

Wissenschaftliche Softwareentwicklung

4. Editoren/IDEs

Till Biskup

Physikalische Chemie

Universität Rostock

03.11.2023





- 🔑 Programmierung bedeutet (meist) das Erzeugen von reinem Text. Im Prinzip ist also jeder Texteditor geeignet.
- 🔑 Programmierer verbringen viel Zeit mit dem Editor. Die Wahl kann für die Produktivität entscheidend sein.
- 🔑 Moderne Editoren bringen Funktionalität mit, die die Programmierung erleichtert (und Fehlern vorbeugt).
- 🔑 IDEs integrieren viele Werkzeuge in einer Oberfläche. Die Komplexität bedingt eine steile initiale Lernkurve.
- 🔑 IDEs ersetzen nicht die solide Kenntnis des Umgangs mit den grundlegenden Programmierwerkzeugen.

Motivation: Warum ist die Wahl des Editors wichtig?

Mindestanforderungen an einen Editor/eine IDE

Editoren vs. Entwicklungsumgebungen (IDEs)

Warum ist die Wahl des Editors wichtig?

Der Programmierer verbringt die meiste Zeit mit seinem Editor.



Gründe für eine sorgfältige Wahl des Editors

- Programme bestehen (meist) aus reinem Text.
 - Im Prinzip ist jeder Texteditor geeignet.
 - Editoren zum Programmieren sind Texteditoren.
- Programmierer verbringen die meiste Zeit mit dem Editor.
 - Das Programm kann die Produktivität entscheidend beeinflussen.
 - Vertrautheit mit dem verwendeten Editor ist entscheidend.
- Code wird viel häufiger gelesen als geschrieben.
 - Gute Editoren erleichtern (erheblich) die Lesbarkeit und erhöhen dadurch die Produktivität.
- Gute Editoren steigern die Codequalität
 - Unterstützung von Refactoring und automatisierten Tests

👉 Tipp: sorgfältig ausprobieren und bewusst entscheiden

Warum ist die Wahl des Editors wichtig?

Der Editor ist das wichtigste Werkzeug des Programmierers.



Der Wert guter Werkzeuge

“ *Every craftsman starts his or her journey with a basic set of good-quality tools. [...]*

Tools amplify your talent.

The better your tools,

and the better you know how to use them,

the more productive you can be.

Start with a basic set of generally applicable tools.

– Andrew Hunt, David Thomas

☛ Der Editor ist das wichtigste Werkzeug des Programmierers.

Motivation: Warum ist die Wahl des Editors wichtig?

Mindestanforderungen an einen Editor/eine IDE

Editoren vs. Entwicklungsumgebungen (IDEs)

Grundlegende Eigenschaften guter Editoren

- (komplett) mit der Tastatur bedienbar
 - Hände können immer auf der Tastatur bleiben.
 - beschleunigt (erheblich) das Arbeiten
- konfigurierbar
 - Jeder hat andere Vorstellungen und Vorlieben.
 - Bsp.: Schriftarten, Farben, Fenstergrößen, Tastaturkürzel
- erweiterbar
 - Unterstützung weiterer (Programmier-)Sprachen
 - Integration mit beliebigen Compilern
- programmierbar
 - Automatisierung wiederkehrender Abläufe
 - Möglichkeiten: Makros oder (eingebaute) Skriptsprachen

Spezifische Eigenschaften für jede Programmiersprache

- Syntaxhervorhebung
 - erhöht (wesentlich) die Lesbarkeit
- Autovervollständigung
 - beugt (konsequent genutzt) Tippfehlern vor
 - erspart mitunter den Blick in die Dokumentation
- automatische Einrückung
 - konsistentes Erscheinungsbild
 - wichtig: sollte konfigurierbar sein
- Vorlagen/Textbausteine
 - spart Tipparbeit (Beschleunigung) und sorgt für Konsistenz
- Refactoring
 - erhöht wesentlich die Lesbarkeit und Qualität von Code

Listing: Beispiel mit Syntax-Hervorhebung

```
def redo(self):
    """Reapply previously undone processing step.

    Raises
    -----
    RedoAlreadyAtLatestChangeError
        Raised when trying to redo with empty history
    """
    if self._history_pointer == len(self.history) - 1:
        raise RedoAlreadyAtLatestChangeError
    processing_step_record = \
        self.history[self._history_pointer + 1].processing
    processing_step = processing_step_record.create_processing_step()
    processing_step.process(self)
    self._increment_history_pointer()

def _has_leading_history(self):
    if len(self.history) - 1 > self._history_pointer:
        return True
    else:
        return False
```

Motivation: Warum ist die Wahl des Editors wichtig?

Mindestanforderungen an einen Editor/eine IDE

Editoren vs. Entwicklungsumgebungen (IDEs)

integrierte Entwicklungsumgebung

engl. *integrated development environment* (IDE),
gemeinsame Oberfläche für Werkzeuge zur Programmierung,
um Softwareentwicklung ohne Medienbrüche zu ermöglichen

- eine Oberfläche für (fast) alle Aufgaben
 - Editor, Compiler, Interpreter, Debugger, VCS, ...
 - Hilfe/Dokumentation, Steuerung externer Services, ...
- Integration existierender Werkzeuge
 - Compiler, Interpreter, Debugger, ...
 - meist weitestgehend konfigurierbar

Editoren vs. IDEs

Was unterscheidet eine IDE von einem Editor?



The screenshot displays the ASpecD IDE environment. The main window shows a Python file named `dataset.py` with the following code:

```
244
245
246
247
248 def has_leading_history(self):
249     if len(self.history) - 1 > self.history_pointer:
250         return True
251     else:
252         return False
253
254 @staticmethod
255 def _create_processing_history_record(processing_step):
256     historyrecord = history.ProcessingHistoryRecord(processing_step)
257     return historyrecord
258
259 def _append_processing_history_record(self, history_record):
260     self.history.append(history_record)
261     self._increment_history_pointer()
262
263 def _increment_history_pointer(self):
264     self.history_pointer += 1
265
266 def _decrement_history_pointer(self):
267     self.history_pointer -= 1
268
269 def _replay_history(self):
270     self.data = self._origdata
271     for historyentry in self.history[self.history_pointer]:
272         historyentry.replay(self)
273
274 def strip_history(self):
275     """Remove leading history, if any.
```

The interface also includes a file explorer on the left showing a project structure with files like `__init__.py`, `analysis.py`, `annotation.py`, `axis.py`, `data.py`, `dataset.py`, `history.py`, `importer.py`, `infofile.py`, `internal.py`, `metadata.py`, `plotting.py`, `processing.py`, `system.py`, `utils.py`, `ASpecD.egg-info`, and `docs`. At the bottom, a terminal window shows the command `python -m unittest discover` and the output `Tests passed: 259 of 259 tests - 24 ms`. An event log on the right shows the test execution timeline.

Einige zusätzliche Fähigkeiten von IDEs

- Refactoring
 - komplexe Ersetzungen/Umbenennungen, Modularisierung, ...
 - Integration der Buildumgebung
 - Compiler, Linker, ggf. Interpreter
 - Debugger
 - interaktive Fehlerdetektion und Behebung
 - Hilfe
 - kontextspezifisch und Sprachdokumentation
 - Integration weiterer externer Komponenten
 - Bsp.: VCS, Datenbanken, Webserver; Tests
- ☞ Grenze zwischen Editor und IDE mitunter fließend

Editoren vs. IDEs

Beispiel: Autovervollständigung von Code und Dokumentationsvorschau



```
1 usage
2
3 def _create_splash_file(self):
4     if not self.configuration.gui['splash']:
5         return
6     template = utils.Template(
7         path='',
8         template='splash.12.svg',
9         context=self.configuration_to_dict(),
10        destination=os.path.join(self._src_dir, 'gui', 'images',
11                                'splash.svg'),
12    )
13
14    template.create(self)
15
16    1 usage
17    2
18    3 append(self)
19    4 path
20    5 def _copy
21    6 _template
22    7 _render(self)
23    8 for in
24    9 context
25    10 destination
26    11 environment
27    12 ___init__(self, path, template, context, destination)
28    13 ___annotations__
29    14 ___class__
30    15 object
31    16 object
32
33 1 usage
34 def _create_documentation_index()
35 self._create_documentation_index()
36 self._add_submodule_documentation()
37 modules = ['app', 'mainwindow']
38 for module in modules:
39     self._create_api_documentation(module)
40 self._add_api_documentation_to_toctree()
41
42 1 usage
43 def _create_documentation_index(self):
44     package_name = self.configuration.package['name']
45     context = self.configuration_to_dict()
46     context['subpackage'] = {'name': f'{package_name}.gui'}
```

```
1
2
3 contents = utils.get_package_data('Makefile.gui')
4 filepath = os.path.join(self._src_dir, 'gui', 'Makefile')
5 with open(filepath, 'w', encoding='utf-8') as file:
6     file.write(contents)
7
8 1 usage
9
10 def _create_test_modules(self):
11     modules = ['app', 'mainwindow']
12     for module in modules:
13         template = utils.Template(
14             path='code',
15             template=f'gui_{module}.j2',
16             context=self.configuration_to_dict(),
17             destination=os.path.join(self._src_dir, 'code',
18                                     f'{module}.py')
19         )
20         template.create()
21
22 1 usage
23
24 def _create_test_modules(self):
25     modules = ['app', 'mainwindow']
26     for module in modules:
27         context = self.configuration_to_dict()
28         context[module] = {}
29         template = utils.Template(
30             path='code',
31             template=f'test_{module}.j2',
32             context=context,
33             destination=os.path.join(self._src_dir, 'code',
34                                     f'test_{module}.py')
35         )
36         template.create()
37
38 1 usage
39
40 def _create_splash_file(self):
41     if not self.configuration.gui['splash']:
42         return
43     template = utils.Template(
44         path='',
45         template='splash.12.svg',
46         context=self.configuration_to_dict(),
47         destination=os.path.join(self._src_dir, 'gui', 'images',
48                                 'splash.svg'),
49     )
50     template.create(self)
51
52 1 usage
53
54 def _create_documentation_index(self):
55     package_name = self.configuration.package['name']
56     context = self.configuration_to_dict()
57     context['subpackage'] = {'name': f'{package_name}.gui'}
```

pyemacode.utils.Template

`def __init__(self, path: str = '', template: str = '', context: Any = None, destination: str = '') -> None`

Wrapper for using the template engine (Jinja2).

Dealing with templates requires a number of settings to be made, namely the source of the template, its name, the context (i.e. the dictionary containing the variables to be replaced within the template), and the destination the rendered template should be output to.

Note

Regarding the path of a template, three different places are looked up, using the function `get_package_data`: first in the user data directory, then in the site data directory, and finally within the package (distribution). For details see the description of the function `get_package_data`.

Examples

Probably the best way to use the class is to instantiate an object providing all necessary parameters:

```
template = Template(
    path='some/relative/path/to/the/template',
    template='name_of_the_template',
    context=dict(),
    destination='name_of_the_file_to_output_rendered_template_to',
)
```

```
Run: Python tests in tests >
Tests failed: 1, passed: 2,594, ignored: 10 of 2,605 tests - 25 sec 23 ms

Test Results 25 sec 23 ms
  test_utils 4 ms
    TestGetVersion 2 ms
      test_version_correct_for_aspecd_2ms
        Expected :0.9.0.dev72
        Actual   :0.9.0.dev69
        <Click to see difference>

Traceback (most recent call last):
  File "/home/till/Programmierung/Python/aspecd/tests/test_utils.py", line 324, in test_version_correct_for_aspecd
    self.assertEqual(utils.get_aspecd_version(), version)
AssertionError: '0.9.0.dev72' != '0.9.0.dev69'
```

```
Run: Python tests in tests >
Tests passed: 2,595, ignored: 10 of 2,605 tests - 25 sec 563 ms

Test Results 25 sec 563 ms
Testing started at 09:07 ...
Launching unittests with arguments python -m unittest discover -s /home/till/Programmierung/Python/aspecd/tests -t

Skipped
/home/till/Programmierung/Python/aspecd/aspecd/plotting.py:1027: RuntimeWarning: More than 20 figures have been open
  self.figure, self.axes = plt.subplots()

Skipped

Skipped
```

Vorteile von IDEs

- alles aus einer gemeinsamen Oberfläche erreichbar
- Beschleunigung und Vereinfachung von Abläufen
- manche Aspekte schwer in reinem Editor realisierbar

Nachteile von IDEs

- steile (initiale) Lernkurve
- verstecken sehr viel vor dem Nutzer
- ☛ Kosten-Nutzen-Abwägung, abhängig von
 - Umfang und Dauer des Projekts
 - Erfahrung des Nutzers

“ *Many new programmers make the mistake of adopting a single power tool, such as a particular integrated development environment (IDE), and never leave its cozy interface.*

This really is a mistake.

We need to be comfortable beyond the limits imposed by an IDE. The only way to do this is to keep the basic tool set sharp and ready to use.

– Andrew Hunt, David Thomas

- ☛ Nichts kann die solide Kenntnis des Umgangs mit grundlegenden Programmierwerkzeugen ersetzen.



- 🔑 Programmierung bedeutet (meist) das Erzeugen von reinem Text. Im Prinzip ist also jeder Texteditor geeignet.
- 🔑 Programmierer verbringen viel Zeit mit dem Editor. Die Wahl kann für die Produktivität entscheidend sein.
- 🔑 Moderne Editoren bringen Funktionalität mit, die die Programmierung erleichtert (und Fehlern vorbeugt).
- 🔑 IDEs integrieren viele Werkzeuge in einer Oberfläche. Die Komplexität bedingt eine steile initiale Lernkurve.
- 🔑 IDEs ersetzen nicht die solide Kenntnis des Umgangs mit den grundlegenden Programmierwerkzeugen.