

Wissenschaftliche Softwareentwicklung

14. Dokumentation im Code

Till Biskup

Physikalische Chemie

Universität Rostock

08.12.2023





- 🔑 Dokumentation im Code sollte auf das notwendige Minimum beschränkt werden.
- 🔑 Veraltete oder falsche Dokumentation schadet mitunter mehr als sie nützt
- 🔑 Formulierungen sollten so knapp und präzise wie möglich, die Formatierung konsistent und übersichtlich sein.
- 🔑 Unklarer Code sollte nicht dokumentiert, sondern umgeschrieben werden.
- 🔑 Dokumentationswerkzeuge erleichtern die Arbeit, setzen aber konsistente Formatierung voraus.



“ *Real Programmers don't comment their code.
It was hard to write, it should be hard to read.*

– Ed Post

“ *Real Programmers don't need comments—
the code is obvious.*

– Ed Post

Listing: linux-2.2.16/fs/buffer.c

```
/*  
 * After several hours of tedious analysis, the following  
 * hash function won. Do not mess with it... -DaveM  
 */
```

Arten von Dokumentation im Code

Argumente für und gegen Kommentare

Regeln für Kommentare

Werkzeuge zur automatischen Erzeugung

Dokumentation im Code

Dokumentation direkt im Quellcode in Form von Kommentaren, die nicht ausgeführt (und vom Compiler/Interpreter ignoriert) wird

- Formatierung abhängig von der Programmiersprache
 - meist durch spezielles Zeichen vor dem Kommentar
 - mitunter unterschiedliche Zeichen für Beginn und Ende
- Unterscheidung von Zeilen- und Blockkommentaren
 - mitunter durch unterschiedliche Zeichen realisiert
 - dienen unterschiedlichen Einsatzgebieten

Listing: Beispiele für Block- und Zeilencommentare in C

```
/*  
 * Blockcommentar in C  
 */  
  
// Zeilencommentar in C
```

Listing: Beispiele für Block- und Zeilencommentare in Python

```
"""  
Blockcommentar in Python  
"""  
  
# Zeilencommentar in Python
```

- 👉 Block- und Zeilencommentare haben unterschiedliche Einsatzgebiete.

- **Kommentarköpfe**
 - am Anfang von (öffentlichen) Klassen- und Funktionsdefinitionen
 - dienen u.a. der Schnittstellenbeschreibung
 - Hilfestellung zur Benutzung der Klasse/Funktion
 - werden häufig vom Hilfesystem automatisch verwendet
 - **Copyright-Hinweise**
 - Autor und Jahr
 - ggf. Name der Lizenz
 - zusätzlich oft Datum der letzten Änderung
 - **(meist) einzeilige Kommentare im Code**
 - Erläuterung einer Codezeile oder eines Codeblocks
- ☞ Der Kontext entscheidet, wie sinnvoll ein Kommentar ist.

- Auskommentieren von Codeteilen
 - kurzzeitig während der aktiven Entwicklung erlaubt
 - sollte *nie* in die Versionsverwaltung wandern
 - Archivierung durch Versionsverwaltung obsolet geworden
- Änderungshistorie
 - vor allem in alten Programmen zu finden
 - komplette Historie der Änderungen mit Kurzkomentar
 - durch Versionsverwaltung obsolet geworden
 - wenn gewünscht, dann in externer Datei („CHANGELOG“)
- komplette Lizenztexte
 - meist viel zu lang
 - gehören in eigene Datei („LICENSE“, ggf. beim Quellcode)
 - im Code nur Verweis auf die Lizenz

Arten von Dokumentation im Code

Argumente für und gegen Kommentare

Regeln für Kommentare

Werkzeuge zur automatischen Erzeugung

“ *Nothing can be quite so helpful as a well-placed comment.
Nothing can clutter up a module more than
frivolous dogmatic comments.
Nothing can be quite so damaging as an old crufty comment
that propagates lies and misinformation.*

– Robert C. Martin

drei Argumente

- Kommentare können extrem hilfreich sein.
- Kommentare können Code unnötig aufblähen.
- Kommentare können lügen.

- Kommentarkopf von Funktionen und Klassen
 - kurze Beschreibung: warum? wie zu nutzen?
 - sollte nicht lediglich die Namen wiederholen
 - oft vom integrierten Hilfesystem genutzt
 - für alle vom Nutzer aufrufbaren Funktionen/Klassen
 - zielgruppengerecht: Nutzer brauchen andere Infos als Entwickler
- Aspekte, die sich nicht in Code ausdrücken lassen
 - Code beschreibt das „Was“, selten das „Warum“.
 - Leser kennen die Gedanken des Autors nicht (immer).
- Begründung für eine bestimmte Implementierung
 - bewahrt andere vor vorschneller „Optimierung“
- ☞ Erfahrung und Fähigkeiten des Programmierers spielen ebenfalls eine Rolle.

- Code und Kommentare sollten im Verhältnis stehen.
 - Code hat eine wesentlich höhere Informationsdichte.
 - Code ist exakt – ein Kommentar in der Regel nicht.
 - Kommentare ergänzen den Code, nicht umgekehrt.
 - Zu viele Kommentare schränken die Lesbarkeit ein.
- Kommentare sollten einen echten Mehrwert haben.
 - Nur das kommentieren, was nicht im Code steht.
 - Immer überlegen: Warum steht es nicht im Code?
- Code sollte für sich sprechen.
 - Das Ziel ist selbstdokumentierender Code.
- ☛ Typisches Problem unerfahrener Programmierer:
Erst wird gar nicht, dann viel zu viel kommentiert.
- ☛ Gute Nutzerdokumentation über Kommentarköpfe ist selten.

- Gründe für inkorrekte Kommentare
 - Code ändert sich.
 - Kommentare werden nicht automatisch überprüft.
 - Programmierer sind Menschen:
Synchronität von Code und Kommentaren ist illusorisch.
 - Folgen – wenn der Fehler erkannt wird
 - Verwirrung – Was wollte der Autor?
 - Verunsicherung – Kann man dem Code vertrauen?
 - Folgen – wenn der Fehler unentdeckt bleibt
 - falsche Verwendung des Codes
 - unerklärliches Verhalten des Programms
- 👉 Vertrauensverlust durch zu viele falsche Kommentare

“ *The proper use of comments is to compensate for our failure to express ourself in code. [...]
Comments are always failures.*

– Robert C. Martin

- Kommentare sind viel zu oft falsch.
 - Synchronität zwischen Code und Kommentaren ist eine unerreichbare Illusion.
- Kommentare lenken vom Code ab.
 - Widersprüche zwischen Kommentaren und Code sind im besten Fall verstörend, im schlimmsten Fall irreführend.
- Die Wahrheit liegt im Code.
 - Kommentare werden nicht automatisch überprüft.

Arten von Dokumentation im Code

Argumente für und gegen Kommentare

Regeln für Kommentare

Werkzeuge zur automatischen Erzeugung

- nicht das Offensichtliche kommentieren
 - nicht den Code wiederholen
 - gilt auch für Schnittstellen von Funktionen
- Funktionen und globale Daten kommentieren
 - wenn der Kontext nicht aus dem Namen klar wird
- schlechten Code nicht kommentieren – neu schreiben!
 - Kommentare kompensieren nicht mangelnde Qualität.
- dem Code nicht widersprechen
 - auf Aktualität und Korrektheit der Kommentare achten
- verdeutlichen, nicht verwirren
 - Code ist präzise, Kommentare sollten es ebenso sein.

- Code wird viel häufiger gelesen als geschrieben.
 - Daseinsberechtigung nur für relevante Kommentare
 - Code hat eine hohe Informationsdichte und ist präzise.
 - Kommentare sollten ähnlich präzise sein wie Code.
 - Akt der Höflichkeit gegenüber dem Leser
 - Code zu lesen erfordert Konzentration.
 - Konzentration sollte nicht durch unwichtige Informationen oder unnötig lange Kommentare verringert werden
 - Wertschätzung: Gute Kommentare kosten Zeit.
 - ähnlich wie die Benennung von Variablen, Funktionen etc.
 - Der erste Versuch ist selten gut genug.
 - Zeit für gute Kommentare zahlt sich vielfach aus.
- ☛ Sprachbeherrschung ist eine notwendige Voraussetzung.

- Idealfall: geschliffenes Englisch
 - erfordert sehr gute Sprachbeherrschung
 - präzise und einfache Formulierungen
- Normalfall: einfaches Englisch
 - Die Leserschaft von Code ist oft international.
 - auch von Nicht-Muttersprachlern erreichbar
- Notfall: geschliffene Muttersprache
 - sehr viel besser als schlechtes (falsches) Englisch
 - Es gibt Übersetzer . . .
- unbedingt vermeiden: schlechte sprachliche Qualität
 - gilt für Fremd- und Muttersprache
 - Zeichen für mangelnde Professionalität

- Erster Satz beschreibt die Funktion
 - *nicht* nur die Signatur (wörtlich) wiederholen
 - hilfreich für den korrekten Zuschnitt der Funktion
- Konsistenz
 - Voraussetzung für den Einsatz von Werkzeugen zur automatischen Erzeugung von Dokumentation
 - Hilfestellung durch Vorlagen im Editor
- sprachspezifische Konventionen beachten
 - gilt für Formatierung und Inhalte
 - Beispiel: PEP 257 in Python
- keine unnötigen Formatierungsstrings
 - stören (empfindlich) die Lesbarkeit und Übersichtlichkeit
 - Kommentare sollten *immer* als reiner Text lesbar sein.

Listing: (Gekürzter) Kommentarkopf einer Python-Funktion

```
def add_to_toctree(filename="", entries=None, sort=False, after=""):
    """
    Add entries to toc tree.

    Adding lines to toctrees in documentation generated with Sphinx is a
    frequent task for a metacode package. A toctree in Sphinx typically
    looks similar to the following:

    .. code-block:: rst

        .. toctree::
           :maxdepth: 1

           first_entry

    The ``.. toctree::`` directive is followed by (optional) parameters,
    the actual entries appear after a blank line and are indented.

    This function allows to add arbitrary entries to a given toctree,
    as entries are given as list.

    ...
    """
```

Formatierung von Kommentarköpfen

Anzeige eines Kommentarkopfs in der IDE PyCharm



```
1061      utils.add_to_tocree(
1062          filename=pymetacode.utils
1063          entries=
1064          sort=
1065      )
1066
1067  def _create_m
1068  window_cr
1069  window_cr
1070  window_cr
1071
1072  def _update_m
1073  package_n
1074  context =
1075  context[
1076  template
1077  templ
1078  conte
```

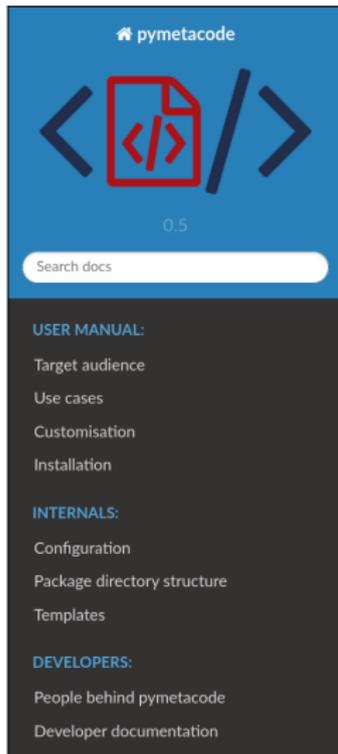
Add entries to toc tree.

Adding lines to toctrees in documentation generated with Sphinx is a frequent task for a metacode package. A toctree in Sphinx typically looks similar to the following:

```
.. code-block:: rst
.. toctree::
:   maxdepth: 1
   first_entry
```

The ``.. toctree::`` directive is followed by (optional) parameters, the actual entries appear after a blank line and are indented.

This function allows to add arbitrary entries to a given toctree, as entries are given as list.



```
pymetacode.utils.add_to_toctree(filename="", entries=None, sort=False, after=")
```

Add entries to toc tree.

Adding lines to toctrees in documentation generated with Sphinx is a frequent task for a metacode package. A toctree in Sphinx typically looks similar to the following:

```
.. toctree::
   :maxdepth: 1

   first_entry
```

The `.. toctree::` directive is followed by (optional) parameters, the actual entries appear after a blank line and are indented.

This function allows to add arbitrary entries to a given toctree, as entries are given as list.

If you would like to have the toctree entries sorted alphabetically, make sure to set the `sort` parameter to `True`.

In case of several toctrees in one document, you may use the `after` parameter to provide a string as a marker. This string is searched for treating it as substring through all lines of the file, and the first matching line used as actual offset.

In case of the file not containing any toctree directive or the string provided by `after` not being found, exceptions will be thrown.

- Parameters:**
- `filename (str)` - Name of the file containing the toctree
 - `entries (list)` - lines to be added to the toctree
- Note that regardless of the leading whitespace, entries are left stripped

“ *Don't comment bad code – rewrite it.*

– Brian W. Kernighan, P. J. Plauger

- Kommentare sind keine Lösung für schlechten Code.
 - Guter Code braucht keine erläuternden Kommentare.
 - Guter Code ist lesbar und selbsterklärend.
- Guter Code ist das Ergebnis harter Arbeit.
 - setzt Bewusstsein, Wissen und Disziplin voraus
 - Fähigkeiten nehmen mit der Erfahrung zu.
- Kommentare sind eine temporäre Notlösung.
 - Schlechter Code *ohne* Kommentare ist noch schlimmer.
 - Schlechter Code mit *falschen* Kommentaren ist das Schlimmste.

- **Kommentarköpfe**
 - hilfreich für alle vom Nutzer aufrufbaren Funktionen
 - selten(er) wichtig für private Funktionen/Methoden
 - werden bei selbstdokumentierenden Signatures überflüssig (gilt i.d.R. nicht für vom Nutzer aufrufbare Funktionen)
 - nicht nur Namen von Funktion und Parametern wiederholen
- **Copyright-Hinweise**
 - sinnvoll (aber: durch VCS und LICENSE nicht zwingend)
 - wenn, dann mit Hinweis auf Lizenz
- **einzeilige Kommentare im Code**
 - vermeiden: Code umschreiben und lesbarer machen
 - müssen zwingend einen Mehrwert haben
 - besonders auf Korrektheit und Aktualität achten

Arten von Dokumentation im Code

Argumente für und gegen Kommentare

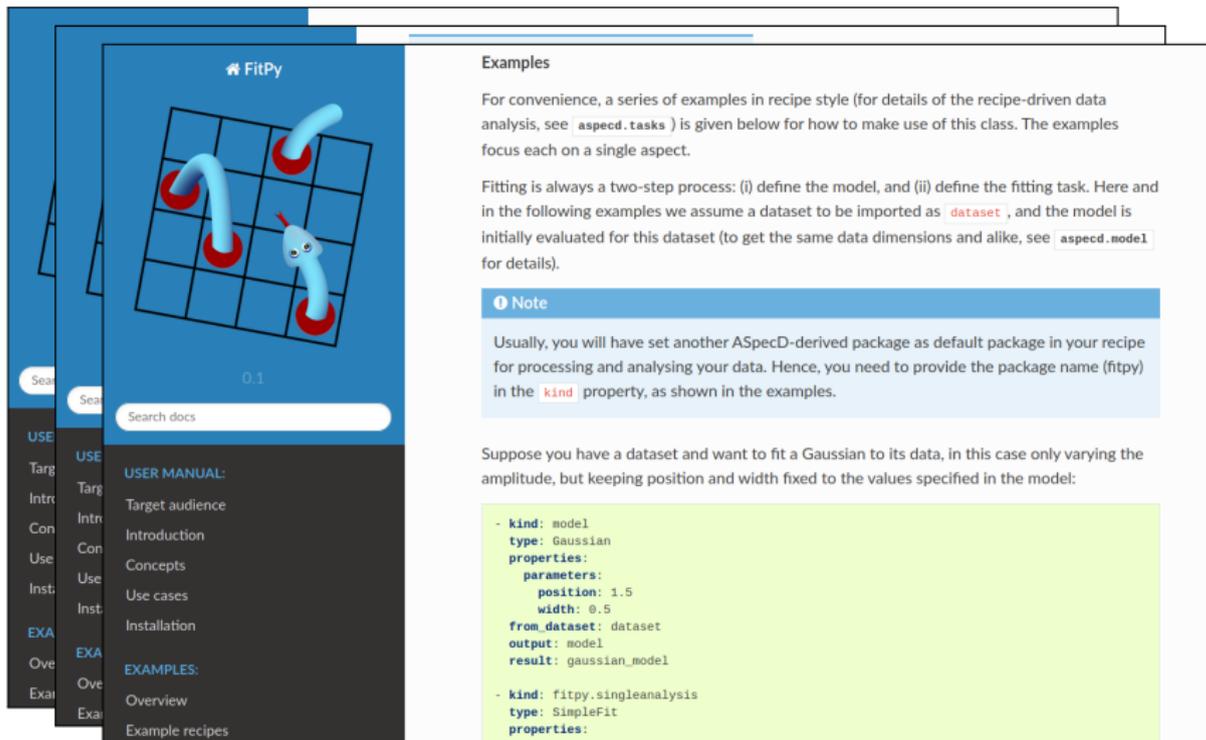
Regeln für Kommentare

Werkzeuge zur automatischen Erzeugung

- Einsatzgebiete
 - automatische Erzeugung einer Entwicklerdokumentation
 - Übersicht über die Schnittstellen aller Funktionen
 - Übersicht über alle Klassen
 - Vor- und Nachteile
 - immer so aktuell wie die Kommentare im Code
 - potentiell große Wirkung mit eher geringem Aufwand
 - automatische Erzeugung einer lesbaren Dokumentation
 - eher nicht für Nutzerdokumentation geeignet
 - Voraussetzungen
 - konsistente Formatierung
 - Verwendung spezifischer Formatierungen
- ☛ Werkzeuge erstellen „nur“ eine gut formatierte Ausgabe.
Die Kommentarköpfe müssen selbst geschrieben werden!

- Jede Sprache hat ihre Konventionen.
 - betrifft Struktur, Formatierung, verwendete Werkzeuge
 - Konventionen der jeweiligen Sprache verwenden
 - Bsp.: Java – JavaDoc; Python – Sphinx
 - Unterschiede in Inhalt und Zielstellung
 - entwicklerzentriert: reine Schnittstellendokumentation
 - nutzerzentriert: inkl. Erklärungen und Beispiele
 - unterschiedliche Möglichkeiten der Textauszeichnung
 - meist spezifische Formatierungsanweisungen
 - sollte die Lesbarkeit nicht beeinträchtigen
 - Beispiele: Markdown, reStructuredText
- 👉 Konventionen der jeweiligen Sprache zumindest kennen

- Integration in den Editor
 - Unterstützung sprachspezifischer Kommentarköpfe
 - Vorlagen, die nur noch ausgefüllt werden müssen
 - Vorlagen sollten anpassbar sein
 - erspart sehr viel Tipparbeit
 - sorgt für (weitgehend) konsistente Formatierung
 - Integration in die Build-Infrastruktur
 - Generation der Dokumentation automatisierbar
 - weiterführende Werkzeuge: Aufrufgraphen (*call graphs*) etc.
 - Integration konzeptioneller (externer) Dokumentation ggf. möglich
 - Nutzerdokumentation, allgemeinere Beschreibung, ...
- ☞ Automatisierte und häufige Erzeugung gut formatierter Dokumentation motiviert: Ja, man kann schon was sehen!



FitPy

0.1

Search docs

Examples

For convenience, a series of examples in recipe style (for details of the recipe-driven data analysis, see `aspecd.tasks`) is given below for how to make use of this class. The examples focus each on a single aspect.

Fitting is always a two-step process: (i) define the model, and (ii) define the fitting task. Here and in the following examples we assume a dataset to be imported as `dataset`, and the model is initially evaluated for this dataset (to get the same data dimensions and alike, see `aspecd.model1` for details).

Note

Usually, you will have set another ASpecD-derived package as default package in your recipe for processing and analysing your data. Hence, you need to provide the package name (fitpy) in the `kind` property, as shown in the examples.

Suppose you have a dataset and want to fit a Gaussian to its data, in this case only varying the amplitude, but keeping position and width fixed to the values specified in the model:

```
- kind: model
  type: Gaussian
  properties:
    parameters:
      position: 1.5
      width: 0.5
  from_dataset: dataset
  output: model
  result: gaussian_model

- kind: fitpy.singleanalysis
  type: SimpleFit
  properties:
```



- 🔑 Dokumentation im Code sollte auf das notwendige Minimum beschränkt werden.
- 🔑 Veraltete oder falsche Dokumentation schadet mitunter mehr als sie nützt
- 🔑 Formulierungen sollten so knapp und präzise wie möglich, die Formatierung konsistent und übersichtlich sein.
- 🔑 Unklarer Code sollte nicht dokumentiert, sondern umgeschrieben werden.
- 🔑 Dokumentationswerkzeuge erleichtern die Arbeit, setzen aber konsistente Formatierung voraus.