Prinzipiensprachen

Entwurfsentscheidungen treffen und darüber reden

Christian Rehn aka R2C2

Delphi-Tage 2013

Vorstellung [Folie 2]

- Christian Rehn aka R2C2
- 2001 angefangen zu programmieren
- Informatikstudium an der TU Kaiserslautern
- Moderator und Redakteur bei Delphi-Treff
- Seit Mai im 1&1 Source Center
- http://www.christian-rehn.de/

Organisatorisches [Folie 3]

- Folien enthalten nur wenig Text
  - Besser für Präsentation
  - Zusätzlich ausführlichere Vortragsnotizen online: http://www.christian-rehn.de/
- Feedback erwünscht (persönlich, per Mail, im Forum, im Blog, ...)

Inhaltsverzeichnis

1 Geschichte

Es war einmal...

In diesem Vortrag möchte ich eine Geschichte erzählen. Meine Geschichte... gewissermaßen. Es begann an einem verregneten Novembertag im Jahre 2001 (wobei ich mich an Regen nicht erinnern kann). Irgendwie bekam ich „Internet“ und klaute in einer Nacht-und-Nebel-Aktion (wahrscheinlich irgendwann nachmittags) einen QBasic-Compiler aus einem alten Backup...

QBASIC

Kurze Zeit später konnte ich für meine Schwester ein Programm zum Rechnen üben schreiben.

Delphi

Von QBASIC kam ich zu Pascal von Pascal zu Delphi. Dann habe ich alles Mögliche gemacht.

Und dann habe ich programmiert.

Aber der Code. . .

Meine ersten Programme waren fürchterlich. Schwer zu lesen, voller Redundanz und Workarounds. . .

BibDB

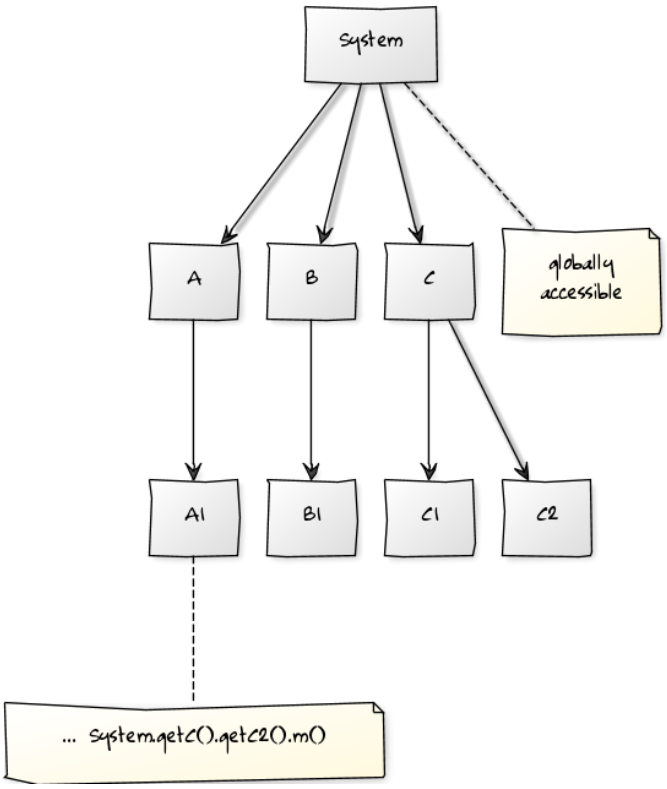
Eines meiner schon fortgeschritteneren Programme war ein kleines Bibliotheksverwaltungsprogramm für die Schulbibliothek, Dieses dient sein einiger Zeit bei meinen Vorträgen als abschreckendes Beispiel. Das folgende Stück Code zeigt die zentrale architektonische Idee:

```
TBib = class(TObject)
public
  property Systemteil  ...;
// ... aggregiert alle wichtigen Systemteile ...
end;

var
  Bib: TBib;

// Zugriff:
Bib.Systemteil.Methode();
```

Allgemein sieht das so aus:



Es gibt eine hübsche, kleine, global erreichbares Systemklasse, die die einzelnen Systemteile instanziiert und verbindet.Jeder Aufruf eines anderen Systemteils wird dadurch kinderleicht. Möchte A1 eine Methode von C2 aufrufen geschieht das in etwa so: System.C.C2.method() Jeder Aufruf geschieht fein säuberlich über ein zentrales Objekt, das alles verbindet.

Das hört sich toll an. Eine prima Idee... Dachte ich. Ich war sogar so überzeugt davon, dass ich angefangen habe, anderen das zu empfehlen. Aber es war eine wirklich, wirklich schlechte Idee und ich hatte keine Ahnung warum. Wenn man keine Ahnung hat, wie ich damals vor acht Jahren, kann man problemlos den schlechtesten Ansatz mit schönen Worten beschreiben und merkt es noch nicht mal.

Warum?

Der Ansatz war also schlecht. Genau genommen war das ganze Programm eine ziemliche Katastrophe. Aber warum? Warum war dieser Ansatz schlecht und ein anderer vielleicht besser? Diese Frage stelle ich mir und sie führte mich zu...

2 Prinzipien

Warum?

Oder anders formuliert:

Was unterscheidet gute Lösungen von schlechten?

Analytisch

Die Frage ist also nicht: „Wie komme ich auf eine Lösung?“ Als Entwickler sind wir es gewohnt Lösungen zu finden. Meist ist das gar nicht so das Problem. Die Frage ist eher, ob die Lösung, die wir gerade favorisieren eine gute ist.

Ich begann mich also mit dieser Frage zu beschäftigen. Aber natürlich war ich nicht der einzige, der sich die Frage stellte. Wie sich herausstellte, hatten schon viele vor mir gelernt, diese Unterscheidung zu treffen. Und praktischerweise haben einige von ihnen Ihre Erfahrungen in Daumenregeln oder „Prinzipien“ ausgedrückt.

Ein paar bekannte Prinzipien [Folie 21]

- KISS
- Murphy's Law
- Starke Bindung, lose Kopplung
- DRY

- SOLID (SRP, OCP, LSP, ISP, DIP)
- Kapselung/Information Hiding
- ...

- Es gibt sehr viele solcher Prinzipien (dutzende, wahrscheinlich hunderte)
- Manche Prinzipien existieren schon seit Ewigkeiten, sind quasi Folklore der Softwareentwicklung

Unter „Prinzip“ oder „Daumenregel“ verstehe ich dabei folgendes:

Prinzipien [Folie 22]

Definition
Ein **Prinzip** ist eine Daumenregel, die gute von schlechten Lösungen unterscheidet – in Bezug auf *einen* Entwurfsaspekt.

- Prinzipien sind informelle Daumenregeln, keine in Stein gemeißelten Gesetze
- Prinzipien helfen, gute von schlechten Lösungen zu unterscheiden
- Dabei beschränken sie sich aber auf einen einzigen abgegrenzten Entwurfsaspekt. In der Regel muss man deshalb immer mehrere Prinzipien bedenken, wenn man Entwurfsentscheidungen trifft.

Sehen wir uns mal ein Beispiel im Detail an:

Murphy's Law (ML) [Folie 23]

„Whatever can go wrong, will go wrong“

Murphy's Law (ML) [Folie 24]

Aussage Alles was schief gehen kann, wird irgendwann schiefgehen. Also ist eine Lösung dann besser, wenn sie weniger Möglichkeiten zulässt, dass etwas schiefgeht.

Begründung Menschen machen Fehler und das wird sich nie grundlegend ändern. Statistisch gesehen wird also auf lange Sicht, eine Fehler-Möglichkeit auch zu einem Fehler führen.

Beispiel `Date date1 = new Date(2013, 01, 12);`

Ein Beispiel aus Java [Folie 25]

```
new Date(2013, 01, 12);
```

Dieser Aufruf erzeugt ein Datumsobjekt, das den 12. Februar 3913 repräsentiert. Das ist unerwartet. Man sieht, dass man hier sehr viel Falsch machen kann.

- Man kann die Parameter vertauschen
- Man kann fälschlicherweise annehmen, dass der Monatswert 1-basiert sei
- Man kann das Jahr fälschlicherweise mit 4 Ziffern angeben

Nach Murphy's Law wird man bei der Benutzung von `java.util.Date` zwangsläufig irgendwann Fehler machen. Deshalb ist die Klasse schlecht entworfen. BTW: Es gibt noch diverse andere Schwachstellen in der Date-API. Deshalb wurde die Calendar-API eingeführt. Diese ersetzt manche der Schwachstellen durch ein paar andere. Mit Java 8 soll endlich eine gutelösung folgen.

Man sieht hier: Das Prinzip sagt nicht, wie man eine gute Lösung konstruiert. Vielmehr unterscheidet es gute von schlechten Lösungen. Es sagt ganz klar, dass der Konstruktor von `java.util.Date` sehr schlecht entworfen ist. Und wenn man mehrere Möglichkeiten zur Auswahl hat, kann das Prinzip dasjenige herausfinden, das am besten (im Bezug auf den einen Aspekt „Fehlermöglichkeiten") ist.

Prinzipien widersprechen einander

Das bedeutet:

- Wir müssen immer mehrere Prinzipien gleichzeitig bedenken
- Wir können aber nicht alle Prinzipien gleichermaßen erfüllen
- Deshalb müssen wir immer Mittelwege und Kompromisse suchen

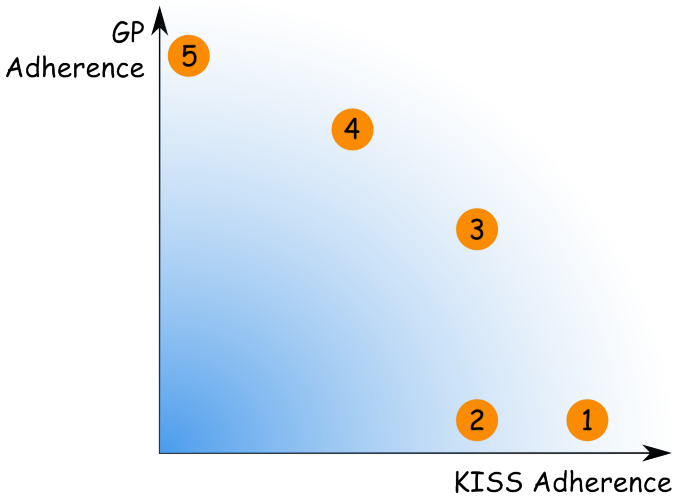
Anforderung: $\sqrt{2}$ wird benötigt [Folie 27]

1. `const SQRT_2 = 1.4142135623730951;`
2. `function sqrt_2: Real;`
3. `function sqrt(r: Real): Real;`
4. `function power(base, exponent: Real): Real;`
5. `class TComplexPolynomRootCalculator`

Was ist der beste Weg, wenn ich den Wert $\sqrt{2}$ in meinem Programm benötige?

1. Eine Konstante `const SQRT_2 = 1.4142135623730951;`
2. Eine Funktion, die $\sqrt{2}$ berechnet: `function sqrt_2: Real;`
3. Eine Funktion, die beliebige Quadratwurzeln aus reellen Zahlen berechnet `function sqrt(r: Real): Real;`
4. Eine Funktion, die beliebige Potenzen aus reellen Zahlen berechnet: `function power(base, exponent: Real): Real;`
5. Eine Klasse, die beliebige Wurzeln aus komplexen Polynomen berechnen kann: `TComplexPolynomRootCalculator`

Wir betrachten hier zwei Prinzipien: KISS und GP. KISS verlangt nach einer möglichst einfachen Lösung, GP nach einer möglichst allgemeinen. Wir können aber nicht beides haben. Entweder ist eine Lösung allgemein (Variante 5) oder einfach (Variante 1), aber nicht beides. Typischerweise implementiert man einen Mittelweg (Varianten 3 und 4)). Variante 2 ist eine schlechte Lösung, da 1 einfacher ist ()bei gleicher Mächtigkeit) und 3 mehr kann (bei gleicher Komplexität).


GP
Adherence
5
4
3
2
1
KISS Adherence

3 Prinzipiensprachen

Mit Prinzipien. Mit mehreren Prinzipien, weil jedes nur einen einzigen Aspekt beleuchtet. Andere Aspekte, die auf Vor- und Nachteile hinweisen können, werden durch andere Prinzipien beschrieben.

Wie findet man passende Prinzipien?

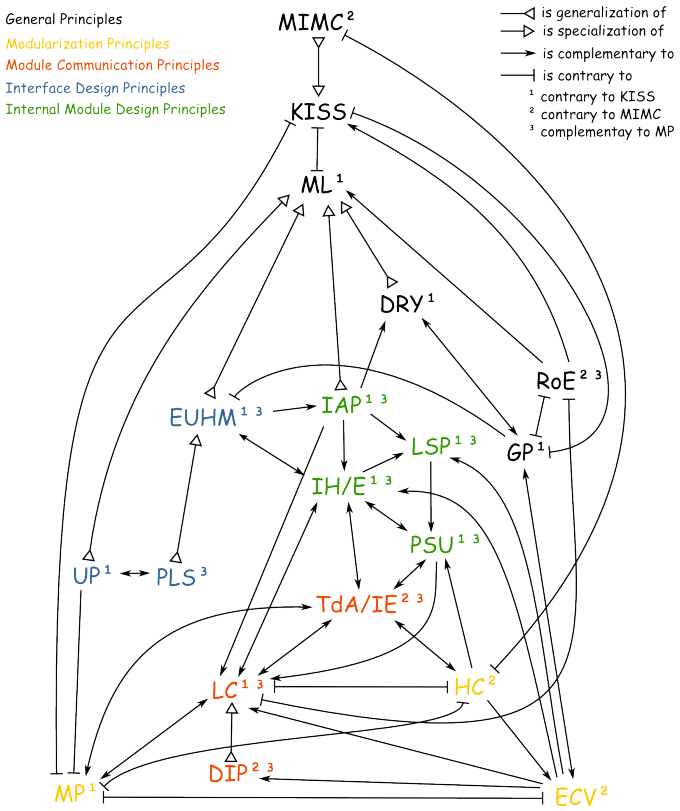
Prinzipiensprachen

Prinzipiensprachen vernetzen Prinzipien so, dass das Betrachten eines Prinzips automatisch zu weiteren Prinzipien führt, die wahrscheinlich im gegebenen Kontext ebenfalls relevant sind. Man fängt also bei einem Prinzip an, das offensichtlich zum Entwurfsproblem passt und die Prinzipiensprache sagt einem, welche anderen Prinzipien man noch bedenken sollte.

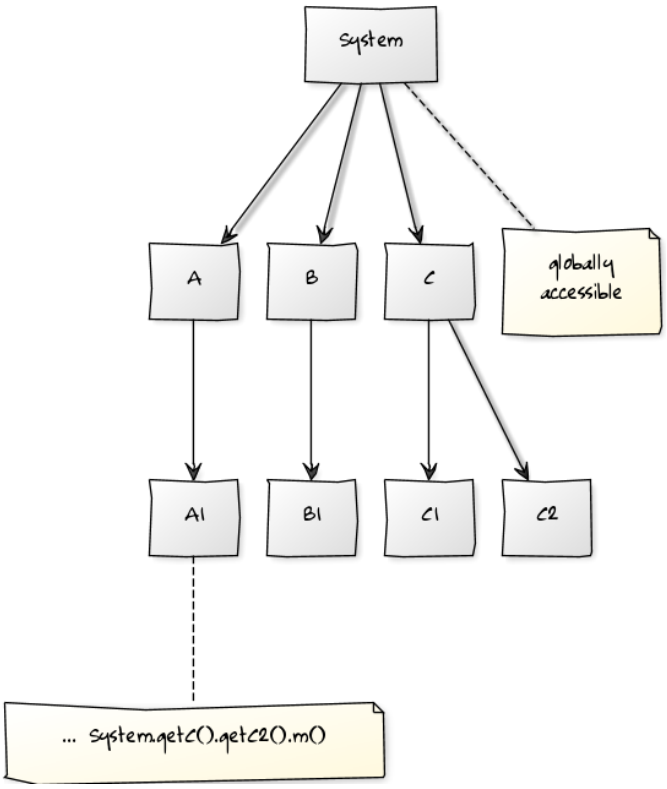
Zusätzlich bilden Prinzipiensprachen ein in sich abgeschlossenes Vokabular – eine Sprache – um über Entwurfsentscheidungen zu reden. Die einzelnen Entwurfsaspekte erhalten Namen.

Für meine Masterarbeit habe ich eine Prinzipiensprache für den Objektorientierten Entwurf (OOD) erstellt:

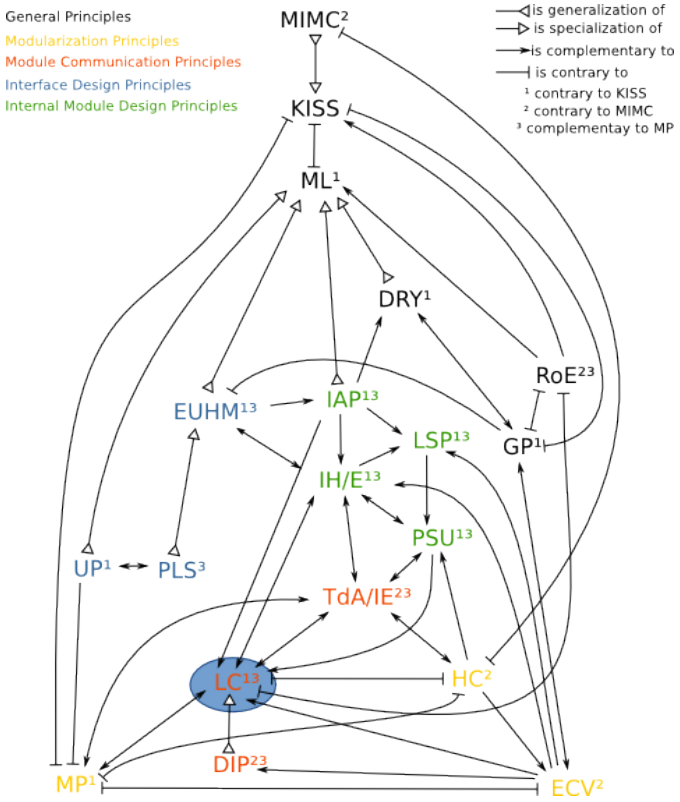
ODD Principle Language [Folie 32]



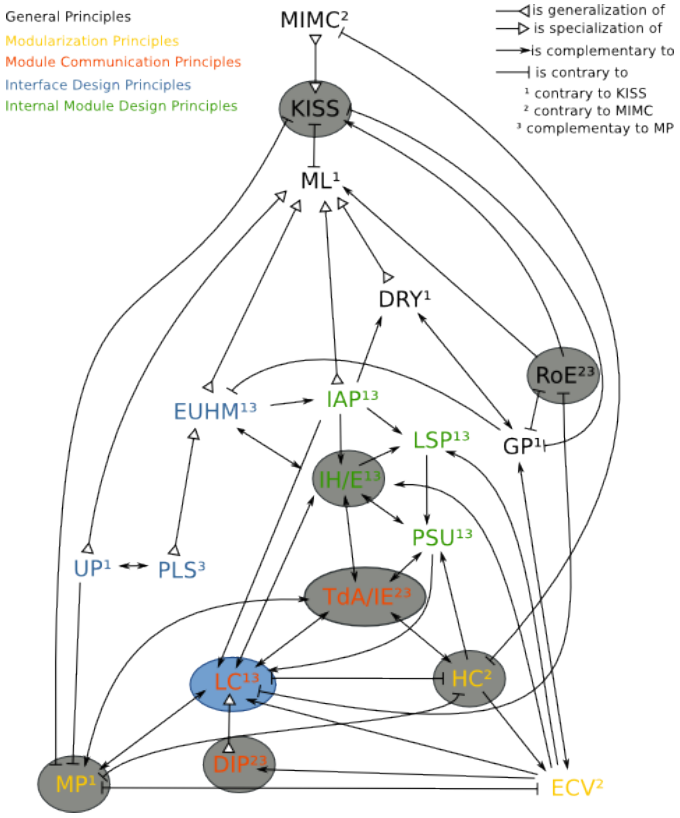
Schauen wir uns nun einmal an, wie wir die Prinzipiensprache nutzen können, um relevante Aspekte zu finden. Wir nehmen dazu gleich mal ein relativ komplexes Beispiel, an dem man viel zeigen kann:



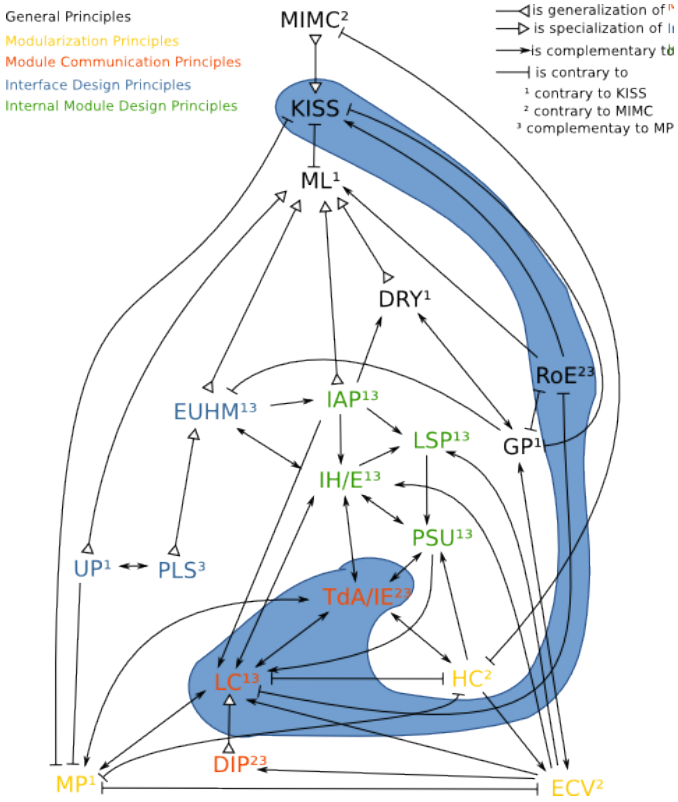
Warum ist das schlecht? befragen wir die Prinzipiensprache:



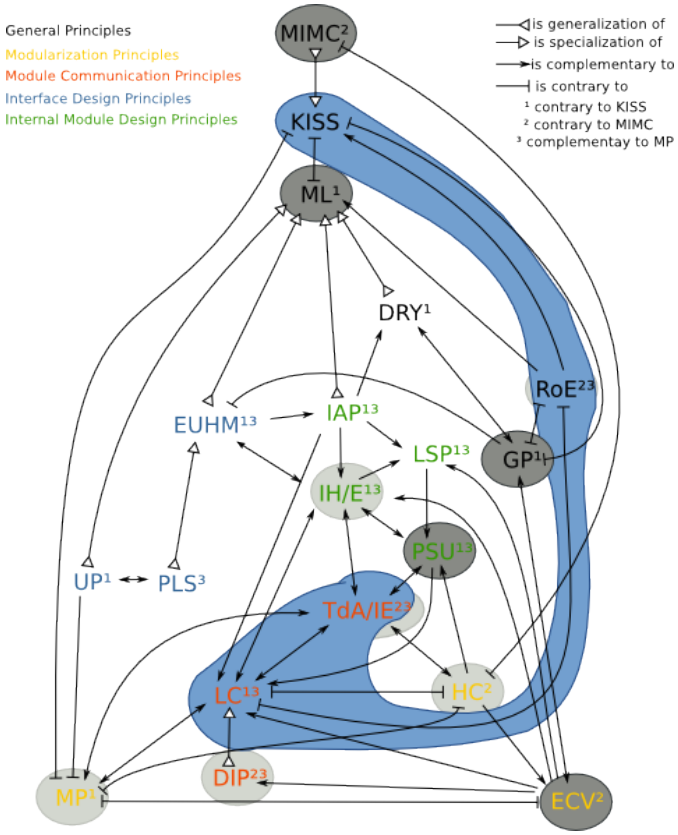
Unser Startprinzip ist LC. Das erscheint uns am passendsten. Deshalb fangen wir damit an.



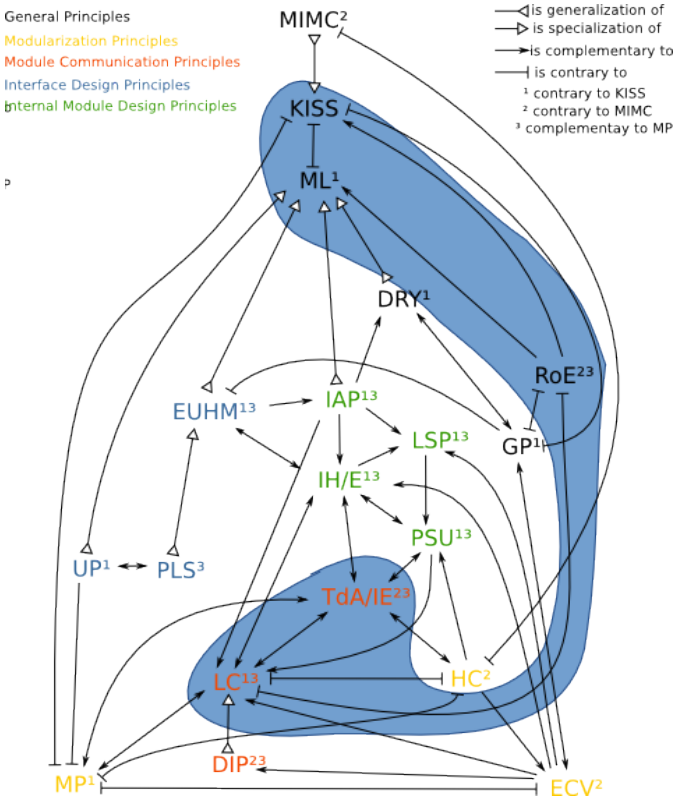
Wir betrachten nun die Prinzipien, mit denen LC in Beziehung steht.



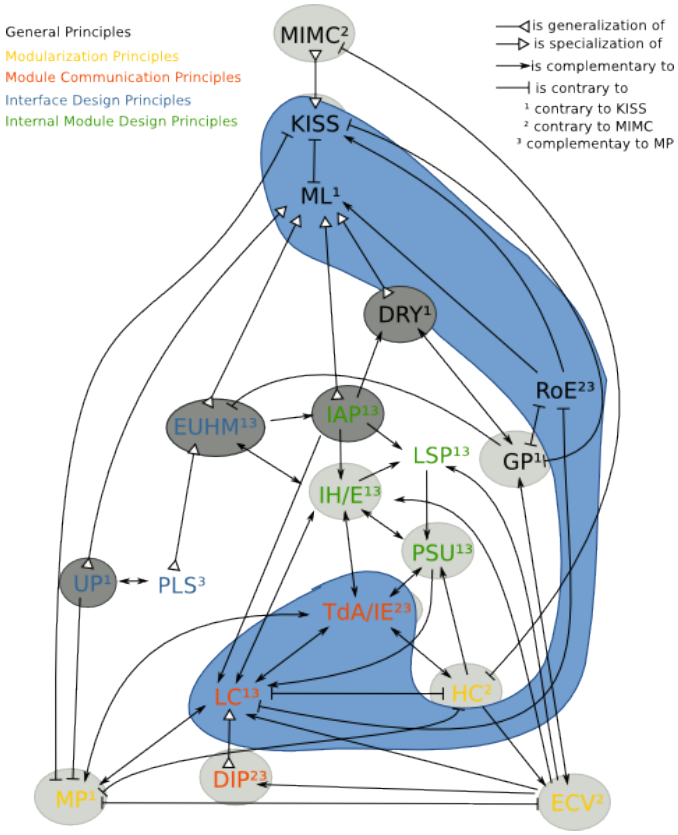
KISS, RoE und TdA/IE passen.



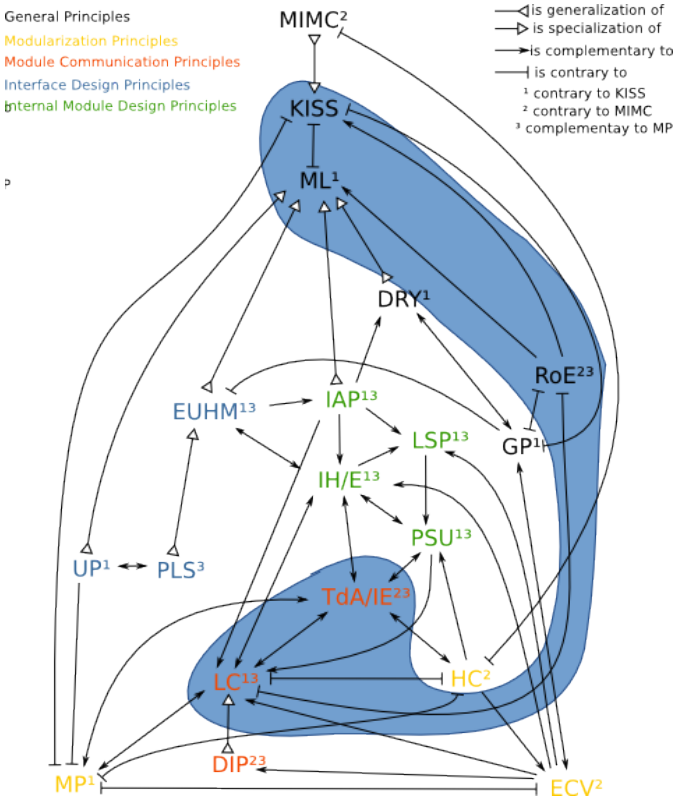
Und wieder sehen wir uns die benachbarten Prinzipien an.


General Principles
Modularization Principles
Module Communication Principles
Interface Design Principles
Internal Module Design Principles
is generalization of
is specialization of
is complementary to
is contrary to
¹ contrary to KISS
² contrary to MIMC
³ complementay to MP
MIMC²
KISS
ML¹
DRY¹
RoE²³
EUHM¹³
IAP¹³
LSP¹³
GP¹
IH/E¹³
PSU¹³
UP¹
PLS³
TdA/IE²³
LC¹³
HC²
DIP²³
MP¹
ECV²

ML passt.



Die nächsten bringen keine weiteren Erkenntnisse.



Somit bleibt es bei den 5 Prinzipien LC, KISS, RoE, TdA/IE und ML. Diese charakterisieren unser Entwurfsproblem. Jetzt geht es an die Bewertung:

- LC ✗
- KISS ✓
- RoE ✗
- TdA/IE ✗
- ML ✓

- LC, RoE und TdA/IE sind gegen die Lösung, KISS ist dafür. ML hat zumindest mal nichts dagegen.

- Natürlich führt einfaches Abzählen der Prinzipien nicht zu einer guten Lösung. Der Ansatz nimmt dem Designer keine Entscheidung ab. Er zeigt nur, welche Aspekte zu bedenken sind.
- Im konkreten Fall würde man wohl zum Schluss kommen, dass die Lösung zwar einfach ist, aber die Einfachheit macht die Nachteile durch Kopplung und fehlender Explizität nicht wett.

Was ist besser?

Dependency Injection

Warum das nun wirklich besser ist, kann man auch wiederum mit den obigen Prinzipien begründen. Das Beispiel findet sind deutlich detaillierter im Wiki: http://www.principles-wiki.net/about:navigating_principle_languages

4 Das Wiki

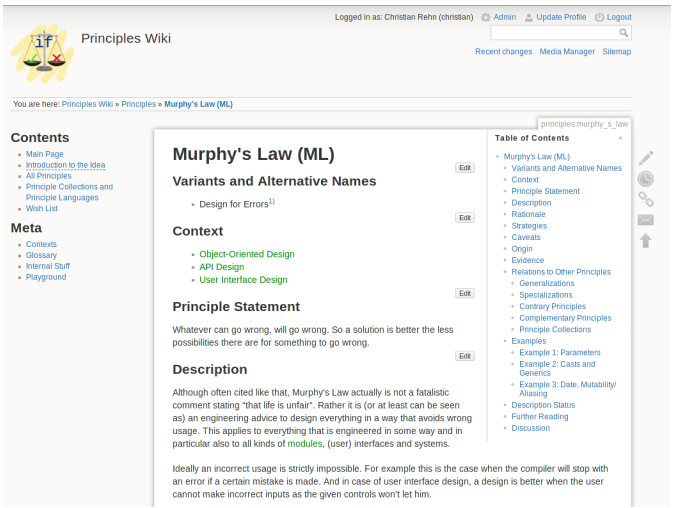
Für meine Masterarbeit habe ich u. a. ein Wiki erstellt und dort Prinzipien und die Prinzipiensprache dokumentiert.

Das Wiki [Folie 45]

www.principles-wiki.net

---

www.principles-wiki.net [Folie 46]



Variants and Alternative Names

Context Prinzipien gibt es nicht nur für den Objektorientierten Entwurf, sondern auch fürs Kodieren, die Architektur, GUI Design, etc.

Principle Statement Die Aussage des Prinzips kurz zusammengefasst in 1–2 Sätzen

Description Eine detailliertere Beschreibung

Rationale Warum das Prinzip gilt

Strategies Was man tun kann um seinen Entwurf so an zu passen, dass es besser dem Prinzip genügt

Caveats Worauf man aufpassen muss

Origin Wo das Prinzip herkommt

Evidence Wie sehr wir wissen, dass das Prinzip gilt

Relations to Other Principles Welche Anderen Prinzipien noch zu bedenken sind

Examples Beispiele, die das Prinzip verdeutlichen

Description Status Wie fertig der Wiki-Eintrag ist

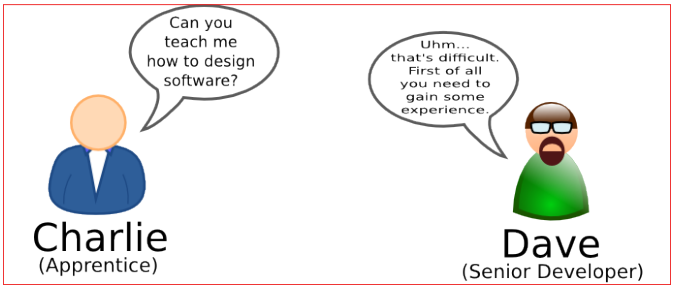
Further Reading Weiterführende Links und Literatur

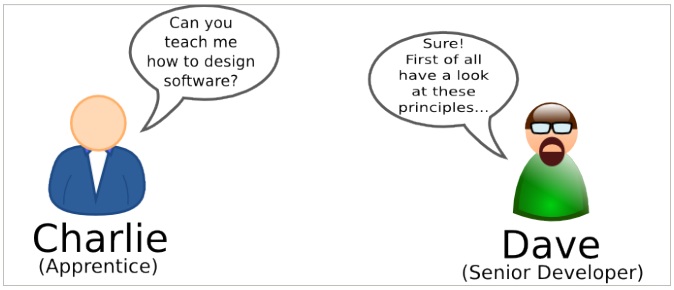
5 Vorteile

Vorteile [Folie 48]

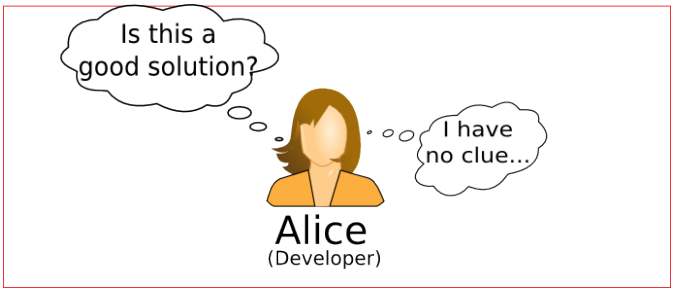
- Lernen
- Entscheiden
- Kommunizieren

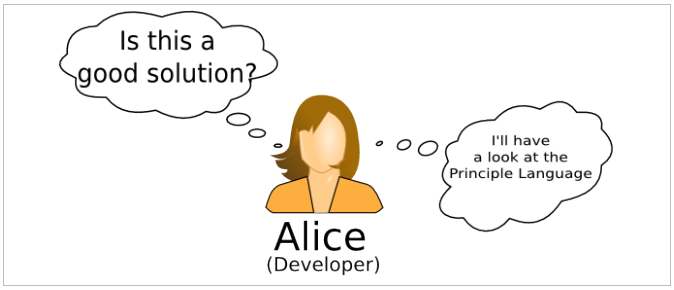
Lernen



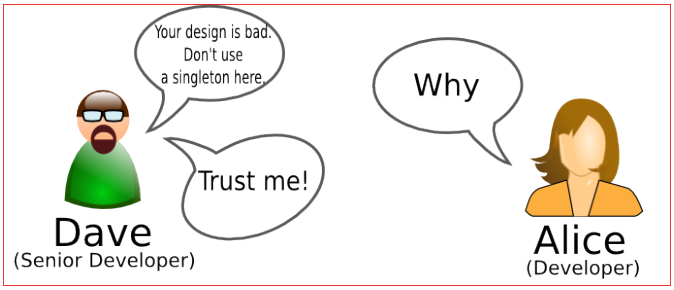


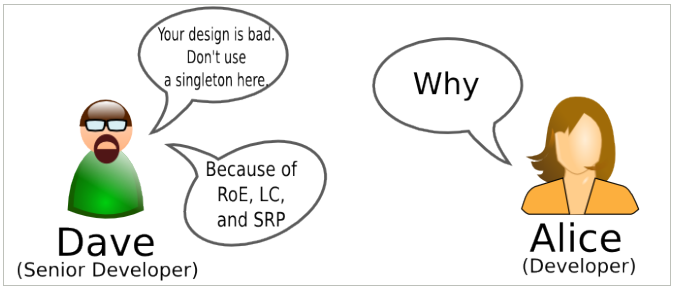
Entscheiden





Kommunizieren





Prinzipien-*Sprachen*

Prinzipiensprachen bilden also eine Hilfestellung zum Treffen von Entwurfsentscheidungen. Das ist insbesondere dann hilfreich, wenn Erfahrung fehlt. Entwickler, die genügend Erfahrung haben, brauchen diese Hilfestellung nicht. Aber auch hier können Prinzipiensprachen hilfreich sein. Sie definieren nämlich eine „Sprache“, ein Vokabular mit dem man über Entwürfe und Entwurfsentscheidungen reden kann. Einzelne Aspekte haben auf einmal einen Namen. Also selbst, wenn man keine Prinzipiensprache braucht, um Software zu entwerfen, kann man damit seine Entscheidungen gegenüber anderen Begründen, Vor- und Nachteile diskutieren, etc.

Auch die Vorteile sind im Wiki deutlich detaillierter beschrieben.

Fazit [Folie 59]

- Prinzipien/Daumenregeln sind wie Patterns Wiederverwendung von Erfahrung
- Mit Prinzipien kann man Entwurfsentscheidungen begründen
- Prinzipiensprachen vernetzen Prinzipien und zeigen weiterführende Aspekte auf
- Prinzipiensprachen bilden ein Vokabular
- www.principles-wiki.net

Ausblick [Folie 60]

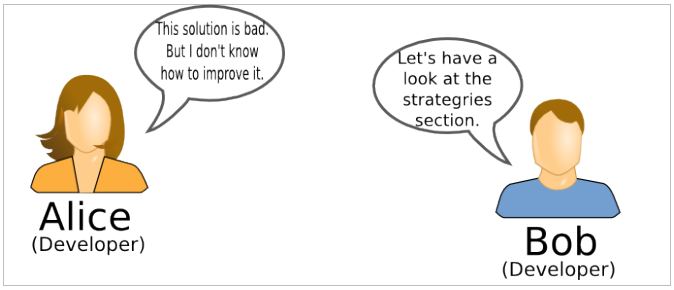
- Das Wiki wird ständig erweitert und ausgebaut
- Weitere Prinzipien und Prinzipiensprachen werden folgen
- Vernetzung von Patterns mit Prinzipien wird folgen
- Mitarbeit willkomemn

Vielen Dank! [Folie 61]

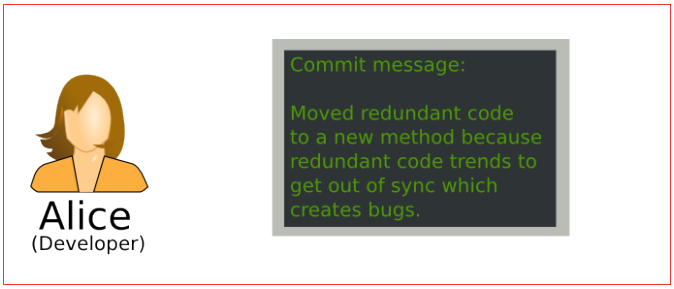
Fragen?

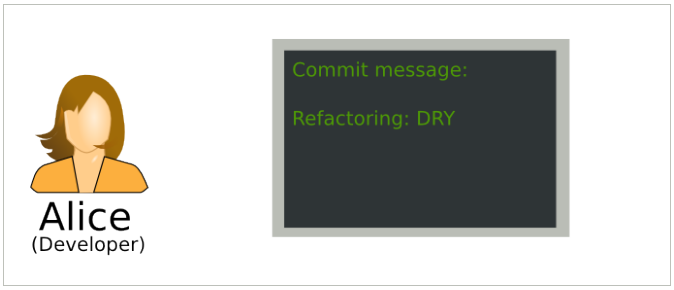
Anhang

Strategien



Commit





Das Große Ganze [Folie 70]

- Prinzipien
- Patterns
- Anti-Patterns
- Refactorings
- Glossary Terms
- Non-Principles

ODD Principle Language [Folie 71]