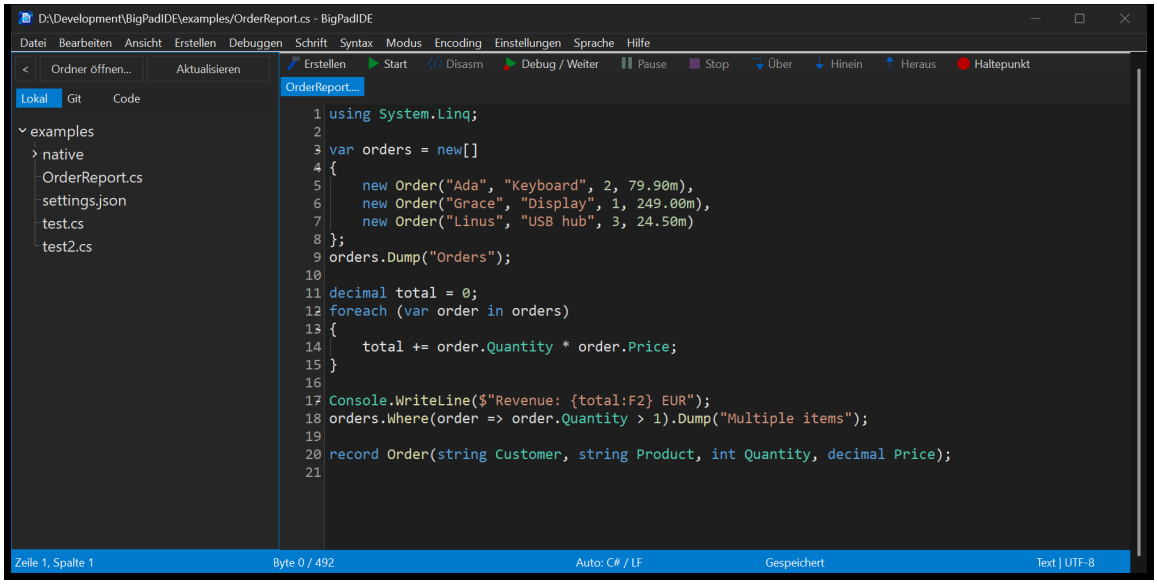
BigPadIDE

Das umfassende Benutzerhandbuch

Vom ersten Dokument bis zu Debugging, Git und IL-/C#-Analyse



Native Windows-Arbeitsumgebung mit Explorer und C#-Editor.

Deutsch | Alle vier Oberflächenstile

Entwicklung: **Rudolf Stepan**
rstepan123@gmail.com

Ausgabe 14. November 2026 | Version 1.2.4

Mit Praxisanleitungen, Screenshots, Tastaturreferenz und Fehlerhilfe.

Vorwort

BigPadIDE will bestehende Werkzeuge nicht ersetzen. Bewährte Entwicklungsumgebungen, Editoren, Debugger und spezialisierte Analyseprogramme haben ihren festen Platz. Dieses Tool versteht sich als Ergänzung: als ein Schweizer Taschenmesser für den Entwicklungsalltag.

Die Idee dahinter ist einfach: Viele Aufgaben beginnen mit einer kleinen Frage. Was steht in dieser Datei? Was macht dieser Code? Welche Bytes liegen an diesem Offset? Wie sieht eine Methode als IL oder rekonstruiertes C# aus? Welche Änderung ist im Git-Diff tatsächlich enthalten? Für solche Fragen bündelt BigPadIDE viele nützliche Funktionen in einer gemeinsamen, direkt zugänglichen Oberfläche.

Wie bei einem Schweizer Taschenmesser liegt der Nutzen im Zusammenspiel der Werkzeuge. Ein kurzes C#-Programm ausprobieren, eine Binärdatei untersuchen, einen Build starten oder eine Änderung vergleichen: Das soll ohne großen Umweg möglich sein. Für umfangreiche oder besonders spezialisierte Aufgaben bleiben die dafür gewählten Werkzeuge selbstverständlich Teil der eigenen Arbeitsweise.

BigPadIDE soll Raum zum Ausprobieren, Lernen und Verstehen bieten. Sie entscheiden, welche Funktionen Ihnen helfen und wo andere Programme besser zu Ihrer Aufgabe passen. Es geht um eine praktische Erweiterung Ihrer Werkzeugkiste.

Dieses Handbuch hilft Ihnen dabei, die einzelnen Werkzeuge kennenzulernen und sie sinnvoll miteinander zu verbinden.

Inhaltsverzeichnis

Kapitel, Abschnitte und Seitenzahlen sind anklickbar. Zusätzlich können Sie die PDF-Lesezeichen verwenden.

Vorwort 2

1. Über dieses Handbuch und der erste Start 11

1.1 So finden Sie den richtigen Einstieg . . . 11

1.2 Bedienkonventionen . . . 12

1.3 In zehn Minuten zum ersten Ergebnis . . . 12

2. Installation und Arbeitsumgebung 13

2.1 Setup und Installationsumfang . . . 13

2.2 Welche Komponente wird wofür gebraucht? . . . 14

2.3 Erstkonfiguration prüfen . . . 14

3. Oberfläche, Themes und Ansichten 15

3.1 Schrift, Syntax und Sprache . . . 15

3.2 Themes und Hexansicht . . . 16

3.3 Schriftgröße der Ordneransicht . . . 19

3.4 Mehrere Bereiche im Split-Screen . . . 19

3.5 Ansichten als Tab oder Fenster öffnen . . . 20

3.6 Einstellungen bewusst speichern . . . 21

4. Dateien bearbeiten und sicher speichern 22

4.1 Auswahl, Bearbeitung und Dateiformat . . . 22

4.2 Datei-Tabs und ungespeicherte Änderungen . . . 22

4.3 Externe Dateiänderungen . . . 22

4.4 Dateien und Ordner im Explorer anlegen . . . 23

4.5 Dateien und Ordner umbenennen und verschieben . . . 23

4.6 Große Dateien und lange Zeilen . . . 24

4.7 Sicherer Umgang mit externen Änderungen . . . 25

5. Navigation und Suche 26

5.1 Suche im Dokument . . . 26

5.2 Gehe zu Zeile oder Offset . . . 26

5.3 Schnell öffnen . . . 26

5.4 Projektweite Suche und Regex . . . 27

6. C# schreiben, bauen und ausführen 30

6.1 Compilerfehler . . . 30

6.2 Einzeldatei oder Projekt? . . . 31

6.3 Beispiel mit Eingabe und Prüfung . . . 31

7. Solutions, aktives Projekt und C#-Werkzeuge 32

7.1 Aktives Projekt per Maus und Tastatur . . . 32

7.2 Typischer Ablauf bei mehreren Projekten . . . 32

7.3 C#-Projektwerkzeuge . . . 33

8. Programmausgabe, Ergebnisse und Startprofile 37

8.1 Tabellen und Programmausgabe . . . 37

8.2 Build-/Run-Profile und Eingabe . . . 38

8.3 Beispiel einer Ergebnistabelle . . . 40

8.4 Beispiel eines gespeicherten Run-Profils . . . 41

9. C#-Debugging 42

9.1 Die erste Debugsitzung . . . 42

9.2 Haltepunkte und Watch-Ausdrücke . . . 42

9.3 Starten und Anhängen unterscheiden . . . 43

9.4 Werte unter dem Mauszeiger, Objektbrowser und Variablen ändern . . . 44

10. Native Projekte: C, C++ und Assembler 46

10.1 ASM, C und C++ bauen und starten . . . 46

10.2 Ein Projekt aus mehreren Quellen . . . 46
10.3 Die sechs mitgelieferten Demos . . . 47
10.4 Werkzeugpfade ausdrücklich setzen . . . 47

11. Natives Debugging 48

11.1 Natives Debugging . . . 48
11.2 Fehlende Quellpositionen einordnen . . . 49

12. Hex-Editor und Bytewerkzeuge 50

12.1 Text, Bytes und Offset . . . 50
12.2 Hex-Werkzeuge: Bytesuche, Werte und Export . . . 50
12.3 Zahlen richtig interpretieren . . . 51
12.4 Bereichsexport als eigener Arbeitsstand . . . 51

13. Nativer Disassembler und PE/ELF 52

13.1 Disassembler . . . 52
13.2 PE-/ELF-Disassembler und Sprungziele . . . 52
13.3 Welche Ansicht passt zur Datei? . . . 54

14. .NET-Assemblies: IL und C# rekonstruieren 55

14.1 .NET-Assemblies: IL und rekonstruiertes C# . . . 55
14.2 Ein vollständiger Analyseablauf . . . 57
14.3 Was die Rekonstruktion aussagt . . . 57
14.4 Schrift und Fensterplatzierung . . . 57

15. Bilder, Frames und Pixelwerte 58

15.1 Bilder bedienen . . . 58
15.2 Ein Bild systematisch prüfen . . . 59

16. Git, Diffs und Zusammenarbeit 60

16.1 Der übliche Commit-Ablauf . . . 60

16.2 Git-Änderungen und Diffs . . . 60
16.3 Einzelne Diff-Abschnitte bearbeiten . . . 61
16.4 Git-Konflikte auflösen . . . 62
16.5 Fetch, Pull und Push . . . 63
16.6 Branches verwalten . . . 65
16.7 Git automatisch aktuell halten . . . 66
16.8 Commit und Dateistaging im Detail . . . 66
16.9 Diff-Fenster und tatsächlicher Dateistand . . . 67

17. Sitzungen und Wiederherstellung 68

17.1 Sitzungswiederherstellung . . . 68
17.2 Absturzwiederherstellung . . . 68
17.3 Sitzungsgrenzen beim Öffnen . . . 69
17.4 Fenster und Ordnerbreite merken . . . 69
17.5 Nach einer Wiederherstellung weiterarbeiten . . . 69

18. Praxisworkshops 70

18.1 Workshop: C#-Fehler finden und korrigieren . . . 70
18.2 Workshop: Das Startprojekt gezielt wechseln . . . 70
18.3 Workshop: Debuggen statt zusätzliche Ausgaben einbauen . . . 70
18.4 Workshop: Eine eigene Assembly untersuchen . . . 71
18.5 Workshop: Native Pipeline und Maschinencode . . . 71
18.6 Workshop: Nur eine Änderung in Git übernehmen . . . 71
18.7 Workshop: Bytebereich und Zahlenwert . . . 72

19. Fehlerhilfe und Diagnose 73

19.1 Installation, C# und Projekte . . . 73
19.2 Ausgabe und Debugging . . . 74
19.3 Dateien, Ansichten und große Daten . . . 75

19.4 Git und Wiederherstellung . . . 76

19.5 Angaben für einen reproduzierbaren Fehlerbericht . . . 76

20. Tastatur- und Menüreferenz 77

20.1 Dateien und Navigation . . . 77

20.2 Bearbeiten und Suchen . . . 78

20.3 Explorer, Ansichten und Git . . . 79

20.4 Build und Debug . . . 80

20.5 Bilder und Disassembler . . . 80

20.6 Die wichtigsten Menüs . . . 81

21. Grenzen und gespeicherte Daten 82

21.1 Wichtige Arbeitsgrenzen . . . 82

21.2 Wo Zustände liegen . . . 83

21.3 Datenstand bewusst unterscheiden . . . 83

22. Glossar, Quellen und weiterführende Hilfe 84

22.1 Begriffe . . . 84

22.2 Herkunft und Gültigkeit . . . 85

22.3 Hilfe und Projekt . . . 85

22.4 Impressum und Kontakt . . . 85

23. Testphase, Kauf und Product-Keys 86

23.1 Testphase und Kaufberechtigung . . . 86

23.2 Product-Key aktivieren . . . 86

23.3 Umzug, Sicherung und Fehlerhilfe . . . 86

24. Die eigene Arbeitsumgebung einrichten 87

24.1 Vier Oberflächen und horizontale Menüs . . . 87

24.2 Schriften und Farben anpassen . . . 87

24.3 Mit Strg und Mausrad bequem lesen . . . 88

24.4 Die gesamte Konfiguration als JSON austauschen . . . 89

24.5 Piece Table oder vollständig im Arbeitsspeicher . . . 89

24.6 Ein persönliches Profil sicher aufbauen . . . 90

25. Toolsets, Compiler und Build-Konfiguration 92

25.1 Einrichten . . . 92

25.2 Gemeinsame Werkzeuge . . . 92

25.3 Compiler und Assembler . . . 93

25.4 Build-Konfiguration und Compileroptionen . . . 93

25.5 Installation zusätzlicher Toolsets . . . 93

26. Semantische Codehilfe und lokale KI 95

26.1 C#-Vervollständigung und graue Vorschläge . . . 95

26.2 Lokale KI einrichten . . . 95

26.3 Inline-Vorschlag und Chat verwenden . . . 96

27. Architektur und grafische Diagramme 97

27.1 Gespeicherten Code analysieren . . . 97

27.2 UML, Ablauf und Systemlandschaft zeichnen . . . 97

27.3 Speichern und austauschen . . . 98

27.4 Explorer und Assembly-Vorschau . . . 98

28. Lizenzen und Veröffentlichungsstand 99

Bildreferenz: weitere Ansichten 100

Abbildungsverzeichnis

Abbildung 1. Moderne dunkle Oberfläche mit Explorer, Editor und Toolbar 15

Abbildung 2. Moderne helle Oberfläche derselben Arbeitsumgebung 16

Abbildung 3. Sepia mit warmen Editor- und Explorerfarben 17

Abbildung 4. Mitternacht mit dunkelblauen Arbeitsflächen 17

Abbildung 5. JSON im Dunkelmodus mit Einrückungslinien 18

Abbildung 6. Dieselbe Datei in der editierbaren Hexansicht 19

Abbildung 7. Assembly-Ansicht in der App 20

Abbildung 8. Separates Diff-Fenster 21

Abbildung 9. Fortschritt der Originalsicherung 23

Abbildung 10. Umbenannter Tab mit erhaltenen ungespeicherten Änderungen 24

Abbildung 11. Projektweite Regexsuche mit Dateifilter und Trefferpositionen 28

Abbildung 12. Das Suchfenster übernimmt das Mitternacht-Theme 29

Abbildung 13. Rote Unterwellung und zugehöriger Compilerfehler 31

Abbildung 14. Native Arbeitsumgebung mit zugehörigem Projektziel 32

Abbildung 15. Verweise mit Filter und Quelltextvorschau. 34

Abbildung 16. Treffer bleiben in einem eigenen Reiter im Ausgabepanel sichtbar. 34

Abbildung 17. Code Map mit aufgelösten Methodenaufrufen. 35

Abbildung 18. Semantische Codeaktion mit Vorher-/Nachher-Kontext und exakter Ersetzung. 36

Abbildung 19. Bestellungen als Zeilen und Spalten im Ergebnispanel 37

Abbildung 20. Run-Profil mit Arbeitsverzeichnis, Argumenten und Umgebung 39

Abbildung 21. Laufendes Programm mit sichtbarem Prompt und Eingabezeile 40

Abbildung 22. Echter Haltepunkt mit Aufrufliste, lokalen Variablen und ausgewertetem Ausdruck 42

Abbildung 23. Dialog für bedingte Haltepunkte 43

Abbildung 24. Aufgeklappte Collection mit Unterobjekten im Wertefenster 44

Abbildung 25. C#-Laufzeitwert direkt über einer Variablen im Quelltext 45

Abbildung 26. Dialog zum Ändern eines tatsächlichen Debugger-Variablenwerts 45

Abbildung 27. Nativer Haltepunkt, Aufrufliste und ausgewerteter Watch-Ausdruck 49

Abbildung 28. Byte-Interpretation, exakte Suche und Bereichsexport 51

Abbildung 29. PE/ELF 53

Abbildung 30. Symbols 54

Abbildung 31. Rekonstruiertes C# und Typliste 56

Abbildung 32. IL-Ansicht desselben Typs 56

Abbildung 33. Zoom, Pixelinformationen und zweite TIFF-Seite 59

Abbildung 34. Abschnittsnavigation und Git-Aktionen 62

Abbildung 35. Basis, eigene und eingehende Konfliktversion 63

Abbildung 36. Git-Synchronisationsdialog 64

Abbildung 37. Lokale und Remote-Branches verwalten 66

Abbildung 38. Dialog für getrennte Schriften und Anwendungsfarben 88

Abbildung 39. Mit der Maus vergrößertes Debuggerpanel mit echten Sitzungsdaten 89

Abbildung 40. Arbeitsumgebung nach dem Wechsel auf vollständige RAM-Puffer 90

Abbildung 41. Toolsets und gemeinsame Werkzeugpfade 94

Abbildung 42. Semantische Vervollständigung und Tab-Vorschlag 95

Abbildung 43. Natives Chatfenster 96

Abbildung 44. Architekturanalyse aus gespeichertem Code 97

Abbildung 45. Grafischer Diagramm-Editor 98

Abbildung 46. Drittanbieter-Lizenzen offline lesen 99

Abbildung 47. BigPadIDE mit Ordnerbaum und C#-Syntaxfarben 100

Abbildung 48. C#-Debugger mit Haltepunkt und Auswertung 101

Abbildung 49. Debugger im Mitternacht-Theme 101

1. Über dieses Handbuch und der erste Start

Dieses Handbuch beschreibt BigPadIDE als native Windows-Arbeitsumgebung für Text, Quellcode, Binärdaten und Bilder. Es richtet sich sowohl an Einsteiger, die einzelne C#-Programme ausführen möchten, als auch an Entwickler, die mit Solutions, nativen Toolchains, Git und Disassemblern arbeiten.

Dieses Handbuch beschreibt **BigPadIDE 1.2.4, Funktionsstand 14. November 2026**. Es gehört zur überarbeiteten gemeinsamen Anwendung mit vier Oberflächenstilen, einstellbaren Schriften und Farben, JSON-Konfiguration und zwei Dokument-Speichermodi. Die Abbildungen wurden am selben Tag automatisch aus der tatsächlichen Anwendung aufgenommen. Sie zeigen die deutsche Programmoberfläche in unterschiedlichen Stilen. Menüpfade im Text beziehen sich auf die deutsche Programmsprache. Ein älteres Installationspaket kann einen früheren Funktionsstand enthalten.

1.1 So finden Sie den richtigen Einstieg

Ihr Vorhaben	Passendes Kapitel
App einrichten und die Oberfläche verstehen	2 und 3
Dateien bearbeiten, suchen und sicher speichern	4 und 5
C# ausführen oder eine Solution bearbeiten	6 bis 8
Haltepunkte und Variablen untersuchen	9 und 11
C, C++ oder Assembler bauen	10
Bytes, native Programme oder .NET-Assemblies analysieren	12 bis 14
Bilder, GIF-Frames oder TIFF-Seiten betrachten	15
Änderungen vergleichen und mit Git arbeiten	16
Nach einem Neustart oder Absturz weiterarbeiten	17
Etwas praktisch ausprobieren oder ein Problem lösen	18 und 19
Tastenkürzel, Grenzen und Begriffe nachschlagen	20 bis 22
Testphase, Kauf oder Product-Key verstehen	23
Schriften, Farben, Panelhöhe, JSON und Speichermodus anpassen	24
Compiler, SDK und Debugger einrichten	25

1.2 Bedienkonventionen

Fettdruck bezeichnet Schaltflächen und Menüpunkte. Ein Menüpfad wie **Datei** → **Ordner öffnen** wird von links nach rechts gelesen. `Strg+Umschalt+B` bedeutet, dass die Tasten gleichzeitig gedrückt werden. Dateinamen, Befehle und Quellcode stehen in Schreibmaschinenschrift.

Einstellungen → **Oberfläche / Style** wechselt in derselben `BigPadIDE.exe` zwischen Klassisch, Workbench (Retro), Modern - Hell und Modern - Dunkel. Dateien und Arbeitsabläufe bleiben beim Wechsel erhalten. **F1** öffnet die lokale Hilfe in der unter **Sprache** gewählten Sprache. Der Link im Kopf der Hilfeseite wechselt zur anderen Sprache; beide Hilfen und Handbücher funktionieren offline.

1.3 In zehn Minuten zum ersten Ergebnis

1. BigPadIDE starten und unter **Sprache** Deutsch wählen.
2. Unter **Datei** → **Ordner öffnen** einen beschreibbaren eigenen Arbeitsordner laden.
3. Mit **Strg+N** ein Dokument anlegen und den folgenden Code eingeben.
4. Mit **Strg+Umschalt+S** als `Hello.cs` speichern.
5. Mit **Strg+J** das Ausgabepanel öffnen, `Hello.cs` als Ziel wählen und **F6** drücken.
6. In **Programmausgabe** die Ausgabe prüfen. Ein Fehlertext im Build-Tab weist beispielsweise auf ein fehlendes SDK hin; Kapitel 19 hilft bei der Einordnung.

```
using System;

Console.WriteLine("Hallo aus BigPadIDE!");
```

Einzelfile-Ausführung erzeugt ein temporäres Konsolenprojekt. Dieser einfache Einstieg benötigt keine eigene Projektdatei. Für Bibliotheken, NuGet-Pakete und mehrere zusammengehörige Quellen verwenden Sie ein SDK-Projekt; Kapitel 7 erklärt den Unterschied zwischen aktivem Projekt und Analysekontext.

2. Installation und Arbeitsumgebung

Die Anwendung ist für Windows ausgelegt. Die konkrete Verfügbarkeit von Compiler, SDK und Debugadapter bestimmt, welche Sprach- und Debugfunktionen nutzbar sind. Ein reines Textdokument benötigt keine Compilerinstallation.

2.1 Setup und Installationsumfang

Das EXE-Setup kann für den aktuellen Windows-Benutzer oder für alle Benutzer installieren. Die gemeinsame `BigPadIDE.exe` enthält alle vier Oberflächenstile. Zum vollständigen Paket gehören Offline-Hilfe und PDF-Handbücher, der C#-Helfer, ein privates .NET 10 SDK, NetCoreDbg und CodeLLDB. Zusätzliche Projekt-Workloads, NuGet-Pakete und native Compiler richten Sie passend zu Ihren Projekten ein. Das private SDK verändert weder den systemweiten PATH noch andere .NET-Installationen.

Native GCC-/ARM-Toolchains sind optionale Downloads des EXE-Setups. Für separat installierte MSVC-/Intel-Werkzeuge lässt sich anschließend ein Toolset importieren. Kapitel 25 erklärt Auswahl, Erkennung und Prüfung. Ein aus dem Quellstand neu gebautes Setup enthält diesen Stand; bereits vorhandene ältere Pakete werden durch das Erzeugen der Dokumentation nicht nachträglich aktualisiert.

Bei einer Microsoft-Store-Installation liegen die Programmdateien in einem schreibgeschützten Paket. Beispiele vor dem Bearbeiten in einen eigenen Projektordner kopieren. Externe Compiler separat installieren und unter **Einstellungen → Toolsets und Werkzeugpfade** auswählen. Store-Verfügbarkeit und Preis ergeben sich aus dem jeweiligen veröffentlichten Angebot.

Eigene Dateien in einem eigenen Arbeitsordner speichern. Beim manuellen Kopieren einer Installation die Hilfsordner einschließlich `csharp`, `help` und der Debugger beibehalten; eine einzelne EXE ist kein vollständiges Programmpaket.

2.2 Welche Komponente wird wofür gebraucht?

Aufgabe	Benötigte Komponente
Text, Hex und Bilder anzeigen	BigPadIDE und seine Anwendungsdateien
Einzelne C#-Datei bauen und ausführen	Passendes .NET SDK; Einzeldateien ab SDK 8
C#-Projekt analysieren	C#-Helfer, passendes SDK, wiederhergestellte Projektverweise
C# debuggen	C#-Helfer und NetCoreDbg sowie passende Symbole
IL oder C# aus einer Assembly anzeigen	Mitgelieferter C#-Helfer mit Decompiler-Engine
x64 C/C++ bauen	MinGW-w64 GCC/G++, bei Bedarf GNU objcopy
NASM-Quellen bauen	Zusätzlich NASM
ARM-/ARM64-Rohziele bauen	Passende GNU-Cross-Toolchain
Native Programme debuggen	Mitgeliefertes CodeLLDB oder eine vollständig konfigurierte Installation
Git-Status, Commit, Branches und Synchronisation	Git mit `git.exe` im PATH

Bei manueller Verteilung die mitgebauten Hilfsordner nicht von der EXE trennen. Insbesondere benötigt Roslyn den vollständigen `csharp`-Ordner einschließlich `BuildHost-netcore`. Ein allein kopiertes EXE ersetzt kein vollständiges Paket.

2.3 Erstkonfiguration prüfen

1. Eine kleine Textdatei öffnen und speichern.
2. Das C#-Beispiel aus Kapitel 1 ausführen.
3. Für Git einen vorhandenen Repository-Ordner öffnen und zum Git-Tab wechseln.
4. Für native Entwicklung ein Demo in einen eigenen beschreibbaren Ordner kopieren und zunächst nur bauen. Danach Start und Debugging einzeln erproben.

Eine Toolchain, die nach dem Start neu auf dem System-PATH eingetragen wurde, erfordert einen Neustart der App, damit der geerbte PATH aktuell ist. Neue Dateien in den von BigPadIDE direkt durchsuchten Toolchain-Ordnern werden beim nächsten Build ermittelt.

3. Oberfläche, Themes und Ansichten

Der Arbeitsbereich besteht aus Menü/Toolbar, optionalem Explorer links, Dokumenttabs, einem oder mehreren Editorbereichen, Ausgabepanel und Statusleiste. Die Fokusposition entscheidet bei einigen Tasten über ihre Wirkung: **F5** aktualisiert im Explorer, während es im Editor Debugging startet oder fortsetzt.

3.1 Schrift, Syntax und Sprache

Einstellungen → **Schriften und Farben** bietet getrennte Schriftarten und Größen für 13 Bereiche, darunter Editor, Explorer, Menüs, Tabs, Status, Panels, Ausgabetext, Dialoge, IL-/C#-, Hex-, Diff- und Suchansichten. Auch Syntax- und Oberflächenfarben lassen sich anpassen. **Strg + Mausrad** ändert die Schrift im Bereich unter dem Mauszeiger in Ein-Punkt-Schritten, von 6 bis 72 pt. Die bisherigen Schrift-Menüs bleiben als schnelle Auswahl verfügbar. Kapitel 24 erklärt Vorschau, Zurücksetzen und den Austausch der Einstellungen.

Syntax → **Automatisch** bestimmt die Sprache anhand der Datei. Alternativ stehen unter anderem C/C++, C#, Assembler, JSON, Python, XML/HTML und Logdarstellung bereit. Syntaxfarben allein wählen keinen Compiler. Unter **Sprache** Deutsch oder English wählen; die Programmhilfe folgt dieser Einstellung.

Ansicht → **Virtueller Zeilenumbruch** passt die Darstellung an die Bereichsbreite an, ohne zusätzliche Zeilenenden in die Datei zu schreiben.

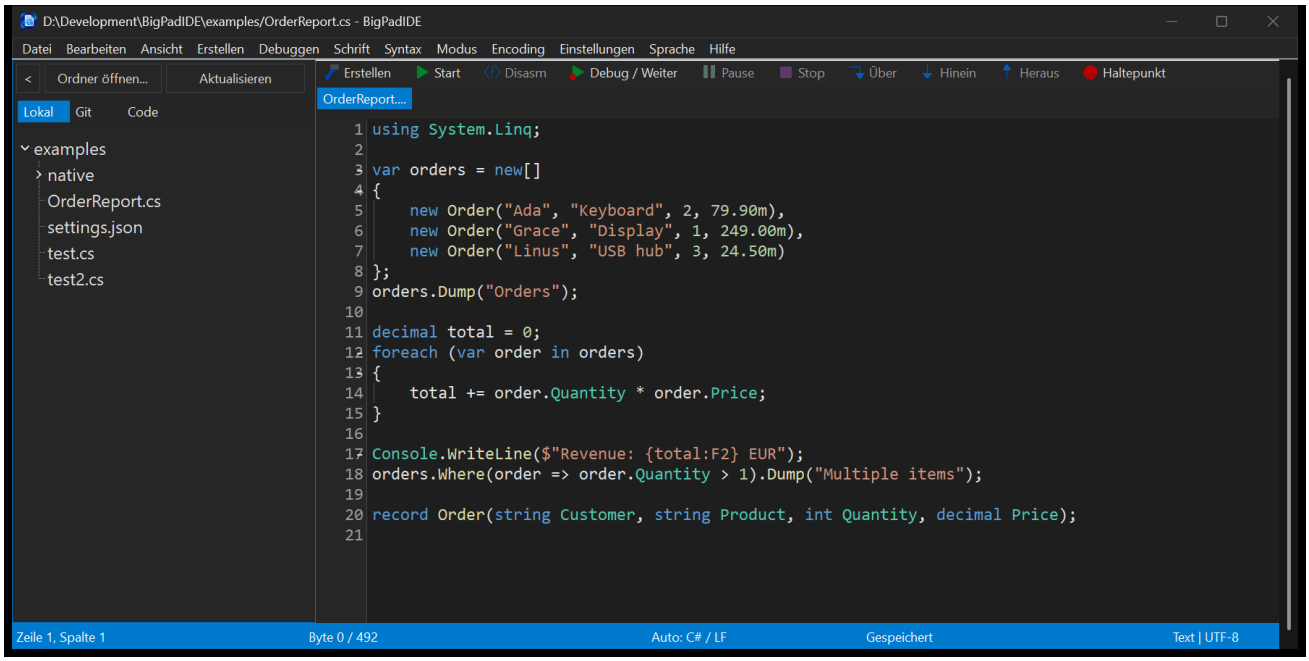


Abbildung 1. Moderne dunkle Oberfläche mit Explorer, Editor und Toolbar

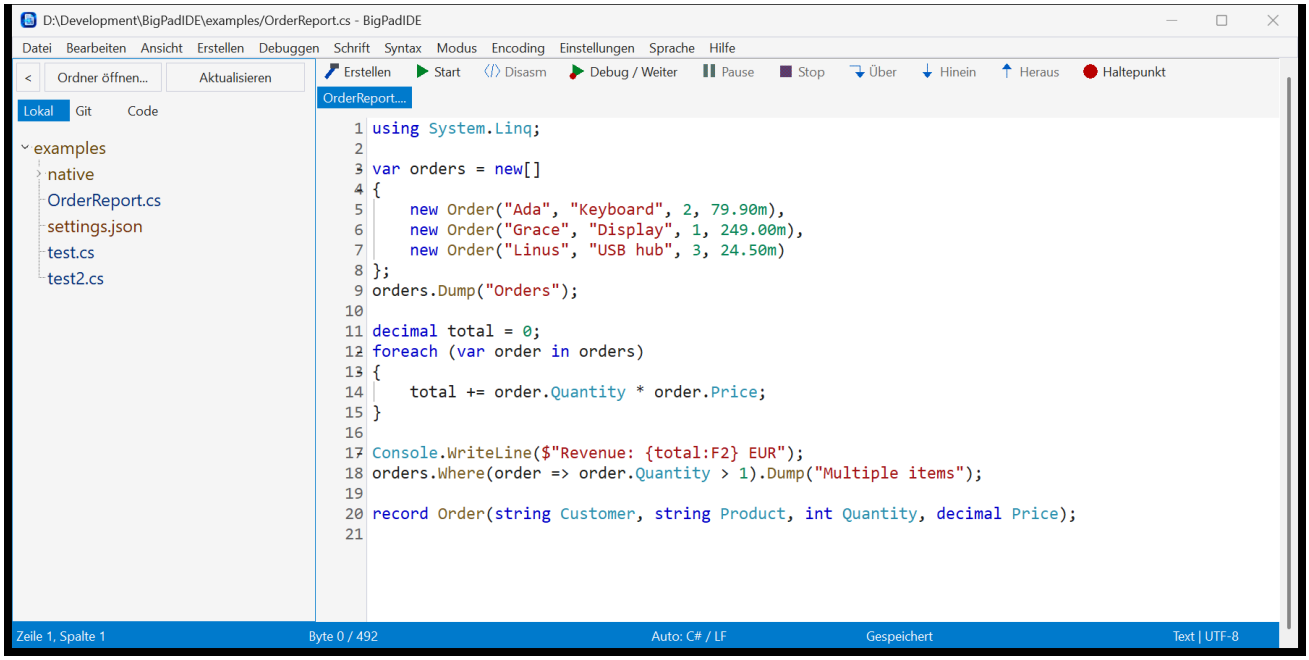


Abbildung 2. Moderne helle Oberfläche derselben Arbeitsumgebung

3.2 Themes und Hexansicht

Unter **Ansicht → Theme** zwischen **Hell**, **Dunkel**, **Sepia** und **Mitternacht** wechseln. Die Farben ändern sich sofort für Text-/Hexeditor, Zeilennummern, Explorer und Ausgabe-/Debugger-Inhalte. Geöffnete Dateien und Undo bleiben erhalten. Das gewählte Theme wird mit den Einstellungen gespeichert und beim nächsten Start wiederhergestellt. Ist automatisches Speichern deaktiviert, **Einstellungen → Einstellungen jetzt speichern** verwenden. **Standard wiederherstellen** setzt das Theme auf Hell zurück.

Der Oberflächenstil steuert Rahmen und Bedienelemente; die Theme-Auswahl steuert die Arbeitsflächen. Eigene Farben aus Kapitel 24 können diese Grundfarben ergänzen.

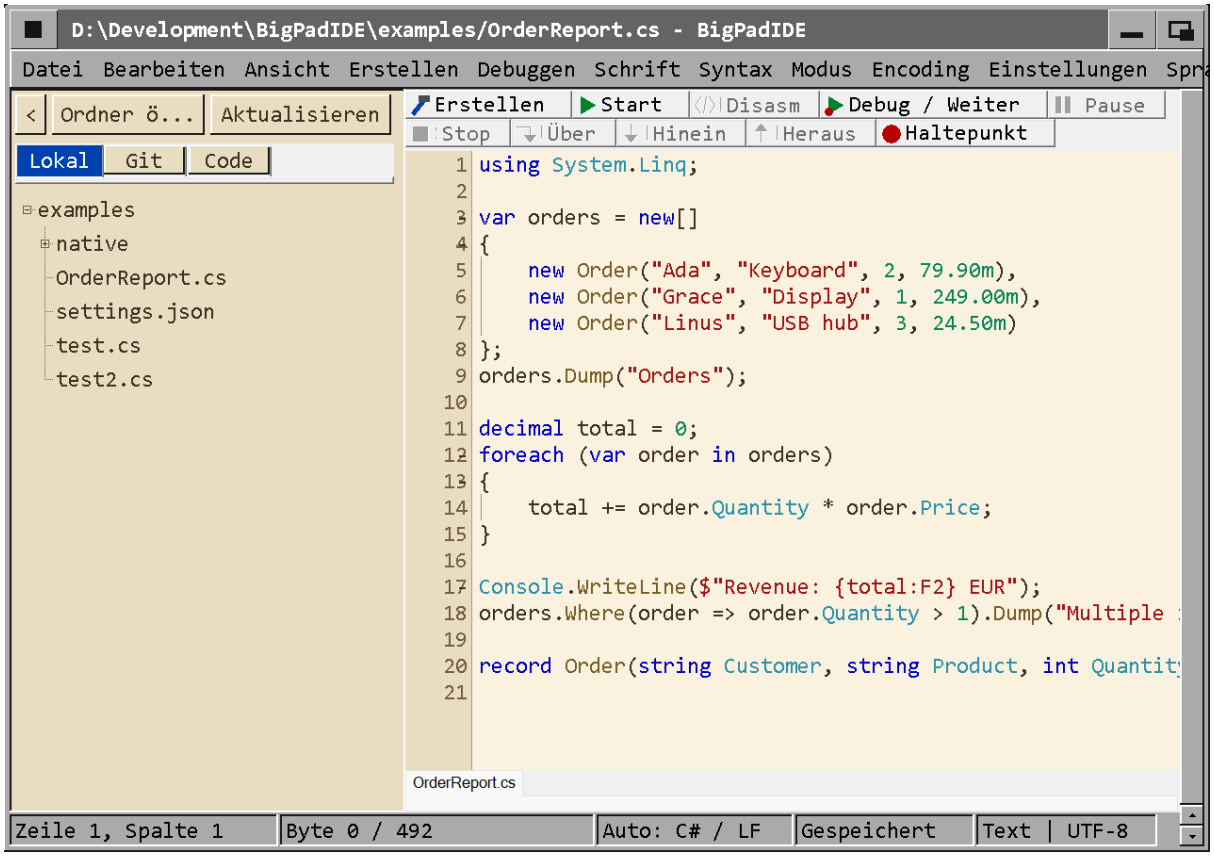


Abbildung 3. Sepia mit warmen Editor- und Explorerfarben

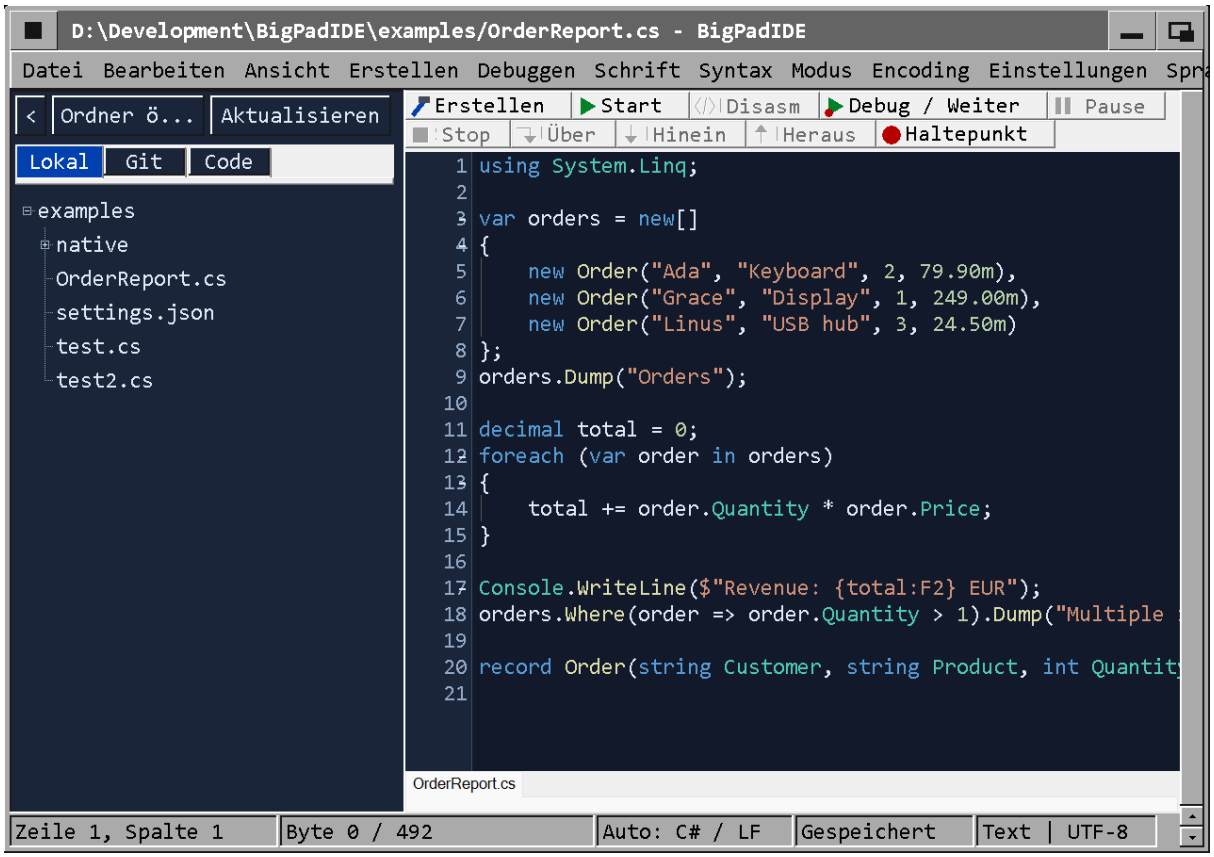


Abbildung 4. Mitternacht mit dunkelblauen Arbeitsflächen

Im Menü **Ansicht** Theme und Zeilenumbruch wählen. **Modus** → **Hex** zeigt Byte-Offsets, Hexwerte und die entsprechende Textdarstellung.

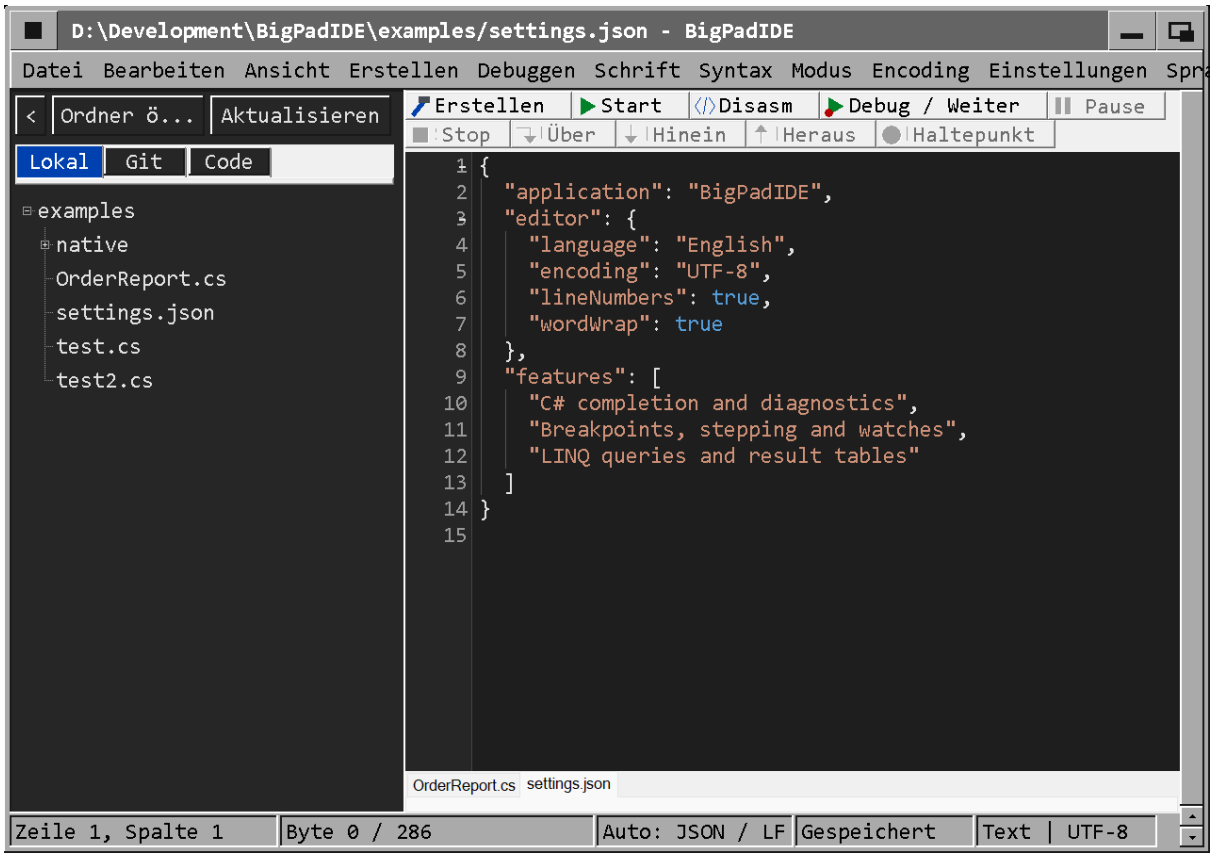


Abbildung 5. JSON im Dunkelmodus mit Einrückungslinien

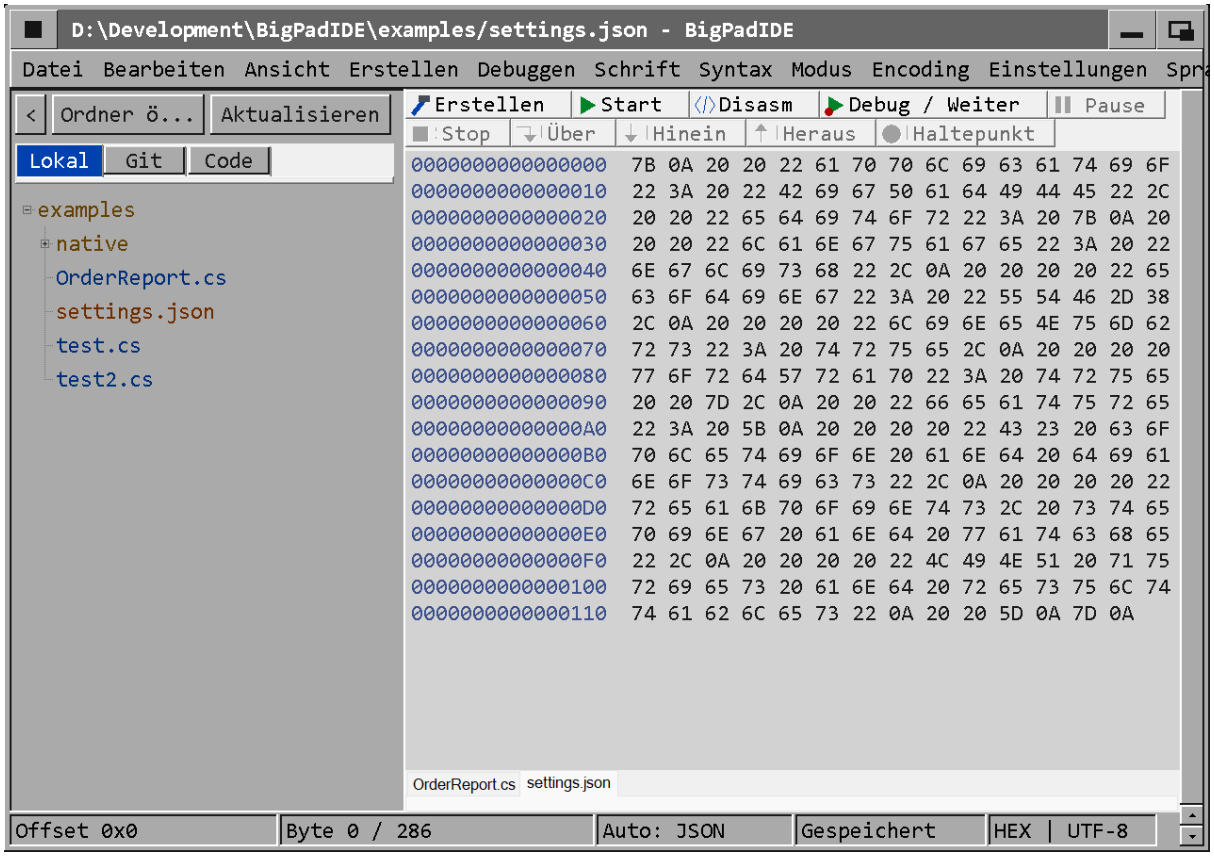


Abbildung 6. Dieselbe Datei in der editierbaren Hexansicht

3.3 Schriftgröße der Ordneransicht

Unter **Schrift → Schriftgröße der Ordneransicht** wählen Sie unabhängig vom Editor 10, 12, 14, 16, 18, 20, 24 oder 28 pt. Standard sind 12 pt. Zeilenhöhe und Kopfzeile passen sich der Größe an; bei wenig Platz werden die Kopf-Schaltflächen zweizeilig angeordnet. Auswahl und aufgeklappte Ordner bleiben erhalten.

Die Größe wird wie andere Einstellungen automatisch gespeichert, wenn die automatische Speicherung aktiv ist. Andernfalls verwenden Sie **Einstellungen → Einstellungen jetzt speichern**. **Standard wiederherstellen** setzt die Ordneransicht auf 12 pt zurück. Strg + Mausrad über dem Dateibaum verändert gezielt dessen Schrift; über dem Editor verändert es die Editorschrift. Panelhöhe und Schriftgröße sind unabhängig.

3.4 Mehrere Bereiche im Split-Screen

Unter **Ansicht → Bereich vertikal teilen (links/rechts)** bzw. **Bereich horizontal teilen (oben/unten)** teilen Sie den aktiven Editorbereich. Beide Teilungen lassen sich kombinieren und weiter verschachteln. Trennlinien können Sie mit der Maus ziehen.

Klicken Sie in einen Bereich oder wählen Sie **Nächster Bereich** (F7), um ihn zu aktivieren. Die blaue Überschrift zeigt den aktiven Bereich. Ein noch nicht sichtbarer Tab wird im aktiven Bereich angezeigt; die Auswahl eines bereits sichtbaren Tabs aktiviert dessen Bereich. Eine neue Teilung verwendet einen bisher nicht sichtbaren Tab oder legt ein leeres Dokument an. Änderungen und Scrollpositionen bleiben je Dokument erhalten. Mausrad und Bildlauf arbeiten für den jeweils aktiven Bereich; das Mausrad aktiviert den Bereich unter dem Mauszeiger.

Text, Bilder, Hex-Editor und Disassembler können gleichzeitig angezeigt werden. **Aktuellen Bereich schließen** entfernt nur den Bereich und lässt seine Tabs geöffnet. **Einzelansicht wiederherstellen** zeigt den aktiven Bereich wieder allein. Das Schließen einer Datei entfernt auch deren Bereich, wenn mehrere Bereiche sichtbar sind. Die Aufteilung gilt für die aktuelle Sitzung.

3.5 Ansichten als Tab oder Fenster öffnen

Einstellungen → IL / C# öffnen und **Einstellungen → Diff öffnen** wählen unabhängig **Als Tab in der App** oder **Eigenes Fenster**. Die normalen App-Einstellungen speichern beide Werte für die gemeinsame Anwendung. Standard: separates Assembly-Fenster und Diff-Tabs. Die Auswahl gilt für neu geöffnete Ansichten; bestehende Ansichten zum Umplatzieren schließen und neu öffnen.

Die Assembly-Ansicht im Tab nutzt das normale Tab-/Split-Layout und die eingestellte Assembly-Schrift, bei automatischer Schriftwahl die Editorschrift. Schließen entfernt ihren Tab. Die Diff-Einstellung gilt auch für Abschnitts- und Konfliktansichten; vorhandene Ansichten derselben Git-Anfrage werden wiederverwendet. Separate Fenster gehören zur App und schließen mit ihr. Während einer Git-Schreibaktion lassen sich deren Fenster und die App nicht schließen. Der Schutz ungespeicherter Dokumente bleibt erhalten.

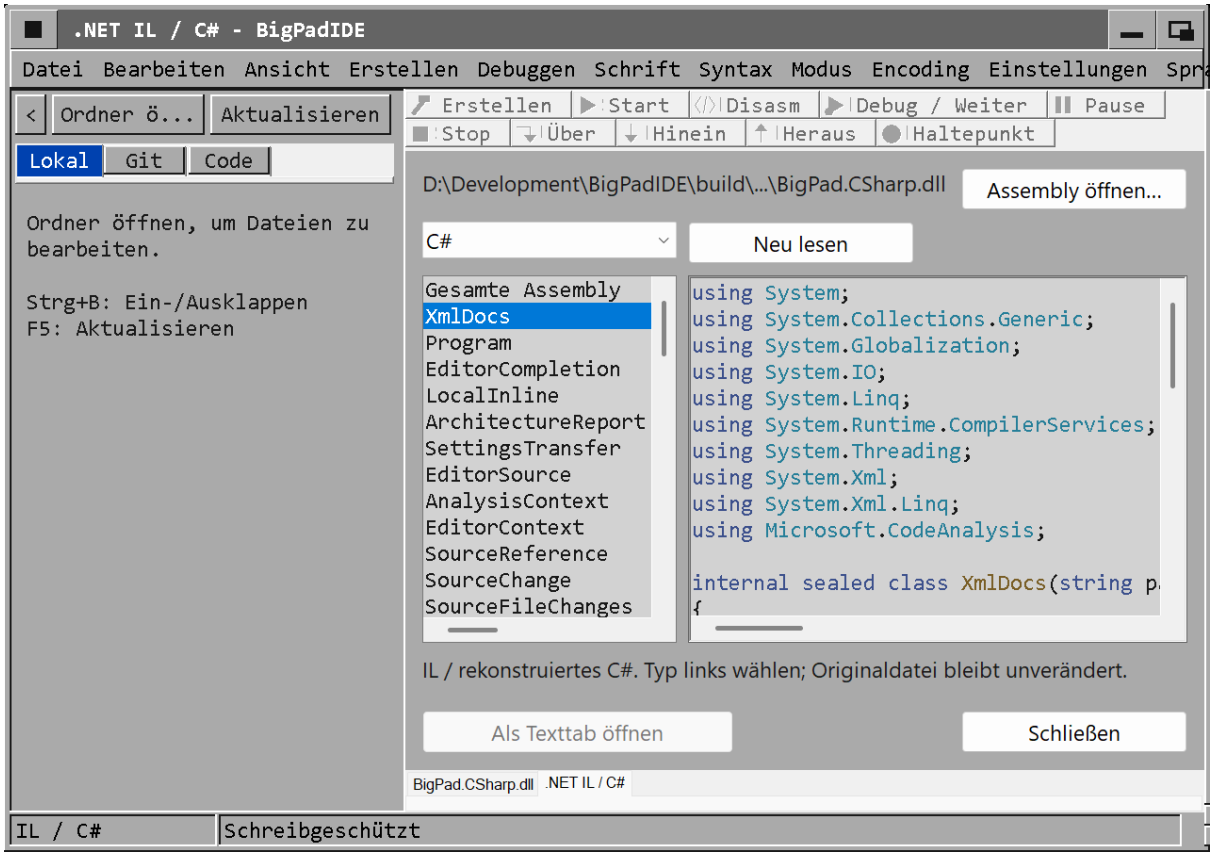


Abbildung 7. Assembly-Ansicht in der App

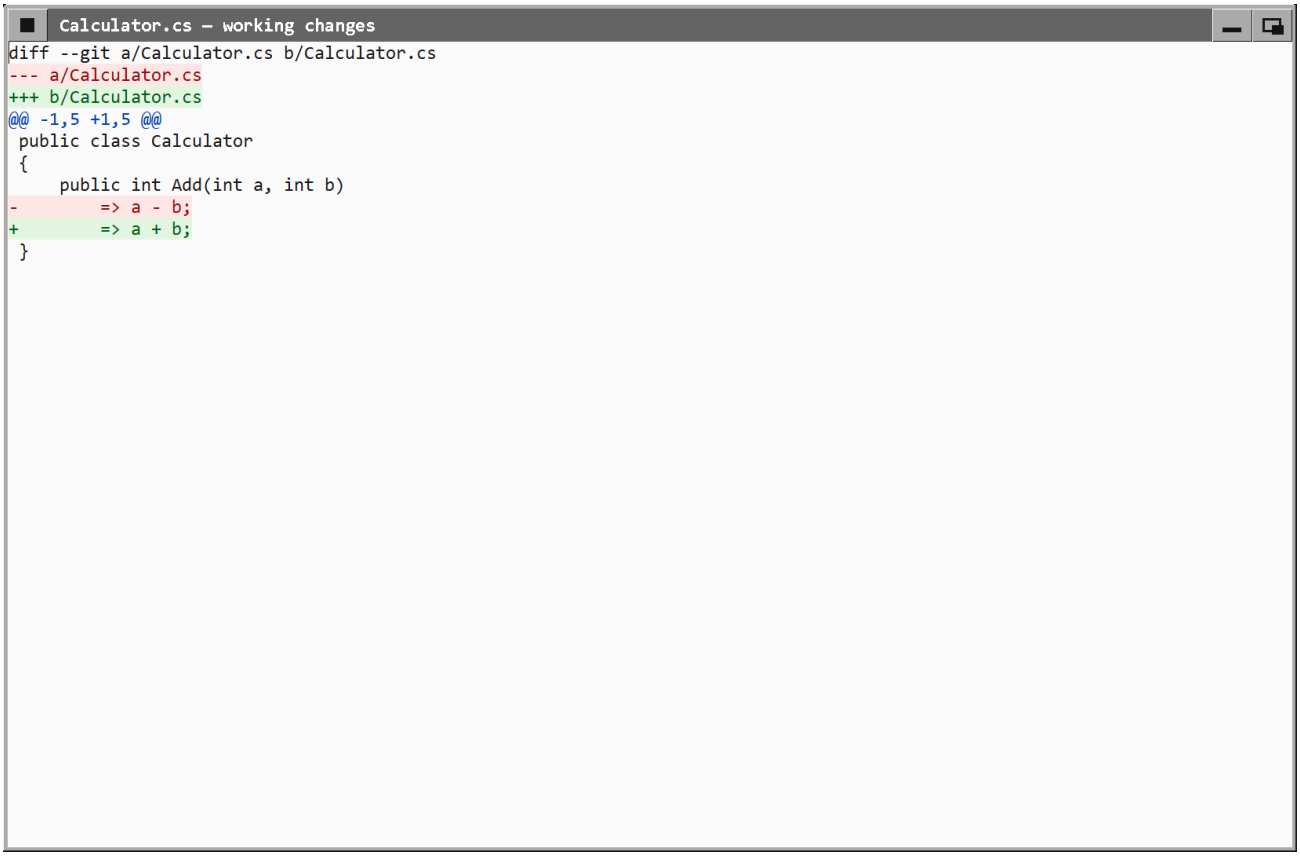


Abbildung 8. Separates Diff-Fenster

3.6 Einstellungen bewusst speichern

Einstellungen → **Automatisch speichern** steuert die automatische Speicherung der UI-Einstellungen. Ist die Option aus, verwenden Sie **Einstellungen jetzt speichern**. **Standard wiederherstellen** setzt die UI-Vorgaben zurück. Dateiverlauf, Sitzungszustand und Haltepunkte haben eigene Speichermechanismen; das Ausschalten der UI-Autospeicherung deaktiviert diese nicht pauschal.

Für die neue Platzierungswahl gilt: **IL / C# öffnen** und **Diff öffnen** sind zwei getrennte Einstellungen. Eine bereits offene Ansicht wird durch das Ändern des Radioschalters nicht umgehängt. Schließen und erneut öffnen wendet die neue Platzierung an. Standardmäßig ist IL/C# separat und Diff in einem Tab.

4. Dateien bearbeiten und sicher speichern

Strg+O öffnet eine Datei, **Strg+N** erzeugt ein leeres Dokument. Speichern erfolgt mit **Strg+S**, ein anderer Dateiname wird über **Strg+Umschalt+S** gewählt. Ordner und Dateien können außerdem auf die Anwendung gezogen werden.

4.1 Auswahl, Bearbeitung und Dateiformat

Die üblichen Befehle **Strg+A/X/C/V**, **Strg+Z** und **Strg+Y** wählen aus, schneiden aus, kopieren, fügen ein und führen Undo/Redo aus. Ist ein Eingabefeld im Ausgabe- oder Werkzeugbereich fokussiert, beziehen sich Bearbeitungsbefehle auf dieses Feld. Für Quelltextänderungen vorher den Editor fokussieren.

Das **Encoding**-Menü bietet UTF-8, Windows-ANSI und die UTF-8-BOM-Option. Die Statusleiste zeigt das aktuelle Encoding. Die BOM wird im Text nicht als Zeichen angezeigt. Windows-ANSI bezieht sich auf die aktive Windows-Codepage, nicht auf eine frei wählbare beliebige Zeichenkodierung.

Unter **Bearbeiten → Zeilenenden konvertieren** sind CRLF, LF und CR wählbar. Diese Konvertierung ändert gespeicherte Zeichen; der virtuelle Zeilenumbruch aus Kapitel 3 betrifft hingegen nur die Anzeige. Prüfen Sie bei Änderungen in einem Repository vor dem Commit, ob eine Encoding- oder Zeilenendenkonvertierung gewollt war.

4.2 Datei-Tabs und ungespeicherte Änderungen

Mehrere Dateien bleiben gleichzeitig in der Tab-Leiste geöffnet. Ein * neben dem Dateinamen zeigt ungespeicherte Änderungen an. Mit Strg+Tab beziehungsweise Strg+Umschalt+Tab wechseln Sie zwischen Tabs; Strg+W oder ein Rechtsklick auf einen Tab schließt ihn. Beim Schließen werden ungespeicherte Änderungen abgefragt.

4.3 Externe Dateiänderungen

BigPadIDE prüft geöffnete Dateien ungefähr jede Sekunde und beim erneuten Aktivieren der Anwendung. Verglichen werden Dateikennung, Größe sowie Schreib- und Änderungszeit, soweit das Dateisystem diese bereitstellt. Eine externe Änderung wird zunächst markiert; nach einer weiteren unveränderten Beobachtung lädt die IDE saubere Dokumente automatisch neu. Das gilt auch für inaktive Tabs und Split-Bereiche sowie Bild-, Hex- und Disassembler-Ansichten. Fokus und Ansichtsmodus bleiben erhalten; Bytepositionen werden bei Bedarf an die neue Dateigröße angepasst.

Eigene ungespeicherte Änderungen werden niemals automatisch neu geladen. Ihr Tab behält das * und erhält den Zusatz **extern geändert**, **Datei fehlt** oder **Datei nicht lesbar**. Beim Speichern in die ursprüngliche Datei prüft die IDE erneut: Sie können ausdrücklich überschreiben, unter einem anderen Namen speichern oder abbrechen. Ohne Zustimmung wird die externe Version nicht ersetzt. Eine gelöschte Datei wird nicht stillschweigend neu angelegt. Wird sie wiederhergestellt, wird ein unverändertes Dokument nach stabiler Beobachtung neu geladen. Nicht lesbare Dateien und ungültige neue Bilddateien behalten zunächst ihre bisherige Ansicht und ihren Hinweis.

Datei → Vom Datenträger neu laden (**Strg+Umschalt+R**) lädt die aktuelle Datei gezielt neu. Bei ungespeicherten Änderungen fragt die IDE ausdrücklich nach dem Verwerfen; die sichere Voreinstellung verwirft nichts. Ein fehlgeschlagenes Neuladen erhält den bisherigen Editorinhalt.

Im dateibasierten Piece-Table-Modus sichert BigPadIDE vor der ersten Bearbeitung dessen Originalbytes in einer temporären Datei. So bleiben die ursprünglichen Piece-Table- Bereiche auch bei externem Überschreiben konsistent. Diese Sicherung verwendet begrenzte RAM-Puffer. Vorab prüft die IDE freien Platz im Windows-Temp-Verzeichnis in Dateigröße zuzüglich 16 MiB Reserve. Ab 32 MiB zeigt ein Dialog den Sicherungsfortschritt und **Abbrechen**; die Kopie läuft im Hintergrund. Währenddessen ist die Bearbeitung vorübergehend gesperrt. Abbruch verwirft die Teilkopie und führt die ursprüngliche Bearbeitung nicht aus. Danach kann erneut bearbeitet werden. Die Kopie bleibt beim Speichern für Undo erhalten und wird beim Neuladen oder Schließen freigegeben. Scheitert die Sicherung oder wurde die Originaldatei bereits direkt verändert, wird die Bearbeitung mit einer Meldung abgebrochen, statt eine inkonsistente Version zu erzeugen. Neue, noch nicht gespeicherte Dokumente benötigen keine solche Dateikopie. Im vollständigen RAM-Modus liegt die Datei bereits unabhängig im Speicher; die Originalsicherung auf dem Datenträger entfällt. Siehe Kapitel 24.5.

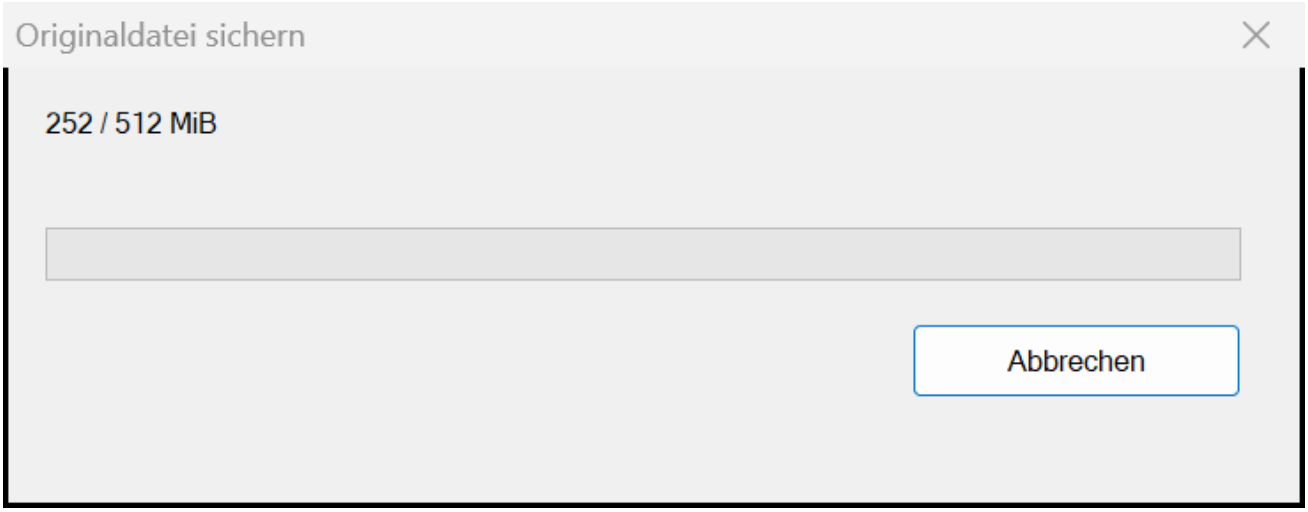


Abbildung 9. Fortschritt der Originalsicherung

4.4 Dateien und Ordner im Explorer anlegen

Im Tab **Lokal** bietet das Kontextmenü **Neue Datei…** und **Neuer Ordner…**. Ein ausgewählter Ordner ist das Ziel; bei einer ausgewählten Datei wird deren Elternordner verwendet. Ohne Auswahl gilt der geöffnete Stammordner. Das Menü lässt sich per Rechtsklick oder über die Kontextmenütaste öffnen.

Geben Sie einen einzelnen Namen ein und bestätigen Sie mit Enter. Ungültige Windows-Namen bleiben zur Korrektur im Dialog; Escape bricht ab. Bestehende Dateien und Ordner werden nicht überschrieben. Nach Erfolg wird der neue Eintrag im aktualisierten Baum ausgewählt und sichtbar gemacht. Neue Dateien öffnen im Editor; Ordner benötigen keinen zusätzlichen Tab. Die Sitzungsprüfung erfolgt vor dem Anlegen einer Datei. Die Ordnerauswahl in Dateidialogen bleibt lesend.

4.5 Dateien und Ordner umbenennen und verschieben

Im lokalen Explorer den Eintrag auswählen und **F2** bzw. **Umbenennen…** im Kontextmenü verwenden. **Verschieben nach…** öffnet die Auswahl eines vorhandenen Zielordners. Escape bricht ab. Dateien und Ordner können innerhalb desselben Volumes verschoben werden; vorhandene Ziele werden nie überschrieben.

Offene Tabs einschließlich ungespeicherter Inhalte, Undo, Cursor und Ansichtsmodus bleiben erhalten. Bei Ordnern folgen auch darunterliegende Dateien, Explorer-Root, Verlauf, Sitzung, Haltepunkte und das

ausgewählte Build-Ziel dem neuen Pfad. Die Recovery-Sicherung wird mit dem neuen Pfad erneuert; eine vorhandene Sicherung bleibt bis zum erfolgreichen Ersatz erhalten. Im Explorer wird der neue Eintrag aufgeklappt und sichtbar ausgewählt, sofern er innerhalb des geöffneten Roots liegt.

Build, Live-Modus und Debugger vor der Aktion beenden. Konflikte mit anderen Tabs, extern geänderten offenen Dateien, gesperrten Dateien oder Sitzungsgrenzen werden vor der Änderung abgewiesen. Beim Verschieben von Ordnern erhalten offene Dateien zunächst unabhängige Sicherungsdateien; das kann zusätzlichen Plattenplatz und Zeit benötigen. Externe Änderungen während der anschließenden Umbenennung behalten den Konfliktschutz und werden gemeldet. Änderungen am Dateisystem werden bei einem anschließenden Speicherfehler von Sitzung/Verlauf ausdrücklich als bereits erfolgt angezeigt. Git-Ansicht und Ordnerauswahldialoge bieten diese lokalen Aktionen nicht an.

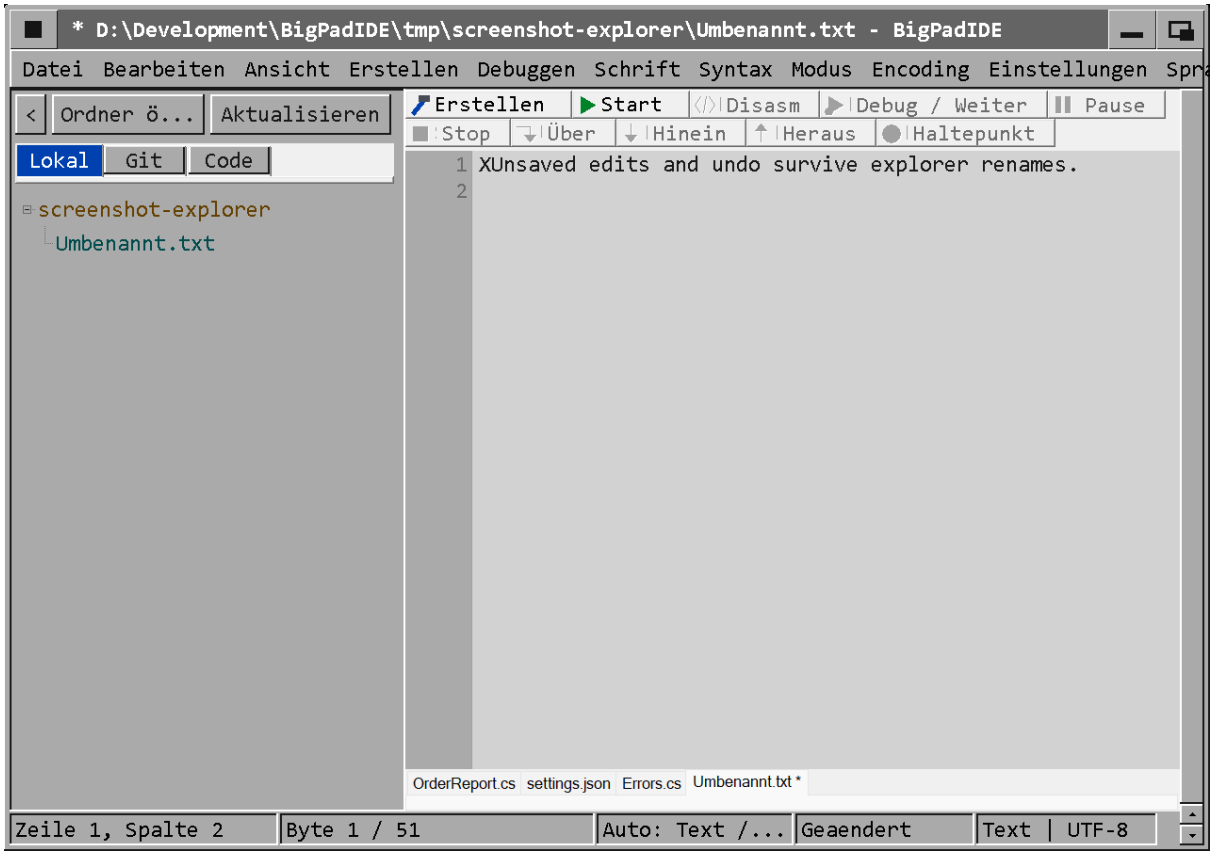


Abbildung 10. Umbenannter Tab mit erhaltenen ungespeicherten Änderungen

4.6 Große Dateien und lange Zeilen

Die Anzeige verwendet begrenzte Falt- und Einrückungsvorschauen, damit lange Zeilen nicht bei jedem Neuzeichnen vollständig durchsucht werden. Bei besonders langen Zeilen kann deshalb eine Faltmarkierung fehlen. Ein ausdrücklicher Klick in den Faltbereich prüft weiterhin die vollständige Struktur. Speichern bündelt auch viele kleine Änderungen in begrenzten Ausgabepuffern; Encoding, BOM und Undo bleiben erhalten.

Reproduzierbare Messungen für 64 MiB bis 1 GiB, Hardware, Grenzen und Rohdaten stehen im Messbericht. Die Zahlen beziehen sich auf das dort beschriebene lokale Testsystem und die ausgewählten Textfixtures.

4.7 Sicherer Umgang mit externen Änderungen

Wenn ein Generator, Git oder eine andere Anwendung dieselbe Datei geändert hat, prüfen Sie zuerst den Tabzustand. Ein sauberer Tab kann nach kurzer stabiler Beobachtung automatisch neu geladen werden. Eigene ungespeicherte Bearbeitungen bleiben erhalten. Im Konfliktfall bei Bedarf zunächst unter einem neuen Namen speichern und die Stände anschließend vergleichen.

Die Wiederherstellung nach einem Absturz ist eine zusätzliche Sicherung des letzten vollständig geschriebenen Snapshots. Sie ersetzt weder regelmäßiges Speichern noch Versionsverwaltung; Einzelheiten stehen in Kapitel 17.

5. Navigation und Suche

BigPadIDE unterscheidet die Suche im aktuellen Dokument, das schnelle Finden von Dateinamen und die inhaltliche Suche über gespeicherte Projektdateien.

5.1 Suche im Dokument

Mit **Strg+F** die Textsuche und mit **Strg+H** das Ersetzen öffnen. **F3** setzt die Suche fort. Bei fokussierter Hexansicht öffnet Strg+F dagegen die Hex-Werkzeuge für genaue Bytefolgen. Einen Textsuchauftrag deshalb im Textmodus beginnen.

Aufgabe	Werkzeug
Vorkommen in der aktuellen Datei finden	Strg+F / F3
Text im aktuellen Dokument ersetzen	Strg+H
Eine bekannte Zeile oder einen Byteoffset anspringen	Strg+G
Einen Dateinamen oder Pfadfragment finden	Strg+P
Text/Regex über gespeicherte Projektdateien suchen	Strg+Umschalt+F
C#-Symbolverwendungen semantisch finden	Erstellen → C# Referenzen suchen

5.2 Gehe zu Zeile oder Offset

Strg+G öffnet den Dialog im Menü Bearbeiten. Zeilen sind ab 1 nummeriert; Offsets sind Bytepositionen ab 0, dezimal oder mit `0x` als Hexadezimalzahl. Ein UTF-8-BOM zählt nicht zum Offset. Das Dateiende ist ein gültiges Ziel. Zeilenziele wechseln in die Textansicht, Byteziele in die Hexansicht. Im Disassembler bleibt die Ansicht erhalten; das Ziel muss zur Architektur passen. Bild- und Diffansichten unterstützen diese Navigation nicht.

Die Zeilensuche verarbeitet kleine Blöcke mit Fortschrittsanzeige und lässt sich abbrechen. CRLF, LF und CR werden berücksichtigt. Fehlende Ziele, Lesefehler und zwischenzeitliche Dokumentänderungen verschieben den Cursor nicht.

5.3 Schnell öffnen

Strg+P oder Datei → Schnell öffnen durchsucht die Dateinamen im geöffneten Ordner einschließlich Unterordnern. Ohne geöffneten Ordner wird das Verzeichnis der aktuellen Datei verwendet. Leerzeichen trennen Suchbegriffe, die alle im relativen Pfad vorkommen müssen. Groß-/Kleinschreibung spielt keine Rolle; `/` und `\` sind gleichwertig. Exakte Dateinamen stehen vor Präfix-, Inhalts- und Verzeichnistreffern.

Pfeil hoch/runter wählt einen Treffer, Enter öffnet ihn, Escape schließt den Dialog. Die Liste zeigt höchstens 200 Treffer und nennt die Gesamtzahl. Bereits geöffnete Dateien behalten ihren Tab und ungespeicherte Änderungen. Ein Fehler beim Öffnen lässt den Dialog für einen weiteren Versuch offen.

Einlesen und Filterung zeigen Fortschritt. `.git` sowie verlinkte Verzeichnisse werden ausgelassen. Die Liste ist auf 50.000 Dateien, 200.000 besuchte Einträge, 64 Unterordnerebenen und 16 MiB gespeicherten Pfadtext begrenzt. Erreichte Grenzen und nicht lesbare Einträge werden angezeigt. Erneutes Öffnen liest den Ordner neu ein; Datei → Öffnen bleibt für Dateien außerhalb der Liste verfügbar.

5.4 Projektweite Suche und Regex

Bearbeiten → Im Projekt suchen… oder **Strg+Umschalt+F** öffnet ein verschiebbares, skalierbares Suchfenster. Gesucht wird im geöffneten Explorer-Ordner; ohne Ordner dient das Verzeichnis der aktuellen Datei als Root. Die Suche liest **gespeicherte Dateien**. Ungespeicherte Änderungen werden nicht automatisch gespeichert.

Suchtext eingeben, bei Bedarf **Regex** und **Groß-/Kleinschreibung** wählen und **Suchen** drücken. Dateifilter und Ausschlüsse bestehen aus mit `;` getrennten Mustern: `*` steht für beliebige Zeichen und `?` für ein Zeichen. Muster passen auf Dateinamen oder relative Pfade mit `/` bzw. `\`; ausgeschlossene Ordner werden nicht durchsucht. Beispiel: Dateifilter `*.cs;*.cpp`, Ausschlüsse `bin;obj;tests`. `.git` und Reparse Points (Links/Junctions) werden ausgelassen. Der Suchroot muss ein direkt erreichbarer Ordner sein.

Die Trefferliste zeigt **Datei:Zeile:Spalte** und eine Textvorschau. Die Spalte zählt UTF-16-Codeeinheiten wie die C#-Werkzeuge. Doppelklick, Enter in der Liste oder **Treffer öffnen** öffnet den Tab und markiert den genauen Bytebereich. Das Suchfenster bleibt für weitere Treffer geöffnet. Seit der Suche geänderte Dateien sowie Tabs mit Änderungen oder abweichendem Datei-/Encodingstand werden abgewiesen; erst speichern/neu laden bzw. passend öffnen und neu suchen. UTF-8 mit/ohne BOM und ANSI der Windows-Codepage werden unterstützt. NULhaltige Dateien sowie UTF-16/UTF-32 mit BOM werden als binär ausgelassen und in der Zusammenfassung berücksichtigt.

Regexmuster werden **zeilenweise** über CRLF, LF und CR ausgewertet. Gruppen, Alternativen, Zeichenklassen und Quantifizierer sind möglich. Lookarounds, Rückreferenzen und andere vom Nonbacktracking-Modus nicht unterstützte Konstrukte melden einen Fehler. Leere Treffer sind zulässig und blockieren die Suche nicht. **Abbrechen/Escape** stoppt den Auftrag und verwirft Teiltreffer; anschließend schließt **Schließen/Escape** das Fenster. Geänderte Suchoptionen stoppen einen laufenden Auftrag und machen die alte Trefferliste ungültig.

Begrenzung	Wert
Suchtext / Filterfeld	4096 / 1024 Zeichen
Treffer	1000
Dateien / Verzeichniseinträge / Tiefe	10.000 / 100.000 / 32
Dateigröße / Summe berücksichtigter Dateigrößen	64 MiB / 512 MiB
Zeilenlänge	65.536 UTF-16-Codeeinheiten
Suchzeit / Prozess-Frist	80 / 90 Sekunden

Begrenzung	Wert
Regex-Zeitlimit je Match / Helferspeicher / Ausgabe	100 ms / 512 MiB / 8 MiB

Übersprungene Dateien/Zeilen, Lesefehler und erreichte Grenzen bleiben in der Zusammenfassung sichtbar. Große Dateien und lange Zeilen werden nicht vollständig in den Arbeitsspeicher geladen. Bei einer ANSI-Wiederholung kann eine Datei zweimal gelesen werden. Die Zeit-/Prozessgrenzen gelten auch dafür.

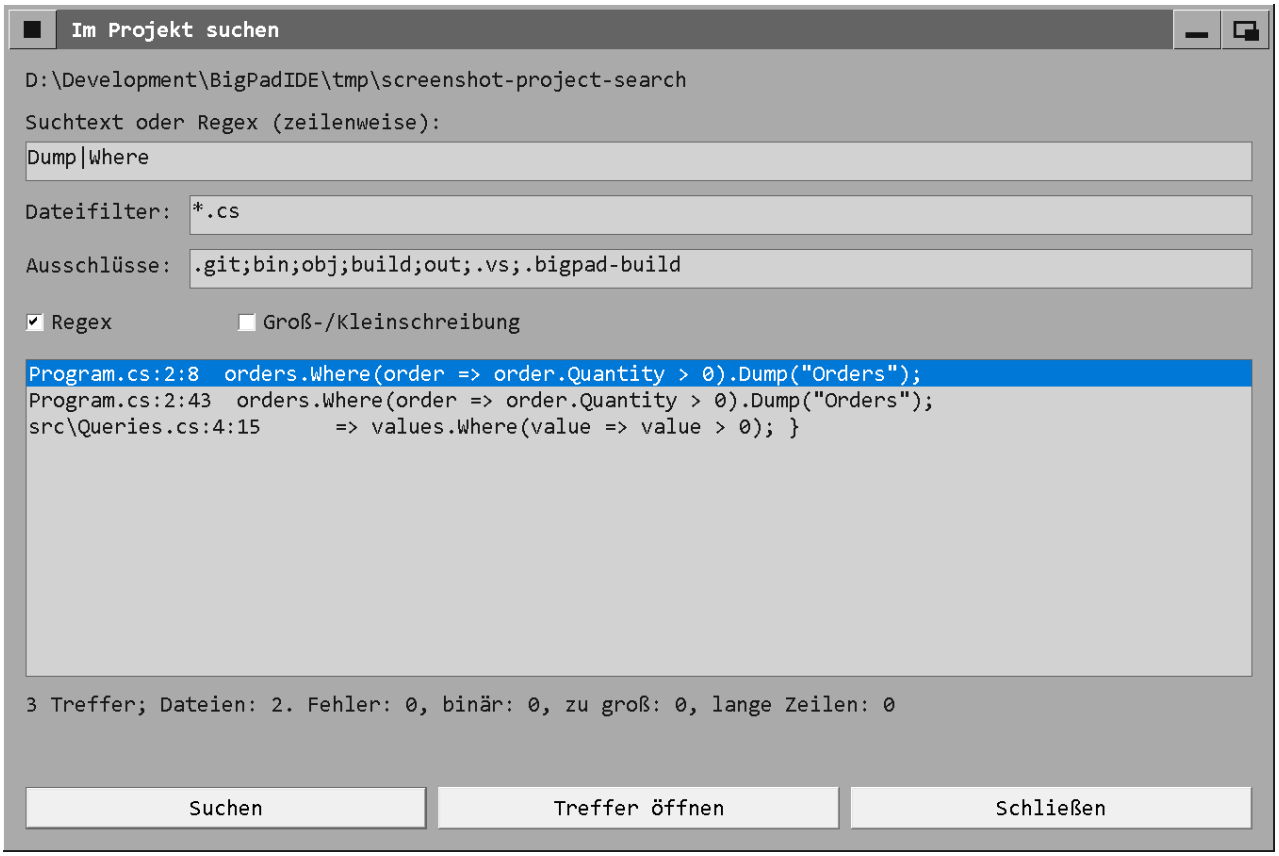


Abbildung 11. Projektweite Regexsuche mit Dateifilter und Trefferpositionen

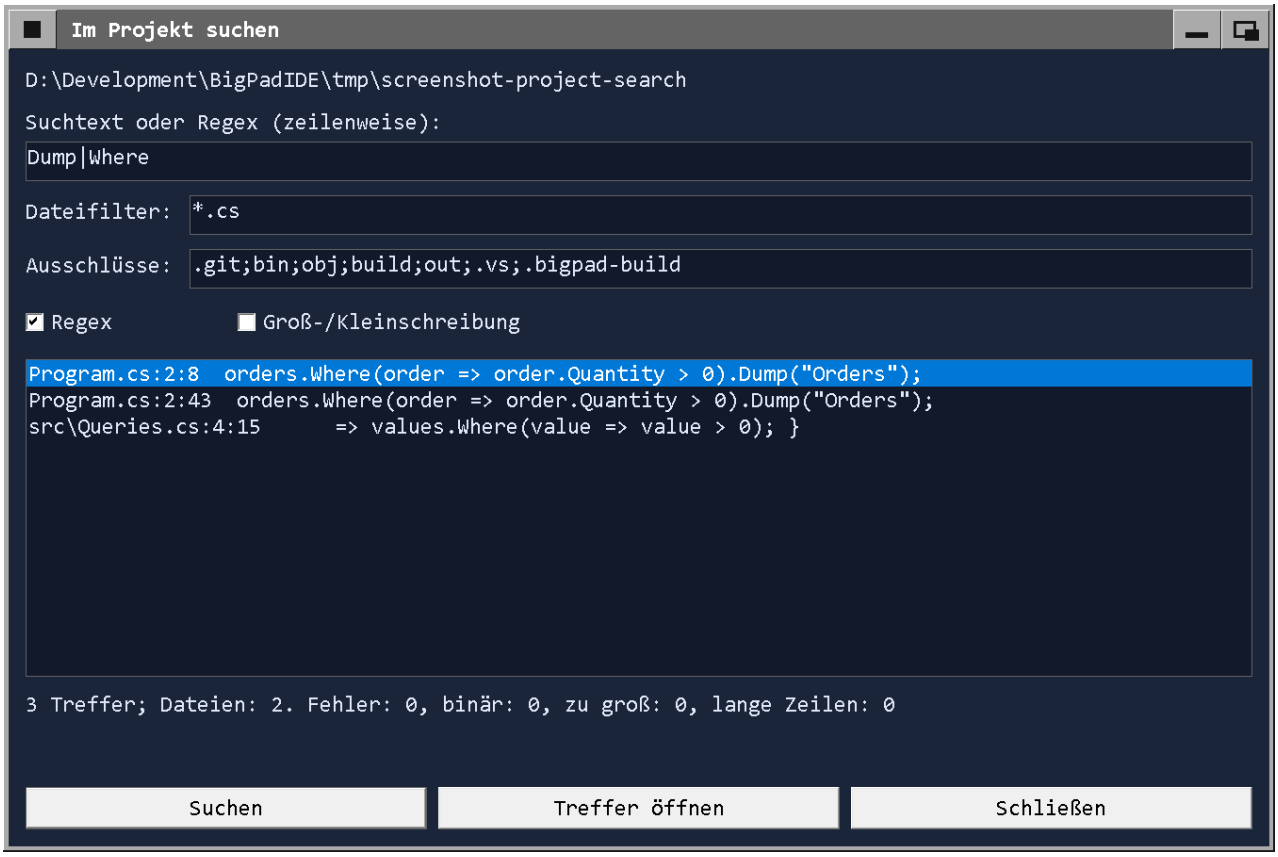


Abbildung 12. Das Suchfenster übernimmt das Mitternacht-Theme

6. C# schreiben, bauen und ausführen

`Strg+J` öffnet oder schließt das untere Panel mit **Build-Ausgabe** und **Programmausgabe**. Oben im Panel das Ziel auswählen, dann mit `Strg+Umschalt+B` bauen oder mit `F6` ausführen. **Stop** beendet den gestarteten Prozess samt Kindprozessen. Auch das **Erstellen**-Menü bietet diese Aktionen an. Ungespeicherte Änderungen der aktuellen Datei werden vor Build/Run gespeichert.

Ein installiertes .NET SDK ist erforderlich. Unterstützt werden SDK-Projekte (`.csproj`), Solution-Builds (`.sln`) und einzelne `.cs`-Programme mit Top-Level-Code oder `Main`. Einzeldateien erhalten ein temporäres Konsolenprojekt für das höchste installierte SDK ab Version 8. Zum Ausführen einer Solution deren ausführbares Projekt in der Liste auswählen. Ausgaben und Compilerfehler erscheinen fortlaufend im zugehörigen Tab, einschließlich Exitcode. Unter **Programmausgabe** sendet **Senden / Enter** eine Eingabezeile; **EOF** beendet die Standardeingabe. `.Dump()` und `.DumpAsync()` liefern im Ergebnis-Tab Grids für Objekte, Auflistungen, Dictionaries und DataTables.

Erstellen → C# Live-Modus führt den aktuellen Editorinhalt nach 800 ms Tipp-Pause automatisch neu aus, auch in einem ungespeicherten Dokument. Dabei wird ein temporärer UTF-8-Snapshot verwendet; die Originaldatei wird nicht überschrieben. Ein neuer Lauf ersetzt den vorherigen. **Stop**, Dateiwechsel und neues Dokument beenden den Live-Modus. Unterstützt werden eigenständige C#-Programme bis 1 MiB Quellcode; Projektverweise und zusätzliche NuGet-Pakete werden nicht übernommen.

Für C# öffnet `Strg+Leertaste` Roslyn-Vervollständigungen; nach einem Punkt werden Vorschläge automatisch angezeigt. Pfeiltasten wählen einen Vorschlag, Enter oder Tab übernehmen ihn, Escape schließt die Liste. Strg+Umschalt+Leertaste zeigt Signaturen, Strg+I Symbolinformationen und F12 eine Quelldefinition. Fehler werden unterwellt und unter **Probleme** aufgelistet.

6.1 Compilerfehler

C#-Fehler werden im Quelltext unterwellt und im Tab **Probleme** aufgelistet. Ein Doppelklick springt zur betroffenen Stelle. Im Beispiel wird einer Ganzzahl eine Zeichenkette zugewiesen; der Compiler meldet CS0029.

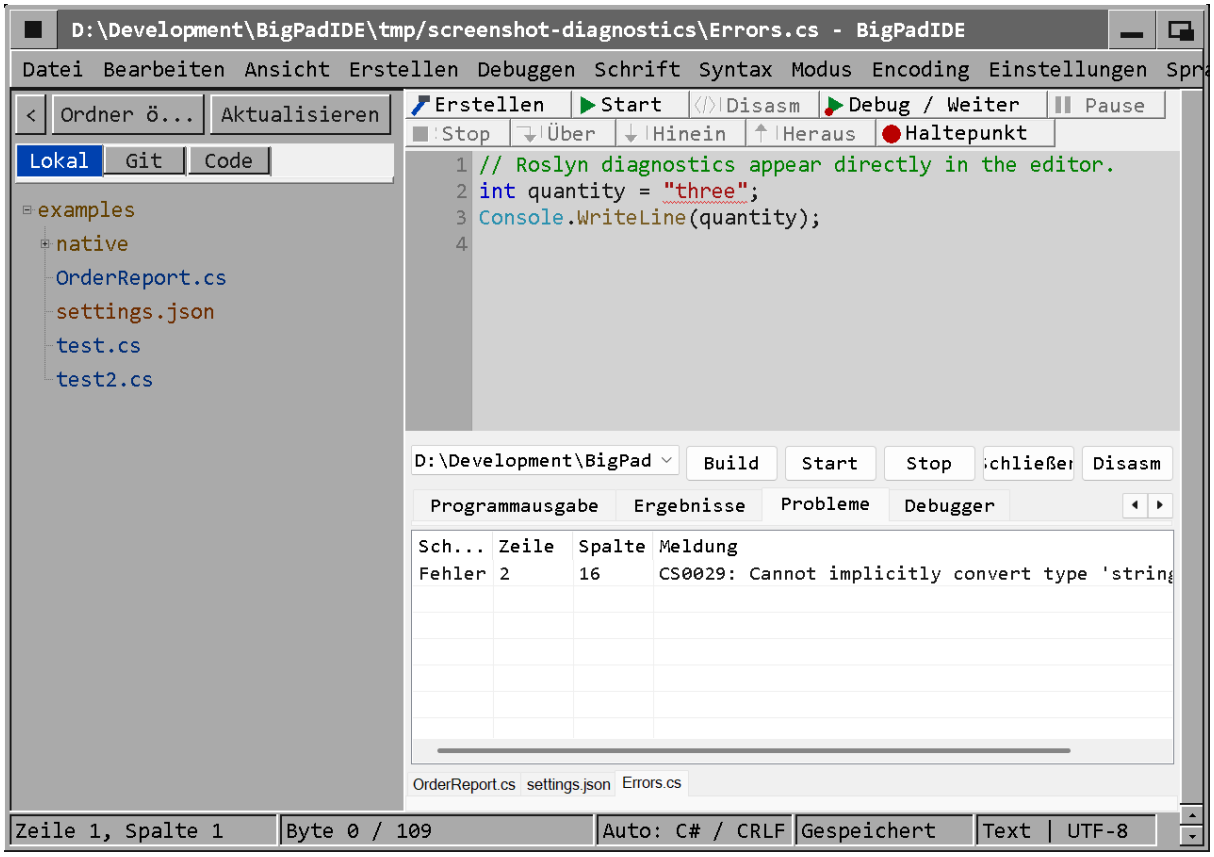


Abbildung 13. Rote Unterwellung und zugehöriger Compilerfehler

6.2 Einzeldatei oder Projekt?

Eine einzelne `.cs`-Datei eignet sich für kleine Programme, Experimente und `.Dump()`-Ausgaben. BigPadIDE legt dafür ein temporäres Konsolenprojekt an. Benötigt ein Programm mehrere Quellen, besondere Build-Einstellungen, externe Pakete oder Projektverweise, öffnen Sie das dazugehörige SDK-Projekt beziehungsweise die Solution und verwenden deren ausführbares Projekt als Ziel.

Das temporäre Einzeldateiprojekt übernimmt keine beliebigen Paketverweise aus einem benachbarten Projekt. Der Live-Modus ist ebenfalls ein eigenständiger Lauf des aktuellen Snapshots. Wenn eine Bibliothek nur im richtigen Projekt verfügbar ist, behebt der Wechsel in dieses Projekt die Ursache zuverlässiger als das Kopieren zusätzlicher Dateien in temporäre Verzeichnisse.

6.3 Beispiel mit Eingabe und Prüfung

```
using System;

Console.Write("Name: ");
string name = Console.ReadLine() ?? "Gast";
Console.WriteLine($"Hallo, {name}!");
```

Als `Greeting.cs` speichern und mit **F6** starten. Im Tab **Programmausgabe** einen Namen eingeben und **Senden** drücken. Die Eingabe ist zeilenweise; ein vollwertiges interaktives Terminal wird nicht emuliert. **EOF** signalisiert das Ende der Eingabe. Vor einem erneuten Lauf muss der vorherige Prozess beendet sein.

7. Solutions, aktives Projekt und C#-Werkzeuge

Eine Solution organisiert mehrere Projekte. Das **aktive Projekt** bestimmt das Ziel von Build, Run, Debug und Startprofilen. Der **C#-Projektkontext** bestimmt dagegen, welche Dokumente und Verweise für Symbolanalyse und Refactoring geladen werden. Diese beiden Entscheidungen erfüllen unterschiedliche Aufgaben.

7.1 Aktives Projekt per Maus und Tastatur

Im lokalen Ordnerbaum einen Projektordner, eine Projektdatei oder eine zugehörige Quelldatei auswählen. Bei nativen Projekten wird die nächstgelegene `bigpad.ini` ermittelt; bei C# ein eindeutiges Projekt beziehungsweise der ausdrückliche Solution-Projektknoten. Mehrdeutige C#-Ordner wählen kein willkürliches Ziel. Mausklick und Tastaturnavigation übernehmen das Ziel in das untere Dropdown. Fettschrift markiert das aktive Projekt. Ein zweiter Auswahlklick unten ist nicht nötig.

Die Auswahl unten zieht umgekehrt Markierung und Auswahl im Ordnerbaum mit, auch bei verschachtelten Projekten und zugeklappter Seitenleiste. Diese bleibt zugeklappt; nach dem Öffnen ist das gewählte Projekt ausgewählt. Das bloße Wechseln eines Dokumenttabs setzt ein ausdrücklich gewähltes Projekt nicht zurück. Laufende Vorgänge vor einem Zielwechsel beenden. Der C#-Analysekontext bleibt eine eigene Entscheidung; Kapitel 7.3 erläutert ihn.

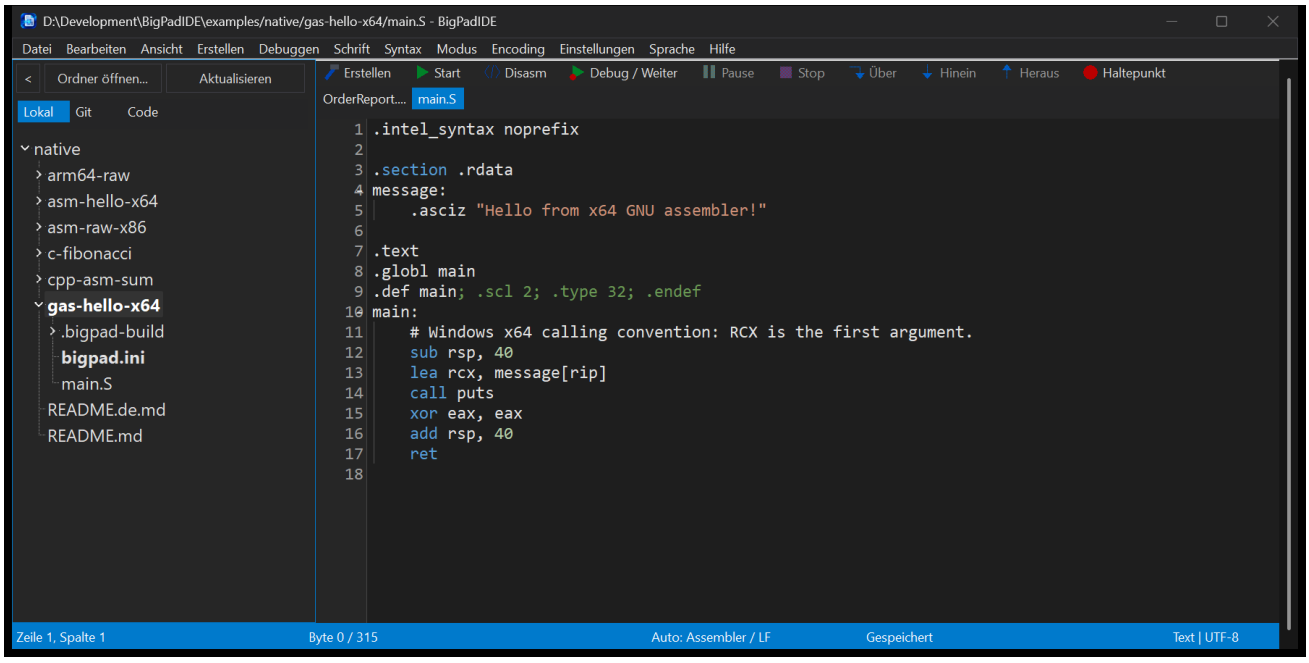


Abbildung 14. Native Arbeitsumgebung mit zugehörigem Projektziel

7.2 Typischer Ablauf bei mehreren Projekten

1. Die `.sln` über **Datei** → **Öffnen** laden. Die Projekte erscheinen im Baum.
2. Das gewünschte ausführbare C#-Projekt anklicken und die Fettschrift prüfen.
3. Im Ausgabepanel kontrollieren, dass dessen `.csproj` als Ziel ausgewählt ist.

4. Mit **Strg+Umschalt+B** bauen, mit **F6** starten oder mit **F5** debuggen.
5. Andere Quelldateitabs öffnen: Das aktive Projekt bleibt dasselbe.
6. Für ein anderes Startziel den entsprechenden Projektknoten anklicken.

Eine Klassenbibliothek kann gebaut und analysiert werden, ist jedoch nicht automatisch ein ausführbares Startprojekt. Wählen Sie zum Starten ein passendes Konsolen-, Desktop- oder anderes ausführbares Projekt.

7.3 C#-Projektwerkzeuge

Unter **Erstellen → C# Projektkontext …** die automatische Ermittlung verwenden oder eine `.csproj`, `.sln` oder `.slnx` auswählen. Die Auswahl gilt für diese Editorinstanz. Automatisch wird die nächste eindeutige Solution, sonst das nächste eindeutige Projekt gesucht (höchstens zwölf Verzeichnisebenen, Ende am Git-Root). Mehrdeutige Kontexte melden einen Fehler. Bei verknüpften Quellen in mehreren Projekten eine `.csproj` ausdrücklich wählen. Mehrdeutige Frameworkkontexte werden abgewiesen. Bei Symboländerungen in gemeinsam genutzten Projekten die ganze Solution laden, damit ihre Verbraucher zum gewählten Kontext gehören.

MSBuild lädt die Debug-Konfiguration mit tatsächlichen Compile-Einträgen, Präprozessorsymbolen und NuGet-/Projektverweisen. Das Projekt vorher wiederherstellen bzw. bauen; fehlende Workloads und unvollständige Kontexte werden gemeldet. Die Analyse führt keinen eigenen Restore aus. Einzeldateien ohne Projekt und unbenannte Quellen verwenden Standard-.NET-Referenzen und die bisherigen Einzeldatei-Hilfen. Build-/Live-Ausführung einzelner Dateien bleibt eigenständig.

C# Referenzen suchen … am Symbol öffnen. Die Liste zeigt Quelldefinitionen und Verwendungen mit Datei, Zeile und Spalte; Enter/Doppelklick/**Öffnen** navigiert. Direkt im Code geht dies über das Kontextmenü oder **Umschalt+F12**. Der Filter grenzt Dateipfade und Quelltext ein, die Vorschau zeigt die ausgewählte Fundstelle. Überladene Methoden werden semantisch unterschieden. **Strg+Klick** beziehungsweise **F12** springt zur Definition des C#-Symbols.

Im Ausgabepanel anzeigen übernimmt die Treffer in den eigenen Reiter **Verweise**. Unter **Einstellungen → Verweise öffnen** lässt sich dieser Reiter als Standard wählen; alternativ bleibt das eigene Suchfenster verfügbar. Diese Einstellung ist auch im JSON-Export enthalten. Im Reiter öffnet Doppelklick oder Enter eine Fundstelle. Dateipfad und Quelltext sind zusätzlich als Tooltip lesbar. Eine neue Suche ersetzt die vorherige Trefferliste. Nach Quelltextänderungen erneut suchen, damit Positionen und Symbole zum aktuellen Puffer passen.

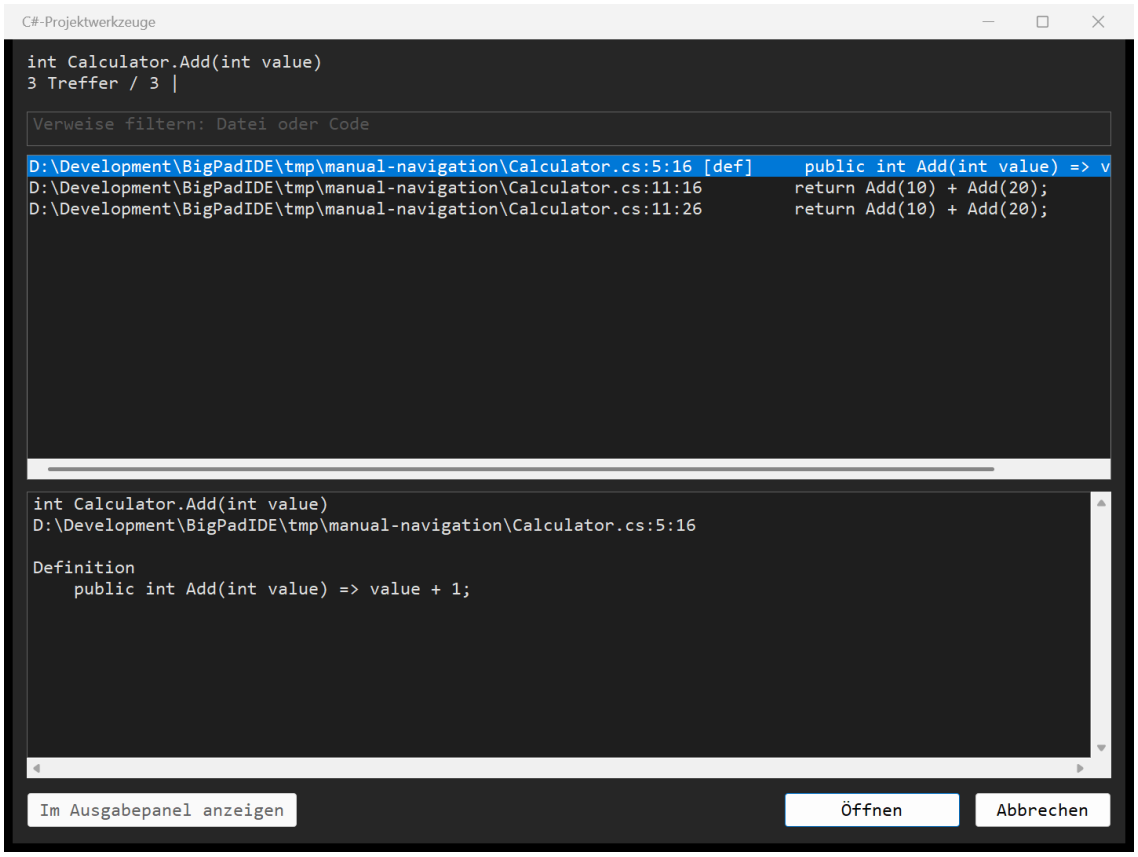


Abbildung 15. Verweise mit Filter und Quelltextvorschau.

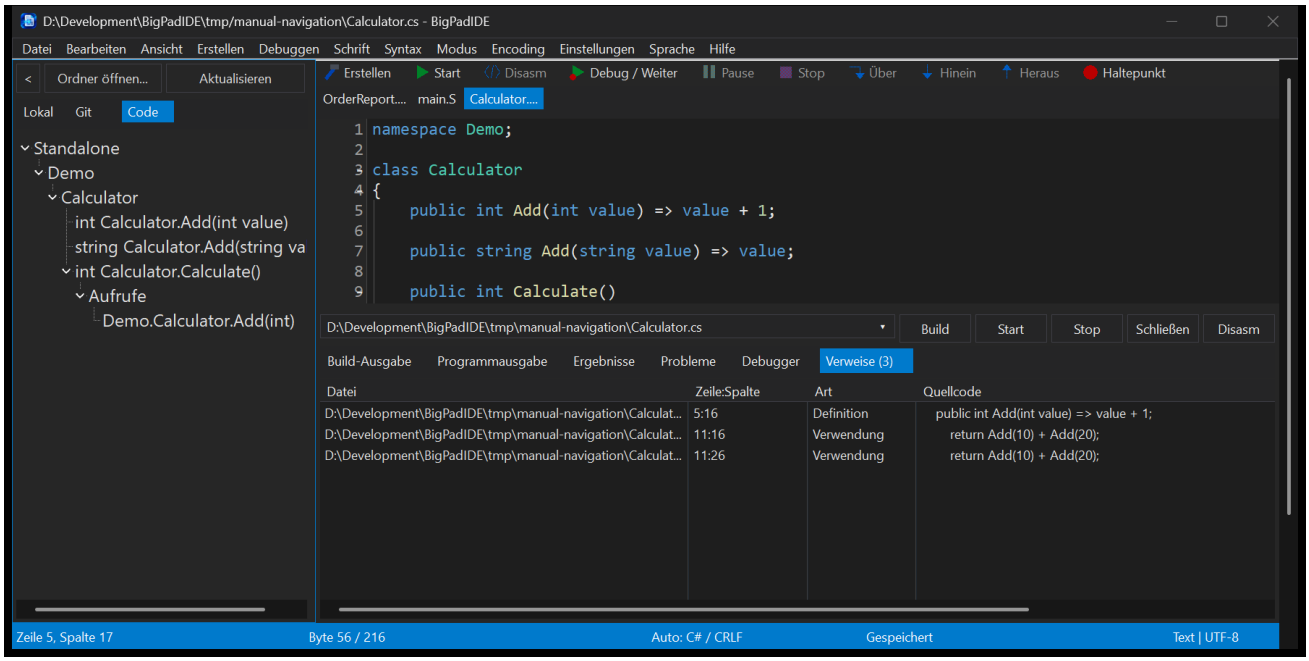


Abbildung 16. Treffer bleiben in einem eigenen Reiter im Ausgabepanel sichtbar.

Der Explorer-Reiter **Code** beziehungsweise **Erstellen → Code Map** analysiert den aktuellen C#-Projektkontext. Der Baum gliedert Projekte, Namensräume, Typen und Member. Gruppen **Aufrufe** und **Abhängigkeiten** zeigen aufgelöste Methodenaufrufe und Typbeziehungen. Ein Klick navigiert zum Symbol beziehungsweise zur Aufrufstelle; das Kontextmenü findet dessen Verweise. **Aktualisieren** berechnet den Baum neu, auch mit ungespeicherten C#-Puffern. Aufrufblätter werden nicht rekursiv vervielfacht. Die Grenze

liegt bei 5.000 Knoten und 10.000 Abhängigkeitskanten. Diese semantischen Werkzeuge gelten für C#. Für andere Sprachen steht die projektweite Textsuche zur Verfügung.

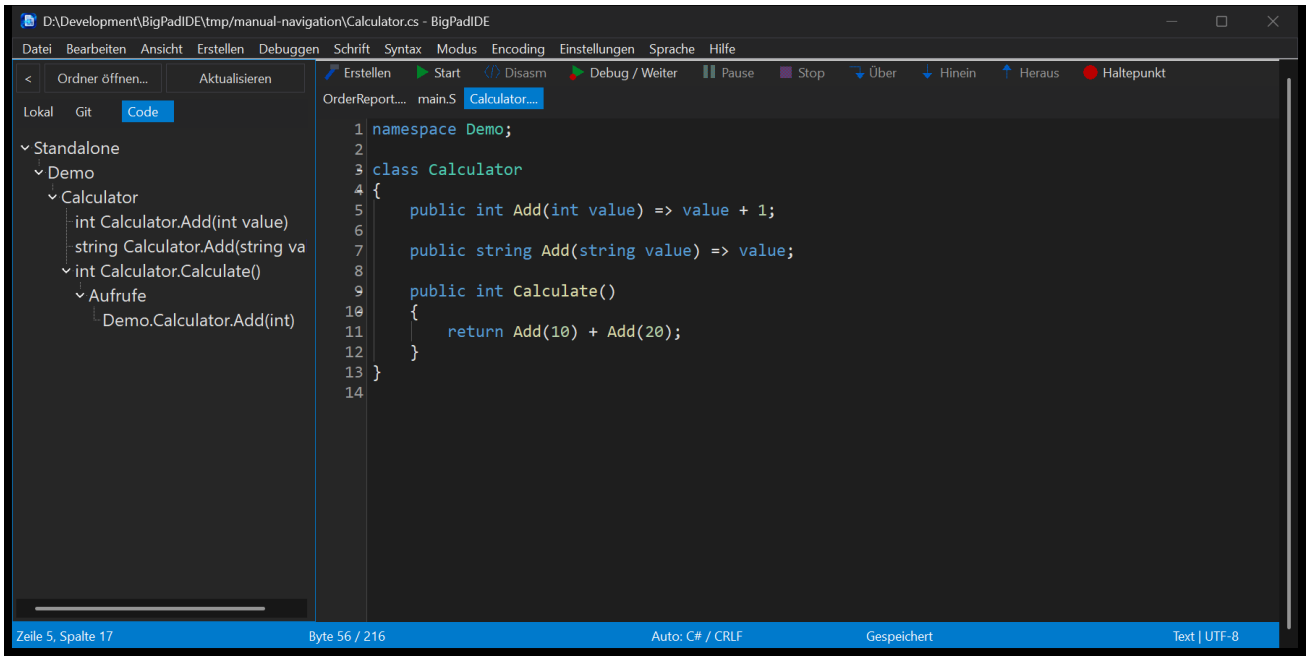


Abbildung 17. Code Map mit aufgelösten Methodenaufrufen.

Im Texteditor setzt ein Klick den Cursor, ein **Doppelklick** markiert ein ganzes Wort. Mit gedrückter Maustaste über weitere Wörter ziehen, um die Auswahl in ganzen Wörtern zu erweitern; dies funktioniert auch rückwärts. Normales Ziehen markiert zeichenweise. Die Darstellung wird gepuffert, sodass Hintergrund und Text beim Tippen und Markieren gemeinsam erscheinen.

C# Symbol umbenennen … fragt den neuen Bezeichner ab und zeigt sämtliche Textänderungen im geladenen Kontext. Metadatensymbole, Änderungen an generiertem Code und neue Compilerfehler werden abgewiesen. Dateien werden nicht umbenannt. **C# Codeaktionen …** bietet am lokalen Variablendeklarationssatz den expliziten Typ oder `var`, sofern der semantische Typ erhalten bleibt. Zusätzlich lassen sich als unnötig erkannte `using`-Direktiven entfernen.

In der Vorschau **Übernehmen** wählen. Der Auftrag wird erneut berechnet; bei geändertem Vorschlag erneut prüfen. Die Übernahme prüft Quellhash, alte Textstellen, Encoding und unveränderte Dokumentgenerationen. Alle Änderungen werden vorbereitet, bevor ein Puffer geändert wird. Geschlossene betroffene Dateien öffnen als normale geänderte Tabs. Bereits ungespeicherte Änderungen bleiben enthalten. Es erfolgt kein automatisches Speichern. **Strg+Z** nimmt die Aktion je Datei zurück. Vorhandene BOMs und Zeilenenden bleiben erhalten; nicht verlustfrei darstellbare ANSI-Änderungen werden abgewiesen. Der Cursor betroffener Tabs beginnt nach der Übernahme bei Dateianfang. Escape/**Abbrechen** beendet auch einen laufenden Helfer samt BuildHost-Prozessen. Während Build, Live-Modus oder Recovery-Rückfrage sind diese Werkzeuge gesperrt. Während einer Debugsitzung bleibt die Verweissuche lesend verfügbar; Änderungen am Quelltext erfordern das Beenden der Sitzung.

Grenzen: aktueller/ungespeicherter Quelltext 1 MiB je Datei, 64 Editor-Snapshots und 8 MiB insgesamt; Projektkontext 2000 Dokumente, 16 MiB je gespeicherter Quelle und 128 MiB insgesamt; Umbenennung/Codeaktion 64 Dateien, 4 MiB je betroffener Datei, 32 MiB Gesamttext, 10.000 Textänderungen, 1 MiB je Einfügung und 8 MiB eingefügter Text. Höchstens 10.000 Referenzorte, 16 MiB Protokollausgabe, 60 Sekunden je Projektauftrag, 1 GiB Helfer-/BuildHost-Speicher insgesamt und acht

Prozesse. Normale Codehilfe hat eine 30-Sekunden-Frist. Größere Aufträge werden ohne Teilübernahme abgewiesen; kleinere Projektkontexte wählen. Die normalen Sitzungsgrenzen gelten ebenfalls.

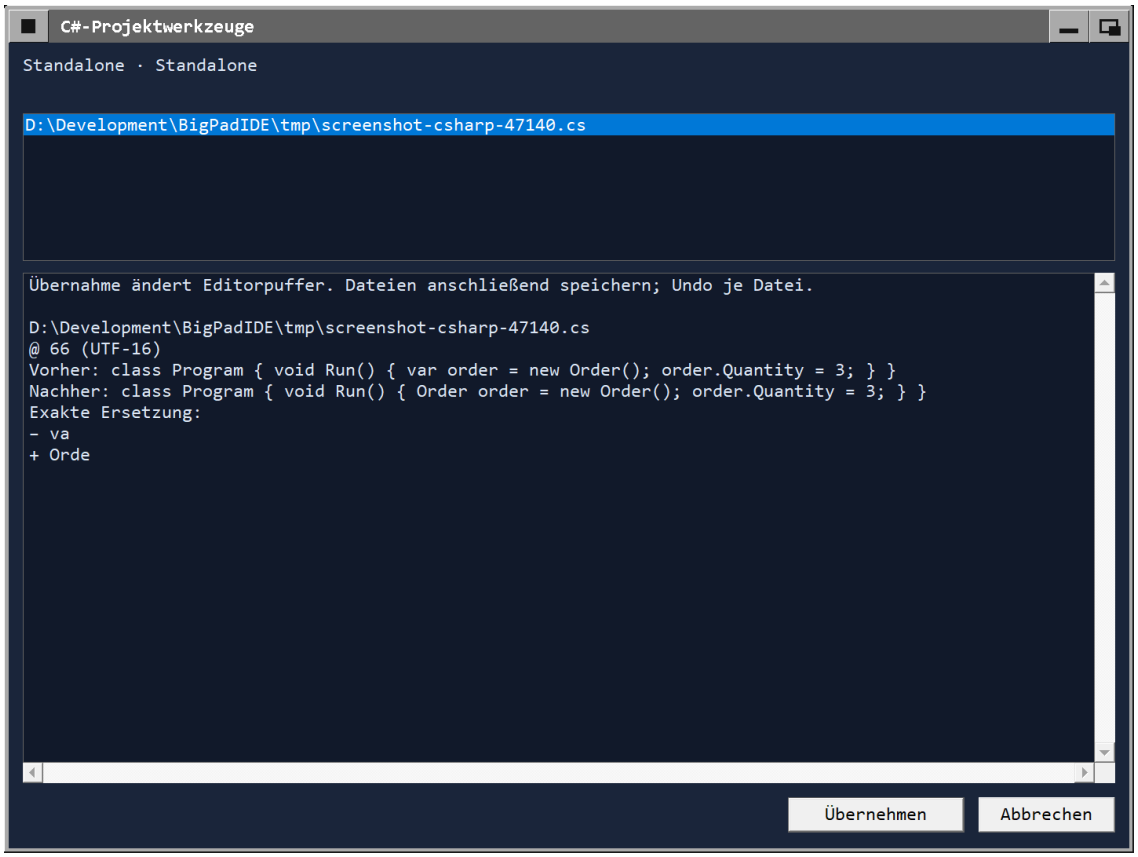


Abbildung 18. Semantische Codeaktion mit Vorher-/Nachher-Kontext und exakter Ersetzung.

8. Programmausgabe, Ergebnisse und Startprofile

Mit **Strg+J** das Ausgabepanel öffnen. Compilerprotokoll, Programmausgabe, Ergebnistabellen, Probleme und Debuggerinformationen sind getrennte Ansichten. Die obere Panelkante mit der Maus nach oben oder unten ziehen, um die Panelhöhe zu ändern. Ein senkrechter Größenpfeil zeigt den ziehbaren Bereich. Das ändert nicht die Schrift. Die Höhe bleibt beim Wiederöffnen erhalten und wird mit den Einstellungen gespeichert. Beim Verkleinern des Hauptfensters wird sie begrenzt; beim Vergrößern kehrt die gewählte Höhe zurück. Ein Mindestbereich für den Editor bleibt sichtbar. Strg + Mausrad ändert dagegen die Schrift unter dem Zeiger. Bei einem fehlgeschlagenen Start zuerst Build-Ausgabe und Exitcode lesen.

8.1 Tabellen und Programmausgabe

`.Dump("Orders")` stellt eine Sequenz mit ihren öffentlichen Eigenschaften als Tabellenspalten dar. Im Tab **Ergebnisse** über die Liste zwischen mehreren Tabellen wechseln. Konsolentext erscheint separat in **Programmausgabe**.

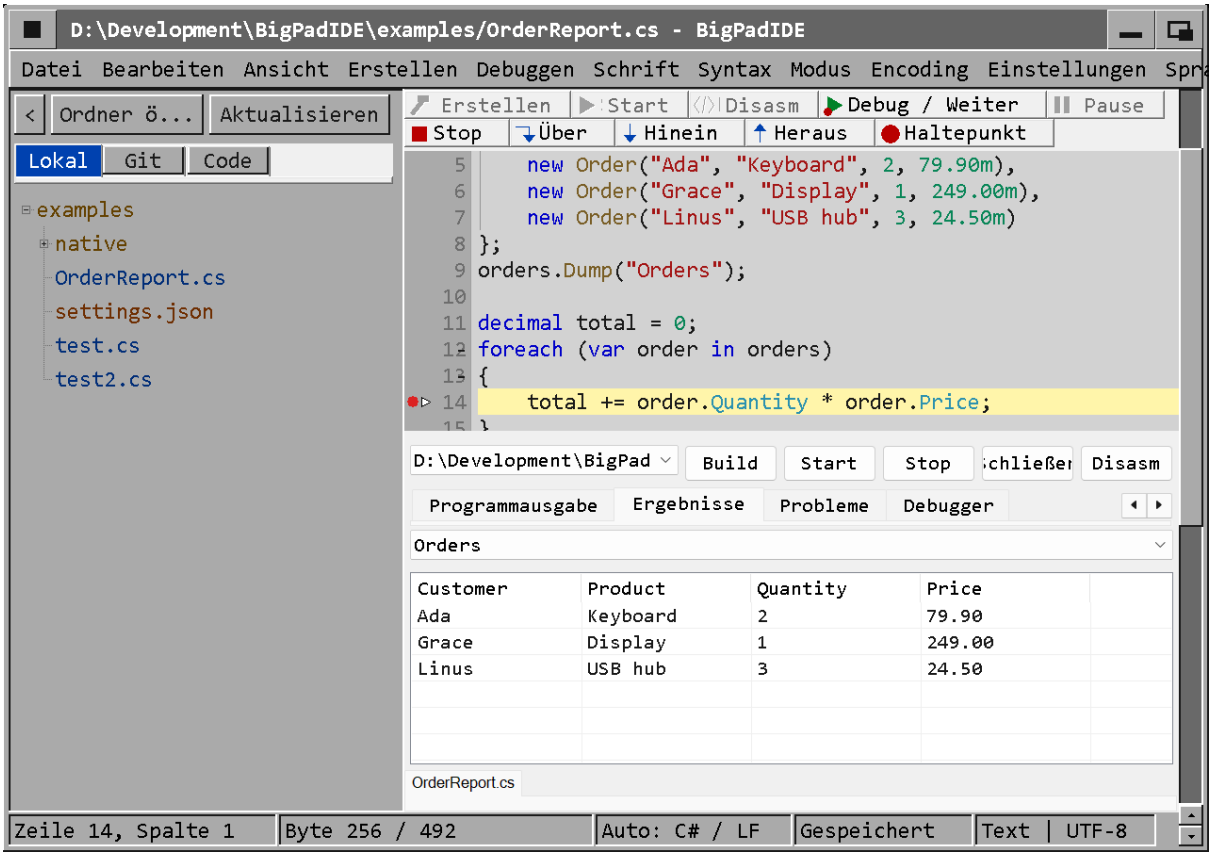


Abbildung 19. Bestellungen als Zeilen und Spalten im Ergebnispanel

8.2 Build-/Run-Profile und Eingabe

Im Ausgabepanel das gewünschte Ziel wählen, dann **Erstellen → Build-Konfiguration und Compileroptionen …** öffnen. **Build** und **Run** besitzen getrennte Entwürfe für Arbeitsverzeichnis, Argumente und Umgebung. Änderungen gelten nach **Speichern** für dieses Ziel; **Abbrechen / Escape** verwirft den Dialogentwurf. Während Build, Debugging, Live-Modus oder Recovery-Rückfrage zuerst den laufenden Vorgang beenden. Ein geänderter, offener Konfigurationstab muss zuerst gespeichert oder geschlossen werden.

Relative Arbeitsverzeichnisse beziehen sich auf den Ordner des gewählten Ziels. Leer lässt den Standard bestehen: Projektordner für Build-Werkzeuge, bei einzelnen C#-Dateien der temporäre Buildordner; zum Ausführen der ursprüngliche Quell-/Projektordner. Das Run-Arbeitsverzeichnis ändert bei .NET den gestarteten Programmkontext, ohne unbeabsichtigt die SDK-Auswahl durch eine fremde global.json zu verändern. Explizites Build-Arbeitsverzeichnis steuert auch das Arbeitsverzeichnis des SDK-Hosts.

Argumente verwenden Windows-Quoting: `"two words"` ist ein Argument, `""` ein leeres. Sie werden direkt an Prozesse übergeben, ohne Shell-Auswertung. Unter **Bauen** stehen zusätzliche Compiler-/Linker- bzw. dotnet-Argumente. Unter **Ausführen** stehen Programmargumente. Bei .NET Build & Run werden Buildargumente vor `--` an `dotnet run` weitergegeben; sie müssen dort unterstützt sein, etwa `--configuration Release` oder `-p:DefineConstants=DEMO`. Bei nativen Pipelines gelten Buildargumente für Compiler und Linker, nicht für die anschließende objcopy-Extraktion.

Umgebung zeilenweise angeben: `NAME=Wert` setzt/ersetzt einen geerbten Wert, `NAME=` setzt einen leeren Wert, `-NAME` entfernt ihn. Namen werden ohne Beachtung der Großschreibung verglichen; Duplikate werden abgewiesen. Der Elternprozess bleibt unverändert. Bei .NET Build & Run verwendet der gemeinsame SDK-/Programmprozess Buildwerte mit Run-Overrides; Run gewinnt. Native Build- und Run-Prozesse verwenden jeweils ihr eigenes Profil. Benannte Live-Snapshots übernehmen das Profil ihrer Originaldatei; unbenannte Live-Quellen nutzen den gewählten Explorerordner, sonst ihren temporären Kontext. Ein neuer Live-Lauf beendet den alten samt Eingabe.

Die Profile werden lokal als UTF-8-Datei `<Ziel>.launch.ini` gespeichert, zum Beispiel `Program.cs.launch.ini` oder `bigpad.ini.launch.ini`. Speichern prüft den geladenen Dateistand und ersetzt eine vollständig geschriebene Datei atomar; fehlgeschlagene oder veraltete Änderungen ersetzen den bisherigen Stand nicht. Verlinkte Sidecars werden nicht ersetzt; diese direkt bearbeiten. Das Format besitzt den Kopf `[BigPadLaunch1]`, die Abschnitte `[build]` und `[run]` sowie `directory=`, beliebig wiederholtes `argument=`, `input=auto|utf8|ansi|oem`, `env.NAME=Wert` und `unset.NAME=`. Argumentwerte im Dateiformat sind wörtlich, ohne zusätzliche Anführungszeichen.

Unter **Programmausgabe** eine Zeile eingeben und **Senden / Enter** wählen. Normale Prompts erscheinen schon vor ihrem Zeilenumbruch. **EOF** leert die Warteschlange und schließt dann die Eingabe; anschließend sind weitere Eingaben deaktiviert. **Stop** beendet den Auftrag einschließlich Kindprozessen. Zwischenablage und Rückgängig im Bearbeiten-Menü betreffen das fokussierte Eingabefeld statt des Quelltexts. Es handelt sich um zeilenweise Pipe-Eingabe mit CRLF, kein interaktives Konsolenterminal.

Im Run-Profil die erwartete Eingabecodierung wählen: **Automatisch** bedeutet bei Windows-C#-Programmen System-OEM, bei nativen Programmen UTF-8. Alternativ UTF-8, Windows-ANSI oder Windows-OEM wählen. Ein C#-Programm, das UTF-8 erwartet, setzt zum Beispiel `Console.InputEncoding = new System.Text.UTF8Encoding(false);`. Nicht verlustfrei darstellbare Zeichen werden abgewiesen; die Eingabe bleibt zur Korrektur im Feld. Fehler und volle Warteschlangen werden ausdrücklich gemeldet.

Grenzen: 256 KiB Konfigurationsdatei, 128 Argumente je Profil, 4096 UTF-16-Zeichen je Argument und 24.000 Zeichen formatierte Argumentliste; die vollständige Windows-Befehlszeile muss unter 32.767 Zeichen bleiben. Bis 64 Umgebungsänderungen je Profil bzw. kombiniertem .NET-Run-Profil, Namen bis 256 und Werte bis 4096 Zeichen; der erzeugte Umgebungsblock ist auf 1.048.576 UTF-16-Zeichen begrenzt. Eingabefeld 16.384 UTF-16-Zeichen, einzelne Nachricht höchstens 64 KiB, wartende Bytes höchstens 1 MiB. Nicht lesende Programme blockieren die Oberfläche nicht.

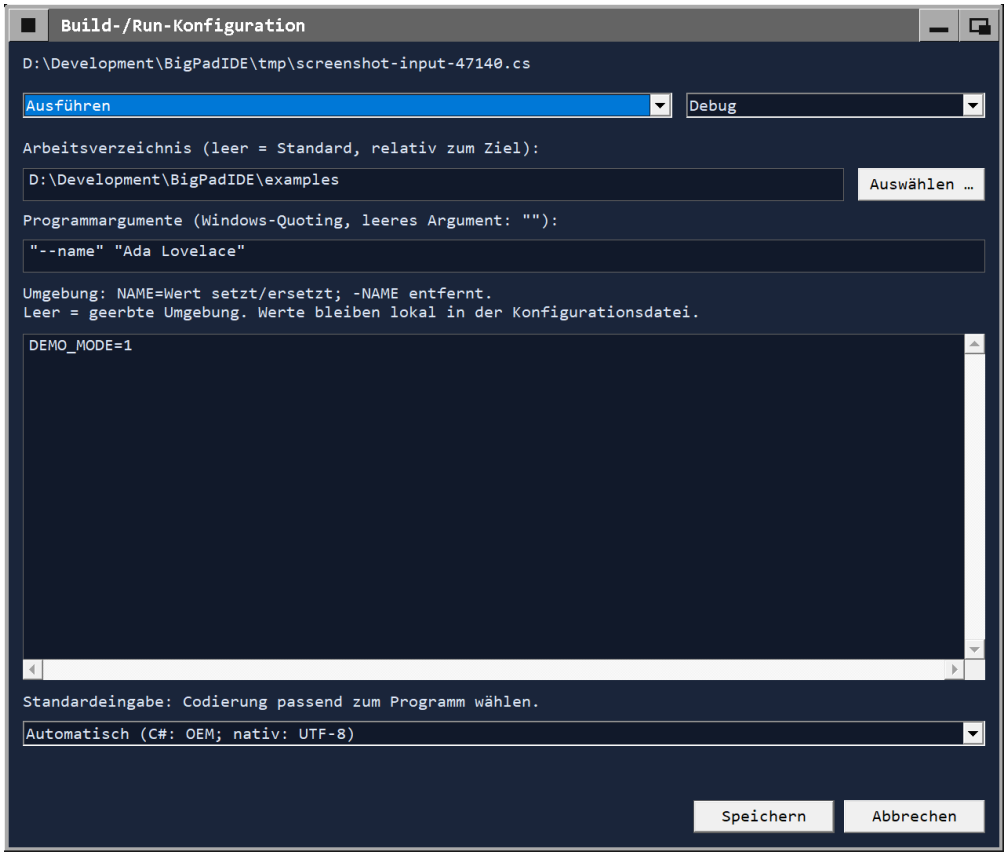


Abbildung 20. Run-Profil mit Arbeitsverzeichnis, Argumenten und Umgebung

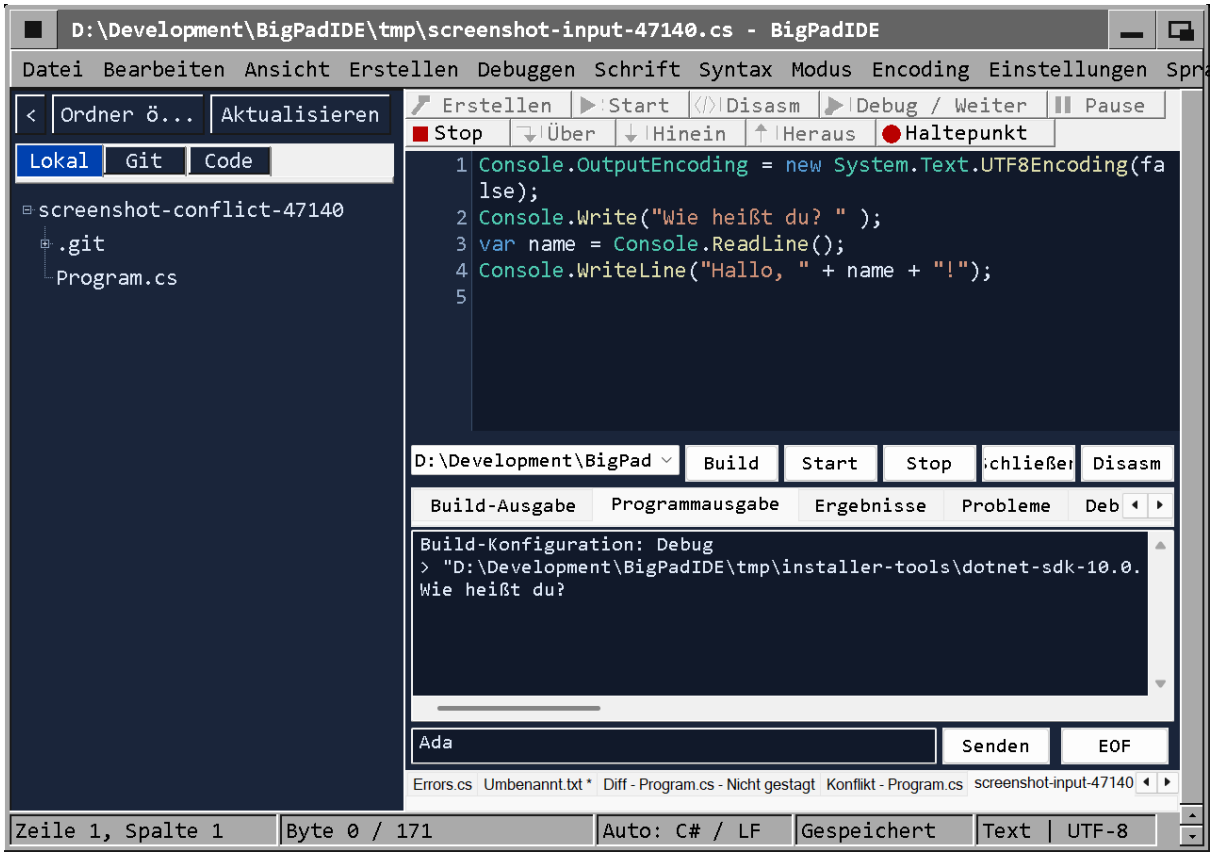


Abbildung 21. Laufendes Programm mit sichtbarem Prompt und Eingabezeile

8.3 Beispiel einer Ergebnistabelle

```
using System;

var orders = new[]
{
    new { Id = 101, Customer = "Anna", Total = 42.50m },
    new { Id = 102, Customer = "Ben", Total = 19.90m }
};

orders.Dump("Bestellungen");
Console.WriteLine("Zwei Bestellungen ausgegeben.");
```

Die BigPadIDE-Dump-Helfer ergänzen die Einzeldateiausführung. Das Ergebnisraster zeigt öffentliche Eigenschaften als Spalten; die Console-Ausgabe bleibt im separaten Ausgabetab. In einem eigenen externen Projekt müssen entsprechende Hilfen im Projekt verfügbar sein; `.Dump()` ist kein eingebauter C#-Sprachbefehl.

8.4 Beispiel eines gespeicherten Run-Profils

```
[BigPadLaunch1]
[build]
argument=--configuration
argument=Debug

[run]
directory=.
argument=erste Eingabe
env.DEMO_MODE=1
input=utf8
```

Die Datei heißt beispielsweise `Demo.csproj.launch.ini`. Jede `argument=`-Zeile enthält genau ein wörtliches Argument; anders als im Dialog werden im Sidecar keine zusätzlichen Anführungszeichen benötigt. `input=utf8` setzt die erwartete Eingabecodierung der App; das Zielprogramm muss dieselbe Codierung verwenden.

9. C#-Debugging

Beim Debugging hält das Programm an definierten Stellen. Aufrufliste, lokale Variablen und Watch-Ausdrücke helfen, den tatsächlich ausgeführten Ablauf zu verstehen. Benötigt werden ein geeigneter Debugadapter und passende Debugdaten.

9.1 Die erste Debugsitzung

1. Das ausführbare C#-Projekt oder eine passende Einzeldatei als Ziel wählen.
2. Eine ausführbare Quelltextzeile wählen und **F9** drücken.
3. Den Editor fokussieren und **F5** starten. Im Explorer hat F5 eine andere Funktion.
4. Beim Halt die gelbe aktuelle Zeile sowie lokale Werte prüfen.
5. Mit **F10** schrittweise weitergehen oder mit **F5** fortsetzen.
6. Mit **Umschalt+F5** die Sitzung beenden, bevor Quellen bearbeitet werden.

9.2 Haltepunkte und Watch-Ausdrücke

Links neben eine Zeilennummer klicken oder **F9** drücken, um einen Haltepunkt umzuschalten. **F5** startet die Sitzung oder setzt sie fort. Der rote Punkt markiert den Haltepunkt, die gelbe Zeile und der Pfeil die angehaltene Anweisung. Der Debugger-Tab enthält Aufrufliste und lokale Variablen. Einen Ausdruck wie `total + 100` eingeben und mit **Auswerten** den aktuellen Wert anzeigen.

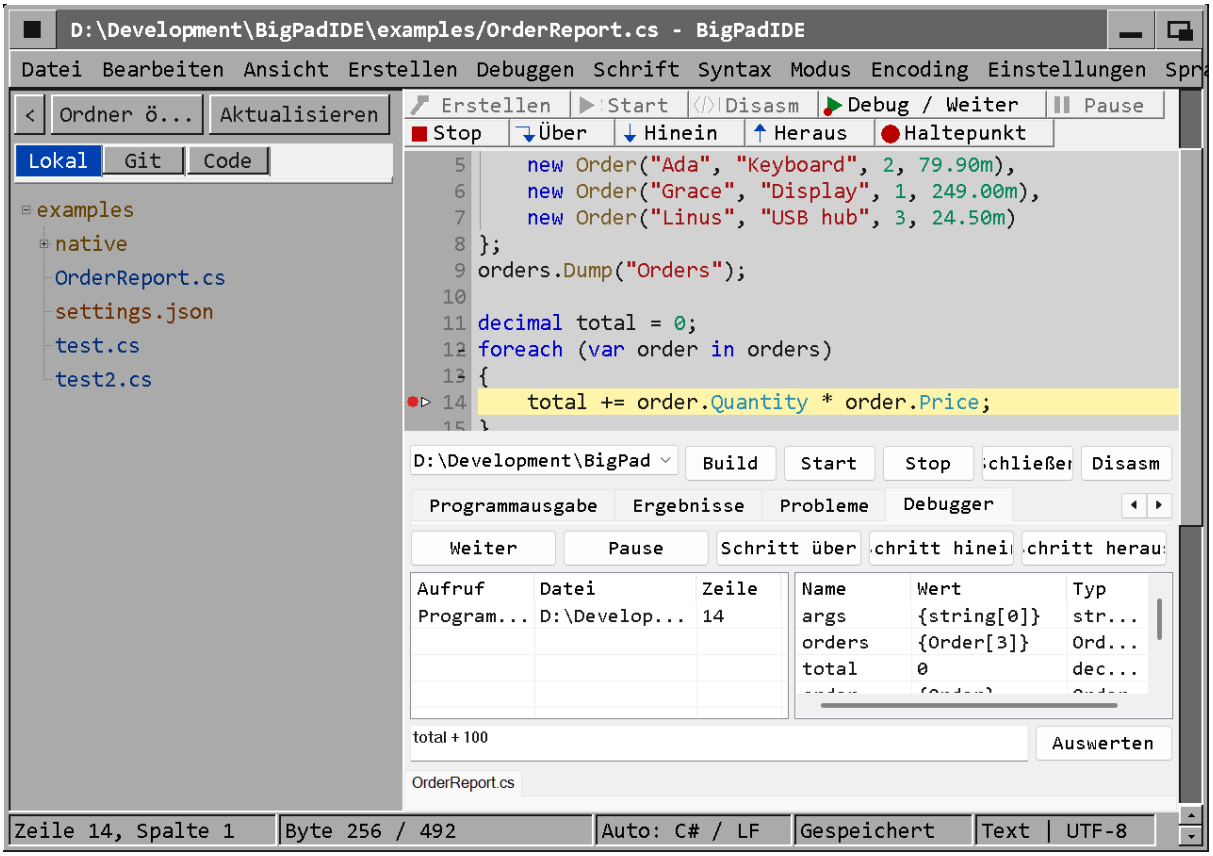


Abbildung 22. Echter Haltepunkt mit Aufrufliste, lokalen Variablen und ausgewertetem Ausdruck

F10 führt einen Schritt über eine Anweisung aus, **F11** springt hinein, **Umschalt+F11** heraus. **F8** pausiert, **Umschalt+F5** beendet die Sitzung. Während des Debuggens ist der Quelltext schreibgeschützt.

Umschalt+F9 / Debuggen → Haltepunkt-Bedingung bearbeitet vor dem Sitzungsstart die aktuelle Zeile. Eine leere Bedingung hält immer an; ein C#-Ausdruck wie `count == 3` nur bei wahrem Ergebnis. Das Kontrollkästchen Aktiviert erhält einen Haltepunkt, ohne ihn zu binden. Bedingte Haltepunkte erscheinen als gelbe Rauten, deaktivierte als gedämpfte Umrisse. Bedingungen und Aktivierung folgen Zeilenänderungen sowie Undo/Redo.

Haltepunkte werden unabhängig vom automatischen Speichern der UI-Einstellungen bei Änderungen dauerhaft in den Benutzereinstellungen gespeichert. Eine Titelwarnung meldet unlesbare oder ungesicherte Daten. Beschädigte Daten bleiben erhalten, bis **Debuggen → Alle Haltepunkte löschen** sie ausdrücklich ersetzt.

Debuggen → C#-Prozess anhängen zeigt zugängliche Prozesse mit PID und vollständigem Programmpfad. Einen bereits laufenden .NET-Prozess wählen; für Quellhaltepunkte müssen passende Quellen und portable PDBs verfügbar sein. Veraltete Auswahlen aktualisieren. Attach prüft die Erstellungszeit und baut oder startet das Ziel nicht neu. **Beenden** trennt die Sitzung und lässt den angehängten Prozess weiterlaufen; eine selbst gestartete Debugsitzung beendet dagegen ihr Ziel. Ungeeignete Ziele und Adapterfehler erscheinen in der Programmausgabe. Attach benötigt dieselbe .NET-SDK-/NetCoreDbg-Einrichtung wie ein Debugstart.

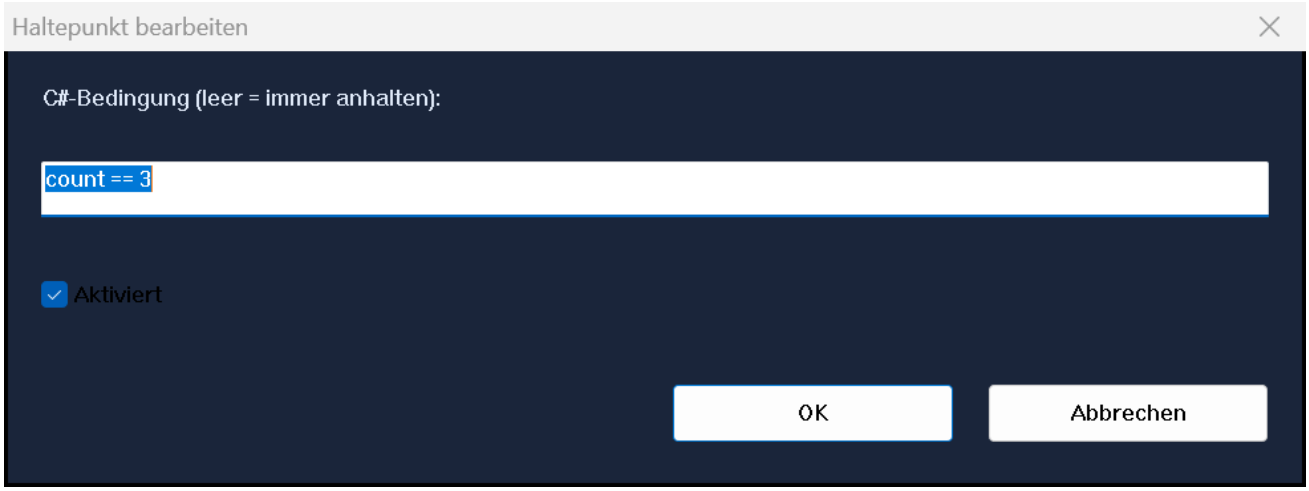


Abbildung 23. Dialog für bedingte Haltepunkte

9.3 Starten und Anhängen unterscheiden

Eine von BigPadIDE gestartete Debugsitzung besitzt ihren Zielprozess und beendet ihn beim Stoppen. Beim Anhängen an einen bereits laufenden Prozess trennt Stop die Verbindung und lässt den Prozess weiterlaufen. Kontrollieren Sie deshalb vor dem Anhängen PID und vollständigen Pfad; mehrere ähnlich benannte Prozesse sind kein eindeutiges Ziel.

Ohne passende Quellen und portable PDBs kann ein angehängtes Programm verbunden sein, ohne dass ein gewünschter Quellhaltepunkt gebunden werden kann. Ein roter Punkt im Editor allein beweist noch keinen gültigen Halt im laufenden Modul.

9.4 Werte unter dem Mauszeiger, Objektbrowser und Variablen ändern

Wenn die Sitzung angehalten ist, den Zeiger kurz über eine Variable im Quelltext halten. Der Tooltip zeigt Namen, aktuellen Laufzeitwert und Typ im ausgewählten Stackframe. Einfache Namen und Punktzugriffe wie `order.Price` werden erkannt; es wird kein beliebig markierter Code automatisch ausgeführt. Ob ein Mitglied ausgewertet werden kann, entscheidet der Debugadapter. Unbekannte oder nicht verfügbare Werte zeigen keinen veralteten Tooltip.

Das funktioniert mit C# über NetCoreDbg sowie mit unterstützten nativen Zielen über CodeLLDB. Beim Fortsetzen, Stackframewechsel, Scrollen oder Verlassen der Ansicht wird die alte Anzeige verworfen. Auch in der Variablenliste zeigt kurzes Verweilen Namen, Wert und Typ; lange Werte sind damit ohne Änderung der Spaltenbreite lesbar.

Das Wertefenster verwendet native Windows-Bedienelemente und eine gut lesbare, DPI-skalierte Schrift, standardmäßig mindestens 12 pt. Die Panels-Schrift aus **Einstellungen → Schriften und Farben** kann diese Vorgabe anpassen. Bei ruhender Maus bleibt es trotz der regelmäßigen Zustandsprüfung sichtbar.

Bei Collections und Objekten den kleinen Pfeil links am Wert anklicken. Der Objektbrowser lädt die Unterelemente erst dann aus dem Debugger. Arrayeinträge und deren Objektfelder können Sie weiter aufklappen. Der Zeiger darf vom Quelltext in dieses Fenster wechseln; erst beim Verlassen beider Bereiche schließt es nach etwa zwei Sekunden. Es öffnet sich dicht am Zeiger. Escape schließt es ebenfalls. Pro Anfrage werden bis zu 200 Elemente angezeigt, insgesamt bis zu 2000 Knoten im Wertefenster. Verfügbar sind die vom Debugadapter gelieferten Kinder; beliebige Enumerator-Methoden werden nicht von der IDE selbst aufgerufen.

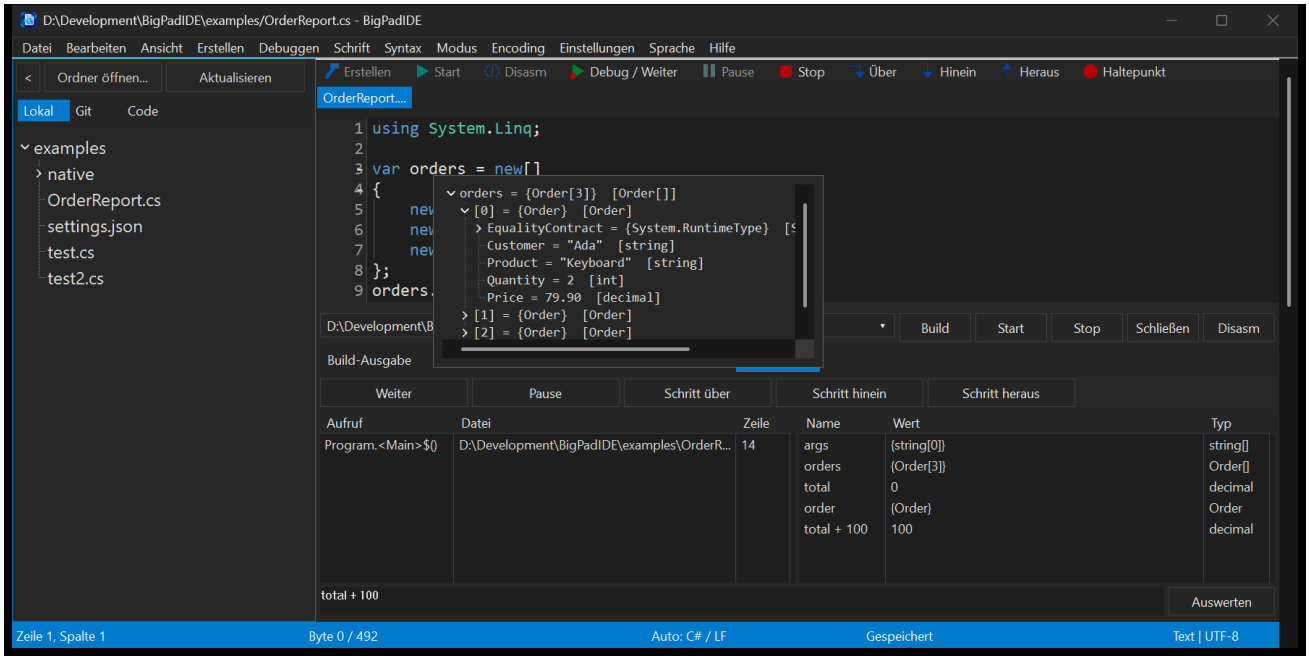


Abbildung 24. Aufgeklappte Collection mit Unterobjekten im Wertefenster

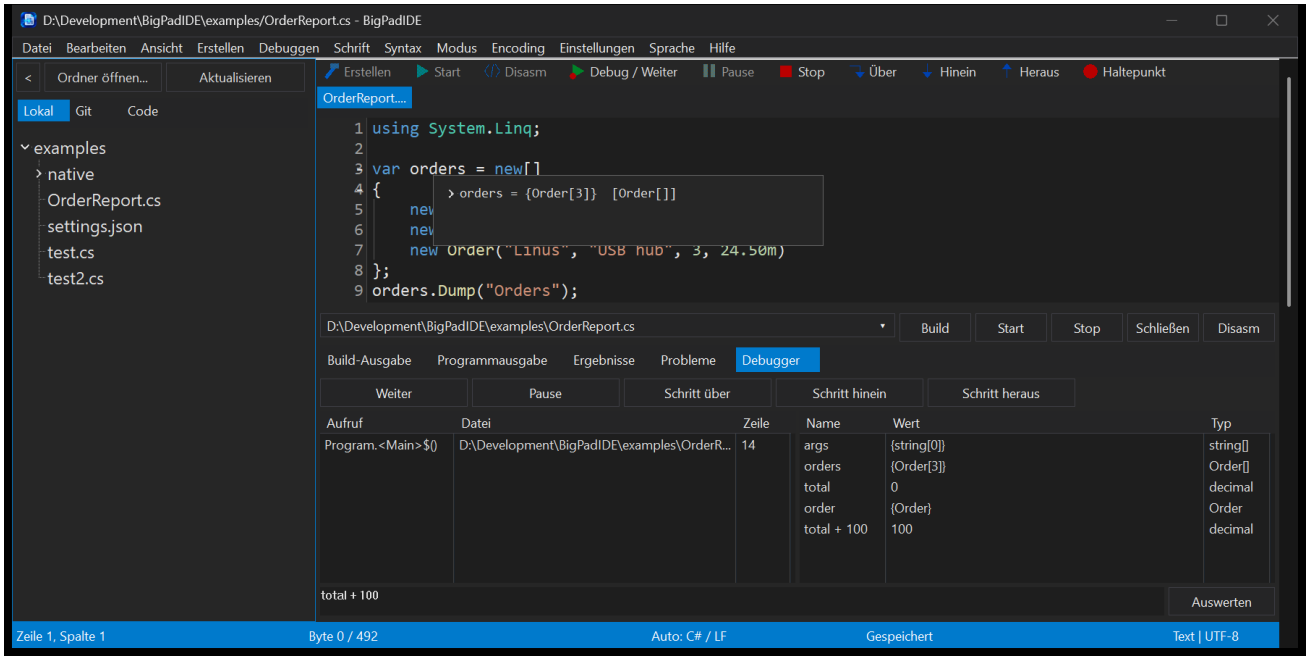


Abbildung 25. C#-Laufzeitwert direkt über einer Variablen im Quelltext

Zum Ändern einer Variable ihre **Wertspalte doppelklicken** oder die Zeile auswählen und **F2** drücken. Den neuen Wert in der Syntax der Debugsprache eingeben, etwa `42` für eine Ganzzahl oder `"Hallo"` für eine Zeichenkette. **Übernehmen** sendet die Änderung an den angehaltenen Prozess; **Abbrechen** verwirft den Entwurf. Der Quelltext und seine Undo-Historie werden dadurch nicht geändert.

Nach einer bestätigten Änderung werden lokale Variablen und vorhandene Watch- Ausdrücke erneut gelesen. Mit F5 läuft das Programm anschließend mit dem neuen Wert weiter. Ein Doppelklick auf den Namen eines aufklappbaren Werts öffnet dagegen seine Unterelemente. Der Wert kann auch dort über seine Wertspalte geändert werden.

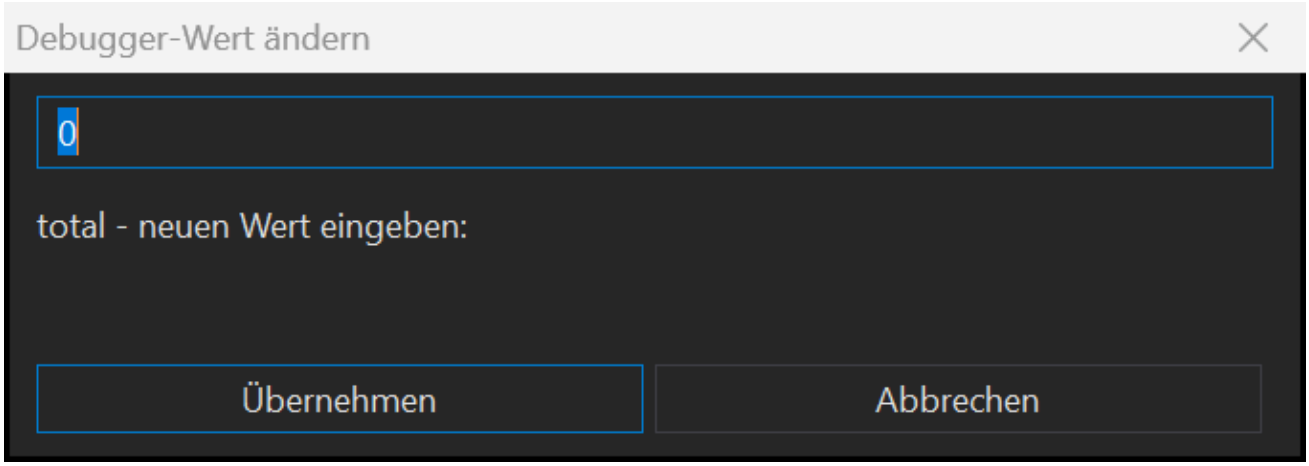


Abbildung 26. Dialog zum Ändern eines tatsächlichen Debugger-Variablenwerts

Die Änderung gilt für die Debugsitzung, nicht dauerhaft für die Quelldatei. Schreibgeschützte Werte, vom Adapter nicht unterstützte Zuweisungen und ungültige Eingaben werden abgewiesen. Watch-Ausdrücke sind nur schreibbar, wenn der Adapter das Zuweisen von Ausdrücken unterstützt; andernfalls die lokale Variable bearbeiten. Wenn während eines geöffneten Dialogs ein anderer Frame oder Laufzustand aktiv wird, ist der alte Bezug ungültig. Den Wert dann im aktuellen Kontext erneut auswählen. Adapterfehler erscheinen ausdrücklich in der Programmausgabe.

10. Native Projekte: C, C++ und Assembler

Die native Pipeline übersetzt Quellen mit externen Werkzeugen, linkt Windows- Programme oder erzeugt Rohbinaries. Build & Disassemblieren öffnet den erzeugten Maschinencode mit der passenden Architektur. Die Entwicklung benötigt keine Umstellung auf eine andere Editoroberfläche.

10.1 ASM, C und C++ bauen und starten

Öffnen Sie eine `.c`-, `.cpp`-, `.cc`-, `.cxx`-, `.asm`- oder `.s`/`.S`-Datei oder ein `bigpad.ini`-Projekt. Unter **Erstellen** bzw. im Ausgabepanel stehen Kompilieren (Strg+Umschalt+B), Build & Start (F6), Build & Disassemblieren (Strg+F6) und Stop bereit. Vor dem Build speichert die IDE geöffnete, geänderte Quelldateien des gewählten Projekts. Compilerfehler stoppen die Pipeline; ein veraltetes Programm wird dann nicht gestartet.

Die native Unterstützung verwendet GCC/G++, NASM und GNU objcopy, die beim Build-Start automatisch im PATH, unter `toolchains` neben der EXE und in üblichen Installationsordnern gesucht werden; `BIGPAD_CC`, `BIGPAD_CXX`, `BIGPAD_NASM` und `BIGPAD_OBJCOPY` können die ausführbaren Werkzeuge auswählen. Die Compiler sind nicht im Installer enthalten, er bietet fehlende Toolchains aber zum Herunterladen an. C#-Builds, Live C# und der C#-Debugger bleiben verfügbar. Native Programme haben erfasste Ausgabe, und zeilenweise Standardeingabe; natives Debugging ist im folgenden Abschnitt beschrieben.

Sechs Projekte mit Anleitung liegen in examples/native und werden unter `examples/native` neben die gebauten Programme kopiert. Für fünf davon reichen die lokal vorhandenen Werkzeuge; das ARM64-Rohbeispiel benötigt eine zusätzliche Cross-Toolchain. Die Projektdatei erlaubt mehrere Quellen, eine Architektur und die Wahl zwischen Windows-Konsolenprogramm und Rohbinary. Ausgaben landen in `.bigpad-build`. Build & Disassemblieren öffnet automatisch die extrahierte `.text`-Sektion bzw. das Rohbinary mit der passenden Decoder-Architektur.

Bei einer Installation für alle Benutzer bleiben die Toolchain-Pakete immer anwählbar, da im Benutzerprofil des Installierenden gefundene Compiler anderen Konten fehlen können. Die Deinstallation entfernt nur protokollierte installierte Dateien; zusätzliche manuell abgelegte Toolchain-Dateien bleiben erhalten.

10.2 Ein Projekt aus mehreren Quellen

Im Projektordner eine Datei `bigpad.ini` anlegen:

```
[project]
name=cpp-asm-sum
architecture=x64
kind=console
sources=main.cpp;sum.asm
```

`sources` enthält mit Semikolon getrennte relative Pfade. Die Erweiterungen bestimmen das Werkzeug: `.c` wird als C17, `.cpp`/`.cc`/`.cxx` als C++20 übersetzt, `.asm` durch NASM und `.s`/`.S` über den GNU-Compiler-Treiber. Bei C++-Quellen übernimmt G++ das Linken. Die Projektarchitektur und NASM-`bits`-Direktive müssen zusammenpassen.

`architecture` unterstützt `x64`, `x86`, `arm`, `thumb` und `arm64`. `kind=console` ist für Windows-x86/x64-Programme vorgesehen. `kind=raw` benötigt genau eine Quelle; ARM/Thumb/ARM64 werden als Rohziele behandelt. Ein Rohbinary ist kein Windows-Programm und lässt sich nicht wie eine EXE starten.

10.3 Die sechs mitgelieferten Demos

Projektordner	Lernziel	Erwartetes Ergebnis
`asm-hello-x64`	NASM und Windows-x64-Aufruf	Hello from x64 NASM!
`gas-hello-x64`	GNU-Assembler in Intel-Syntax	Hello from x64 GNU assembler!
`cpp-asm-sum`	C++ ruft eine ASM-Funktion	ASM sum 1..10 = 55
`c-fibonacci`	C17 und Funktionsaufrufe	C Fibonacci(10) = 55
`asm-raw-x86`	Rohbytes und Befehlsdekodierung	Nur Build/Disassembler
`arm64-raw`	ARM64-Cross-Build	Nur Build/Disassembler; Cross-Toolchain nötig

Vor dem Arbeiten einen Demoordner aus `examples/native` in ein beschreibbares eigenes Verzeichnis kopieren. Die Ausgaben entstehen in `.bigpad-build`. Die aktuelle App unterstützt zeilenweise Eingabe im Run-Modus und natives Debugging; ältere Hinweise in einzelnen Demoanleitungen können diese später ergänzten Möglichkeiten noch nicht berücksichtigen.

10.4 Werkzeugpfade ausdrücklich setzen

Die Variablen `BIGPAD_CC`, `BIGPAD_CXX`, `BIGPAD_NASM` und `BIGPAD_OBJCOPY` enthalten jeweils den Pfad zur ausführbaren Datei, ohne zusätzliche Argumente. Für ARM64 existieren außerdem `BIGPAD_ARM64_CC`, `BIGPAD_ARM64_CXX` und `BIGPAD_ARM64_OBJCOPY`. Ein gesetzter ungültiger Pfad wird gemeldet und nicht stillschweigend durch einen anderen Compiler ersetzt.

Ein PATH-Eintrag, ein Toolchain-Ordner neben der EXE oder eine erkannte Installation kann die Werkzeuge ebenfalls liefern. Bei mehreren Toolchains empfiehlt sich die ausdrückliche Auswahl, damit Compiler und objcopy zur gleichen Architektur gehören.

11. Natives Debugging

Für native Windows-Programme verwendet BigPadIDE CodeLLDB. Die Bedienung von Haltepunkten, Schritten, Aufrufliste und Watches entspricht weitgehend dem C#-Debugger. Symbolformat, Architektur und Quellpfade müssen zum Programm passen.

11.1 Natives Debugging

`tools/Setup-NativeDebugger.ps1` installiert einmalig das festgelegte, per SHA-256 geprüfte CodeLLDB-Paket 1.12.3 unter `build/debugger`. Alternativ `BIGPAD_CODELLDB` auf `extension/adapter/codelldb.exe` einer vollständigen Installation setzen; die benachbarten LLDB-/Python-Dateien werden benötigt. Natives Debugging nutzt dieselbe .NET-Helper-Runtime wie die C#-Werkzeuge. Der vorhandene Installer wird nicht neu gepackt.

Für native Quellen oder ein `bigpad.ini`-Ziel setzt **F9** Quellhaltepunkte; **F5** speichert geänderte Projektquellen, baut und startet das Ergebnis mit CodeLLDB. Compilerfehler verhindern den Start. Debug-Builds hängen `-g -O0` nach den Build-Profilargumenten an; NASM fordert CodeView-Informationen an. **Debuggen → Natives Programm starten** wählt eine vorhandene ausführbare Datei ohne Neubau. Passende Quellen und Debugdaten werden benötigt; geprüft ist MinGW/GCC mit DWARF auf Windows x64. NASM-Quellpositionen hängen von den verfügbaren Symbolinformationen ab. Das Run-Profil liefert Argumente, Umgebung und Arbeitsverzeichnis; ohne Vorgabe gilt das Quell-/Projektverzeichnis. Vorhandene Programme lesen ihr eigenes Sidecar-Profil. Rohbinaries und ARM-Cross-Ziele bleiben Disassembler-Ziele.

Die bekannten Tasten **F10 / F11 / Umschalt+F11 / F8** steuern Schritte und Pause. Der Debugger-Tab zeigt Aufrufliste, lokale Variablen, Register und Watch-Werte; Doppelklick wählt einen Frame oder erweitert einen Wert. **Umschalt+F9** bearbeitet native Bedingungen vor dem Sitzungsstart. Quelltexte bleiben während des Debuggens schreibgeschützt. Ein anfänglicher Windows-Runtime-Haltepunkt kann ohne eigene Quellposition erscheinen; Weiter führt zum Quellhaltepunkt. Programmausgabe und Debugger-Protokoll sind getrennt. Interaktive Standardeingabe steht im Run-Modus bereit; der Debugger verwendet seinen Eingabekanal für Debuggerbefehle.

Debuggen → Nativen Prozess anhängen wählt einen bestehenden Prozess mit PID und Programmpfad, prüft dessen Erstellungszeit und hängt sich ohne Neubau an. Beenden trennt angehängte Ziele und lässt sie weiterlaufen; selbst gestartete Ziele werden beendet. Veraltete Auswahlen aktualisieren und ein für das aktuelle Windowskonto zugängliches Ziel wählen.

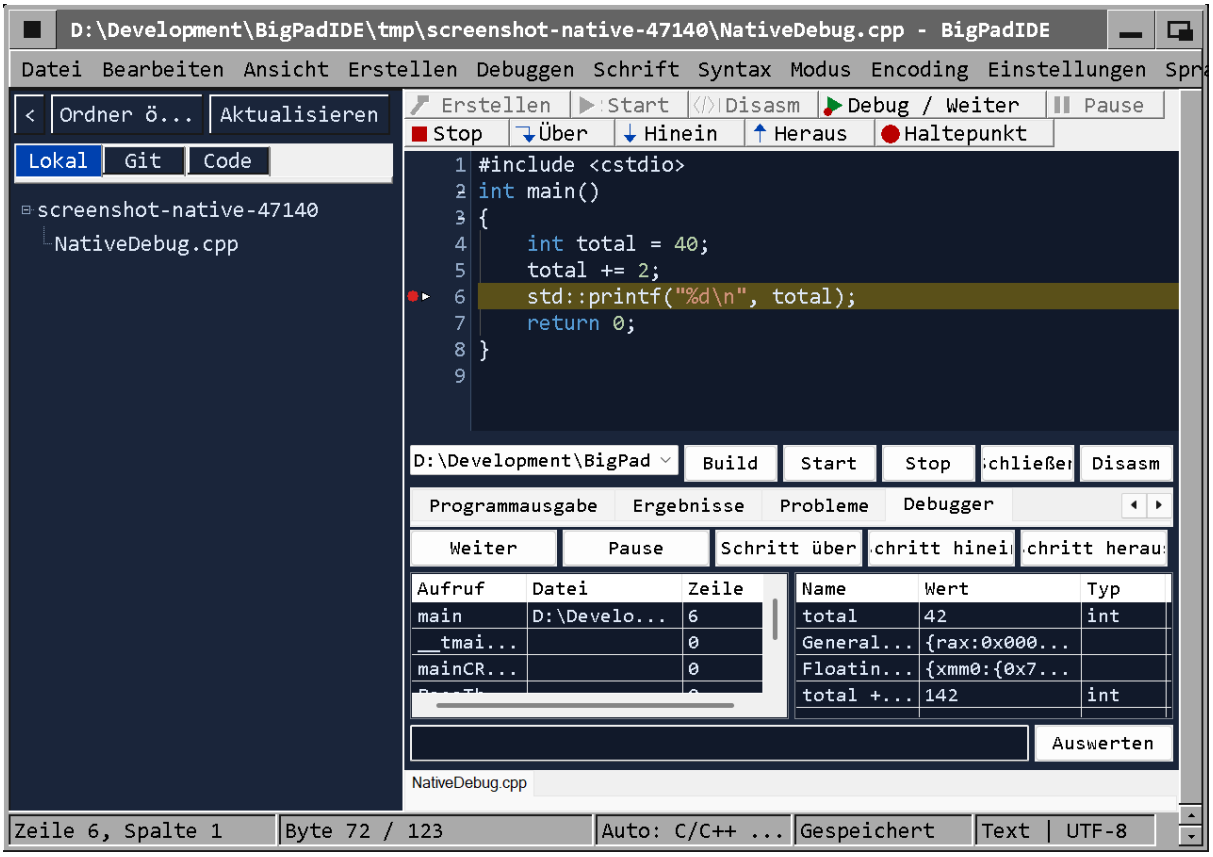


Abbildung 27. Nativer Haltepunkt, Aufrufliste und ausgewerteter Watch-Ausdruck

Adapter: CodeLLDB-Release, Windows-Debugformate.

11.2 Fehlende Quellpositionen einordnen

Bei einem ersten Halt in einer Windows-Laufzeitfunktion kann die eigene Quelle fehlen. **Weiter** führt zum gesetzten Quellhaltepunkt, sofern er gebunden wurde. Optimierte oder ohne passende Debugdaten gebaute Programme liefern weniger verlässliche Variablen und Zeilenpositionen. Für reproduzierbare Versuche zuerst ein mit der IDE gebautes x64-Demo im Debugmodus verwenden.

Ein CodeLLDB-Adapter ist mehr als eine einzelne EXE. Die benachbarten LLDB- und Python-Dateien gehören dazu. Ein auf eine herauskopierte Adapterdatei gesetztes `BIGPAD_CODELLDB` kann deshalb trotz existierender Datei unvollständig sein.

12. Hex-Editor und Bytewerkzeuge

Im Hexmodus bearbeiten Sie Bytes statt dekodierter Textzeichen. Das ist nützlich, um Dateiköpfe zu prüfen, Datenbereiche zu finden und Zahlen in unterschiedlichen Byte-Reihenfolgen zu interpretieren. Vor größeren Experimenten eine Kopie speichern.

12.1 Text, Bytes und Offset

Modus → **Hex-Editor** zeigt Offset, Hexwerte und eine Textdarstellung. Ein Byte wird durch zwei Hexziffern dargestellt: `41` entspricht beispielsweise dem ASCII-Zeichen A. Ein Offset ist eine Position, kein virtueller Speicherwert. In Textnavigation zählt ein UTF-8-BOM nicht zum logischen Dokumentoffset; für Containeradressen im Disassembler gelten die dort beschriebenen Zuordnungen.

Die aktuelle Darstellung ändert nicht automatisch das Encoding. Auch im Hexmodus bleibt die Encoding-Anzeige sichtbar. Für gewünschte Textänderungen ist der Textmodus meist übersichtlicher, für genaue Binäränderungen der Hexmodus.

12.2 Hex-Werkzeuge: Bytesuche, Werte und Export

Im **Modus** → **Hex-Editor** mit **Modus** → **Hex-Werkzeuge** oder **Strg+F** öffnen. Vollständige Hex-Bytepaare wie `DE AD 00 FF` eingeben; Leerzeichen und kompakte Schreibweise sind erlaubt. Die exakte Suche berücksichtigt NUL-/hohe Bytes und ungespeicherte Änderungen. Die Suchfolge ist auf 64 KiB begrenzt. **Weitersuchen / F3** beginnt hinter der Auswahl und läuft einmal über das Dateiende zurück. Abbrechen stoppt die blockweise Suche; Textersetzungen bleiben im Textmodus.

Die Zahlenliste folgt dem Auswahlbeginn oder ohne Auswahl dem Cursor. **Little endian** und **Big endian** steuern vorzeichenlose/vorzeichenbehaftete Ganzzahlen mit 8, 16, 32 und 64 Bit sowie IEEE float32/float64. Höchstens acht Bytes werden gelesen; bei kürzerer Auswahl oder Dateiende stehen nicht verfügbare Werte als Strich. NaN, Unendlich und negative Null bleiben erkennbar. Die Byte-Reihenfolge ändert nur die Interpretation.

Bereich exportieren verwendet Startoffset und Byteanzahl als Dezimalzahl oder mit `0x`-Präfix, zunächst aus der Auswahl übernommen. Einen neuen Dateinamen wählen: Vorhandene Ziele einschließlich der Quelle bleiben erhalten. Exportiert werden die sichtbaren logischen Bytes einschließlich ungespeicherter Änderungen, ohne BOM-Metadaten voranzustellen oder Encoding/Byte-Reihenfolge zu konvertieren. Erst nach vollständigem Schreiben wird das Ergebnis atomar veröffentlicht. Abbruch, Lesefehler oder eine geänderte Quelle hinterlassen kein Teilziel. Unveränderte Quelldateien sind während des Exports gegen direkte Schreibzugriffe gesperrt. Schließen des Fensters bricht seine Arbeit ab. Eingabefelder haben eigenes Text-Undo und Strg+A unabhängig vom Quelltext-Undo.

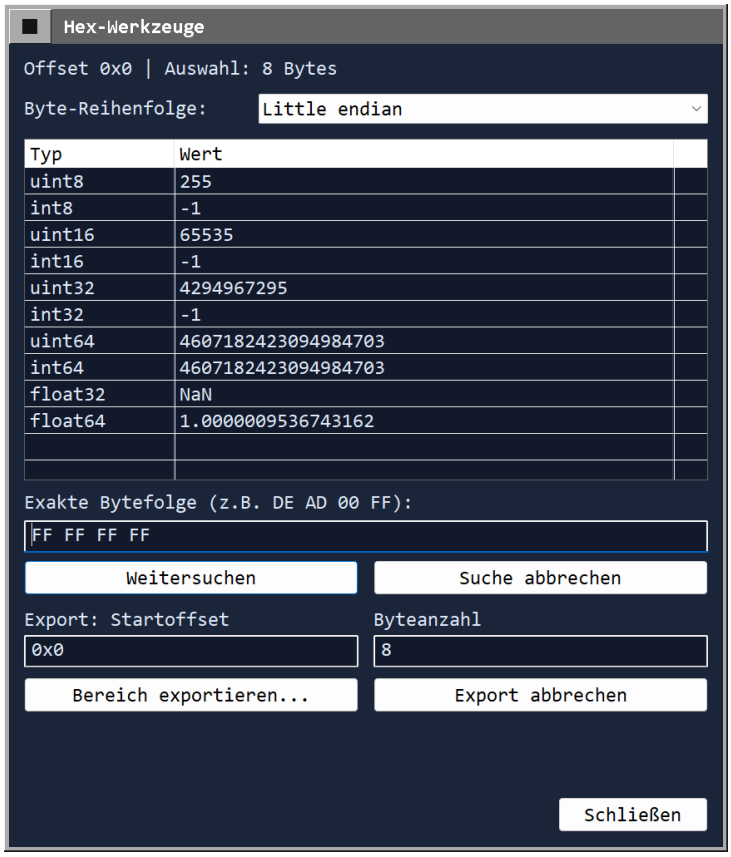


Abbildung 28. Byte-Interpretation, exakte Suche und Bereichsexport

12.3 Zahlen richtig interpretieren

Die Bytes `01 00 00 00` ergeben als vorzeichenlose 32-Bit-Zahl Little Endian den Wert 1, Big Endian dagegen 16.777.216. Das Umschalten der Interpretation ändert keine Bytes. Ein sinnvoll erscheinender Wert ist noch kein Beweis für das Format; verwenden Sie die Spezifikation der untersuchten Datei oder bekannte Testdaten.

12.4 Bereichsexport als eigener Arbeitsstand

1. Den gewünschten Startoffset und die Byteanzahl bestimmen.
2. In den Hex-Werkzeugen **Bereich exportieren** wählen.
3. Einen neuen Zielnamen angeben; vorhandene Dateien werden nicht überschrieben.
4. Nach Abschluss die erzeugte Datei separat öffnen und Größe/Inhalt prüfen.

Der Export liest die logischen Dokumentbytes einschließlich ungespeicherter Änderungen. Er ersetzt nicht das Speichern der ursprünglichen Datei.

13. Nativer Disassembler und PE/ELF

Der native Disassembler übersetzt Maschinenbytes in Assemblerbefehle. Er erzeugt keinen C#-Quellcode aus einem nativen Programm. Für verwaltete .NET-Assemblies steht die gesonderte IL-/C#-Ansicht in Kapitel 14 zur Verfügung.

13.1 Disassembler

Öffnen Sie eine Binärdatei und wählen Sie **Modus → Disassembler**. Verfügbar sind x86 (16/32 Bit), x64, ARM, ARM Thumb und ARM64. ARM-Modi verwenden Little Endian. Jede Zeile zeigt den Dateioffset, die Maschinenbytes und den Assemblerbefehl. Ungültige oder unvollständige Befehle erscheinen als `.byte`. Abwechselnd helle und grünliche Zeilen über die gesamte Breite verbessern wie Endlospapier die Lesbarkeit; im Dunkelmodus werden entsprechend dunkle Streifen verwendet.

Pfeiltasten und Mausrad bewegen die Ansicht durch die Befehle; Bild auf/ab blättert. Pos1 springt zum Anfang, Ende zum letzten Dateiausschnitt. Über den Bildlaufbalken können Sie zu einem Byteoffset springen. Bei x86 muss der Startoffset auf einem Befehlsanfang liegen; nach einem beliebigen Sprung ist die Synchronisation nicht eindeutig. Einen Startpunkt können Sie auch im Hex-Editor auswählen und danach den Disassembler aktivieren. ARM und ARM64 werden auf vier Bytes, Thumb auf zwei Bytes ausgerichtet.

Die Ansicht ist schreibgeschützt und merkt sich den Modus je Tab. Zum Bearbeiten wechseln Sie zurück zu **Modus → Hex-Editor**. Es werden nur begrenzte Ausschnitte decodiert. Die Ansicht interpretiert Rohbytes: PE-/ELF-Sektionen, virtuelle Ladeadressen, Symbole und ausführbarer Code werden nicht automatisch erkannt.

13.2 PE-/ELF-Disassembler und Sprungziele

Modus → Disassembler → PE/ELF analysieren liest Sektionen, bevorzugte virtuelle Adressen und vorhandene COFF-/ELF-Symbole. Die Architektur wird aus dem Header gewählt. Die erste Spalte bleibt der Dateioffset, danach folgt die virtuelle Adresse. Diese Adresse stammt aus der Datei und berücksichtigt keine ASLR-Ladeverschiebung eines laufenden Prozesses. Relative Operanden werden mit der virtuellen Adresse dekodiert.

Sektionen / Symbole / VA öffnet eine filterbare Liste. Namen ohne Beachtung der Großschreibung filtern oder eine genaue virtuelle Adresse mit `0x` eingeben. Maximal 500 Treffer werden gleichzeitig gezeigt; weiteres Filtern erreicht auch spätere Symbole. Dateilose Bereiche wie BSS und mehrdeutige Zuordnungen sind nicht navigierbar. **Enter / Sprungziel folgen** folgt einem direkten Sprung/Call an der Cursorposition, **Rücktaste / Zurück** kehrt zurück (bis 256 Positionen). Register-/Speicherziele indirekter Sprünge benötigen Laufzeitzustand und werden nicht geraten.

Die sechs manuellen Architektureinträge schalten jederzeit in den bisherigen Rohmodus mit Dateioffsets zurück. PE/ELF-Zuordnung wird pro Tab geführt, bei Wiederherstellung zunächst als Rohmodus geöffnet und bei Bedarf neu analysiert. Metadaten werden aus den aktuellen logischen Dokumentbytes gelesen; keine externen Symboldownloads/PDB-Analyse. Unterstützt sind PE32/PE32+ und ELF32/ELF64; ELF-Metadaten auch Big Endian, die Decoder nur die vorhandenen Little-Endian-Architekturen. Relokierbare ELF-Objekte benötigen ein Linkerlayout und bleiben roh. ELF ohne Sektionstabelle/mit erweiterten Zählfeldern und Metadaten oberhalb der Grenzen werden ausdrücklich abgewiesen: 4096 Sektionen, 50000 Symbolrecords, 1

MiB pro Stringtabelle, 4 MiB Symbolnamen und 16 MiB Gesamtlesen. Beschädigte Header veröffentlichen kein teilgültiges Modell; Rohmodus bleibt verfügbar.

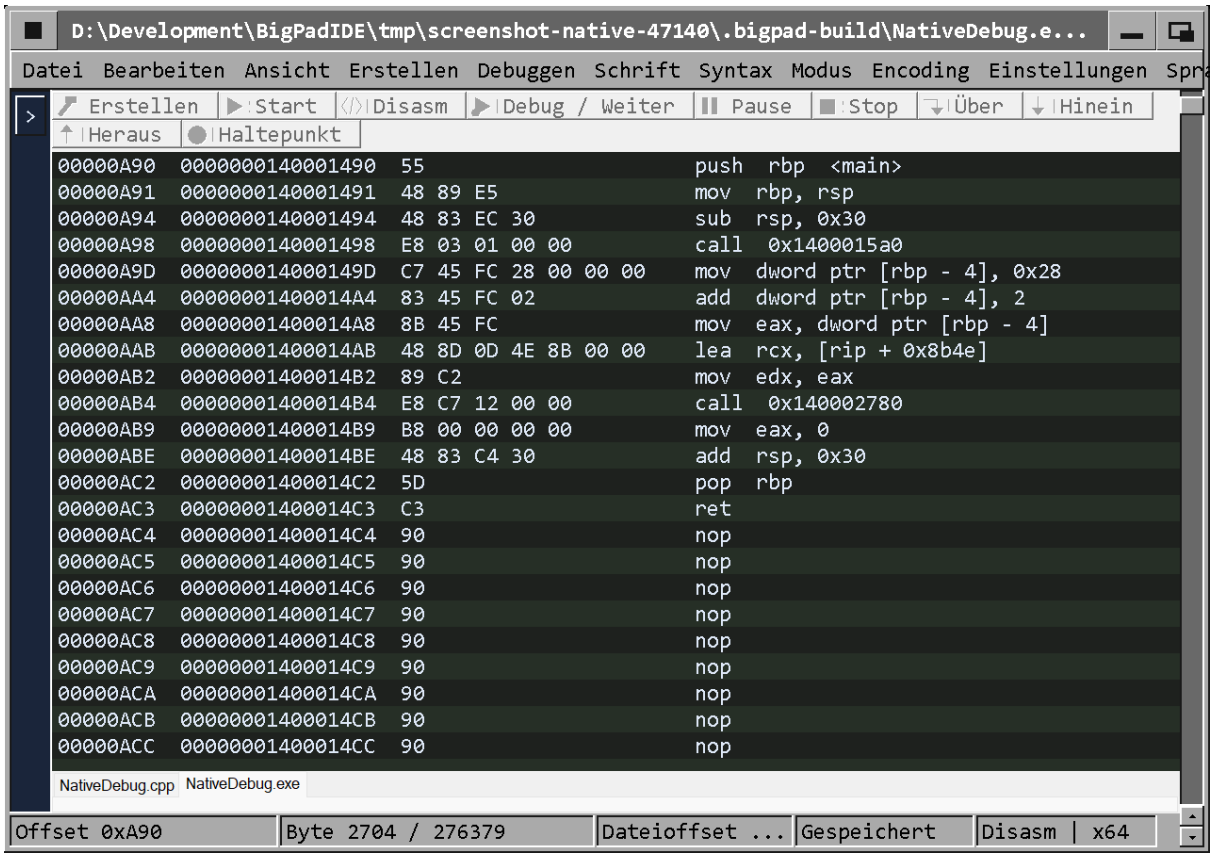


Abbildung 29. PE/ELF

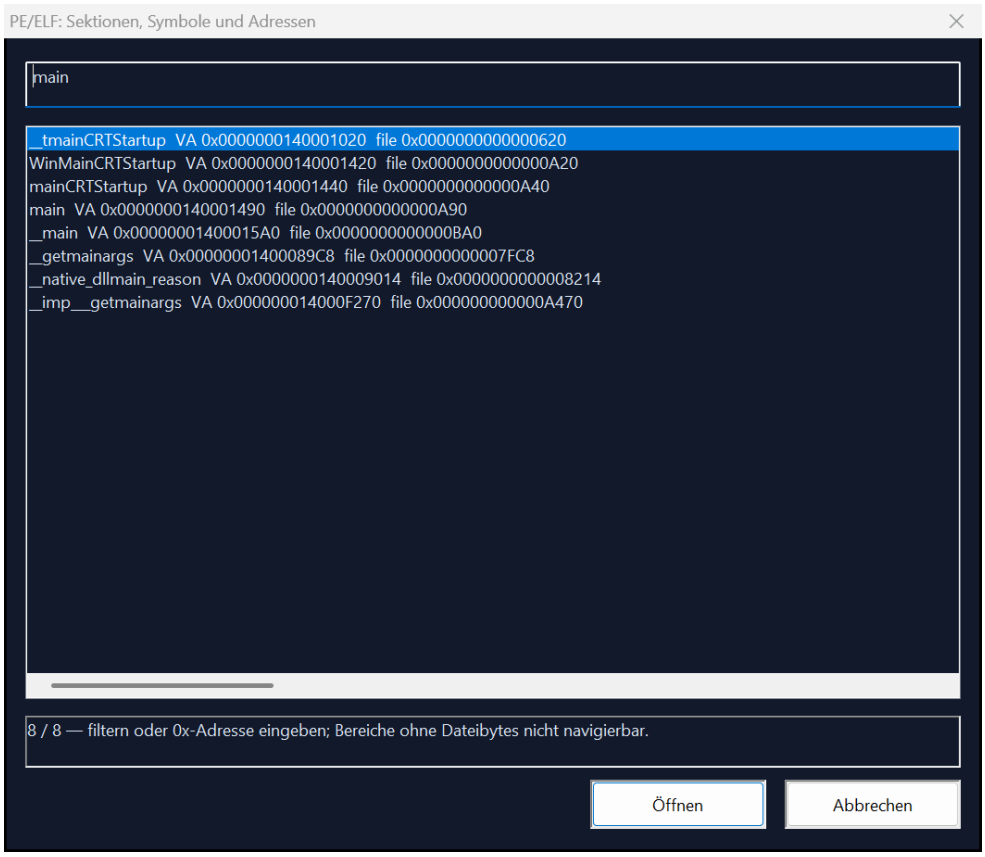


Abbildung 30. Symbols

13.3 Welche Ansicht passt zur Datei?

Eingabe	Vorgehen
Reiner Maschinencode oder extrahierte `.text`-Datei	Architektur manuell im Rohmodus wählen
Native Windows-EXE oder DLL	PE/ELF analysieren; anschließend Sektionen/Symbole nutzen
ELF mit unterstütztem ausführbarem Layout	PE/ELF analysieren; Header bestimmt Architektur
Relokierbares Objekt ohne endgültiges Layout	Rohmodus; keine sichere virtuelle Zuordnung voraussetzen
Verwaltete .NET-DLL	.NET-Assembly: IL / C# öffnen
Moderne native .NET-Apphost-EXE	Zum verwalteten Code die benachbarte .dll öffnen

Die Zeilenoffsets im Rohbinary unterscheiden sich von Offsets in einer ursprünglichen PE-Datei. Virtuelle Adressen sind wiederum eine dritte Größe. Beim Notieren eines Fundorts immer dazuschreiben, welche Datei und welche Art von Adresse gemeint ist.

14. .NET-Assemblies: IL und C# rekonstruieren

Die Assembly-Ansicht liest Typen und Methoden aus einer verwalteten DLL/EXE. Sie bietet sowohl IL als auch rekonstruiertes C#. So lassen sich eigene Builds, Bibliotheken und Compilerergebnisse untersuchen, ohne das untersuchte Programm zu starten.

14.1 .NET-Assemblies: IL und rekonstruiertes C#

Modus → Disassembler → .NET-Assembly: IL / C#… öffnet die Ansicht. Eine bereits geöffnete DLL/EXE dient als Vorgabe; sonst **Assembly öffnen…** verwenden. Moderne SDK-Anwendungen besitzen meist eine native Apphost-EXE: dafür die verwaltete `.dll` auswählen. Native PE-Dateien werden mit einer Fehlermeldung abgewiesen.

Codevorschau und Typliste übernehmen Schriftart und Größe des Editors. Änderungen über die Schrift-/Größenmenüs aktualisieren auch ein bereits geöffnetes IL-/C#-Fenster sofort.

Das Fenster liest zuerst die Typliste und zeigt danach den ersten Benutzertyp. Links einen anderen Typ einschließlich verschachtelter Typen wählen; der erste Eintrag steht für die gesamte Assembly. Die Sprachauswahl wechselt zwischen IL mit Offsets und rekonstruierten C#-Klassen/Methodenkörpern. **Neu lesen** liest den Dateistand erneut. **Abbrechen** beendet den Worker und verwirft unfertigen Code. Aus der schreibgeschützten Vorschau kopieren oder **Als Texttab öffnen** verwenden, um ein separates Dokument zu bearbeiten/speichern. Automatische C#-Live-Ausführung wird beim Öffnen des erzeugten Textes deaktiviert.

C# wird aus IL rekonstruiert; ursprüngliche Kommentare, Formatierung und wegoptimierte Informationen sind nicht wiederherstellbar. Fehlende Referenzen, nicht unterstützte oder verschleierte Methoden können die Rekonstruktion begrenzen; IL bleibt unabhängig verfügbar. Die Ansicht führt weder Eingabecode noch Modulinitialisierer aus.

Grenzen: 128 MiB Eingabe, 10.000 sichtbare Typen, 64 Verschachtelungsebenen, 4 MiB UTF-8 / 2.097.152 UTF-16-Einheiten, 30 Sekunden und 1 GiB pro Helperprozess. Bei großen Assemblies einzelne Typen auswählen. Nach einem Codefehler bleibt eine vollständige Typliste bedienbar. Assemblies und Referenzen werden vom Datenträger statt aus ungespeicherten Tabs gelesen. ILSpy muss nicht separat installiert werden.

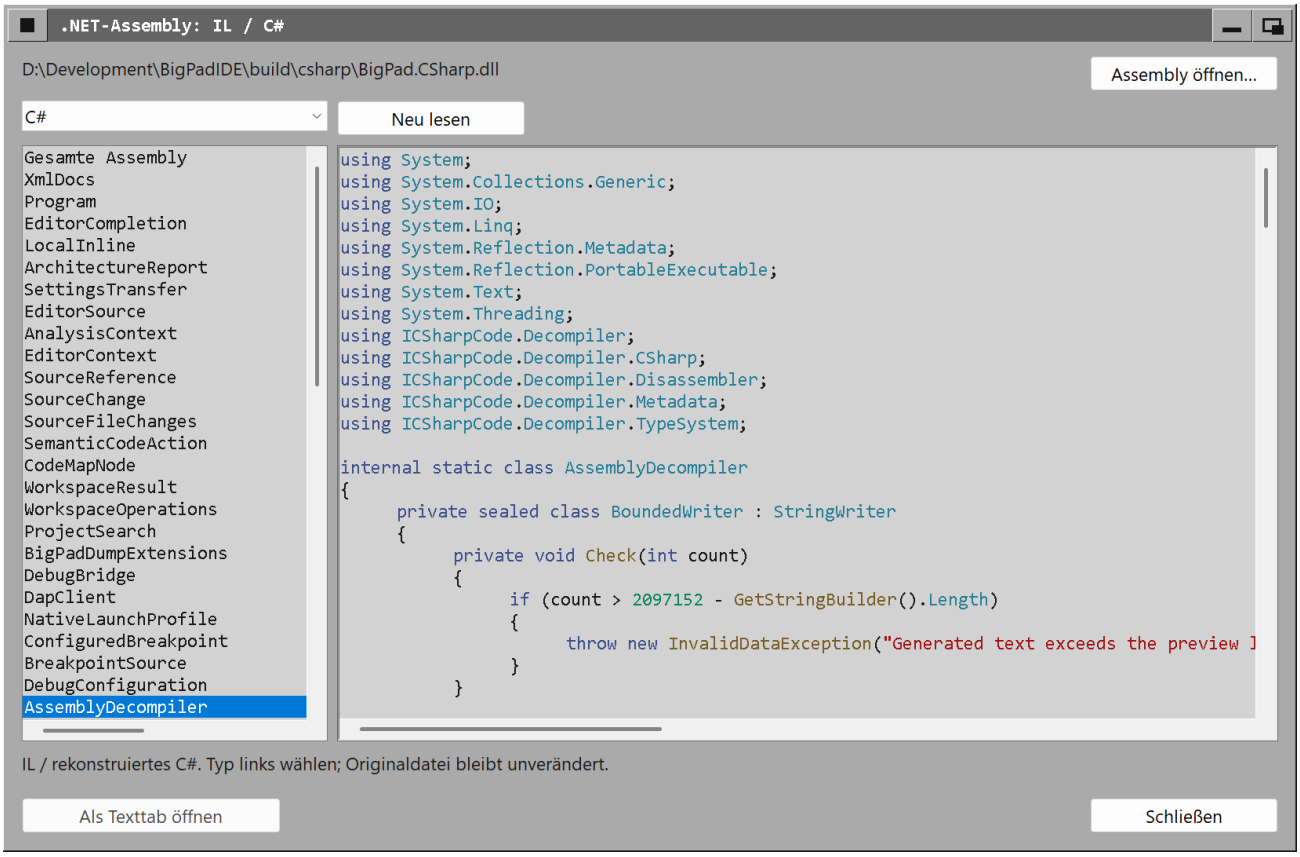


Abbildung 31. Rekonstruiertes C# und Typliste

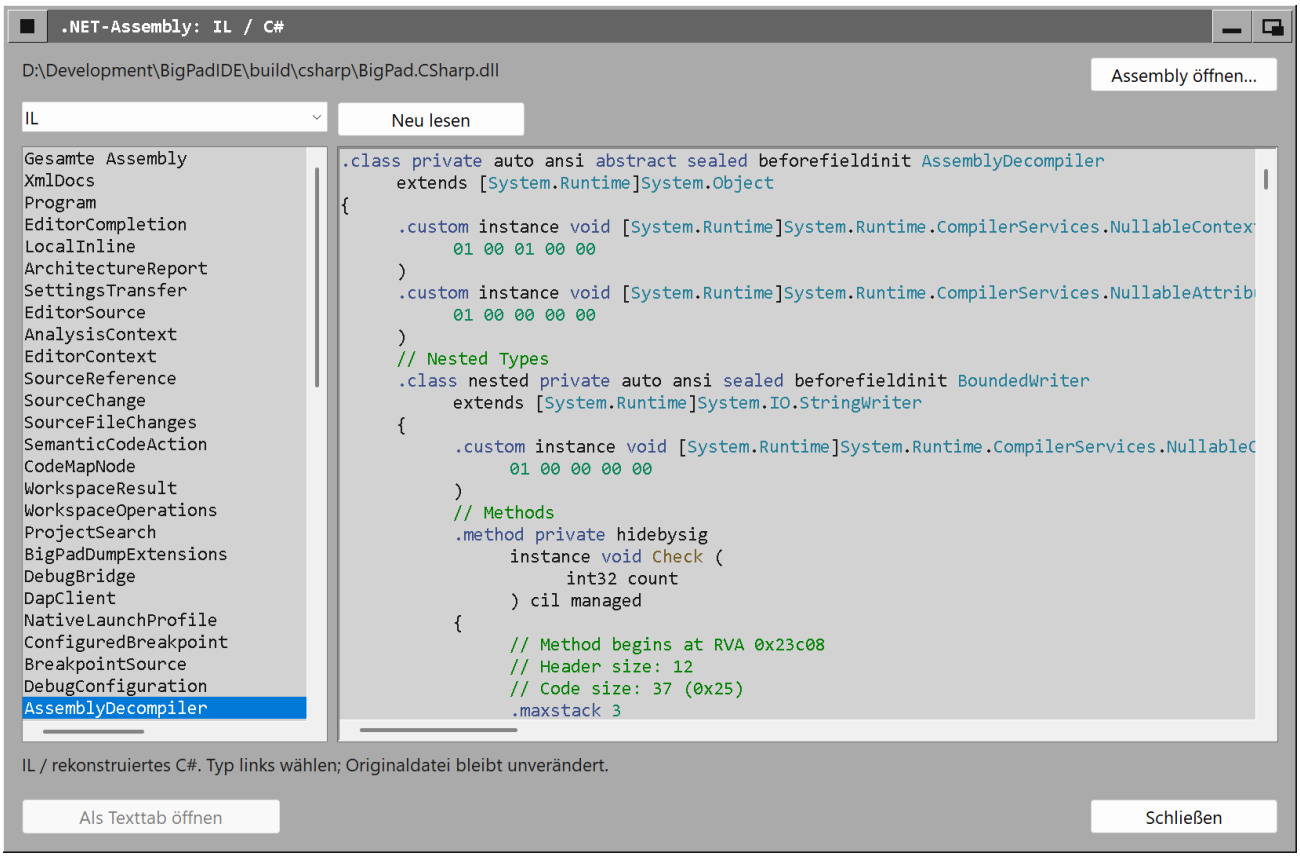


Abbildung 32. IL-Ansicht desselben Typs

14.2 Ein vollständiger Analyseablauf

1. Das betreffende Projekt bauen und den Ausgabeordner bestimmen.
2. **Modus** → **Disassembler** → **.NET-Assembly: IL / C#** wählen.
3. Bei SDK-Anwendungen die verwaltete DLL statt einer nativen Apphost-EXE wählen.
4. Links zunächst einen einzelnen Benutzertyp auswählen.
5. Zwischen **C#** und **IL** wechseln und dieselbe Methode vergleichen.
6. Für die ganze Assembly den ersten Listeneintrag verwenden. Bei umfangreichen Assemblies einzelne Typen bevorzugen, damit Zeit- und Ausgabegrenzen passen.
7. Bei Bedarf **Als Texttab öffnen** und den erzeugten Text separat speichern.

Das Texttab ist ein neues Dokument. Es überschreibt die Assembly oder ursprüngliche Quellen nicht. Ein neuer Build der DLL wird erst mit **Neu lesen** eingelesen. Ungespeicherte Quelldateitabs sind keine Eingabe dieser Ansicht.

14.3 Was die Rekonstruktion aussagt

IL ist die in der Assembly gespeicherte Zwischensprache. C# ist eine daraus abgeleitete verständlichere Darstellung. Beispielsweise können vom Compiler erzeugte Zustandsautomaten als async-Strukturen rekonstruiert werden. Die ursprüngliche Schreibweise, Kommentare und wegoptimierte Informationen stehen dadurch nicht wieder zur Verfügung.

Bei unaufgelösten Referenzen, Verschleierung oder nicht unterstützten Konstrukten eine einzelne Methode beziehungsweise einen einzelnen Typ in IL untersuchen. Eine Fehlermeldung bei C# bedeutet nicht, dass die IL-Ansicht denselben Rekonstruktionsschritt benötigt. Die Codevorschau ist schreibgeschützt; das Texttab dient der eigenen Weiterbearbeitung.

14.4 Schrift und Fensterplatzierung

Schrift → **Schriftart/Schriftgröße** steuert auch Codevorschau und Typliste live. **Einstellungen** → **IL / C# öffnen** bestimmt Tab oder eigenes Fenster. Nach einer Platzierungsänderung eine schon offene Ansicht schließen und erneut öffnen. Für Diff gibt es einen unabhängigen Schalter.

15. Bilder, Frames und Pixelwerte

Die Bildansicht dient zum Betrachten und Untersuchen von PNG, JPEG, BMP, GIF und TIFF. Sie ist keine Bildbearbeitung. Zoom, Verschieben und die Auswahl eines Frames oder einer Seite ändern die Bilddatei nicht.

15.1 Bilder bedienen

PNG, JPEG, BMP, GIF und TIFF öffnen als schreibgeschützte Bildansicht. **Ansicht** → **Bild** enthält Einpassen (**0**), 100 % (**1**), Vergrößern/Verkleinern (**+ / -**) und Frame-/Seitennavigation. Das Mausrad zoomt um den Bildpunkt unter dem Zeiger; linke Maustaste ziehen verschiebt ein vergrößertes Bild. Pfeiltasten verschieben um 32 Bildschirmpixel. Bildzoom ist unabhängig von der Schriftgröße des Editors. 100 % bedeutet ein dekodierter Bildpixel pro Gerätepixel. Einpassen erhält das Seitenverhältnis und vergrößert kleine Bilder nicht. Manueller Zoom ist auf 1/65536 bis 64 begrenzt; beim Einpassen sehr großer Bilder ist kleinere Skalierung möglich. Der Bildrand begrenzt das Verschieben.

Die Statuszeile zeigt Abmessungen, Zoom, aktuellen Frame/Seite und Anzahl. Über dem Bild erscheinen **X/Y und RGBA (0-255)** des dekodierten Pixels unter dem Zeiger. Außerhalb des Bilds wird kein Pixelwert erfunden. Die Werte stammen aus dem aktuellen dekodierten Frame, einschließlich Transparenz; Farbindizes oder komprimierte Quelldaten werden nicht als RGBA ausgegeben.

Bild auf / Bild ab wechseln GIF-Frames bzw. TIFF-Seiten; **Pos1 / Ende** wählen den ersten/letzten Eintrag. Kein automatisches Abspielen oder Umlauf. Unterschiedliche Seitengrößen sind erlaubt. Fehlerhafte Wechsel erhalten die zuvor dargestellten Pixel; Grenzen deaktivieren die betreffenden Menüpunkte. Tabs behalten ihre eigene Ansicht, neu geöffnete/wiederhergestellte Bilder beginnen mit Einpassen und Frame 1. Zoom, Verschieben und Framewahl verändern oder speichern keine Bilddaten.

Die Bildansicht hält eine eigene komprimierte Kopie und einen dekodierten Frame. Der Decoder hält die Quelldatei nicht offen. Grenzen: 256 MiB komprimierte Datei, 67.108.864 Pixel pro Frame und 10.000 Frames/Seiten. Dekoder-/Speicherfehler werden als Fehler behandelt. PNG/JPEG/BMP bleiben in der Regel Einzelbilder.

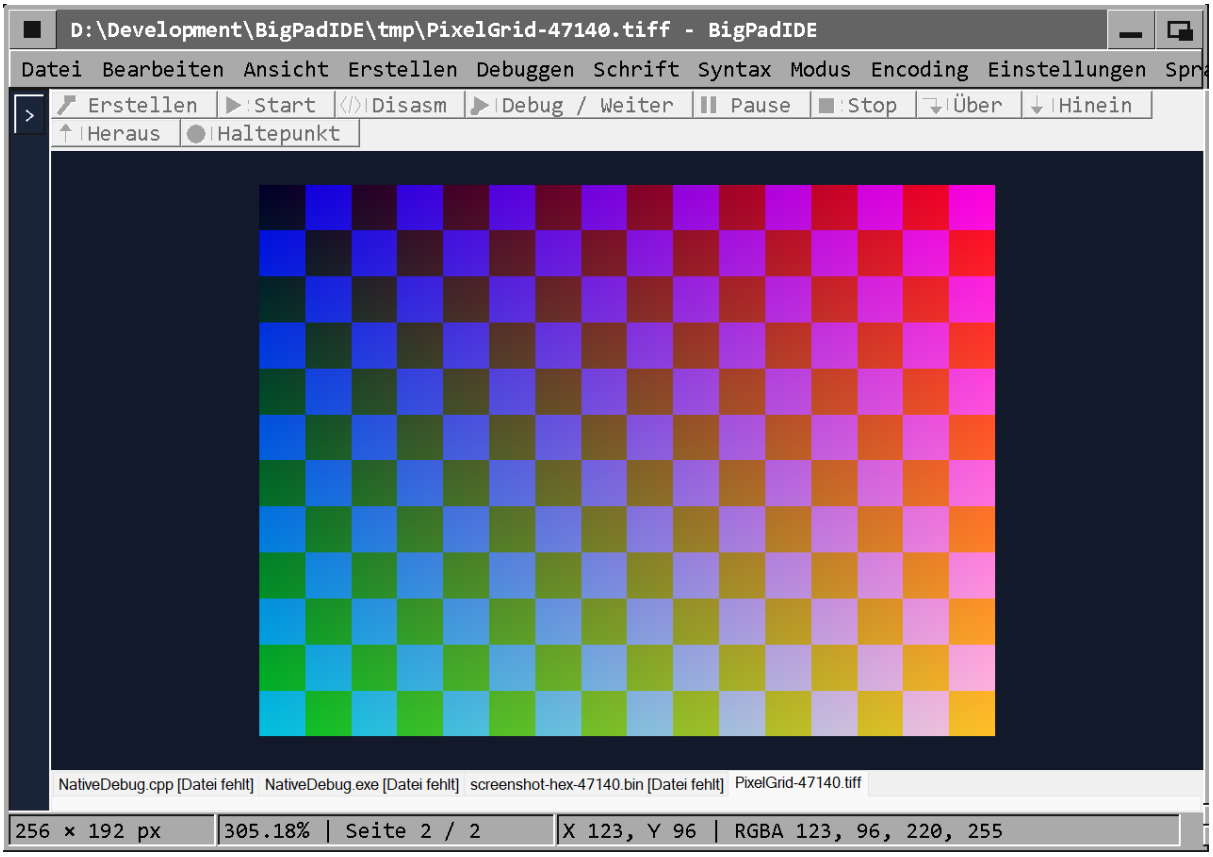


Abbildung 33. Zoom, Pixelinformationen und zweite TIFF-Seite

15.2 Ein Bild systematisch prüfen

1. Die Bilddatei öffnen und in der Statusleiste Abmessungen und Framezahl prüfen.
2. **0** für Einpassen oder **1** für pixelgenaue 100-Prozent-Darstellung drücken.
3. Mit dem Mausrad um einen interessanten Bildpunkt zoomen.
4. Mit gedrückter linker Maustaste einen vergrößerten Ausschnitt verschieben.
5. Den Zeiger auf einen Pixel setzen und X/Y sowie RGBA ablesen.
6. Bei GIF/TIFF mit **Bild auf/Bild ab** die weiteren Frames/Seiten untersuchen.

Eine transparente Stelle kann korrekte RGB-Kanäle und zugleich Alpha 0 besitzen. Die angezeigten RGBA-Werte sind dekodierte Pixelwerte des ausgewählten Frames; sie sind keine unmittelbare Sicht auf komprimierte Datei-Bytes.

16. Git, Diffs und Zusammenarbeit

Git unterscheidet **HEAD** als letzten Commit, den **Index** als vorbereiteten Stand für den nächsten Commit und die **Arbeitskopie** auf dem Datenträger. Ein ungespeicherter Editorpuffer ist ein weiterer Stand und erscheint noch nicht im Git-Diff. Vor der Prüfung eigener Änderungen deshalb bewusst speichern.

16.1 Der übliche Commit-Ablauf

1. Den Repository-Ordner öffnen und im Explorer **Git** wählen.
2. Eigene Dokumentänderungen speichern und die Diff-Anzeige prüfen.
3. Nur die gewünschten Dateien oder Abschnitte stagen.
4. Im Commit-Dialog die tatsächlich gestagten Dateien erneut prüfen.
5. Eine aussagekräftige Nachricht eingeben und den lokalen Commit erstellen.
6. Wenn ein passender Upstream und Zugang eingerichtet sind, den Commit pushen.

Staging ist noch kein Commit, und ein lokaler Commit ist noch kein Push. Ein Fetch lädt Referenzen, verändert jedoch nicht die eigene Arbeitsdatei.

16.2 Git-Änderungen und Diffs

Git installieren und `git.exe` im PATH verfügbar machen. Danach einen lokalen Repository-Ordner oder einen seiner Unterordner öffnen. Die Tabs **Lokal** und **Git** wechseln das linke Panel; der aktive Tab ist blau mit weißer Schrift.

Der Git-Baum umfasst das gesamte Repository und gruppiert offene Änderungen als **Gestagt**, **Nicht gestagt** und **Unversioniert**. Unveränderte committete Dateien werden ausgeblendet. Status: M (geändert), A (hinzugefügt), D (gelöscht), R (umbenannt), U (Konflikt) und ? (unversioniert). Eine Datei kann in beiden Änderungsgruppen stehen, wenn nach dem Staging weitere lokale Änderungen hinzukommen. Ein sauberes Repository zeigt einen Hinweis, dass keine uncommitteten Änderungen vorliegen.

Klick oder Enter auf eine gestagte oder nicht gestagte Datei öffnet einen schreibgeschützten Unified Diff als Tab direkt in der App. Gestagt vergleicht HEAD mit dem Index, nicht gestagt den Index mit der Arbeitskopie. Ergänzungen sind grün, entfernte Zeilen rot und Abschnittsköpfe blau. Gelöschte Dateien zeigen ihren entfernten Inhalt. Unversionierte Dateien öffnen sich regulär im Editor. Binäränderungen zeigen den entsprechenden Git-Hinweis.

Diff-Tabs lassen sich wechseln, schließen und im Split-Screen neben Quelldateien anzeigen. Ungespeicherte Änderungen bestehender Quelltext-Tabs bleiben erhalten. Der Vergleich verwendet den Git-Index und Dateien auf dem Datenträger; eigene Editoränderungen daher zuerst speichern. F5 im Explorer aktualisiert die Liste; erneuter Klick aktualisiert den vorhandenen Diff-Tab derselben Datei und Art. Aktualisieren im Diff beziehungsweise Strg+Umschalt+R liest seinen Git-Stand neu. Externe Änderungen werden weiterhin ausdrücklich neu eingelesen. Fehlendes Git, Ordner außerhalb eines Repositorys und Lesefehler werden im Panel oder als Fehler gemeldet.

16.3 Einzelne Diff-Abschnitte bearbeiten

Ein Git-Diff hat **Aktualisieren**, **Vorherige Änderung**, **Nächste Änderung**, **Abschnitt stagen** und **Abschnitt verwerfen…**. Bei einem gestagten Diff heißt die Index-Aktion **Staging zurücknehmen**; Verwerfen ist dort gesperrt. Ein Klick in einen Abschnitt wählt ihn aus, die Statuszeile zeigt seine Nummer. Vor/Zurück springt zu den Abschnittsköpfen und läuft am Ende um. Bei Fokus im Diff steuern **Alt+Pfeil hoch/runter** die Navigation; **Strg+Umschalt+R** aktualisiert den Diff. Staged und unstaged bleiben getrennte Tabs; erneutes Öffnen aktualisiert jeweils den passenden vorhandenen Tab, auch im Split-Screen.

Stagen überträgt nur den gewählten gespeicherten Abschnitt in den Index. Staging zurücknehmen ändert ebenfalls nur den Index und lässt die Arbeitsdatei unverändert. Verwerfen stellt nur den gewählten Arbeitsdatei-Abschnitt aus dem Index wieder her; es verlangt eine Bestätigung mit Datei und Abschnittsnummer. Alle anderen Abschnitte bleiben erhalten. Vor Verwerfen alle geänderten Tabs speichern oder schließen und Build, Debugging, Live-Modus sowie Wiederherstellung beenden. Ungespeicherte Quelltexte werden beim Stagen nicht automatisch gespeichert und bleiben samt Undo erhalten.

Vor jedem Schreiben werden HEAD/Branch, Index, Arbeitsdatei und vollständiger Diff mit der angezeigten Version verglichen. Veraltete Abschnitte werden abgelehnt und der aktuelle Stand neu gelesen. Der Patch wird geprüft und anschließend angewendet; Git behält seine normalen Pfad- und Kontextprüfungen. Währenddessen ist der Editor gesperrt und Schließen wartet auf den Abschluss. Jeder Git-Befehl hat fünf Sekunden Zeitlimit; nur instabile Leseabfragen werden höchstens dreimal versucht. Schreibaktionen werden nicht automatisch wiederholt. Nach der Aktion aktualisieren sich dieser Diff und die Git-Liste. Andere Diff-Tabs werden bei Bedarf ausdrücklich aktualisiert.

Originale Patch-Bytes bleiben erhalten, auch bei ANSI, CRLF und fehlendem letzten Zeilenumbruch; normale Git-Attribute wie `text eol=crlf` gelten weiterhin. Neue und gelöschte Textdateien bilden jeweils einen vollständigen Abschnitt. Binärdaten, Konflikt-Diffs, Modusänderungen, Symlinks/Submodule und besondere Metadaten bleiben für Abschnittsaktionen gesperrt. Verknüpfte oder nicht erreichbare Pfade sind ebenfalls nicht beschreibbar. Für solche Fälle stehen die vorhandenen Git-Aktionen für ganze Dateien bereit.

Die interaktive Ansicht ist auf 1 MiB Diff-Bytes, 20.000 Zeilen und 2.000 Abschnitte begrenzt. Überschreitungen werden gemeldet und nicht teilweise schreibbar gemacht; die Git-Prozessausgabe bleibt zusätzlich auf 32 MiB begrenzt. Ein fehlgeschlagenes Neuladen wird von einer bereits erfolgreich angewendeten Änderung unterschieden.

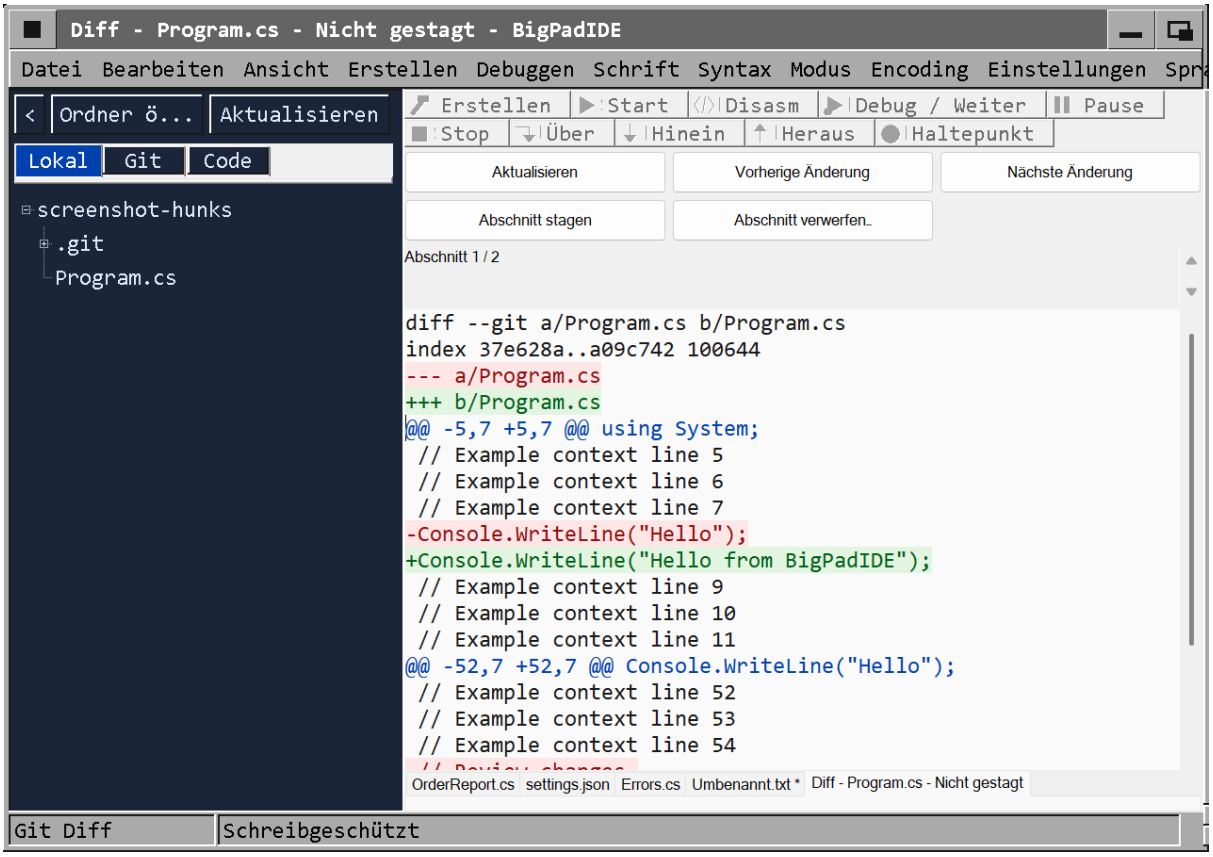


Abbildung 34. Abschnittsnavigation und Git-Aktionen

16.4 Git-Konflikte auflösen

Ein Klick auf eine ungelöste Datei im Git-Baum oder **Konflikt lösen...** im Kontextmenü öffnet einen Konflikt-Tab. **Basis (1)**, **Unsere (2)** und **Eingehende (3)** zeigen die Git-Index-Versionen mit Modus, Objekt-ID und Größe. Die Bezeichnungen folgen den Index-Stufen der aktuellen Git-Operation; vor Übernahme den tatsächlichen Inhalt prüfen. Fehlende Versionen sind ausdrücklich als nicht vorhanden gekennzeichnet, etwa die Basis eines Add/Add-Konflikts oder eine gelöschte Seite.

1. Mit **Ergebnis bearbeiten** die Arbeitsdatei im normalen Editor öffnen. Dort Änderungen zusammenführen und speichern; Encoding, Undo und die üblichen Dateischutzprüfungen bleiben verfügbar. Bei einer fehlenden Arbeitsdatei zuerst eine vorhandene Seite übernehmen oder die Löschung als Ergebnis verwenden.
2. Alternativ übernimmt **Unsere übernehmen** beziehungsweise **Eingehende übernehmen** nach Bestätigung deren Dateiinhalt. Eine fehlende Seite bedeutet eine ausdrücklich bestätigte Löschung. Die Übernahme ändert nur die Arbeitsdatei; der Konflikt bleibt zunächst ungelöst. Git-Arbeitsbaumfilter einschließlich EOL-Konvertierung gelten.
3. Zur Konfliktansicht zurückkehren und **Aktualisieren** oder Strg+Umschalt+R bei Konflikt-Fokus verwenden. **Als gelöst markieren** prüft und stagt nach Bestätigung genau das gespeicherte Ergebnis oder dessen Löschung. Andere Dateien werden nicht gestagt. Der Git-Merge beziehungsweise Rebase wird nicht automatisch abgeschlossen; anschließend den passenden Git-Commit-/Fortsetzungsbefehl ausführen.

Vor schreibenden Konfliktaktionen geänderte Tabs speichern oder schließen und Build, Debugging, Live-Modus sowie Wiederherstellungsdialog beenden. HEAD/Branch, Index-Stufen und Arbeitsdatei werden

gegen die angezeigte Version geprüft. Änderungen seit der Vorschau führen zu einer Ablehnung und einer neuen Anzeige; fehlgeschlagene Writes werden nicht automatisch wiederholt. Während einer Schreibaktion bleibt der Editor gesperrt. Git-Schreib-/Filterbefehle haben 120 Sekunden Zeitlimit, lokale Leseabfragen jeweils fünf Sekunden. Nach Aktionen werden Konfliktansicht und Git-Liste neu geladen.

Die Markerprüfung erkennt Zeilenanfänge mit <<<<<<<, |||||||, ======= und >>>>>>> in UTF-8/ANSI-Daten, einschließlich UTF-8-BOM. Sie ist auf 512 MiB und fünf Sekunden begrenzt. **Konfliktmarker bewusst beibehalten / Prüfung überspringen** erlaubt nach zusätzlichem Hinweis in der Bestätigung eine bewusste Ausnahme, zum Beispiel für wörtliche Marker im Text oder extern geprüfte große Binärergebnisse. Beim erneuten Öffnen einer Vorschau ist diese Option wieder ausgeschaltet.

Pro Version werden maximal 1 MiB und 20.000 Textzeilen dargestellt. Größere Versionen behalten Modus/Objekt-ID/Größe, ihr Ergebnis lässt sich im normalen Editor bearbeiten; die direkte Seitenübernahme bleibt dort gesperrt. Binäre Versionen zeigen einen Hinweis und können extern oder im Hex-Editor bearbeitet werden. Symlink-/Submodul- Konflikte und verknüpfte Pfade bleiben in der Konfliktansicht nur lesbar. Normale Batch-Staging-/Zurücksetzen-Aktionen lösen ungelöste Dateien nicht nebenbei auf.

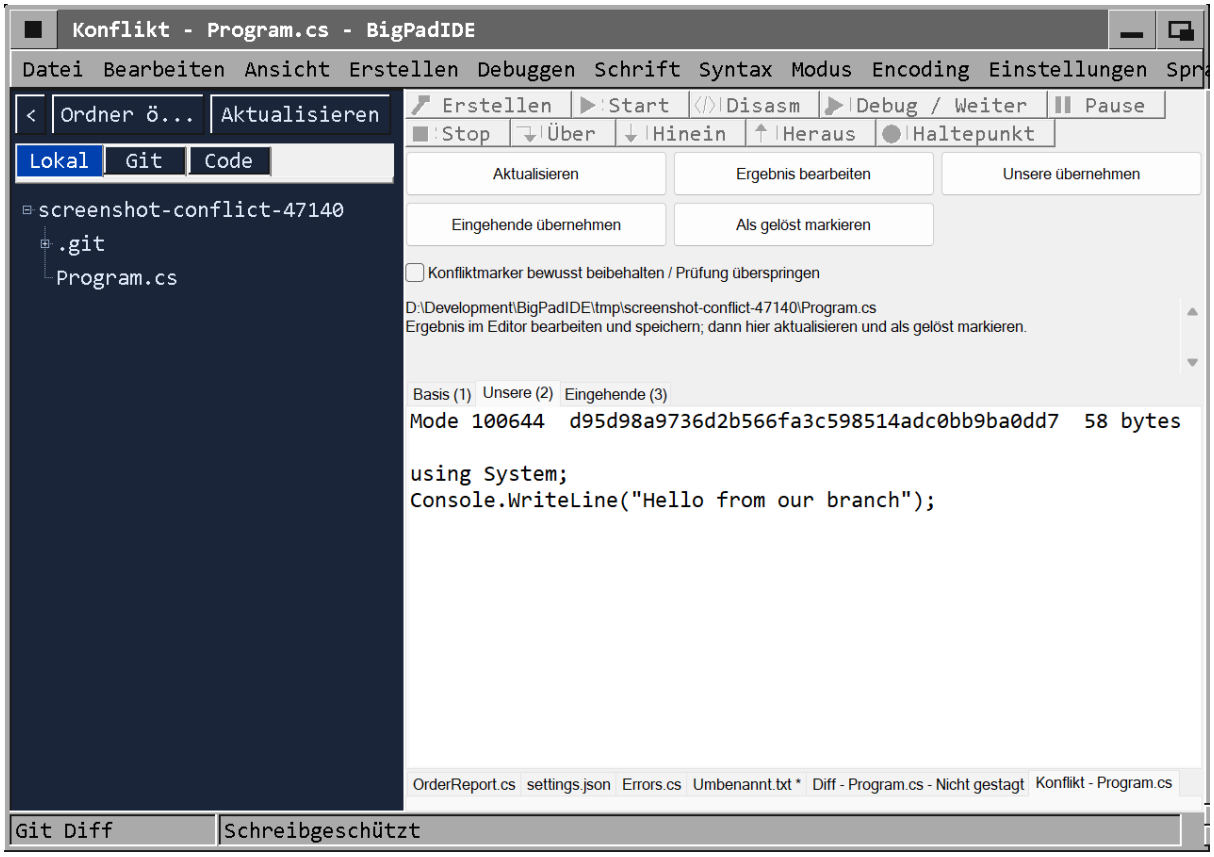


Abbildung 35. Basis, eigene und eingehende Konfliktversion

16.5 Fetch, Pull und Push

Im Git-Panel öffnet **Fetch / Pull / Push...** einen Dialog mit Branch, Commit, Upstream und Zielbranch. **Voraus/Zurück** zählt lokale und eingehende Commits anhand der vorhandenen Tracking-Referenz; **Fetch** aktualisiert diesen Stand. **Aktualisieren** liest nur die lokalen Referenzen. Ohne konfigurierten Upstream, bei detached HEAD oder vor dem ersten Commit bleiben Netzwerkaktionen gesperrt. Eine fehlende Tracking-Referenz kann mit Fetch angelegt werden.

- **Fetch** lädt den konfigurierten Upstream-Remote ohne Submodule.

- **Pull (FF)** erlaubt ausschließlich Fast-Forward. Alle geänderten Editor-Tabs müssen gespeichert oder geschlossen sein; Index und Arbeitskopie müssen auch hinsichtlich unversionierter Dateien sauber sein. Build, Live-Modus, Debugging und Wiederherstellungsdialog müssen beendet sein. Divergenz wird als Fehler angezeigt; es gibt keinen automatischen Merge, Rebase oder Stash.
- **Push** überträgt den angezeigten Commit auf den angezeigten Upstream-Zweig. Andere Branches und Tags sowie Mirror-/Force-Push werden nicht mitgenommen. Der benannte Remote verwendet seine konfigurierten Push-URLs; bei mehreren Push-URLs betrifft die Aktion alle diese Ziele. Den Upstream vorher im Terminal konfigurieren, wenn er noch fehlt.

Git läuft im Hintergrund; der Dialog zeigt verstrichene Zeit und aktuelle Git-Ausgabe. Während eines Befehls bleiben Editor und Dialogaktionen gesperrt; Schließen wartet auf Abschluss oder Zeitlimit. Jeder Netzwerkbefehl hat 120 Sekunden, lokale Abfragen jeweils fünf Sekunden. HTTP-Verbindungen ohne Datentransfer werden nach 30 Sekunden abgebrochen. Ausgaben sind auf 32 MiB begrenzt; das Dialogprotokoll zeigt höchstens die letzten 64 KiB der Abschlussausgabe.

Terminal-/Askpass-Abfragen und interaktive Git-Credential-Manager-Anmeldung sind deaktiviert. Zugangsdaten beziehungsweise SSH-Agent vorher im Terminal einrichten; eigene Credential-/SSH-Helfer bleiben unter dem Prozesszeitlimit. Git-Fehler wie Authentifizierung, unerreichbarer Remote oder abgelehnter Push bleiben im Dialog sichtbar. Nach einem Zeitlimit den Remote-Stand prüfen: Der Server kann Änderungen bereits übernommen haben. Nach jeder Aktion werden Referenzen neu gelesen und beim Schließen die Git-Ansicht aktualisiert. Saubere Quelldatei-Tabs verwenden weiterhin die normale Erkennung externer Änderungen; offene Diff-Tabs bleiben Momentaufnahmen.

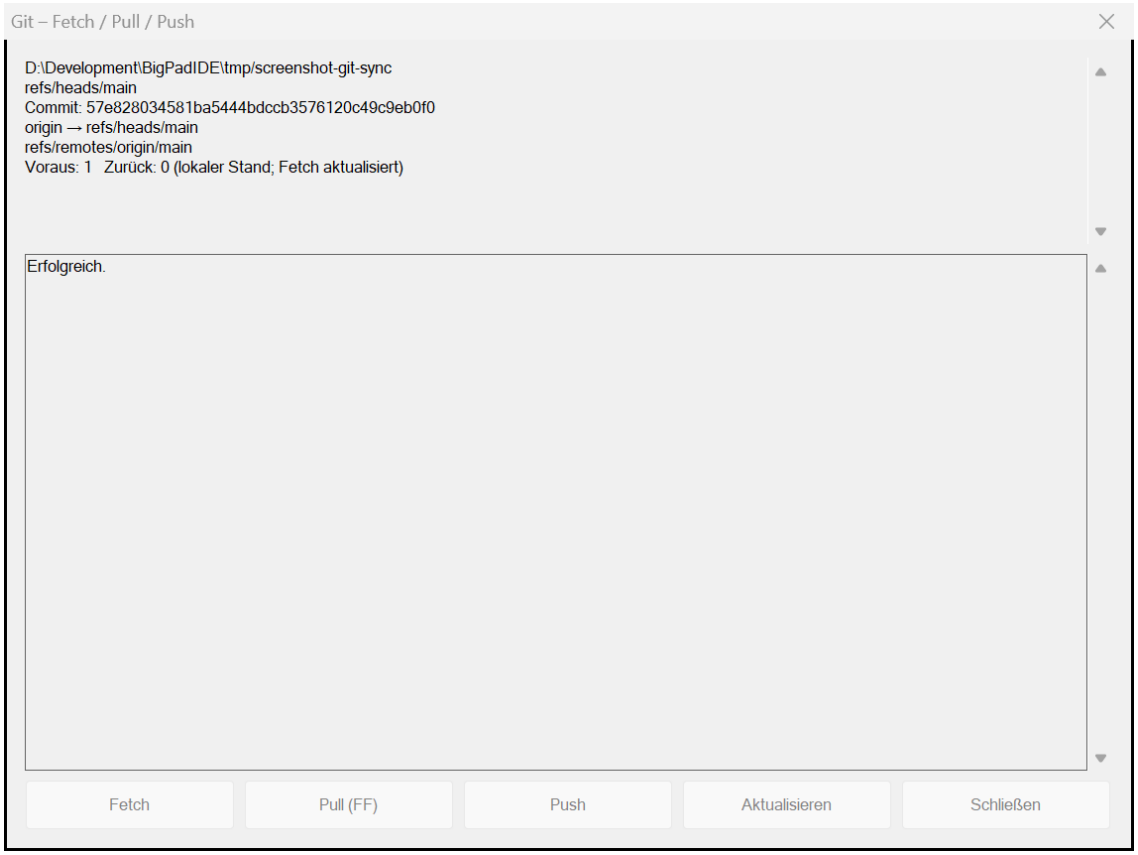


Abbildung 36. Git-Synchronisationsdialog

16.6 Branches verwalten

Branches… im Git-Panel zeigt lokale Branches und bereits geladene Remote-Tracking-Branches. Markierungen: `[*]` aktuell, `[L]` lokal, `[R]` remote, `[W]` in einem anderen Worktree ausgecheckt. Commit-Kürzel und vorhandener Upstream stehen in jeder Zeile. **Aktualisieren** liest ausschließlich lokale Referenzen; neue Remote-Branches zuerst über Fetch laden. Symbolische Remote-HEAD-Aliase werden ausgelassen. Die Liste ist auf 10.000 Branches begrenzt.

- **Erstellen ab Auswahl / HEAD** legt eine lokale Branch am angezeigten Commit an und wechselt nicht dorthin. Ohne Auswahl gilt HEAD, etwa bei detached HEAD. Ein Upstream wird dabei nicht automatisch eingerichtet.
- **Umbenennen** benennt die gewählte lokale Branch um; auch die aktuelle Branch und ihre Upstream-Konfiguration werden durch Git korrekt nachgeführt. Ein vorhandener Zielname wird nicht überschrieben.
- **Löschen…** verlangt eine Bestätigung mit genauem Branch-Namen und Commit. Die Branch muss in HEAD zusammengeführt sein und zusätzlich die normalen Sicherheitsprüfungen von `git branch -d` erfüllen, auch bezüglich ihres Upstreams. Aktuelle und in einem Worktree belegte Branches werden nicht gelöscht.
- **Wechseln** checkt die gewählte lokale Branch aus. Bei `[R]` heißt die Aktion **Remote lokal auschecken**: Im Namensfeld einen neuen lokalen Namen eingeben; Git legt eine Tracking-Branch mit genau diesem Remote-Upstream an und wechselt dorthin. Bestehende lokale Branches werden nicht ersetzt.

Neue Namen dürfen höchstens 1024 Zeichen haben und müssen Git-Refnamenregeln entsprechen. `HEAD`, `@` und führende Bindestriche sind reserviert. Ungültige oder bereits belegte Namen zeigen einen Fehler; die Eingabe bleibt zur Korrektur erhalten. Bereits vorhandene reservierte Sondernamen werden nicht als Git-Kurzschreibweise ausgeführt und müssen bei Bedarf über den vollständigen Ref im Terminal bearbeitet werden. Vor dem ersten Commit stehen keine Branch-Schreibaktionen zur Verfügung.

Die bisherige Branch-Auswahl am unteren Rand verwendet denselben geschützten Hintergrundvorgang. Vor jedem Checkout alle geänderten Tabs speichern oder schließen und Build, Debugging, Live-Modus sowie Wiederherstellungsdialog beenden. Index und Arbeitskopie müssen einschließlich unversionierter Dateien sauber sein. Ignorierte Dateien werden bei einem Konflikt nicht überschrieben. Keine Force- oder Merge-Checkouts, keine Submodul-Rekursion. In anderen Worktrees belegte Branches werden auch nicht umbenannt oder hier ausgecheckt. Andere ungespeicherte Tabs bleiben erhalten, wenn die Aktion abgelehnt wird.

Referenzen, aktuelle Branch und Worktree-Belegung werden vor jeder Aktion erneut geprüft. Schreibbefehle und Hooks laufen mit 120 Sekunden Zeitlimit im Hintergrund; der gesperrte Editor verhindert neue Eingaben während des Wechsels. Schließen wartet auf den Abschluss. Auch nach einem Fehler werden Referenzen neu gelesen: Ein Post-Checkout-Hook kann fehlschlagen, nachdem Git bereits gewechselt hat. Saubere Quelldatei-Tabs nutzen die normale Erkennung externer Dateiänderungen; Diff-Tabs bleiben Momentaufnahmen. Die Verwaltung überträgt oder löscht keine Branches auf einem Remote.

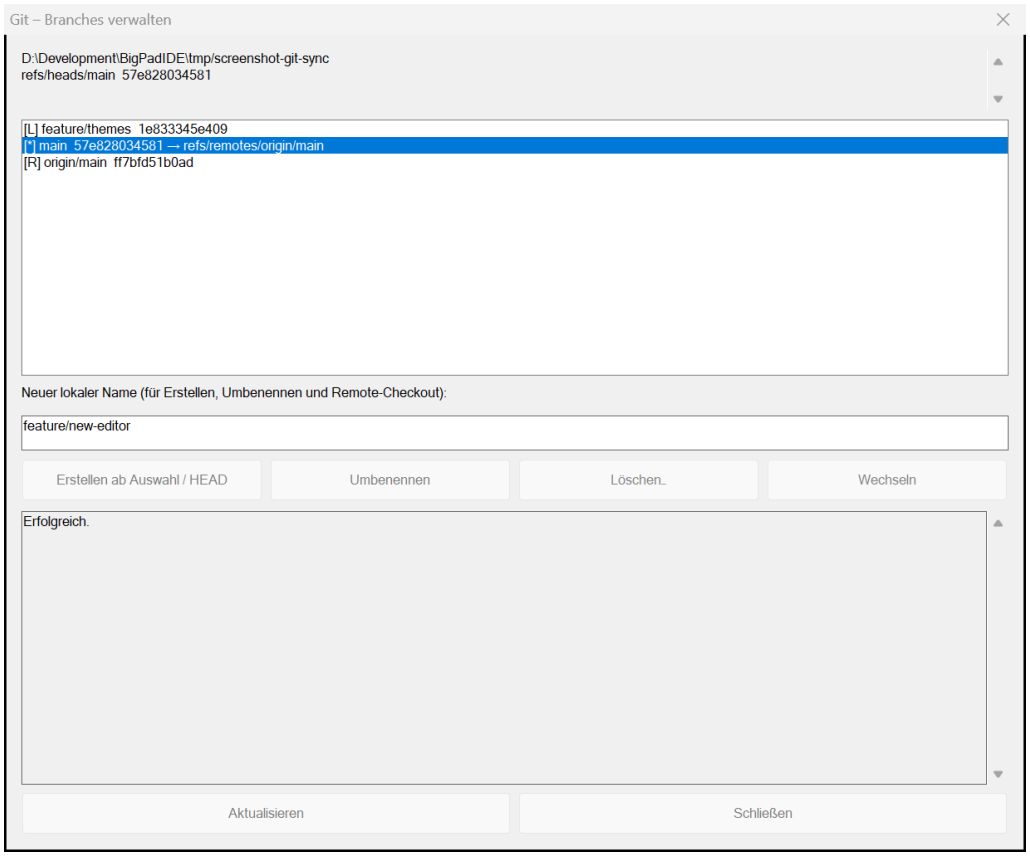


Abbildung 37. Lokale und Remote-Branches verwalten

16.7 Git automatisch aktuell halten

In der Git-Ansicht beobachtet BigPadIDE das Repository und seine Git-Metadaten. Speichern, externe Änderungen und Dateiumbenennungen aktualisieren die Liste gebündelt. Fensteraktivierung verarbeitet ausstehende Ereignisse; ohne Änderungen startet sie bei verfügbarer Dateiüberwachung keine Git-Abfrage. Auch gemeinsame Metadaten verknüpfter Git-Worktrees werden berücksichtigt.

Kurze Ereignisfolgen werden 400 ms gebündelt; bei fortlaufenden Änderungen wird spätestens nach zwei Sekunden eine Aktualisierung angefordert. Zwischen Abfragen liegt mindestens eine Sekunde. Laufende Leseabfragen, Git-Dialoge, Kontextmenüs und Schreibaktionen werden nicht unterbrochen. Änderungen währenddessen werden danach berücksichtigt. Auswahl, aufgeklappte Ordner und Markierungen bleiben soweit möglich erhalten. In der lokalen Ansicht laufen keine automatischen Git-Abfragen.

Kann Windows die Dateiüberwachung nicht starten, weist die Git-Ansicht darauf hin. F5 und Fensteraktivierung bleiben als Aktualisierung verfügbar; es gibt kein regelmäßiges Git-Polling im Leerlauf.

16.8 Commit und Dateistaging im Detail

Im Datei-Kontextmenü übernimmt **Stagen** eine nicht gestagte oder unversionierte Datei in den Index. Kästchen beziehungsweise Leertaste markieren mehrere Dateien. **Markierte stagen** und **Staging markierter Dateien zurücknehmen** arbeiten auf der jeweiligen Änderungsgruppe. Die Arbeitsdateien bleiben beim Zurücknehmen des Stagings erhalten.

Commit öffnet eine Übersicht von Repository, Branch und gestagten Dateien. Mit **Commit erstellen** oder **Strg+Enter** den lokalen Commit anlegen. Die Vorschau verwendet den Index; ein extern veränderter Index

verlangt eine neue Prüfung. Git-Hooks können laufen und Fehler melden. Bei einem Fehler bleiben Nachricht und Entwurf erhalten; vor einem weiteren Versuch die Liste aktualisieren.

Änderungen verwerfen setzt die Arbeitskopie auf den Indexstand zurück. **Alle Änderungen dieser Datei verwerfen** betrifft Index und Arbeitskopie zum HEAD-Stand. Bei unversionierten Dateien bedeutet **Datei löschen** eine echte Dateilöschung. Diese Aktionen deshalb erst nach Prüfung der Vorschau bestätigen.

16.9 Diff-Fenster und tatsächlicher Dateistand

Einstellungen → **Diff öffnen** gilt getrennt von der Assembly-Einstellung und auch für Hunk-/Konfliktansichten. Der Standard ist ein Tab. Eine offene Ansicht kann eine Momentaufnahme darstellen; **Aktualisieren** liest neu. Eine identische erneute Git-Anfrage verwendet eine vorhandene Ansicht wieder, statt beliebig viele Duplikate anzulegen. Während schreibender Git-Aktionen bleiben die betroffenen Fenster und die App gegen vorzeitiges Schließen geschützt.

17. Sitzungen und Wiederherstellung

Ein regulärer Neustart und ein Absturz sind unterschiedliche Situationen. Die Sitzungswiederherstellung merkt sich regulär geöffnete gespeicherte Dateien; die Absturzwiederherstellung bietet gesicherte ungespeicherte Dokumentinhalte an. Prüfen Sie nach einer Wiederherstellung den Inhalt, bevor Sie ihn speichern.

17.1 Sitzungswiederherstellung

Beim regulären Schließen speichert BigPadIDE Tab-Reihenfolge, aktiven Tab, Cursor und Scrollposition jedes Dokuments sowie Split-Achsen und Trennverhältnisse. Beim nächsten Start wird die Sitzung vor zusätzlichen Kommandozeilen-Dateien wiederhergestellt. Text-, Hex- und Disassembler-Ansichten behalten ihren Modus, die Disassembler-Architektur und die gewählte Textkodierung. Alle vier Stile der gemeinsamen Anwendung verwenden dieselbe Sitzung.

Fehlende oder nicht lesbare Dateien werden gesammelt gemeldet; die übrigen Tabs werden geöffnet und betroffene Split-Panes zusammengeführt. Positionen werden an die aktuelle Dateigröße und Zeichen-/Instruktionsgrenzen angepasst. Leere Tabs werden wiederhergestellt; verworfene ungespeicherte Inhalte und temporäre Git-Diffs werden nicht aus dieser Sitzungsdatei rekonstruiert. Die vorhandene Speicherrückfrage bleibt wirksam. Die Absturzwiederherstellung sichert zusätzlich ungespeicherte Inhalte, siehe unten.

Die Sitzung unterstützt bis zu 256 Tabs, 32 Split-Ebenen und 4 MiB Metadaten. Beschädigte oder unbekannte Formate werden mit Fehlermeldung abgewiesen.

17.2 Absturzwiederherstellung

BigPadIDE sichert geänderte und unbenannte Dokumente automatisch. Die Sicherungen werden im Hintergrund geschrieben; pro Dokument beginnen Aufträge frühestens im Abstand von zwei Sekunden. Wiederhergestellt wird die letzte vollständig abgeschlossene Sicherung. Große Dateien benötigen entsprechend mehr Zeit und Speicherplatz auf dem Laufwerk des Benutzerprofils.

Beim Start werden alte Sicherungen im Hintergrund geprüft. Laufende Instanzen bleiben geschützt. Der Dialog bietet **Wiederherstellen**, **Alle verwerfen** und **Behalten**. Schließen, Escape und die anfängliche Enter-Auswahl behalten die Daten. Verwerfen verlangt eine zusätzliche Bestätigung. Beschädigte oder nicht prüfbare Sicherungen werden gekennzeichnet; ohne gültige Daten bleibt Wiederherstellen deaktiviert.

Wiederhergestellte Inhalte öffnen als geänderte Tabs mit dem Hinweis **[Wiederhergestellt]**. Kodierung, BOM, rohe Bytes, Cursor und Scrollposition bleiben erhalten. Bereits bearbeitete Tabs werden nicht ersetzt. Nicht prüfbare Dateien bleiben auch beim Wiederherstellen der gültigen Dateien bestehen. Bei ausgeschöpfter Grenze von 256 Tabs bleiben weitere Sicherungen für einen späteren Start erhalten. Extern geänderte Originaldateien bleiben durch die vorhandene Speicherrückfrage geschützt.

Speichern oder ausdrückliches Verwerfen entfernt die jeweilige Sicherung. Alte wiederhergestellte Stände werden erst nach einer neuen dauerhaften Sicherung, Speichern oder ausdrücklichem Verwerfen entfernt. Leere verwaiste Instanzverzeichnisse werden beim nächsten Start bereinigt. Fehler erscheinen im Titel und in der Statusleiste; ein Klick auf die Statusleiste zeigt Details. Fehlgeschlagene Dokumentaufträge lassen sich dort sofort wiederholen und werden sonst nach 30 Sekunden erneut versucht.

Die gemeinsame Anwendung speichert unter `%LOCALAPPDATA%\BigPadIDEWorkbench\Recovery`; ältere Classic-Installationen können noch `%LOCALAPPDATA%\BigPadIDE\Recovery` verwenden. Die Daten bleiben lokal und werden bei regulärem Schließen nach den Speicherentscheidungen bereinigt.

17.3 Sitzungsgrenzen beim Öffnen

Bis zu 256 Tabs sind möglich, einschließlich temporärer Diff-Tabs. An der Grenze können Sie bereits geöffnete Dateien weiterhin aktivieren oder einen leeren Tab durch eine Datei ersetzen. Splits verwenden zuerst einen ausgeblendeten Tab. Neue Tabs, zu tiefe Splits und Pfade außerhalb des Sitzungsbudgets werden vor der Änderung abgewiesen. Das gilt auch für neue Pfade beim Speichern unter.

17.4 Fenster und Ordnerbreite merken

Beim regulären Beenden speichert BigPadIDE Fensterposition, normale Fenstergröße, Maximierungszustand und die zuletzt gezogene Breite des Ordnerpanels. Beim nächsten Start werden diese Einstellungen für die gemeinsame Anwendung wiederhergestellt. Ein minimiert geschlossenes Fenster öffnet wieder normal beziehungsweise maximiert. Nicht mehr sichtbare Fensterpositionen werden an verfügbare Monitore angepasst. Fehler beim Speichern verhindern das Schließen, damit Sie es erneut versuchen können.

17.5 Nach einer Wiederherstellung weiterarbeiten

1. Den angebotenen Sicherungsstand und seinen Status ansehen.
2. Gewünschte Dokumente wiederherstellen; unsichere Sicherungen zunächst behalten.
3. Die als wiederhergestellt markierten Tabs auf Vollständigkeit prüfen.
4. Bei extern veränderter Originaldatei beide Stände zunächst getrennt sichern.
5. Gewollte Inhalte regulär speichern und anschließend Git-Diffs prüfen.

Nicht prüfbare Daten werden nicht als scheinbar erfolgreiche Wiederherstellung behandelt. **Behalten** ermöglicht eine spätere Entscheidung. **Alle verwerfen** ist eine ausdrückliche Entscheidung und fordert eine zusätzliche Bestätigung.

18. Praxisworkshops

Die folgenden Übungen verbinden die Einzelwerkzeuge zu vollständigen Abläufen. Verwenden Sie einen eigenen beschreibbaren Übungsordner. Die erwarteten Ergebnisse helfen zu erkennen, ob Zielauswahl, Werkzeugkette und Ansicht zusammenpassen.

18.1 Workshop: C#-Fehler finden und korrigieren

1. Ein neues Dokument als `Types.cs` speichern.
2. `int value = "Text";` eingeben und die C#-Analyse abwarten.
3. Die Unterwellung und die Liste **Probleme** prüfen. Eine Zeichenkette lässt sich nicht direkt einer Ganzzahl zuweisen.
4. Die Zeile zu `int value = 42;` ändern und eine Console-Ausgabe ergänzen.
5. Speichern und **F6** starten. Die Fehlerausgabe muss verschwinden, das Programm soll den Wert 42 ausgeben.

Die Analyse unterstützt beim Bearbeiten; maßgeblich für den tatsächlichen Build ist die Ausgabe des ausgewählten Ziels. Falls ein anderer Projektfehler erscheint, zuerst die Zielauswahl prüfen.

18.2 Workshop: Das Startprojekt gezielt wechseln

Ausgangspunkt ist eine vorhandene `.sln` mit zwei ausführbaren C#-Projekten, die unterschiedlichen Text ausgeben, beispielsweise `Client` und `Tools`.

1. Die Solution öffnen und `Client` im Baum anklicken.
2. Fettschrift und Ziel `Client.csproj` im Ausgabepanel kontrollieren.
3. **F6** drücken und die erwartete Client-Ausgabe prüfen.
4. Einen Quelltab aus `Tools` öffnen. Das Startziel soll weiterhin Client sein.
5. Nun den Projektknoten `Tools` anklicken und den Lauf wiederholen.
6. Die Tool-Ausgabe bestätigt den bewussten Zielwechsel.

So unterscheiden Sie Dokumentnavigation von einer bewussten Startprojektwahl. Für semantische Solution-Refactorings zusätzlich einen geeigneten C#-Projektkontext wählen und die Vorschau vor der Übernahme prüfen.

18.3 Workshop: Debuggen statt zusätzliche Ausgaben einbauen

```
using System;

int total = 0;
for (int i = 1; i <= 5; i++)
{
    total += i;
}
Console.WriteLine(total);
```

1. Als eigene C#-Datei speichern und an `total += i;` einen Haltepunkt setzen.

2. Mit **F5** starten und die lokalen Werte `i` und `total` ansehen.
3. Mit **F10** genau eine Anweisung weitergehen und die Änderung verfolgen.
4. Vor dem nächsten Lauf mit **Umschalt+F9** die Bedingung `i == 3` setzen.
5. Neu starten und den Halt erst bei i = 3 prüfen.
6. Fortsetzen; die abschließende Ausgabe soll 15 sein.

Bei unerwarteter F5-Wirkung den Fokus vom Explorer zurück in den Editor setzen.

18.4 Workshop: Eine eigene Assembly untersuchen

1. Ein C#-Projekt mit einer kleinen Klasse bauen.
2. Die entstandene DLL in der IL-/C#-Ansicht öffnen.
3. Die eigene Klasse statt der gesamten Assembly auswählen.
4. Die rekonstruierte Methode in C# und anschließend in IL ansehen.
5. **Als Texttab öffnen**, einen neuen Dateinamen wählen und den Text speichern.
6. In **Einstellungen → IL / C# öffnen** die andere Platzierung wählen.
7. Die vorhandene Ansicht schließen, erneut öffnen und die neue Platzierung prüfen.

Erwartet wird nachvollziehbarer Code, nicht die originalgetreue Wiederherstellung von Kommentaren oder Formatierung. Eine Apphost-EXE enthält nicht automatisch den verwalteten Programmcode; dafür die entsprechende DLL verwenden.

18.5 Workshop: Native Pipeline und Maschinencode

1. `examples/native/c-fibonacci` in einen eigenen Ordner kopieren.
2. Diesen Ordner und die Projektdatei `bigpad.ini` öffnen.
3. Mit **Strg+Umschalt+B** bauen und das Protokoll auf Fehler prüfen.
4. Mit **F6** ausführen. Erwartet wird `C Fibonacci(10) = 55`.
5. Mit **Strg+F6** bauen und disassemblieren.
6. Die erzeugte `.text`-Datei und ihren Offsetraum von der ursprünglichen EXE unterscheiden. Für PE-Adressen die EXE separat analysieren.

Fehlt GCC oder objcopy, die Toolchain zuerst nach Kapitel 10 einrichten. Ein Compilerfehler ist kein Anlass, eine ältere EXE als neuen Build zu interpretieren.

18.6 Workshop: Nur eine Änderung in Git übernehmen

1. In einer versionierten Textdatei zwei voneinander entfernte Änderungen machen.
2. Speichern und im Git-Tab den nicht gestagten Diff öffnen.
3. Den gewünschten Abschnitt auswählen und nur diesen stagen.
4. Im gestagten Diff kontrollieren, dass ausschließlich die gewünschte Änderung enthalten ist. Die andere bleibt nicht gestagt.

5. Im Commit-Dialog den Index prüfen und einen lokalen Commit erstellen.
6. Anschließend den verbliebenen Arbeitsstand erneut ansehen.

Falls Git beide Änderungen in einem Abschnitt zusammenfasst, verwenden Sie für die Übung weiter auseinanderliegende Stellen. Diese Aktion ist abschnittsweise, nicht eine beliebige interaktive Einzelzeilen-Patchbearbeitung.

18.7 Workshop: Bytebereich und Zahlenwert

1. Eine kleine eigene Binärdatei oder Kopie öffnen und Hexmodus wählen.
2. Eine bekannte Bytefolge wie `01 00 00 00` suchen.
3. Die 32-Bit-Interpretation in Little und Big Endian vergleichen.
4. Den Bereich in eine neue Datei exportieren.
5. Die neue Dateigröße und die enthaltenen Bytes erneut prüfen.

Ein Interpretationwechsel ändert keine Quelldaten. Der Export berücksichtigt ungespeicherte Bytes und schreibt in ein neues Ziel; er speichert nicht automatisch das bearbeitete Original.

19. Fehlerhilfe und Diagnose

Beginnen Sie mit der konkreten Meldung und dem betroffenen Ziel. Build-Ausgabe, Programmausgabe, Probleme und Debugger-Protokoll beantworten unterschiedliche Fragen. Ein Fehler beim Start kann bereits aus dem Build stammen.

19.1 Installation, C# und Projekte

Symptom	Prüfen und beheben
C# startet nicht, SDK fehlt	Vollständige Installation und .NET-SDK-Verfügbarkeit prüfen; bei eigener Verteilung die Hilfsordner mitnehmen.
Nur eine DLL ist nicht startbar	Ein ausführbares Projekt als Startziel wählen; eine Klassenbibliothek besitzt keinen eigenen Einstiegspunkt.
Das falsche Projekt startet	Fettschrift im Baum und Ziel im Ausgabepanel kontrollieren; gewünschten Projektknoten anklicken.
Vervollständigung kennt einen Pakettyp nicht	Passenden C#-Projektkontext wählen, Projekt außerhalb der Analyse wiederherstellen/bauen und Paketverweise prüfen.
Mehrdeutiger Projekt-/Frameworkkontext	Eine eindeutige .csproj ausdrücklich wählen; bei verknüpften Quellen den gewünschten Kontext festlegen.
Einzeldatei kennt NuGet-Paket nicht	Im eigentlichen SDK-Projekt ausführen; temporäre Einzeldateiprojekte übernehmen keine beliebigen Projektpakete.
Ein Refactoring ist gesperrt	Build, Live-Modus, Debugsitzung und Recovery-Rückfrage beenden.
Refactoring-Vorschau ist veraltet	Neu berechnen und prüfen; die App übernimmt veraltete Quellen nicht teilweise.
ANSI-Änderung lässt sich nicht übernehmen	Darstellbarkeit prüfen und bei Bedarf bewusst nach UTF-8 wechseln, statt Zeichen still zu verlieren.

19.2 Ausgabe und Debugging

Symptom	Prüfen und beheben
Programm wartet ohne neue Ausgabe	Programmausgabe auf einen Prompt prüfen, eine Zeile senden oder bewusst EOF auslösen.
Eingabe-Umlaute kommen falsch an	Erwartete Eingabecodierung im Run-Profil mit der des Programms abstimmen.
Tastenbasierte Konsolenoberfläche funktioniert nicht	Das Panel ist zeilenweise Pipe-Eingabe, kein interaktiver Terminalemulator.
F5 aktualisiert nur den Baum	Den Editor fokussieren; F5 ist fokusabhängig.
Haltepunkt wird nicht erreicht	Tatsächliches Startziel, ausgeführten Codepfad, Symbole, Quelle und Debug-Build prüfen.
Quelle lässt sich beim Halt nicht bearbeiten	Debugsitzung beenden; Quelltexte sind währenddessen schreibgeschützt.
Native Sitzung startet nicht	Vollständige CodeLLDB-Installation sowie passendes Ziel und Debugdaten prüfen.
Angehangener Prozess läuft nach Stop weiter	Das ist das beabsichtigte Trennen einer Attach-Sitzung.

19.3 Dateien, Ansichten und große Daten

Symptom	Prüfen und beheben
Externe Dateiänderung ist noch nicht sichtbar	Sauberen Tab und kurze stabile Beobachtung abwarten; eigene Änderungen verhindern automatisches Neuladen.
Speichern meldet abweichenden Datenträgerstand	Externen Stand prüfen; eigene Bearbeitung gegebenenfalls erst unter neuem Namen sichern.
Neue Fensterplatzierung wirkt nicht	Bestehende Ansicht schließen und neu öffnen; IL/C# und Diff besitzen getrennte Einstellungen.
Assembly-Ansicht lehnt eine EXE ab	Native Apphost-EXE vermeiden; verwaltete DLL des Projekts wählen.
C#-Rekonstruktion schlägt fehl	Einzelnen Typ und IL-Ansicht versuchen; Referenzen, Verschleierung und Ausgabe-/Zeitgrenzen berücksichtigen.
Native Disassembly wirkt unsinnig	Architektur, Endianness und Befehlsgrenze prüfen; Datenbytes sind nicht zwangsläufig Code.
Eine virtuelle Adresse ist nicht anspringbar	Dateilose BSS, Mehrdeutigkeit oder fehlendes endgültiges Layout können eine Dateizuordnung verhindern.
Ein Bild lässt sich nicht öffnen	Format, Decoderfehler und Größen-/Pixelgrenzen prüfen; die gesamte dekodierte Framegröße zählt.
Suche zeigt nicht alle erwarteten Ergebnisse	Root, Dateifilter, Ausschlüsse, gespeicherten Stand und die Zusammenfassung übersprungener Dateien prüfen.
Weitere Tabs werden abgewiesen	Sitzungsgrenze und Pfadbudget prüfen, nicht mehr benötigte Ansichten schließen.

19.4 Git und Wiederherstellung

Symptom	Prüfen und beheben
Git-Ansicht ist leer	Sauberes Repository kann ohne Änderungen sein; andernfalls Git im PATH und Repository-Root prüfen.
Eigene Bearbeitung fehlt im Diff	Zuerst im Editor speichern; Git vergleicht Datenträger, Index und HEAD.
Pull ist gesperrt	Editor-Tabs sowie Index/Arbeitskopie einschließlich unversionierter Dateien bereinigen; laufende Aktionen beenden.
Pull meldet Divergenz	Integrierter Pull erlaubt nur Fast-Forward; gewünschten Merge-/Rebase-Ablauf bewusst außerhalb dieses Dialogs durchführen.
Push oder Fetch kann nicht anmelden	Zugangsdaten/SSH-Agent vorher im Terminal einrichten; interaktive Anmeldeabfragen sind deaktiviert.
Nach Zeitlimit ist der Serverstand unklar	Remote prüfen, bevor erneut übertragen wird; der Server kann eine Änderung bereits übernommen haben.
Hunk-/Konfliktaktion meldet veraltete Vorschau	Aktualisieren und erneut prüfen; HEAD, Index oder Arbeitsdatei haben sich geändert.
App schließt während Git-Schreiben nicht	Abschluss oder gemeldetes Zeitlimit abwarten; der Schreibvorgang ist gegen Abbruch durch Schließen geschützt.
Recovery-Daten sind nicht prüfbar	Daten zunächst behalten; die App bietet keine scheinbar erfolgreiche Teilwiederherstellung an.

19.5 Angaben für einen reproduzierbaren Fehlerbericht

Notieren Sie Oberflächenstil, Funktionsstand oder Paketversion, Datei-/Projektart, ausgewähltes Ziel, die letzten reproduzierbaren Schritte und die genaue Fehlermeldung. Bei Compilerproblemen auch Toolchain, Architektur und den relevanten Build-Auszug nennen. Für Anzeigeprobleme sind Theme, Schriftgröße und betroffene Ansicht hilfreich. Zugangsdaten und vertrauliche Quelltexte vor dem Teilen entfernen; häufig genügt ein kleines eigenes Beispiel.

20. Tastatur- und Menüreferenz

Tastenkürzel wirken im jeweiligen Fokus. Besonders F5, Strg+F und Bildnavigation besitzen in unterschiedlichen Ansichten unterschiedliche Bedeutungen. Bei einem unerwarteten Ergebnis zuerst den gewünschten Arbeitsbereich anklicken.

20.1 Dateien und Navigation

Taste	Funktion
Strg+N	Neues Dokument
Strg+O	Datei öffnen
Strg+Umschalt+O	Ordner öffnen
Strg+P	Schnell öffnen nach Dateiname/Pfad
Strg+S	Speichern
Strg+Umschalt+S	Speichern unter
Strg+Umschalt+R	Vom Datenträger neu laden; bei Diff-/Konfliktfokus deren Stand aktualisieren
Strg+W	Aktuellen Tab schließen
Strg+Tab / Strg+Umschalt+Tab	Nächsten/vorherigen Dokumenttab aktivieren
Strg+G	Zeile oder Byteoffset anspringen
Strg+Pos1 / Strg+Ende	Dokumentanfang/-ende
F1	Mitgelieferte Programmhilfe

20.2 Bearbeiten und Suchen

Taste	Funktion
Strg+A	Alles auswählen
Strg+X / Strg+C / Strg+V	Ausschneiden / Kopieren / Einfügen
Strg+Z	Rückgängig
Strg+Y / Strg+Umschalt+Z	Wiederholen
Strg+F	Im Text suchen; im Hexmodus Bytewerkzeuge
F3	Weitersuchen
Strg+H	Ersetzen
Strg+Umschalt+F	Projektweite Inhaltssuche
Strg+Leertaste	C#-Vervollständigung
Strg+Umschalt+Leertaste	C#-Parameterhilfe
Strg+I	C#-Symbolinformation
F12	C#-Quelldefinition
Umschalt+F12	Alle C#-Verweise auf das Symbol suchen
Strg+Klick	C#-Definition unter dem Mauszeiger öffnen

20.3 Explorer, Ansichten und Git

Taste	Funktion
Strg+B	Ordneransicht ein-/ausklappen
Strg+Umschalt+E	Explorer fokussieren
Strg+R	Ordneransicht aktualisieren
F5 im Explorer	Ordner-/Git-Ansicht aktualisieren
F2 im lokalen Baum	Ausgewählten Eintrag umbenennen
F2 in der Debugger-Variablenliste	Ausgewählten Laufzeitwert ändern
Leertaste im Git-Baum	Datei für Mehrfachaktion markieren
Enter im Baum	Datei beziehungsweise Diff öffnen
F7	Nächsten Split-Bereich aktivieren
Strg+J	Ausgabepanel ein-/ausblenden
Strg + Mausrad	Schrift im Bereich unter dem Zeiger ändern
Obere Ausgabepanelkante ziehen	Panelhöhe ändern, Schrift unverändert lassen
Strg+Enter im Commit-Dialog	Lokalen Commit erstellen

20.4 Build und Debug

Taste	Funktion
Strg+Umschalt+B	Gewähltes Ziel bauen
F6	Build & Start
Strg+F6	Natives Ziel bauen und disassemblieren
F5 im Editor	Debugstart / Fortsetzen
F9	Haltepunkt umschalten
Umschalt+F9	Haltepunktbedingung bearbeiten
F8	Debugsitzung pausieren
F10	Schritt über die nächste Anweisung
F11	Schritt hinein
Umschalt+F11	Schritt heraus
Umschalt+F5	Debugsitzung beenden beziehungsweise Attach trennen

20.5 Bilder und Disassembler

Taste	Funktion
0 / 1 in der Bildansicht	Einpassen / 100 Prozent
+ / - in der Bildansicht	Vergrößern / verkleinern
Pfeiltasten im Bild	Bildausschnitt um 32 Bildschirmpixel verschieben
Bild auf / Bild ab im Bild	Vorheriger/nächster GIF-Frame oder TIFF-Seite
Pos1 / Ende im Bild	Erster/letzter Frame beziehungsweise Seite
Enter im analysierten Disassembler	Direktem Sprung-/Callziel folgen
Rücktaste im analysierten Disassembler	Zur vorigen Position zurück

20.6 Die wichtigsten Menüs

Menü	Inhalt
Datei	Neu, Öffnen, Schnell öffnen, Ordner, Verlauf, Speichern, Beenden
Bearbeiten	Undo/Redo, Navigation, Suche, Ersetzen, Zeilenenden, Zwischenablage
Ansicht	Themes, Zeilenumbruch/-nummern, Explorer, Splits, Ausgabe, Bilder
Erstellen	Build/Run, Profile, Live-C#, C#-Symbol- und Projektwerkzeuge
Debuggen	Start/Attach, Schritte, Haltepunkte und Bedingungen
Schrift	Schriftart, Editorgröße und getrennte Explorergröße
Syntax	Automatische oder ausdrückliche Sprachdarstellung
Modus	Text, Hex, Bytewerkzeuge, native und verwaltete Disassembler
Encoding	UTF-8, Windows-ANSI, UTF-8-BOM
Einstellungen	Style, Schriften/Farben, JSON, Speichermodus, Toolsets, Ansichtsplatzierung, Speichern und Defaults
Sprache / Hilfe	Deutsch/English, Offline-Hilfe, Projekt und About

21. Grenzen und gespeicherte Daten

Die folgenden Werte sind bewusst begrenzte Arbeitsbudgets des beschriebenen Funktionsstands. Eine Grenze ist keine allgemeine Zusage über alle Dateiformate oder verfügbare Systemressourcen. Die jeweilige Funktion meldet Fehler oder übersprungene Eingaben ausdrücklich. MiB bezeichnet 1.048.576 Byte.

21.1 Wichtige Arbeitsgrenzen

Bereich	Grenze beziehungsweise Verhalten
Dokumente/Ansichten	Bis 256 Tabs; temporäre und externe Ansichten zählen zur Kapazität
Schnell öffnen	50.000 Dateien, 200.000 besuchte Einträge, 64 Ebenen, 200 angezeigte Treffer
Projektweite Suche	1000 Treffer, 10.000 Dateien, 64 MiB je Datei, 512 MiB summiert
Regexsuche	Zeilenweise, keine Lookarounds/Rückreferenzen im Nonbacktracking-Modus
Live-C#	Eigenständige Quellen bis 1 MiB; kein automatischer Import von Projektpaketen
C#-Editor-Snapshots	Bis 64 Snapshots und 8 MiB insgesamt; weitere Projektgrenzen in Kapitel 7
Assembly-Eingabe	128 MiB und 10.000 sichtbare Typen, 64 Ebenen
Assembly-Codeausgabe	4 MiB UTF-8 beziehungsweise 2.097.152 UTF-16-Einheiten
Assembly-Worker	30 Sekunden und 1 GiB Speicherbudget
Komprimierte Bilddatei	256 MiB
Dekodierter Bildframe	67.108.864 Pixel; bis 10.000 Frames/Seiten
PE/ELF-Metadaten	4096 Sektionen, 50.000 Symbolrecords, 16 MiB Gesamtlesen
Run-/Build-Profil	256 KiB Datei, 128 Argumente je Profil, 64 Umgebungsänderungen
Standard-Eingabe	16.384 UTF-16-Zeichen im Feld, höchstens 1 MiB wartende Bytes
Git-Netzwerk-/Schreibbefehl	Im Regelfall 120 Sekunden; Fehler/Timeout im Dialog prüfen

21.2 Wo Zustände liegen

Zustand	Speicherort beziehungsweise Geltung
Eigene Dokumente	Von Ihnen gewählte Pfade
Native Build-Ausgaben	`.bigpad-build` im jeweiligen Projektordner
Zielbezogene Startprofile	`<Ziel>.launch.ini` neben dem Zielpfad
UI-Einstellungen, Verlauf, Sitzungen	Lokaler Benutzerzustand der gemeinsamen Anwendung
JSON-Konfiguration	Ausdrücklich gewählter Exportpfad; keine Lizenz-/Trial-Daten
Recovery der gemeinsamen Anwendung	`%LOCALAPPDATA%\BigPadIDEWorkbench\Recovery`
Ältere Classic-Recovery	`%LOCALAPPDATA%\BigPadIDE\Recovery`
Aktives C#-Projekt	Aktuelle Arbeitsumgebung; keine neue persistierte Solution-Eigenschaft
Projektkontext für C#-Analyse	Aktuelle Editorinstanz
Separate IL-/Diff-Fenster	Gehören zur App; keine eigenständigen Instanzen

21.3 Datenstand bewusst unterscheiden

Funktion	Welche Daten werden gelesen?
Editor und Hex-Export	Logischer Dokumentinhalt einschließlich ungespeicherter Änderungen
Projektweite Dateisuche	Gespeicherte Dateien auf dem Datenträger
C#-Analyse und Refactoring	Geladener Projektkontext plus unterstützte Editor-Snapshots
Git-Diff	HEAD, Index und Arbeitsdateien; keine ungespeicherten Editorpuffer
Assembly-IL/C#	Assembly und Referenzen vom Datenträger
Bildansicht	Dekodierter ausgewählter Frame/Seite
Wiederherstellung	Letzter vollständig geschriebener Sicherungsstand

Diese Unterscheidung erklärt viele scheinbare Widersprüche: Ein Symbol kann im Editor bereits geändert sein, während eine Suche über Dateien oder ein Git-Diff noch den alten gespeicherten Stand zeigt. Speichern, neu lesen oder neu berechnen sind jeweils bewusste Schritte für das passende Werkzeug.

22. Glossar, Quellen und weiterführende Hilfe

22.1 Begriffe

Begriff	Bedeutung in diesem Handbuch
Assembly	Verwaltete .NET-Datei mit Metadaten und typischerweise IL-Code
Apphost	Native Start-EXE einer SDK-Anwendung; verwalteter Code liegt meist in einer DLL
BOM	Byte Order Mark; bei UTF-8 optionale Signatur am Dateianfang
Build	Übersetzen und gegebenenfalls Linken eines ausgewählten Ziels
Decompiler	Rekonstruiert höherstufigen Code aus kompiliertem Code
Disassembler	Dekodiert Maschinenbefehle beziehungsweise zeigt IL statt ursprünglicher Quellen
DWARF / PDB	Formate für Debug-/Symbolinformationen
EOF	Ende der Standardeingabe
HEAD	Aktueller Git-Commit beziehungsweise die entsprechende Referenz
Hunk / Abschnitt	Zusammengehöriger Änderungsblock eines Diffs
IL	Common Intermediate Language einer verwalteten .NET-Assembly
Index / Staging	Vorbereiteter Git-Dateistand für den nächsten Commit
Little/Big Endian	Reihenfolge nieder- und höherwertiger Bytes eines mehrbyteigen Werts
Offset	Byteposition relativ zu einer Datei oder einem angegebenen Datenbereich
PE / ELF	Containerformate für Programme, Bibliotheken und weitere Binärdateien
Projektkontext	Für C#-Analyse geladene Quellen, Symbole und Verweise
Recovery	Wiederherstellungsdaten ungespeicherter Dokumente
Rohbinary	Bytes ohne den vollständigen ausführbaren PE-/ELF-Container
Sidecar	Zusätzliche Konfigurationsdatei neben einer Quelldatei oder einem Projekt
Snapshot	Zu einem Zeitpunkt festgehaltener Dokument- oder Dateistand
Solution	Zusammenfassung mehrerer Projekte; üblicherweise `.sln`

Begriff	Bedeutung in diesem Handbuch
Upstream	Zugeordneter entfernter Tracking-Branch für Git-Synchronisation
VA	Virtuelle Adresse aus dem Layout einer ausführbaren Datei

22.2 Herkunft und Gültigkeit

Dieses Handbuch wurde aus der deutschsprachigen Projektdokumentation und dem aktuellen Anwendungsverhalten zusammengestellt. Menüpfade und Tastenkürzel wurden mit der Implementierung abgeglichen. Es beschreibt verfügbare Funktionen des angegebenen Quellstands und keine Zusage, dass ein älteres Setup bereits denselben Stand enthält.

Die ursprünglichen Beschreibungen finden Sie in der Benutzeranleitung, in der Projektübersicht und in den nativen Demoprojekten. Das Architekturdokument erklärt die technischen Hintergründe. Die Messdokumentation beschreibt reproduzierbare Versuche für große Dateien und die Grenzen ihrer Übertragbarkeit.

Alle im PDF enthaltenen Abbildungen sind Aufnahmen aus dem Repository unter `docs/help/screenshots`. Sie sind Beispiele, keine ausgedachten Bedienoberflächen. Die PDF-Lesezeichen öffnen Kapitel und Abschnitte direkt; die Textsuche des PDF-Readers eignet sich zum Nachschlagen von Menübegriffen und Fehlersymptomen.

22.3 Hilfe und Projekt

F1 öffnet die mitgelieferte Offline-Programmhilfe. **Hilfe** → **GitHub-Projekt** führt zum BigPadIDE-Repository. Der About-Dialog nennt Entwickler und Lizenzinformationen. Lizenzen gebündelter Drittkomponenten gehören zur jeweiligen Distribution.

22.4 Impressum und Kontakt

Entwickler von BigPadIDE: Rudolf Stepan

E-Mail: rstepan123@gmail.com

23. Testphase, Kauf und Product-Keys

23.1 Testphase und Kaufberechtigung

Ohne gültige Vollversion beginnt beim ersten Start eine 14-tägige Testphase. Beginn und letzter Zeitstand werden je Windows-Benutzer geschützt gespeichert; Neustarts, Updates und Neuinstallation beginnen keine neue Frist. Eine beschädigte Lizenzdatei oder deutlich zurückgestellte Uhr wird nicht als neuer Erststart behandelt.

Nach dem Ende der Testphase ist für weitere Arbeit eine gültige Berechtigung nötig. Bei Ablauf während einer Sitzung bleiben geöffnete Dokumente erhalten; Speichern, Speichern unter und Beenden bleiben möglich. Vor dem Schließen eigene Arbeit sichern. Die App zeigt den jeweiligen Status und eine Möglichkeit zur Lizenzierung an.

Eine regulär über den Microsoft Store erworbene und von der App erkannte Vollversion benötigt keinen zusätzlichen Product-Key. Das Store-Konto und die Paketinstallation müssen zur gekauften App passen. Eine herauskopierte EXE erbt keine Store-Berechtigung. Eine noch nicht veröffentlichte Paketvorschau ist keine Kaufzusage; maßgeblich ist das tatsächlich verfügbare Angebot.

23.2 Product-Key aktivieren

Für den separaten EXE-Vertrieb sind signierte 30-, 90-, 365-Tage- und unbefristete Keys unterstützt. Die tatsächlich angebotene Variante steht in Ihrem Kaufbeleg.

1. **Hilfe** → **Product-Key / Lizenz** öffnen.
2. Aktivieren beziehungsweise Wechseln bestätigen und den vollständigen gelieferten `BP1.…`-Schlüssel einfügen.
3. Den angezeigten Kundennamen und das Ablaufdatum in UTC kontrollieren.

Die Laufzeit eines befristeten Keys beginnt bei seiner Ausstellung, nicht beim Installieren oder Einfügen. Eine vorhandene Aktivierung wird beim Neustart weiterverwendet. Bereits verwendete Lizenz-IDs können nicht erneut manuell aktiviert werden; eine Verlängerung benötigt einen neu ausgestellten Key.

23.3 Umzug, Sicherung und Fehlerhilfe

Aktivierung und Trial sind an den lokalen Windows-Benutzerzustand gebunden. Ein JSON-Konfigurationsexport überträgt weder Kaufberechtigung noch Product-Key-Aktivierung oder Testphase. Auf einem anderen Windows-Konto die für Ihren Vertriebsweg vorgesehene Aktivierung verwenden und bei Unklarheiten den Herausgeber kontaktieren.

Bei einer Fehlermeldung Systemdatum, Programmausgabe und den angezeigten Lizenzstatus prüfen. Keine Aktivierungsdaten veröffentlichen und keinen vollständigen Product-Key in Screenshots oder Fehlerberichte aufnehmen. Für eine Rückfrage reichen zunächst Programmversion, Vertriebsweg, Windows-Version und der Wortlaut der Meldung.

Kontakt: rstepan123@gmail.com.

24. Die eigene Arbeitsumgebung einrichten

24.1 Vier Oberflächen und horizontale Menüs

Unter **Einstellungen** → **Oberfläche / Style** den bevorzugten Stil wählen. Modern - Hell und Modern - Dunkel verwenden zurückhaltende Flächen, Workbench eine Retro-Darstellung und Klassisch die Windows-Bedienelemente. Der Wechsel gilt sofort und lässt offene Dokumente einschließlich ungespeicherter Änderungen erhalten.

Ein Hauptmenü einmal öffnen. Danach den Zeiger horizontal über die anderen Menüüberschriften bewegen, beispielsweise von **Encoding** nach **Einstellungen**: Das nächste Menü öffnet ohne weiteren Klick. Escape schließt das Menü.

24.2 Schriften und Farben anpassen

1. **Einstellungen** → **Schriften und Farben** öffnen.
2. Den gewünschten Bereich wählen und **Schrift wählen** anklicken. Schriftart, Stil und Größe getrennt für Editor, Explorer, Menüs, Toolbar, Status, Tabs, Panels, Ausgabetext, Dialoge, Assembly, Hex, Diff und Suche festlegen.
3. Für Farben den Bereich und die angezeigte Ausgangsfarbe auswählen. **Farbe wählen** ändert diesen Eintrag, die Vorschau zeigt den gewählten Farbwert.
4. **Übernehmen** zeigt Änderungen sofort; **OK** übernimmt und schließt.
5. **Bereich zurücksetzen**, **Farbe zurücksetzen** oder **Alles zurücksetzen** stellen gezielt die Standarddarstellung wieder her.

Farben sind nach Bereich und ursprünglichem RGB-Wert zusammengefasst. Wird dieselbe Ausgangsfarbe innerhalb eines Bereichs mehrfach verwendet, betrifft die Anpassung diese Stellen gemeinsam. Windows-eigene Titelleisten und Standard-Dateidialoge folgen weiterhin den Betriebssystemeinstellungen. Neu geöffnete Dialoge verwenden die eingestellte Dialogschrift; einige bereits offene Dialoge erst nach erneutem Öffnen.

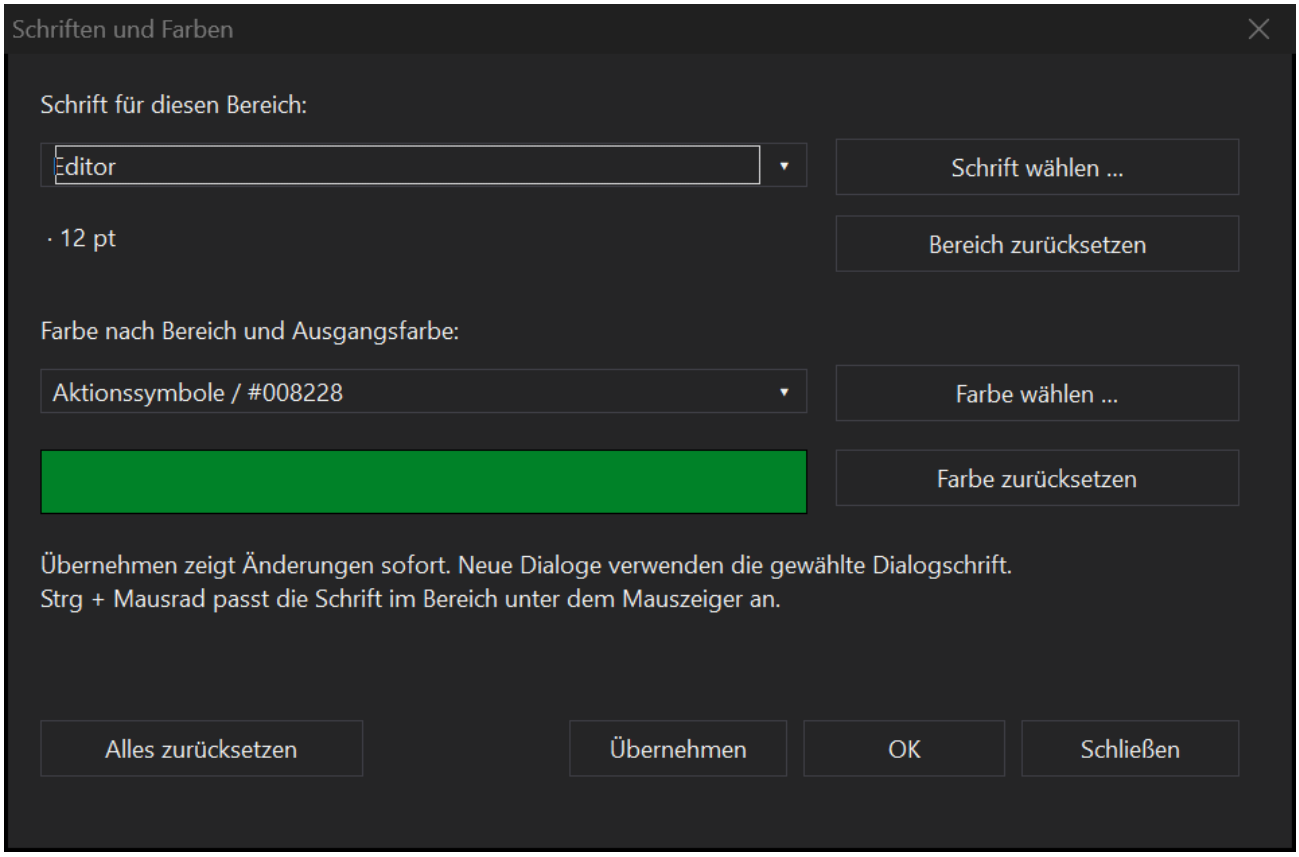


Abbildung 38. Dialog für getrennte Schriften und Anwendungsfarben

24.3 Mit Strg und Mausrad bequem lesen

Den Zeiger über den gewünschten Bereich bewegen, Strg halten und das Mausrad drehen. Jede volle Rastung ändert die Größe um einen Punkt; fein auflösende Räder sammeln Teilbewegungen. Größen von 6 bis 72 pt sind möglich. Im Explorer verändert das Rad die Baumschrift, in der Ausgabe deren Textschrift; Editor und Panel bleiben getrennt. Die neue Größe wird gespeichert. Ein gewöhnliches Mausrad ohne Strg scrollt weiterhin.

Für mehr Platz im Ausgabepanel dessen obere Kante ziehen. Das vergrößert das Panel selbst und lässt die Schriftgröße unverändert. Debugger-Spalten teilen den Platz nach Inhalt und Schrift auf, bevorzugen Pfade und Werte und zeigen für gekürzte Zellen Tooltips. Manuell gezogene Spaltenbreiten bleiben bis zur nächsten Sitzung. Die gelbe Zeile markiert die aktuelle Ausführungsposition; eine alte blaue Textauswahl wird beim Debuggersprung aufgehoben.

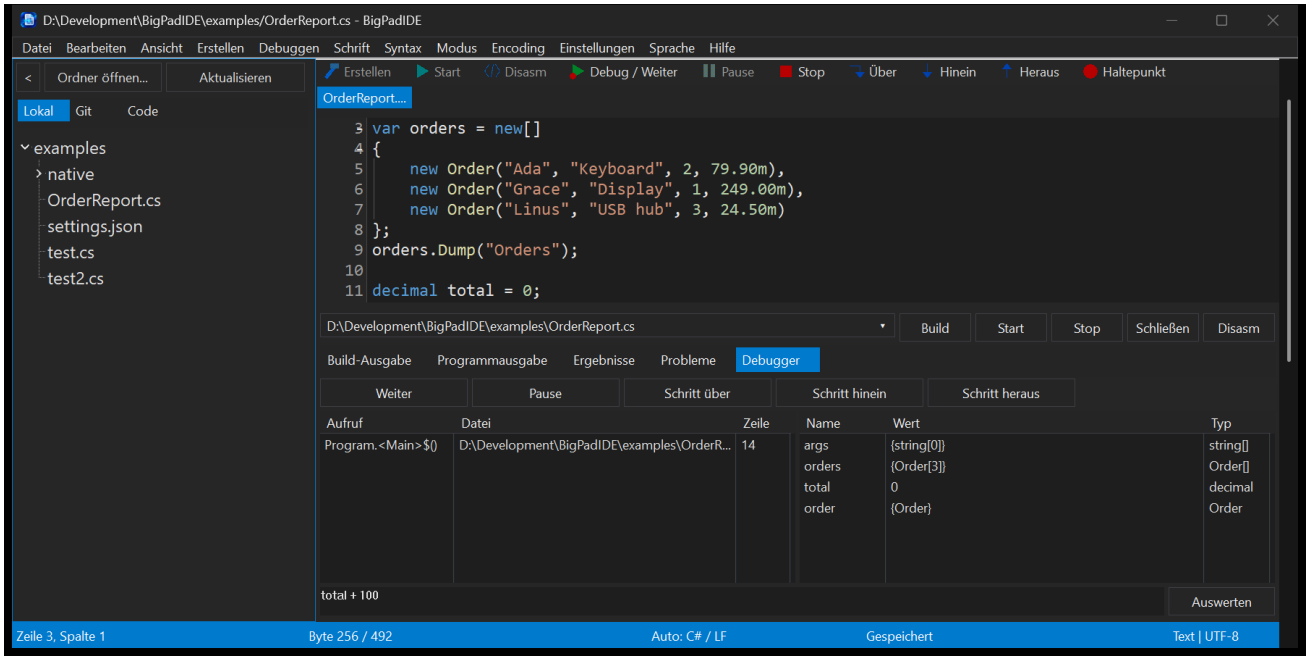


Abbildung 39. Mit der Maus vergrößertes Debuggerpanel mit echten Sitzungsdaten

24.4 Die gesamte Konfiguration als JSON austauschen

Einstellungen → Konfiguration als JSON exportieren schreibt eine UTF-8-Datei. Sie enthält die gespeicherten Programmeinstellungen einschließlich Schriften, Farben, Toolsets, Werkzeugpfaden und Workspace-/Fensterdaten. Offene, ungespeicherte Dokumentinhalte sind kein Bestandteil dieses Exports; dafür gibt es Speichern und Recovery. Projektbezogene `.launch.ini`-Dateien liegen neben den Projekten und werden als Projektdateien mitgenommen, nicht in die Registry-Konfiguration eingebettet.

Einstellungen → Konfiguration aus JSON importieren prüft das Format vor der Übernahme. Ein Import ersetzt den exportierten Einstellungsbestand; vor einem fremden Import die eigene Konfiguration sichern. Ungültige Daten werden abgewiesen, Schreibfehler lösen eine Rücksicherung aus. Lizenz-, Product-Key- und Trial-Daten sind bewusst ausgenommen. Exportiert werden maximal 4 MiB; die Datei ist versioniert.

Auf einem anderen Rechner können SDK-, Compiler-, Projekt- oder Debuggerpfade fehlen. Nach dem Import **Toolsets und Werkzeugpfade** prüfen. Die JSON-Datei kann eigene Ordnernamen und Werkzeugpfade enthalten; vor öffentlicher Weitergabe kontrollieren.

24.5 Piece Table oder vollständig im Arbeitsspeicher

Unter **Einstellungen → Dokument-Speichermodus** stehen **Piece Table (dateibasiert)** und **Vollständig im RAM (Dateien und Undo-Verlauf)** zur Wahl.

Wahl	Geeignet, wenn	Zu beachten
Piece Table	Große Dateien mit begrenztem RAM bearbeitet werden	Unveränderte Bytes bleiben zunächst dateibasiert; die Originalsicherung benötigt temporären Plattenplatz
Vollständig im RAM	Jede geöffnete Datei vollständig im Arbeitsspeicher liegen soll	Dateien und Undo-Versionen benötigen entsprechend Speicher; nach dem Laden wird die Quelldatei nicht mehr als Datenquelle benötigt

Beim Wechsel bereitet die App alle geöffneten Dokumente einschließlich Undo/Redo vor und übernimmt sie gemeinsam. Änderungen, Cursor, Auswahl, Breakpoints und Scrollposition bleiben erhalten, auch in inaktiven Tabs. Bei fehlendem Speicher oder einer zwischenzeitlich geänderten Quelldatei wird der Wechsel abgelehnt. Es gibt keinen stillen Wechsel auf das andere Backend.

Mausrad, Scrollleisten, Zeilenumbruch, Hex und Disassembly verwenden in beiden Modi dieselben Dokumentpositionen. Der Backendwechsel springt nicht an den Dateianfang. RAM-Snapshots teilen unveränderte Puffer; Änderungen kopieren geteilte Versionen. Beim Zurückschalten bleibt der vorhandene Undo-Verlauf erhalten und kann weiterhin RAM belegen. Für eine frisch dateibasierte Datei nach dem Speichern neu öffnen.

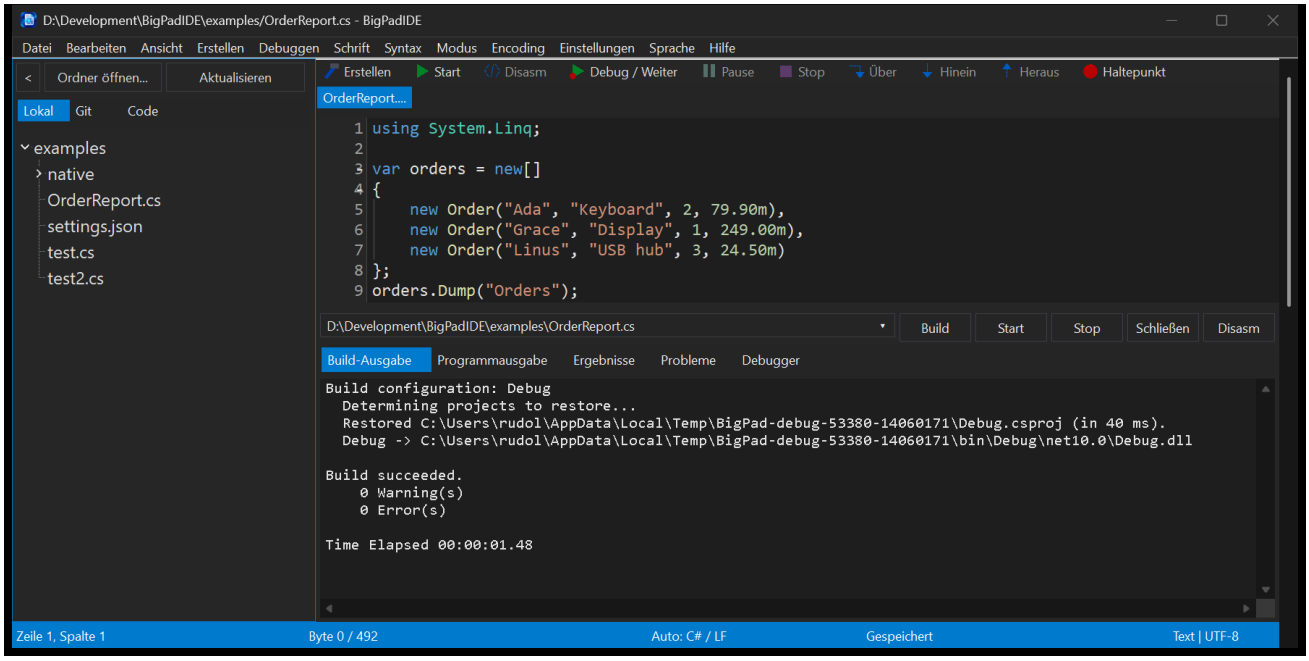


Abbildung 40. Arbeitsumgebung nach dem Wechsel auf vollständige RAM-Puffer

24.6 Ein persönliches Profil sicher aufbauen

Zuerst Stil und gut lesbare Schrift einstellen, dann Panelhöhen und Farben anpassen. Ein kleines eigenes Projekt bauen und die Ausgabe prüfen. Anschließend die Konfiguration als JSON sichern. Für eine zweite Arbeitsumgebung dieses Profil importieren und nur die dort abweichenden Werkzeugpfade ändern. So bleiben die Bedienung und die eigene Arbeitsweise konsistent.

Auch der letzte Öffnungszustand des Ausgabepanels gehört zur Arbeitsumgebung. Beim regulären Beenden wird gespeichert, ob es offen oder geschlossen war; der nächste Start stellt genau diesen Zustand und die

gewählte Höhe wieder her. Diese Sitzungsinformation wird auch bei ausgeschalteter Einstellungsautomatik gesichert. Eine Build-/Debugaktion darf das Panel weiterhin bei Bedarf öffnen.

25. Toolsets, Compiler und Build-Konfiguration

Unter **Einstellungen → Toolsets und Werkzeugpfade** verwaltet BigPadIDE benannte Toolsets für x64, x86, ARM/Thumb und ARM64. Die Einstellungen gelten für den aktuellen Windows-Benutzer und werden unabhängig von der geöffneten Datei gespeichert.

25.1 Einrichten

1. **Neu** wählen und einen Profilnamen vergeben.
2. Compilerfamilie wählen: **GCC / MinGW**, **Microsoft MSVC** oder **Intel oneAPI (icx-cl)**.
3. Zielarchitektur wählen. MSVC verwendet Windows-Projekte für x86/x64, Intel für x64; GNU-ARM-Toolsets unterstützen die bisherigen Rohbinary-Projekte.
4. Mit **Installation erkennen** eine Installation importieren. Für MSVC werden Compiler, MASM, Linker sowie VC-/Windows-SDK-Includes und Bibliotheken ermittelt. Intel ergänzt seine eigenen Header und Laufzeitbibliotheken sowie die MSVC-Abhängigkeiten. Für GCC lässt sich der `bin`-Ordner einer vorhandenen Toolchain auswählen.
5. Einzelne Eigenschaften in der Tabelle auswählen und im Feld darunter bearbeiten. **Auswählen** öffnet eine Datei-/Ordnerauswahl; **Übernehmen** übernimmt den Feldwert in den Entwurf. Include-, Bibliotheks- und zusätzliche PATH-Verzeichnisse enthalten einen absoluten Ordner pro Zeile, in Suchreihenfolge.
6. **Für Zielarchitektur aktiv** einschalten, **Prüfen** und **Speichern** wählen.

Es können mehrere Profile für dieselbe Architektur gespeichert werden. Aktiv ist jeweils eines; ein Stern kennzeichnet es. **Kopieren** erstellt eine unabhängige Profilkopie. Ohne aktives Profil bleibt die bisherige automatische Suche aktiv. Eine laufende Build-/Debugsitzung muss vor dem Wechsel beendet werden.

Leere Werkzeugfelder verwenden den Standardnamen aus dem konfigurierten `bin`-Ordner. Ein vollständig angegebener Werkzeugpfad hat Vorrang vor `bin`, Umgebungsvariablen, PATH und Erkennung. Ein fehlendes explizites Werkzeug erzeugt einen Fehler und wird nicht unbemerkt durch einen anderen Compiler ersetzt. Includes und Bibliothekspfade werden beim jeweiligen Build übergeben; zusätzliche PATH-Ordner gelten nur für den Build-Prozess. Die Windows-Systemumgebung wird nicht dauerhaft verändert.

25.2 Gemeinsame Werkzeuge

Allgemeine Werkzeuge enthält das .NET-SDK (`dotnet.exe`), NetCoreDbg, CodeLLDB, Git und das C#-Helferverzeichnis. Hier lassen sich auch ein anderes SDK oder separat installierte Debugger auswählen. Der SDK-Wechsel gilt für Build/Run, Analyse und Debugger-Helfer einschließlich ihrer Unterprozesse. Das SDK muss die vom mitgelieferten C#-Helfer benötigte .NET-Laufzeit enthalten. Das ausgelieferte Paket enthält .NET 10. Leere Felder aktivieren wieder die bisherigen Standardpfade und die Suche.

25.3 Compiler und Assembler

GCC-Profile verwenden GCC-kompatible Argumente und NASM für `.asm`. MSVC-/Intel-Profile verwenden MSVC-kompatible Argumente, `link.exe` und MASM (`ml64.exe` bzw. `ml.exe`). NASM- und MASM-Syntax sind unterschiedlich; ein Compilerwechsel konvertiert keinen Assemblerquelltext. GNU-ASM-Dateien `.s/.S` benötigen ein GCC-Profil. Für **Build & Disassemblieren** kann in MSVC-/Intel-Profilen zusätzlich GNU-/LLVM-`objcopy.exe` angegeben werden. Linker bedeutet bei GCC den Compiler-Treiber, typischerweise `g++.exe`.

25.4 Build-Konfiguration und Compileroptionen

Unter **Erstellen > Build-Konfiguration und Compileroptionen** lassen sich **Debug** und **Release** für das aktuelle Projekt bzw. die Datei wählen. Die Auswahl gilt für Build, Run und Debugging, auch bei C#. Release aktiviert die Optimierung; ein Debugstart mit Release behält sie bei und ergänzt Debugsymbole.

Build enthält Compiler- bzw. MSBuild-Optionen, beispielsweise `-p:DefineConstants=MY_FEATURE`. Für native Projekte gibt es getrennte **Linkeroptionen** und **Assembleroptionen**. Die Eingabe verwendet Windows-Quoting. **Run** enthält Programmargumente. Die zugehörige `.launch.ini` speichert die Auswahl; ältere Konfigurationen bleiben lesbar.

Gemischte GCC/NASM-Debug-Builds korrigieren die PDB-Dateiinformationen von GNU ld, damit CodeLLDB C++- und ASM-Haltepunkte unabhängig von der Quellpfadlänge auflöst.

25.5 Installation zusätzlicher Toolsets

GCC und ARM bleiben optionale, SHA-256-geprüfte Toolchain-Downloads. Zusätzlich bietet der Installer MSVC Build Tools 2026 und Intel oneAPI an, standardmäßig abgewählt. MSVC lädt einen festgelegten, SHA-256-geprüften Microsoft-Bootstrapper und startet dessen interaktiven Installer mit den C++-Werkzeugen. Intel öffnet die offizielle Compiler-Downloadseite; dort werden Betriebssystem und Paket gewählt. Die Hersteller zeigen ihre eigenen Lizenzbedingungen und gegebenenfalls Administratorabfragen. Diese beiden Aufgaben sind nicht für unbeaufsichtigte Installationen vorgesehen. Anschließend in BigPadIDE ein Profil erkennen/importieren und aktivieren.

Der BigPadIDE-Uninstaller entfernt keine separat installierten Microsoft-/Intel-Produkte. CodeLLDB und das .NET-SDK werden mit dem BigPadIDE-Paket ausgeliefert.

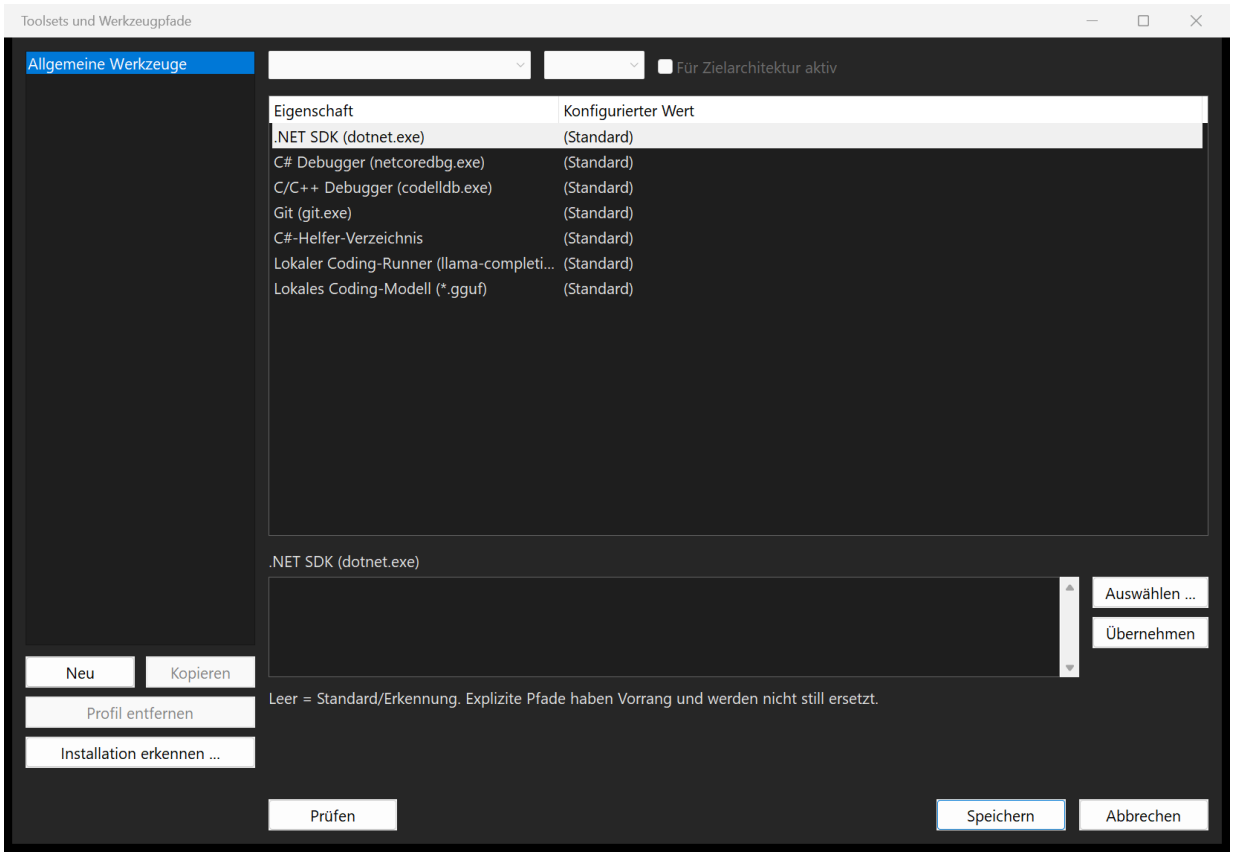


Abbildung 41. Toolsets und gemeinsame Werkzeugpfade

26. Semantische Codehilfe und lokale KI

26.1 C#-Vervollständigung und graue Vorschläge

Die C#-Codehilfe verwendet Roslyns Completion-Provider. Sie berücksichtigt den gewählten Projekt-/Solution-Kontext, Framework-Mitglieder, abgeleitete Typen und Erweiterungsmethoden. Für zuverlässige Projektergebnisse zuerst den passenden C#-Projektkontext auswählen und Buildfehler prüfen. Die schnelle lokale Liste wird asynchron um semantische Treffer ergänzt; sie ist keine Zusage vollständiger Funktionsgleichheit mit Visual Studio.

Grauer Text hinter dem Cursor ist ein Vorschlag und gehört noch nicht zum Dokument. **Tab** übernimmt ihn als eine Undo-Aktion, **Esc** verwirft ihn. Mit Pfeil hoch/runter wird stattdessen ein Eintrag der Vervollständigungsliste gewählt; **Enter** übernimmt den gewählten Namen. Verschieben des Cursors oder Ändern des Dokuments verwirft veraltete Vorschläge. Debugger-Wertetipps und Codehilfe verwenden eine kleinere, an der Editorskalierung orientierte Schrift.

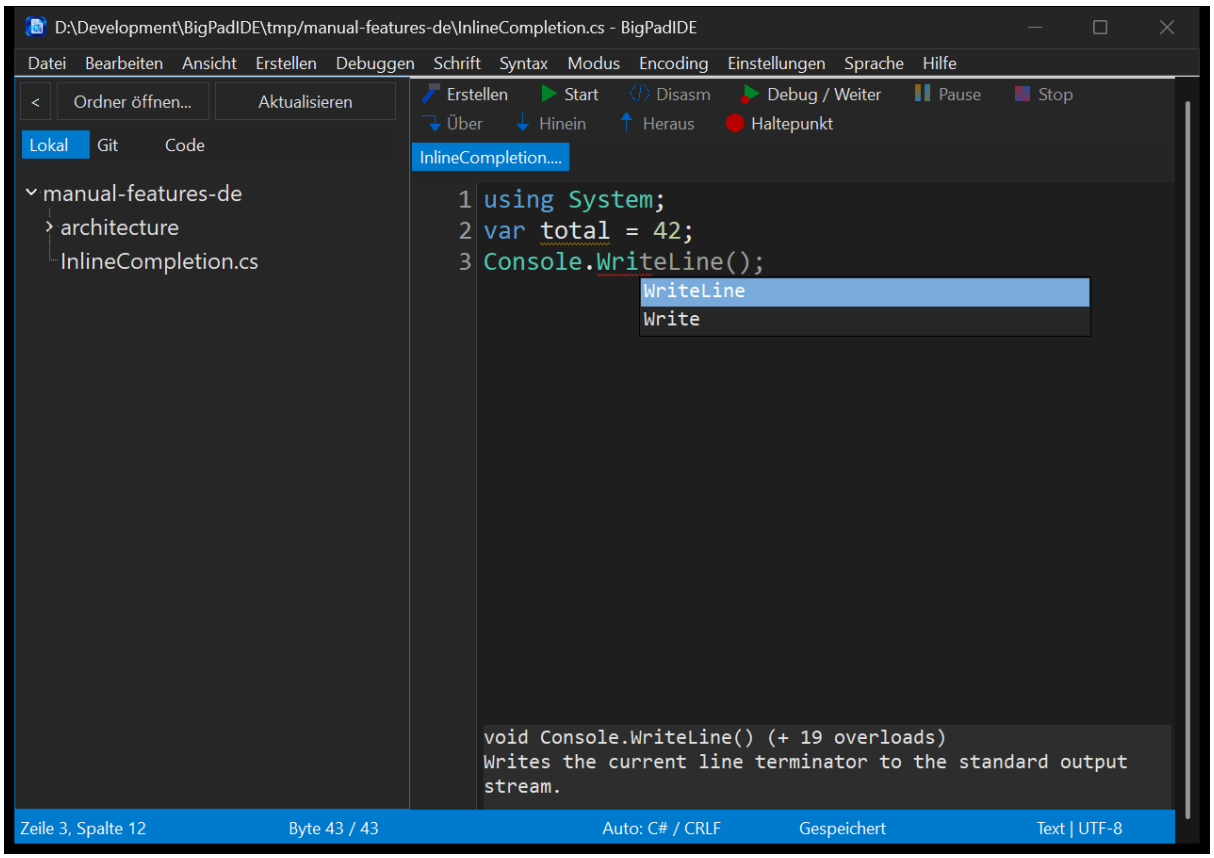


Abbildung 42. Semantische Vervollständigung und Tab-Vorschlag

26.2 Lokale KI einrichten

Im EXE-Setup sind **Lokale KI-Laufzeit** (llama.cpp CPU x64, etwa 25 MB Download) und darunter **Coding- und Chatmodell** (Qwen2.5-Coder 1.5B Instruct Q4_K_M, etwa 1,12 GB) getrennte, zunächst abgewählte Optionen. Nach Installation beider Pakete erkennt BigPadIDE sie ohne manuelle Pfade. Downloads erfolgen nur bei entsprechender Auswahl und werden mit festen SHA-256-Werten geprüft. Die spätere Inferenz bleibt lokal. Für die CPU-Variante werden keine GPU-Treiber benötigt.

Für eigene Pakete unter **Einstellungen → Toolsets und Werkzeugpfade → Allgemeine Werkzeuge** den vollständigen Pfad zur `llama-completion.exe` und zur GGUF-Datei eintragen. Das vollständige llama.cpp-Paket einschließlich DLLs muss erhalten bleiben. Explizite Pfade haben Vorrang vor den mitinstallierten Standardpfaden. Für Entwicklung ohne Setup kann `tools/Setup-LocalCoding.ps1 -ModelPath "E:\Models\modell.gguf"` ein vorhandenes b12377-Vulkan-ZIP verwenden. Danach die IDE neu starten.

Die Promptvorlagen unterstützen Qwen-/StarCoder-FIM, Qwen-Instruct sowie Gemma-4- Textdialoge mit deaktiviertem Thinking. Andere Modellfamilien brauchen passende Vorlagen. Eine `mmproj`-Datei ist ein multimodaler Projektor, kein Sprachmodell. Das Modell verbleibt am ausgewählten Speicherort und wird nicht kopiert.

26.3 Inline-Vorschlag und Chat verwenden

Am Ende einer C#-Zeile **Strg+Alt+Leertaste** drücken, um einen lokalen KI-Vorschlag anzufordern. **Tab** übernimmt die graue Ergänzung, **Esc** verwirft bzw. beendet die Anfrage. Ein laufendes Modell kann merklich Zeit und Arbeitsspeicher benötigen. Anfragen besitzen Zeit- und Ausgabegrenzen; Fehler werden angezeigt.

Erstellen → Lokaler KI-Chat … öffnet das native Chatfenster. Das Eingabefeld ist mehrzeilig; **Strg+Enter** sendet, **Abbrechen** beendet die laufende Anfrage, **Neuer Chat** leert den Verlauf. **Auswahl hinzufügen** übernimmt ausdrücklich markierten Code. Bis zu vier letzte Gesprächspaare werden innerhalb des Kontextlimits an das Modell übergeben. Der Verlauf bleibt in der IDE-Sitzung im Arbeitsspeicher. Antworten werden nicht automatisch ausgeführt oder in Dateien übernommen; vorgeschlagenen Code vor Übernahme prüfen.



Abbildung 43. Natives Chatfenster

27. Architektur und grafische Diagramme

27.1 Gespeicherten Code analysieren

Ordner im Explorer öffnen und **Erstellen → 2D-Architekturanalyse ...** wählen. Die aus CodeCity3D übernommenen Parser analysieren gespeicherte Dateien mehrerer Sprachen. C#-Projekte nutzen zusätzlich MSBuild/Roslyn zur Auflösung erkannter Beziehungen. Nicht aufgelöste oder nicht-C#-Beziehungen können heuristisch sein; Analysefehler werden im Ergebnis ausgewiesen. Ungespeicherte Editoränderungen sind nicht Teil dieser Analyse. Große Ordner gegebenenfalls auf ein Teilprojekt begrenzen.

Das Ergebnis öffnet sich als bearbeitbares natives Diagramm. Zoomen, Einpassen, Elementinformationen und **Quellcode öffnen** helfen bei der Navigation. Der Quellverweis führt zur erkannten Datei/Zeile. Das Modell erfasst Codebeziehungen; eine vollständige Infrastruktur- oder Laufzeitarchitektur lässt sich daraus nicht automatisch ableiten.

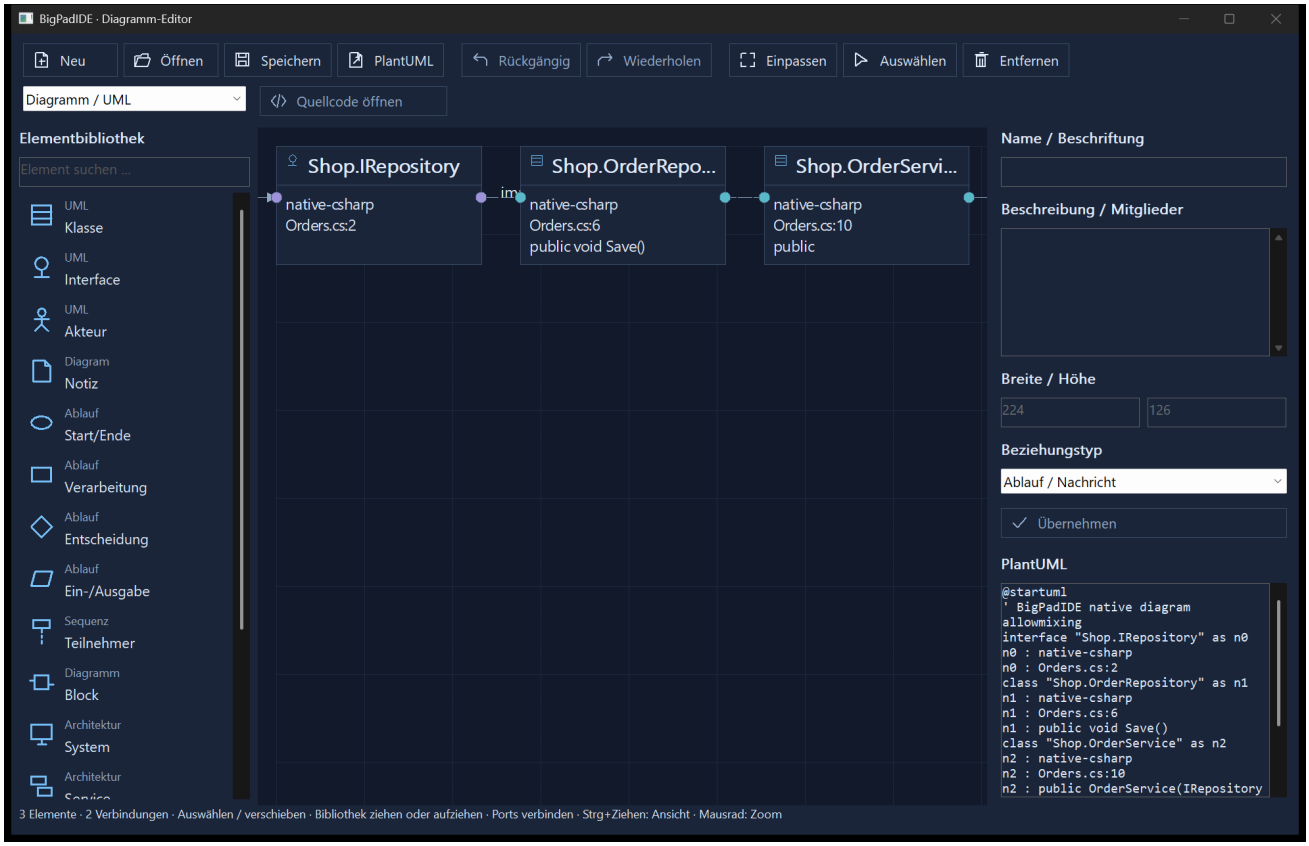


Abbildung 44. Architekturanalyse aus gespeichertem Code

27.2 UML, Ablauf und Systemlandschaft zeichnen

Erstellen → Grafischer Diagramm-Editor ... öffnet die Zeichenoberfläche ohne Browser. Links stehen Elemente bereit, in der Mitte die Zeichenfläche, rechts Eigenschaften und PlantUML-Vorschau. Formen aus der Bibliothek ziehen oder auf der Fläche mit der Maus aufziehen. Elemente verschieben, über Eckanfasser skalieren und über Ports verbinden. Bezeichnungen, Mitglieder und Beziehungstyp im Eigenschaftenbereich bearbeiten und übernehmen.

Die Elementfamilien decken UML-Klassen und Schnittstellen, Abläufe mit Start, Prozess und Entscheidung, Sequenzteilnehmer sowie Systeme, Services, Datenbanken, externe Systeme und Infrastruktur ab.

Infrastrukturbeziehungen werden manuell dokumentiert. Icons, abgestufte Schriften und Farben folgen dem IDE-Theme.

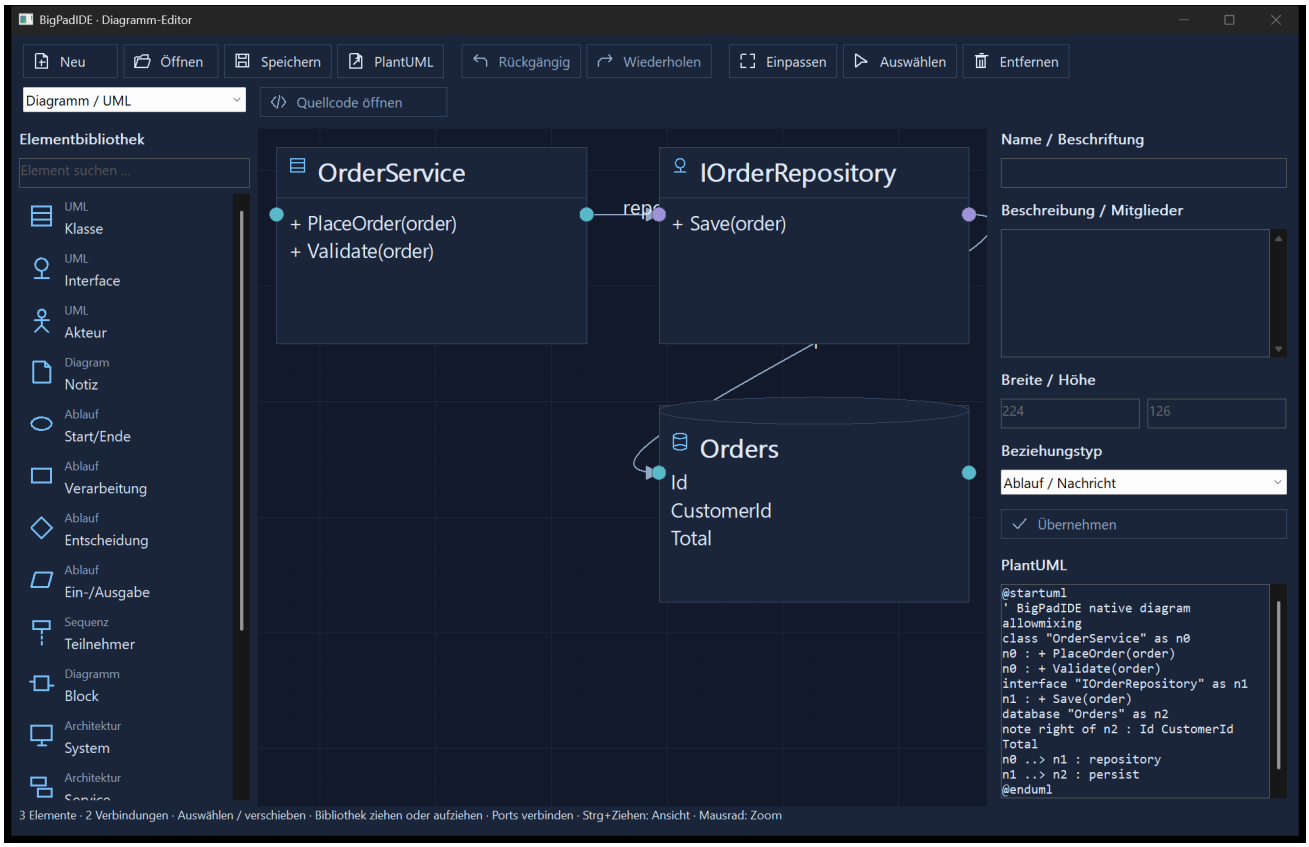


Abbildung 45. Grafischer Diagramm-Editor

27.3 Speichern und austauschen

Als `.bpdiagram` speichern, um Positionen, Größen, Texte, Beziehungen und Quellverweise später weiterzubearbeiten. Undo/Redo erleichtert lokale Änderungen. Der PlantUML-Export erzeugt lesbaren, mit Git versionierbaren Diagrammtext. Er ist kein vollständiges semantisches UML-/XMI-Austauschmodell. XMI und ein hierarchischer C4-Drilldown sind noch nicht implementiert. Diagramme sind auf 1000 Elemente, 3000 Kanten und 4 MiB begrenzt; Teilmodelle verwenden, wenn diese Grenzen erreicht werden.

27.4 Explorer und Assembly-Vorschau

Im Explorer-Kontextmenü zwischen **Baum**, **Liste** und **Details** umschalten. Details zeigen Dateigröße und Änderungsdatum; Ordner lassen sich direkt öffnen. **Im Windows-Explorer anzeigen** öffnet einen Ordner oder markiert eine Datei. Die Ansichtsart bleibt nach Neustart erhalten.

Bei erkannten .NET-Assemblies bietet das Kontextmenü **.NET analysieren: IL / C# ...**, bei unterstützten nativen PE-/ELF-Dateien das Disassemblieren. Dateien können auch im Hex-Editor geöffnet werden. Die IL-/C#-Vorschau ist schreibgeschützt, verwendet Syntaxfarben des Themes und unterstützt **Strg+Mausrad** für die Schriftgröße. Erneutes Öffnen derselben DLL lädt das vorhandene Panel neu. Ist es leer, zuerst Statusmeldung, Lesbarkeit der DLL und Helfer-/SDK-Pfade prüfen.

28. Lizenzen und Veröffentlichungsstand

Hilfe → Lizenzen und Drittanbieter-Komponenten … öffnet eine native Offline- Übersicht. Komponente wählen, um Version, Lizenz, Copyright-Hinweise und mitgelieferte Drittanbietertexte zu lesen. **Projektseite** öffnet den Herausgeberverweis, **Lizenzordner** die Dateien der Installation. Optionale Komponenten sind als solche gekennzeichnet; eigene Modelle unterliegen weiterhin ihren Herausgeberlizenzen.

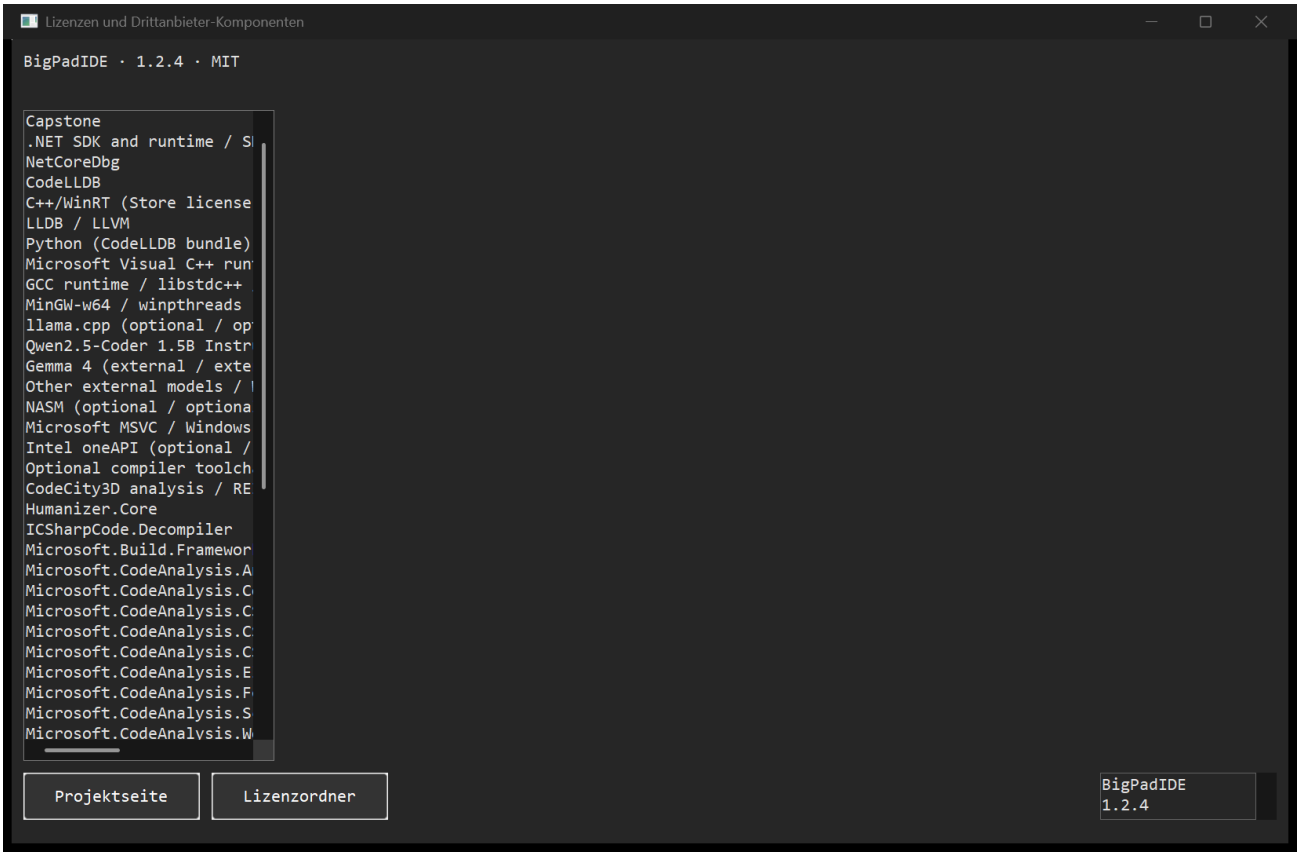


Abbildung 46. Drittanbieter-Lizenzen offline lesen

Die Übersicht wird beim Build aus den aufgelösten NuGet-Paketen, SDK-Hinweisen und hinterlegten nativen Lizenzquellen erzeugt. Sie enthält unter anderem Capstone, Roslyn/ILSpy, Debugadapter, LLVM/Python, GNU-/MinGW-Laufzeiten, C++/WinRT und die optionalen KI-Pakete. Diese Hinweise werden mit dem EXE- und Store-Payload geliefert.

Dieser dokumentierte Stand ist ein erster Abschluss für die Store-Vorbereitung. Ein erfolgreicher lokaler Build oder ein erzeugtes MSIX bedeutet noch keine Store-Zertifizierung oder Veröffentlichung. Store-Einreichung, Signierung, Entitlement-Prüfung im Paket und Freigabe bleiben getrennte Veröffentlichungsschritte.

Bildreferenz: weitere Ansichten

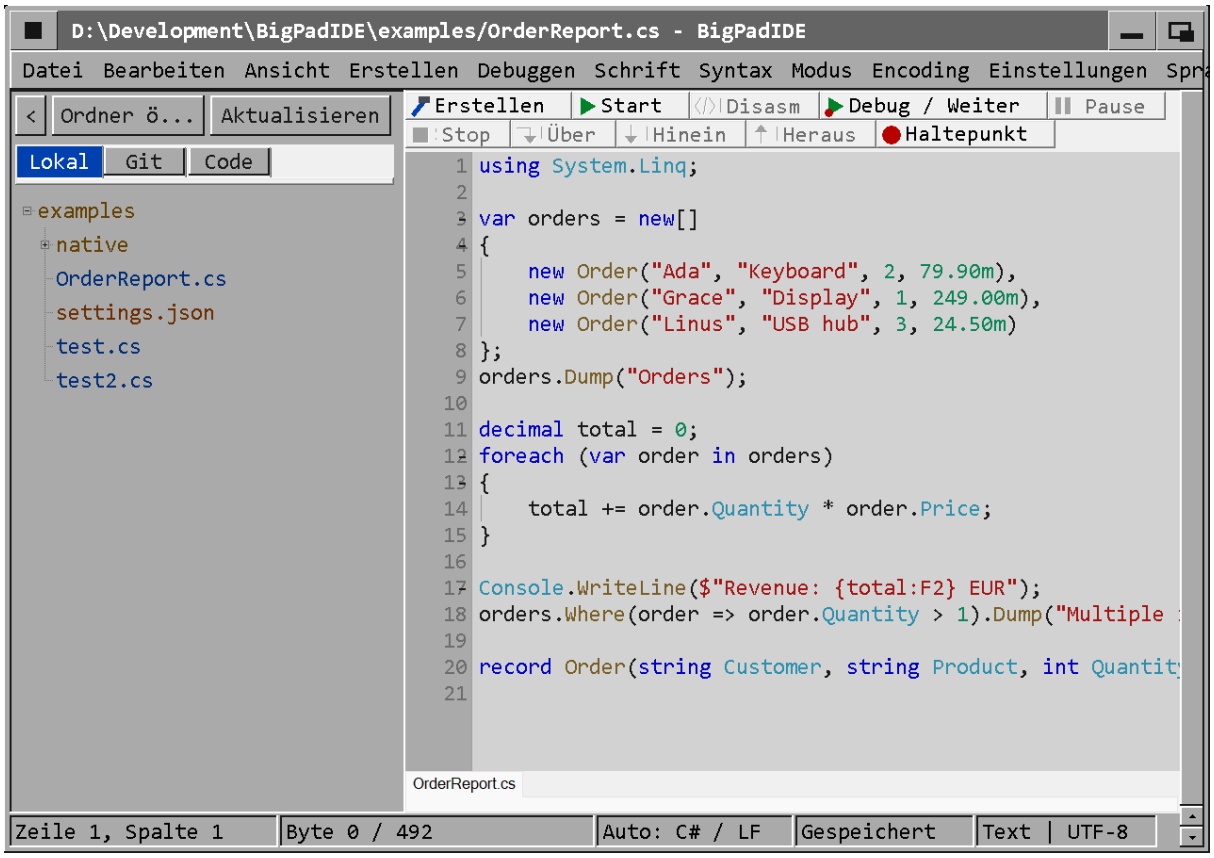


Abbildung 47. BigPadIDE mit Ordnerbaum und C#-Syntaxfarben

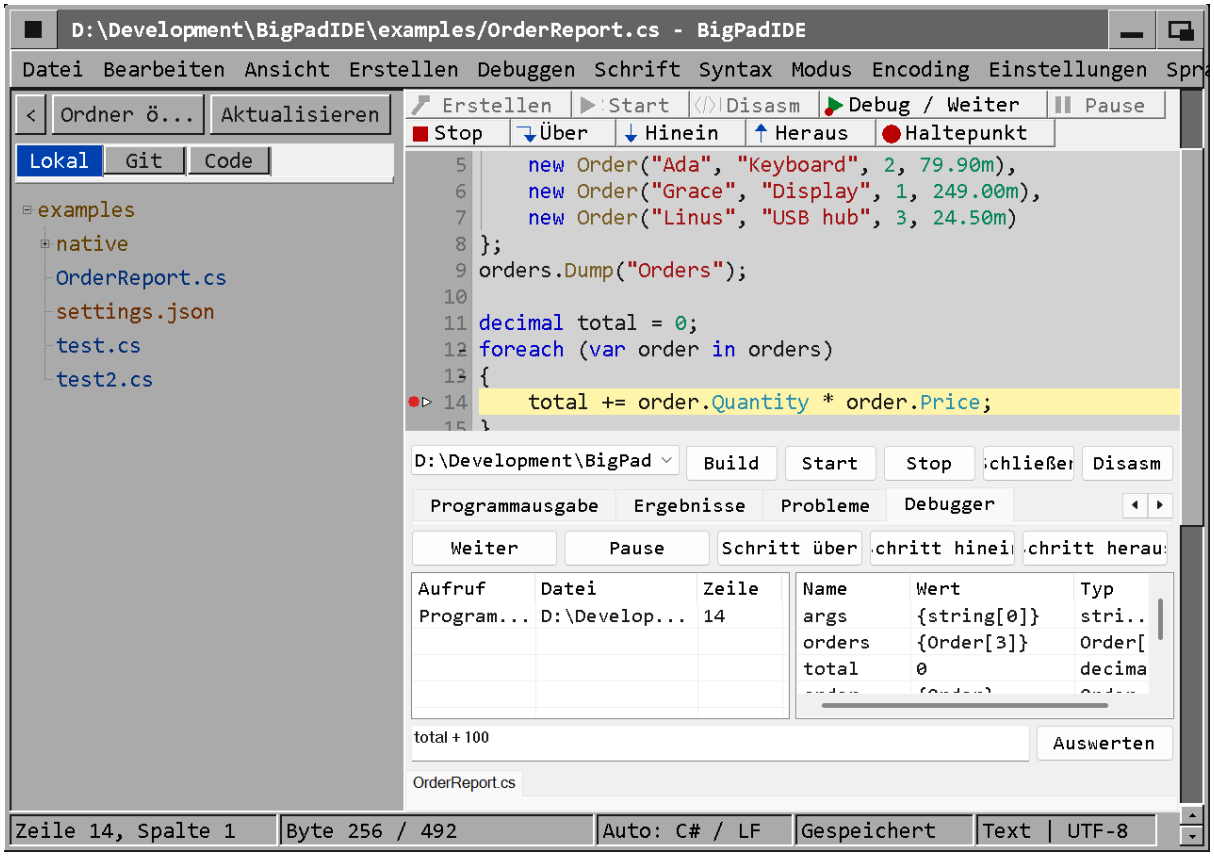


Abbildung 48. C#-Debugger mit Haltepunkt und Auswertung

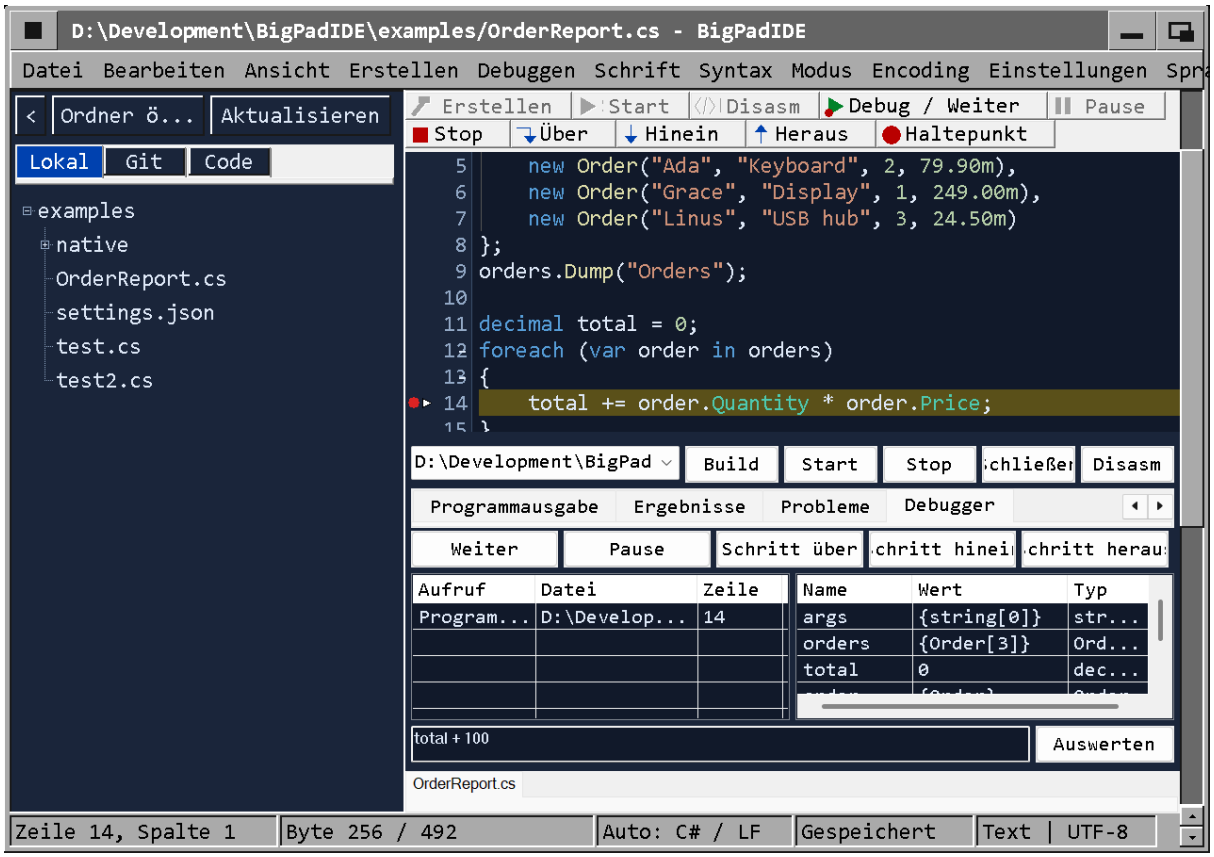


Abbildung 49. Debugger im Mitternacht-Theme