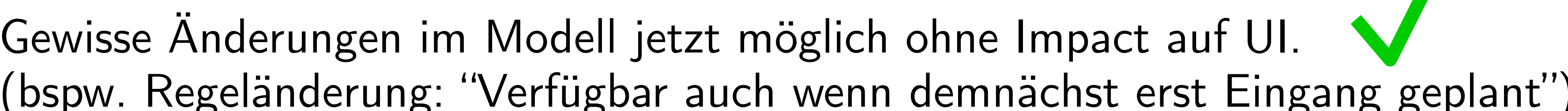
Model

```
int getVerfuegbar(name) {
  return queryDatabase("select anzahl from artikel where name = :name");
}
```

Persistency

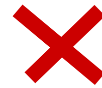
Jetzt aber... Oder immer noch nicht?

Die Oberfläche läßt sich leicht und unabhängig verändern. ✓



Gewisse Änderungen im Modell jetzt möglich ohne Impact auf UI. ✓
(bspw. Regeländerung: "Verfügbar auch wenn demnächst erst Eingang geplant")

Manche Änderungen im Domain-Modell ziehen immer noch Änderungen in der Oberfläche nach sich.



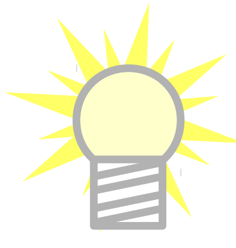
Grundlegende Entscheidung:

> Das Domain-Modell sollte zu Beginn grob bekannt sein!
>
> Das, was wesentlich ist, sollte gekennzeichnet und dokumentiert werden!

View referenziert Namen aus dem ViewModel, und das ViewModel Namen aus dem Modell.

D.h. Model-Change zieht u.U. View-Change nach sich, oder aber der Code wird langsam inkonsistent.

Wie können wir das Problem weiter verringern?



Komplette Abkopplung durch Abstraktion!

Komplett abstrakt

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

```
void notifyVUIChange() {
  feld.setText(vuiModel.getText());
  feld.setColor(vuiModel.getColor());
}
```

Reiner UI-Code

Allgemeine Funktion zur Ermittlung der Farbe

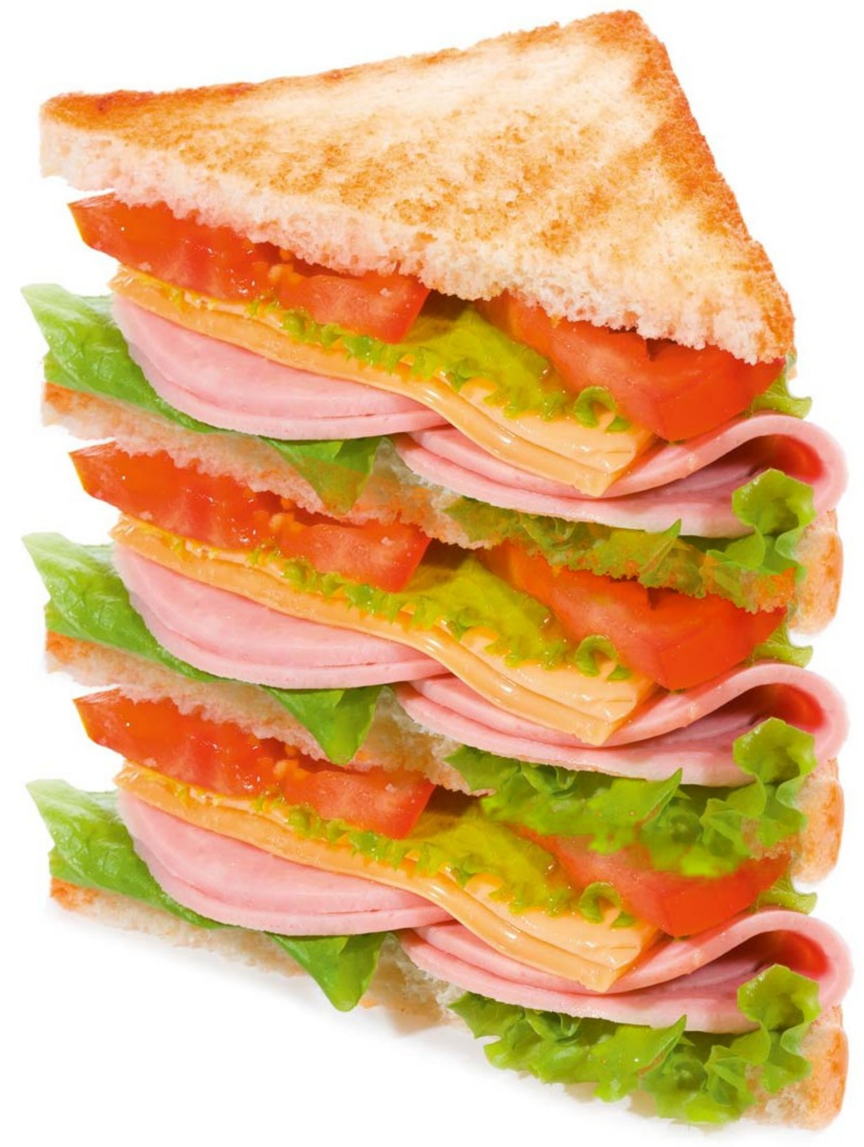
```
String getColor() {
  if (viewModel.getStatus() == OK) {
    return GRUEN;
  } else { ... }
}
```

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt())) {
      setStatus(OK); setStatusText("ok");
    } else { ... }
    notifyViewModelChange();
  }
}
```

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) >= bestellt);
}
```

```
int getVerfuegbar(name) {
  return queryDatabase("select anzahl from artikel where name = :name");
}
```

5 Tiers / M3V
Model View View View



5 Tiers / M3V

Framework

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

Standard

Media View

```
void notifyVUIChange() {
  feld.setText(vuiModel.getText());
  feld.setColor(vuiModel.getColor());
}
```

Standard

Virtual UI View

```
String getColor() {
  if (viewModel.getStatus() == OK) {
    return GRUEN;
  } else { ... }
}
```

Interaction View (ViewModel)

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt())) {
      setStatus(OK);  setStatusText("ok");
    } else { ... }
    notifyViewModelChange();
  }
}
```

Model

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) >= bestellt);
}
```

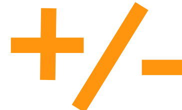
Persistency

```
int getVerfuegbar(name) {
  return queryDatabase("select anzahl from artikel where name = :name");
}
```

Komplett getrennt - ist das jetzt besser?

Oberfläche und Business Logik sind unabhängig!

Aufbau ist nicht mehr selbsterklärend.

VUI und Media Schicht schränkt Möglichkeiten ein. 

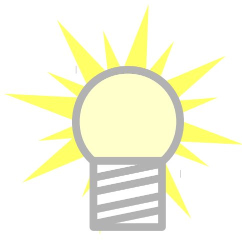
Grundlegende Entscheidung:

> Abstraktions-Schicht und Architektur müssen gut dokumentiert werden!
>
> UI/UX Interaktions-Muster müssen gut durchdacht sein!

Immer noch viel Code, teilweise redundant in Model, Interaction View und Virtual UI View.

Architektur wieder zusammenwürfeln wäre ein Schritt zurück.

Wie können wir das Problem lösen?



Wesentliche Informationen in Domänen-Modell auslagern und von dort automatisch verteilen.

Wesentliches extrahieren

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

```
void notifyVUIChange() {
  feld.setText(vuiModel.getText());
  feld.setColor(vuiModel.getColor());
}
```

```
String getColor() {
  if (viewModel.getStatus() == OK) {
    return GRUEN;
  } else { ... }
}
```

Das ist eigentlich alles.

states:

“ok”, lieferbar

“ausverkauft”, ! lieferbar

type: TrafficLight

lieferbar: verfuegbar >= bestellt

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt()))
    {
      setStatus(OK);  setStatusText(“ok”);
    } else {  ...  }
    notifyViewModelChange();
  }
}
```

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) >= bestellt);
}
```

```
int getVerfuegbar(name) {
   return queryDatabase(“select anzahl from artikel where name = :name”);
}
```

MDSD
Model Driven Software Development

5 Tiers / M3V + MDSD

Framework — Standard

```
while (true) {
  while (getMouseState() != LEFT_PRESSED) { sleep(10); }
  if (getMousePosition() == PRUEF_FELD) {
    callback();
  }
}
```

Media View — Standard

```
void notifyVUIChange() {
  feld.setText(vuiModel.getText());
  feld.setColor(vuiModel.getColor());
}
```

Virtual UI View — Generiert

```
String getColor() {
  if (viewModel.getStatus() == OK) {
    return GRUEN;
  } else { ... }
}
```

Domain Model

states:
“ok”, lieferbar
“ausverkauft”, ! lieferbar
type: TrafficLight
lieferbar: verfuegbar >= bestellt

Interaction View (ViewModel) — Generiert

```
void notifyModelChange() {
  if (verfuegbarFeld.hasChanged()) {
    if (isLieferbar(model.getCurrentArtikel(), model.getBestellt()))
    {
      setStatus(OK); setStatusText("ok");
    } else { ... }
    notifyViewModelChange();
  }
}
```

Model — Generiert

```
boolean isLieferbar(artikel, bestellt) {
  return getVerfuegbar(artikel) ... ellt);
}
```

Persistency

```
int getVerfuegbar(name) {
  return queryDatabase("select anzahl from artikel where name = :name");
}
```

Ein Traum... ;-)

Bewertungskriterien

Was ist fundamental?

(d.h. schwer revidierbar?)

Interessante Beobachtung:

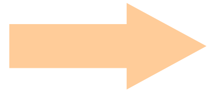
MVC:	BL schwer	UI leicht
M3V+MDSD:	BL leicht	UI mittel

"Fundamental" ist keineswegs eindeutig!

Ziel guter Architektur?

Das war unser Problem!

- Qualität hoch
- Aufwand gering
  (zu Beginn, aber auch bei Änderungen)



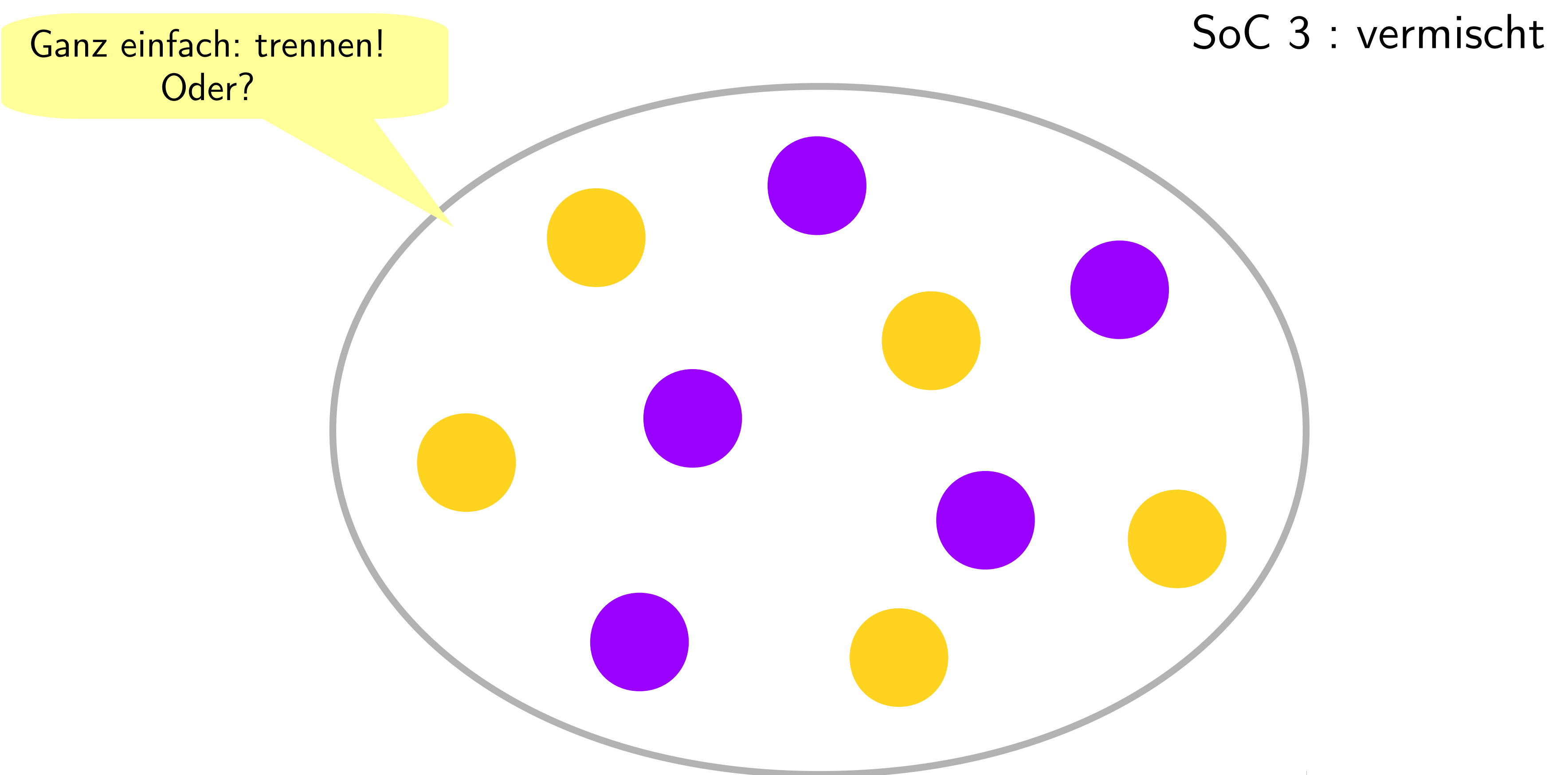
Revidierbarkeit:
1. Verständlichkeit, Komplexität, Auffindbarkeit
2. Impact: Menge der betroffenen Systemteile

!

Code-Stellen, Dokumentation, Komponenten, Deployments, Anwender, Geschäfts-Einheiten, usw.

Prinzipien:
- SoC - Separation of Concerns
- DRY - Don't Repeat Yourself

Separation of Concerns



Separation of Concerns

SoC 2 : getrennt, Dependency

Separation of Concerns

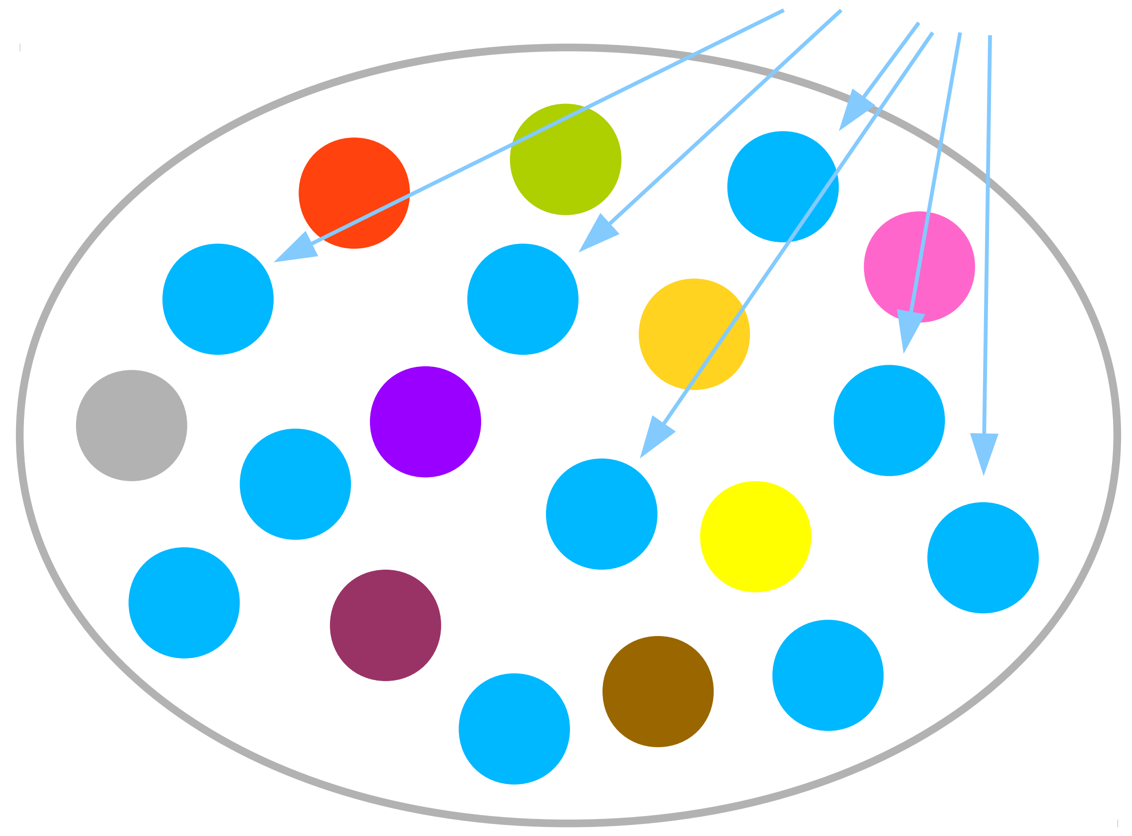
SoC 1 : getrennt, Dependency

Separation of Concerns

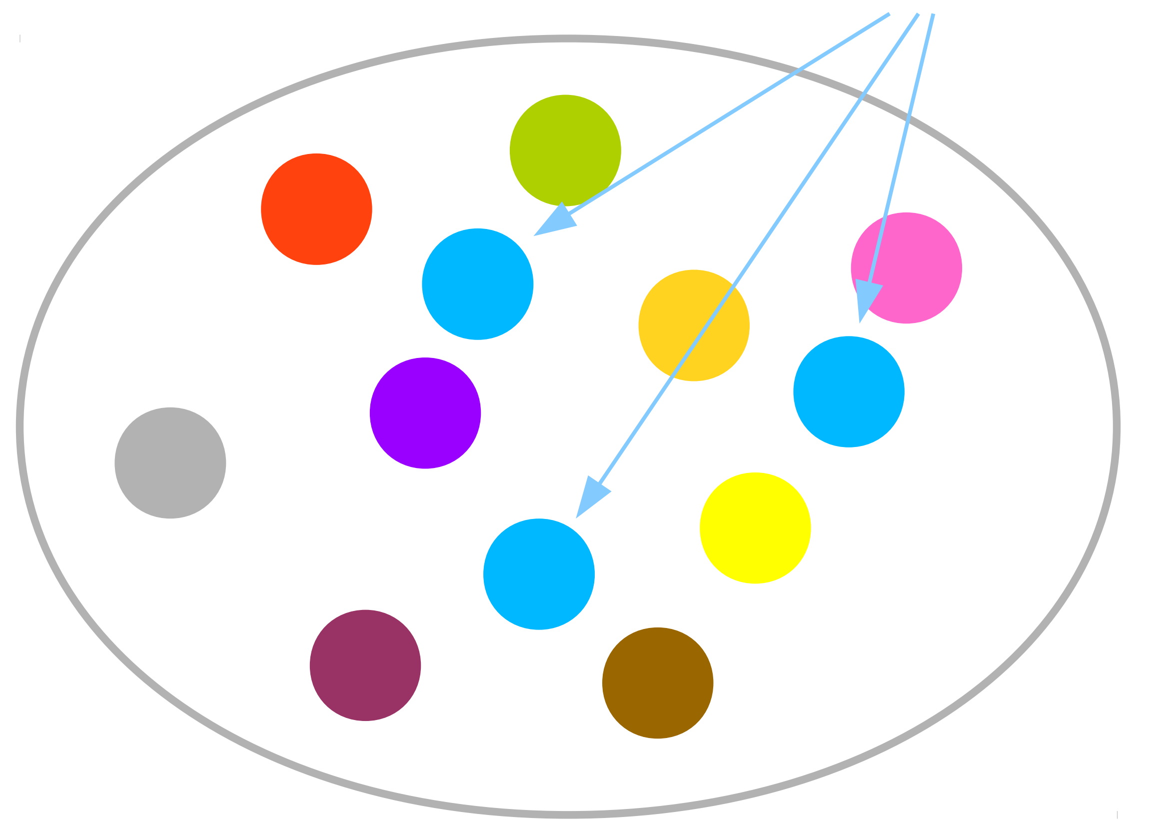
SoC 0 : getrennt, abstract Dependency

Don't Repeat Yourself

DRY 3 : wild verstreut, unwartbar

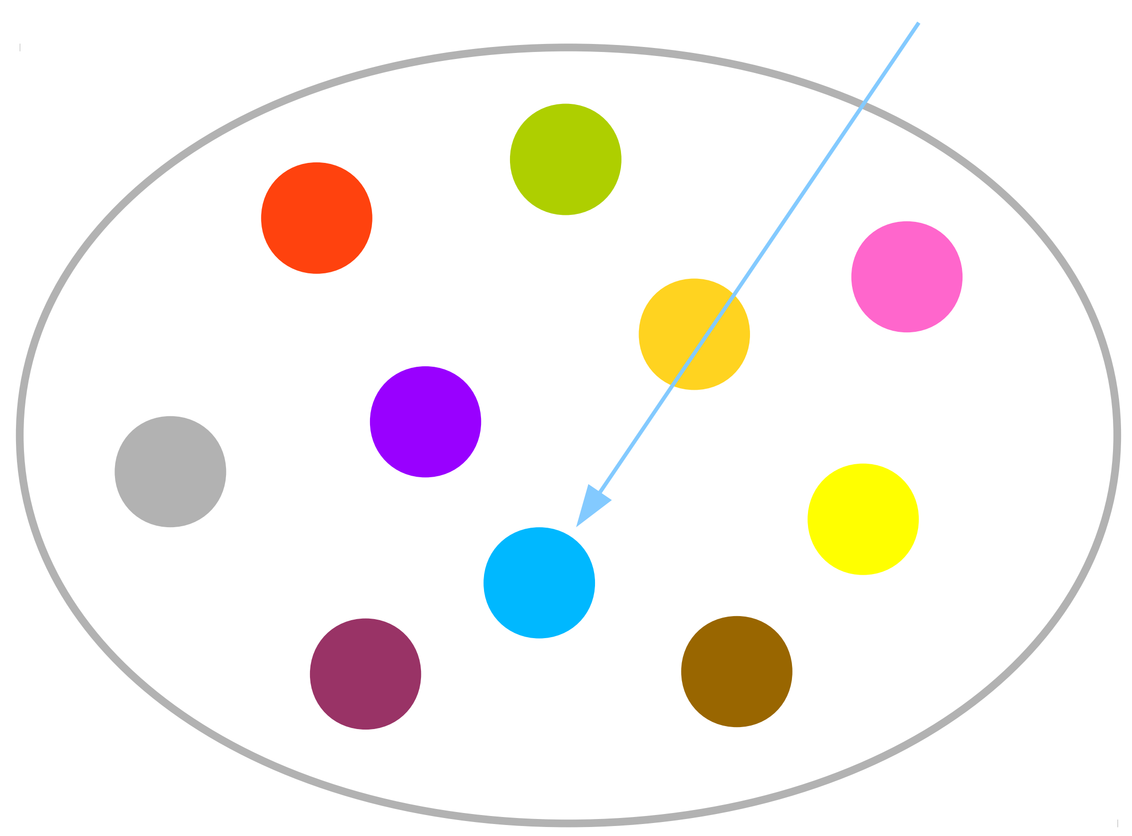


Don't Repeat Yourself



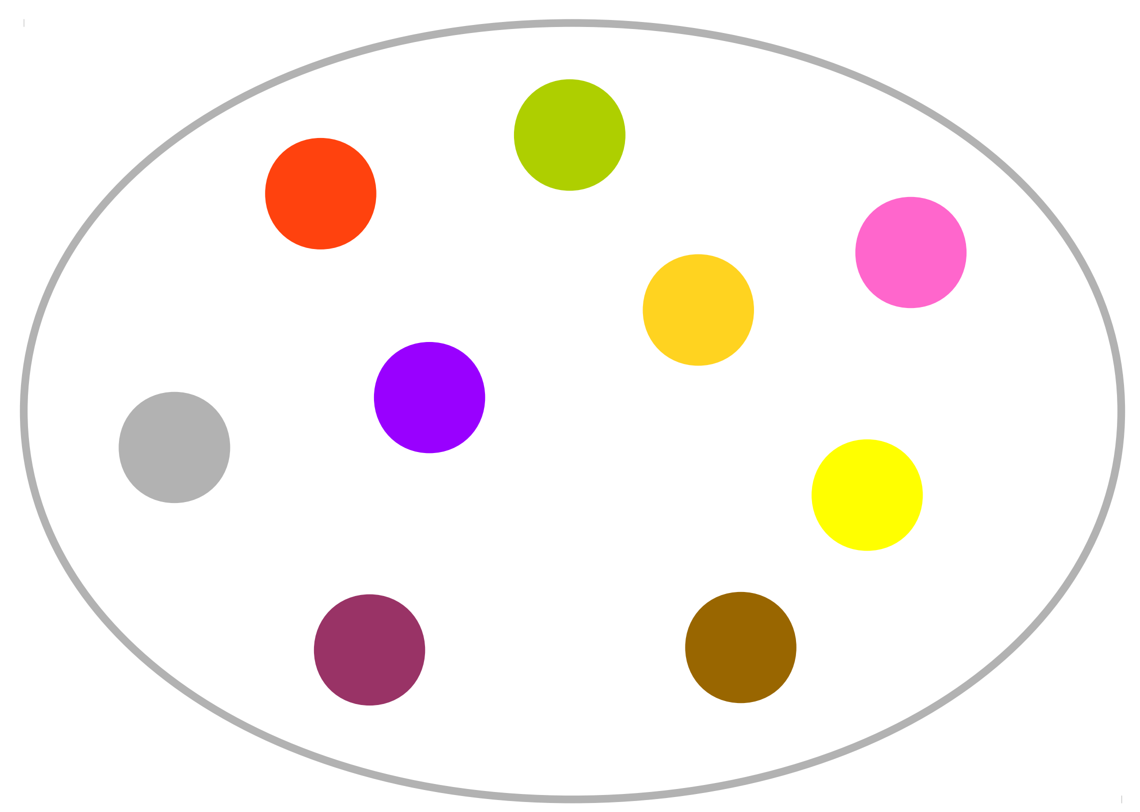
Don't Repeat Yourself

DRY 1 : genau einmal



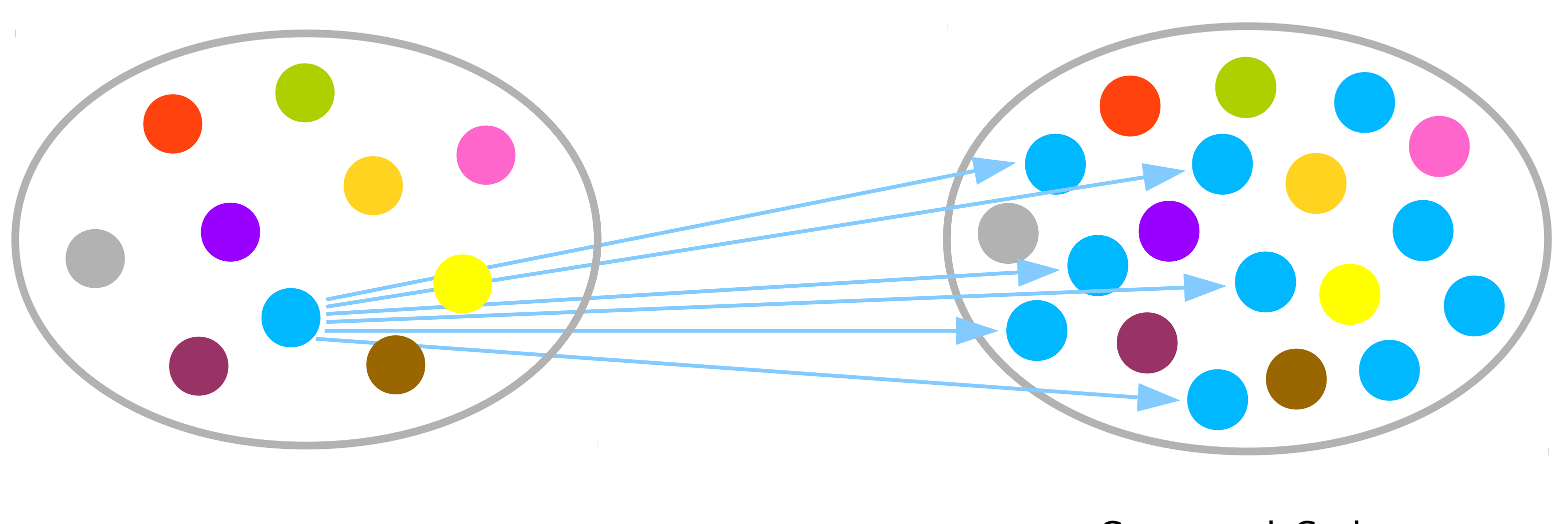
Don't Repeat Yourself

DRY 0 : gar nicht



Don't Repeat Yourself

Auch DRY 1 !



Prinzipien

- SoC - Separation of Concern

  SoC 3 : vermischt
  SoC 2 : getrennt, Dependency <--->
  SoC 1 : getrennt, Dependency --->
  SoC 0 : getrennt, Dependency ---

- DRY - Don't Repeat Yourself

  DRY 3 : wild verstreut, unwartbar
  DRY 2 : mehrfach
  DRY 1 : genau einmal
  DRY 0 : gar nicht

Prinzipien

SoC 0...3, DRY 0...3

Nützlich zur Bewertung und Verbesserung von Szenarien

Z.B. "Agile vs. Modelling ?"

Agile

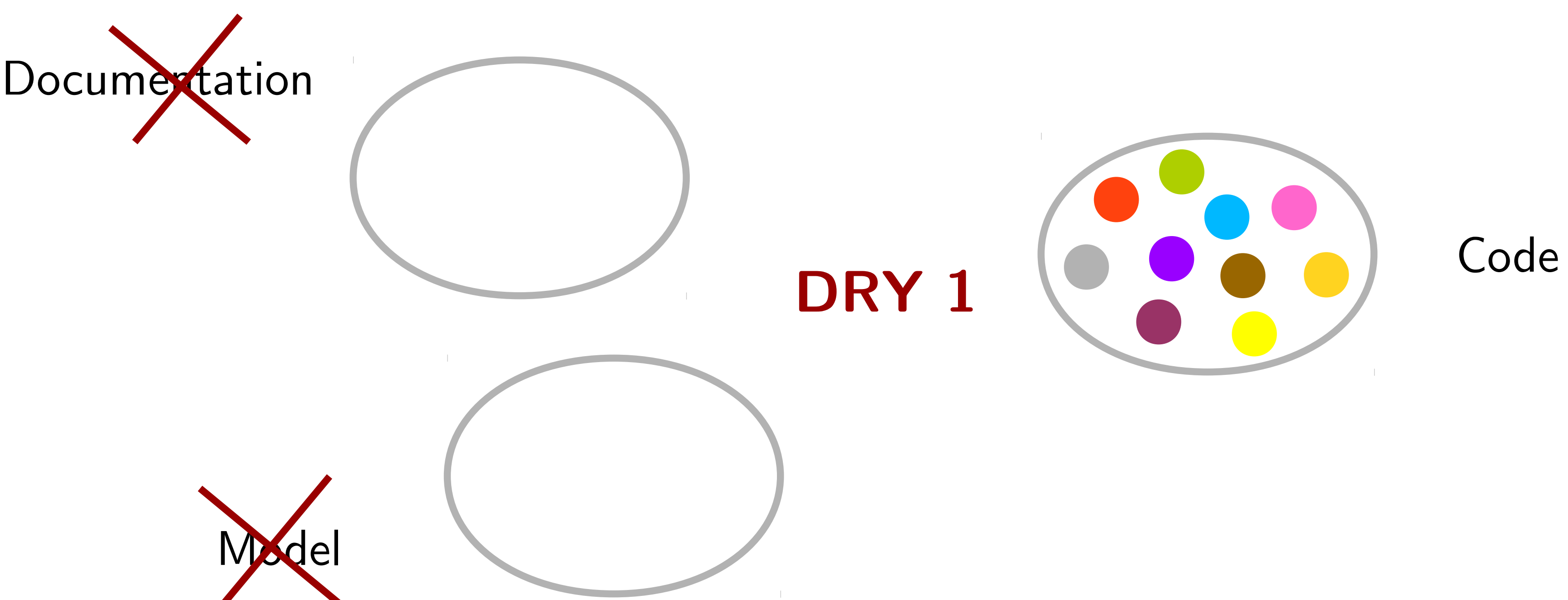
agilemanifesto.org (values)

- Working software over comprehensive documentation
- Responding to change over following a plan
- ...

Agile

agilemanifesto.org

- Working software over comprehensive documentation
- Responding to change over following a plan
- ...



Model Driven Software Development

Model

DRY 1

Code

Documentation

Model Driven Software Development

Model

Code

DRY 1

(immer noch)

Documentation

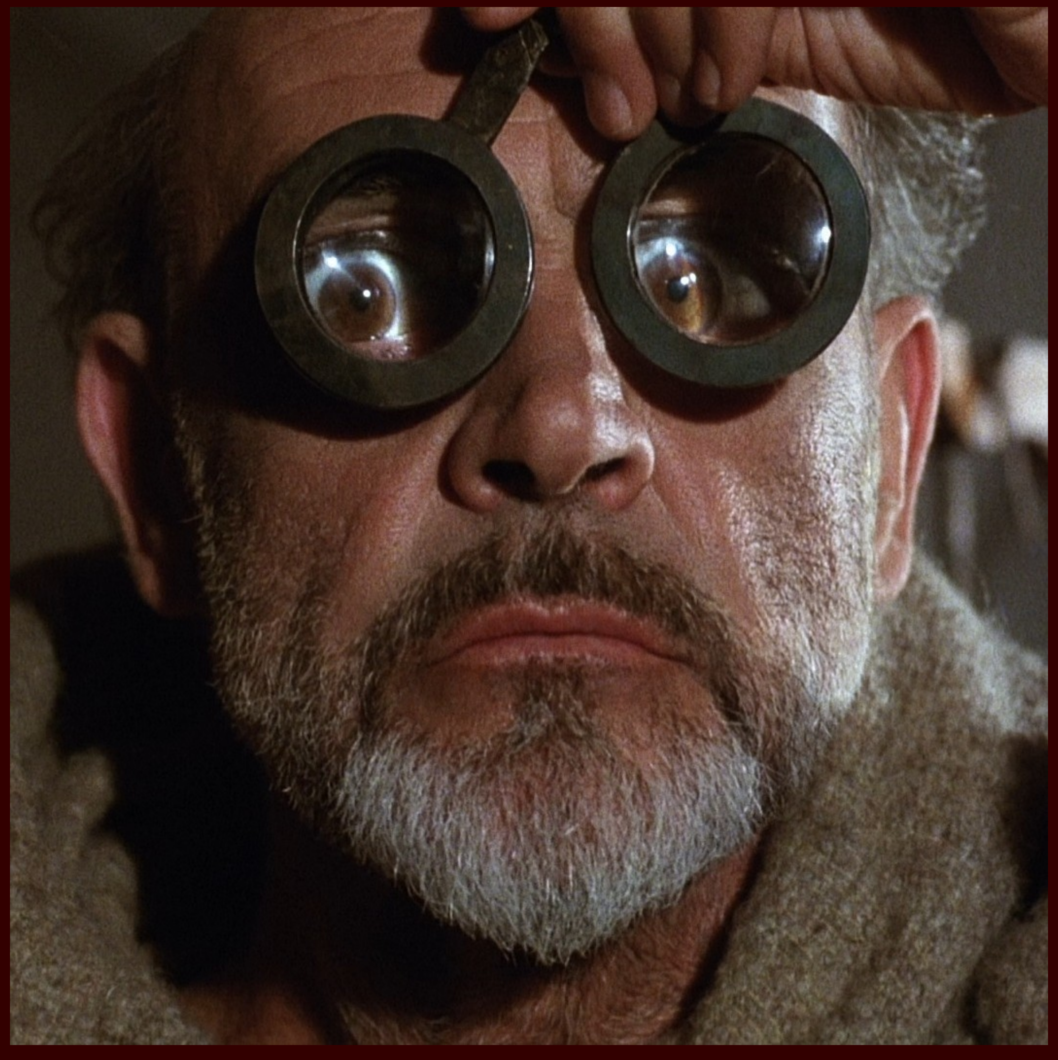
Agile

agilemanifesto.org (principles)

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done - is essential.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
- ...

Kann zu MDSD führen!

Learnings

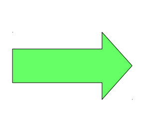


Ich bin wie ein Besessener hinter einem Anschein von Ordnung hergelaufen,
während ich doch hätte wissen müssen, dass es in der Welt keine Ordnung gibt.

William von Baskerville in "Der Name der Rose" von Umberto Eco

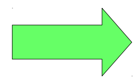
Learnings

- *Welche Entscheidungen sind fundamental ?*



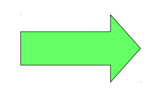
alles andere als festgelegt
kann beeinflußt werden durch andere Entscheidungen
sollte sorgfältig überlegt werden

- *Revidierbarkeit betrifft das ganze System*



Code, Menschen, Installationen... Dokumentation... Modelle...

- *Was soll dokumentiert werden, was nicht ?*



ist jeweils abzuwägen