

Financial Anomaly Interpretability Using LLMs

F.A.I.L
Project Report

Course: MA5741 - Object Oriented Programming

Team Members:

| | |
|------------------|------------|
| Aritra Dasgupta | (MA25M005) |
| Ashwini | (MA25M006) |
| Devharish N | (MA25M010) |
| Rumaiz Ibrahim K | (MA25M023) |

Contents

| | |
|--|----------|
| 1 Project Overview | 3 |
| 1.1 Description | 3 |
| 1.2 Key Features | 3 |
| 1.3 Project Structure | 4 |
| 2 Core OOP Concepts Implementation | 5 |
| 2.1 Encapsulation | 5 |
| 2.1.1 Definition | 5 |
| 2.1.2 Implementation in Project | 5 |
| 2.1.3 Benefits | 6 |
| 2.2 Inheritance | 6 |
| 2.2.1 Definition | 6 |
| 2.2.2 Example: Processor Inheritance Hierarchy | 6 |
| 2.2.3 Benefits | 6 |
| 2.3 Polymorphism | 7 |
| 2.3.1 Definition | 7 |
| 2.3.2 Example: Progress Observer Polymorphism | 7 |
| 2.3.3 Benefits | 7 |
| 2.4 Abstraction | 7 |
| 2.4.1 Definition | 7 |
| 2.4.2 Example: Abstract Base Classes | 8 |
| 2.4.3 Benefits | 8 |
| 3 SOLID Principles Implementation | 9 |
| 3.1 Single Responsibility Principle (SRP) | 9 |
| 3.1.1 Definition | 9 |
| 3.1.2 Implementation | 9 |
| 3.1.3 Focused Classes | 9 |
| 3.2 Open/Closed Principle (OCP) | 10 |
| 3.2.1 Definition | 10 |
| 3.2.2 Example: Strategy Pattern for Extension | 10 |
| 3.3 Liskov Substitution Principle (LSP) | 10 |
| 3.3.1 Definition | 10 |
| 3.3.2 Example: Strategy Substitution | 10 |
| 3.4 Interface Segregation Principle (ISP) | 11 |
| 3.4.1 Definition | 11 |
| 3.4.2 Example: Segregated Interfaces | 11 |
| 3.5 Dependency Inversion Principle (DIP) | 12 |
| 3.5.1 Definition | 12 |

| | |
|---|-----------|
| 3.5.2 Example: Dependency Injection | 12 |
| 4 Design Patterns Implementation | 13 |
| 4.1 Strategy Pattern | 13 |
| 4.1.1 Definition | 13 |
| 4.1.2 Implementation Areas | 13 |
| 4.1.3 Example Code | 13 |
| 4.2 Factory Pattern | 14 |
| 4.2.1 Definition | 14 |
| 4.2.2 Example: News Retriever Factory | 14 |
| 4.3 Observer Pattern | 14 |
| 4.3.1 Definition | 14 |
| 4.3.2 Example: Progress Observer | 14 |
| 4.4 Template Method Pattern | 15 |
| 4.4.1 Definition | 15 |
| 4.4.2 Example: Pipeline Template Method | 15 |
| 4.5 Builder Pattern | 16 |
| 4.5.1 Definition | 16 |
| 4.5.2 Example: Pipeline Builder | 16 |
| 4.6 Repository Pattern | 16 |
| 4.6.1 Definition | 16 |
| 4.6.2 Example: Data Repository | 17 |
| 5 Architecture Overview | 18 |
| 5.1 Layered Architecture | 18 |
| 5.2 UML Diagrams | 18 |
| 6 Code Examples and Demonstrations | 26 |
| 6.1 Complete Pipeline Execution | 26 |
| 6.2 Custom Strategy Implementation | 26 |
| 6.3 Multiple Observers | 27 |
| 7 Benefits and Best Practices | 28 |
| 7.1 Benefits of OOP in This Project | 28 |
| 7.2 Best Practices Demonstrated | 28 |
| 7.3 Design Pattern Benefits Summary | 29 |
| 8 Conclusion | 30 |
| 8.1 Comprehensive OOP Implementation | 30 |
| 8.2 SOLID Principles | 30 |
| 8.3 Design Patterns | 30 |
| 8.4 Professional Software Engineering | 31 |

Chapter 1

Project Overview

1.1 Description

A sophisticated quantitative analysis framework that demonstrates advanced Object-Oriented Programming principles, SOLID principles, and design patterns for comprehensive market analysis. The project implements a complete pipeline for detecting financial anomalies, retrieving relevant news, generating embeddings, analyzing similarities, and generating AI-powered explanations.

1.2 Key Features

- Comprehensive market data analysis
- Multiple anomaly detection strategies
- News retrieval from various sources
- Embedding generation for semantic analysis
- Similarity analysis using advanced algorithms
- AI-powered event explanations
- Progress tracking and monitoring
- App which can be hosted on Streamlit

1.3 Project Structure

```
F.A.I.L_OOPS/
├── core/ (Core framework)
│   ├── base.py
│   ├── interfaces.py
│   └── exceptions.py
├── processors/ (Data processing)
│   ├── data_loader.py
│   ├── anomaly_detector.py
│   ├── news_retriever.py
│   ├── embedding_generator.py
│   ├── similarity_analyzer.py
│   └── ai_explainer.py
├── pipeline/ (Pipeline orchestration)
│   ├── quantitative_pipeline.py
│   ├── progress_observer.py
│   └── pipeline_factory.py
└── ui/ (User interface)
    ├── streamlit_app.py
    └── ui_components.py
```

Chapter 2

Core OOP Concepts Implementation

2.1 Encapsulation

2.1.1 Definition

Bundling data and methods together within a class, hiding internal implementation details and providing controlled access through public methods.

2.1.2 Implementation in Project

Example 1: BaseProcessor Class

The BaseProcessor class demonstrates encapsulation by:

- **Private attributes:** Using underscore prefix (_observers) to indicate internal implementation
- **Public methods:** Providing controlled access through public methods like add_observer(), remove_observer()
- **Protected attributes:** Using protected access for logger attributes

```
1 class BaseProcessor(ABC):
2     def __init__(self, name: str):
3         self.name = name # Public attribute
4         self._observers: List[IPrgressObserver] = [] # Private
5         self.logger = logging.getLogger(f"__name__.{name}")
6
7     def add_observer(self, observer: IProgressObserver) -> None:
8         """Public method for controlled access"""
9         self._observers.append(observer)
10
11    def _notify_progress(self, step: str,
12                         progress: float, message: str) -> None:
13        """Private method - internal implementation"""
14        for observer in self._observers:
15            observer.update_progress(step, progress, message)
```

2.1.3 Benefits

- Prevents external code from directly accessing internal state
- Allows for controlled access through public methods
- Makes the code more maintainable and less error-prone
- Enables future changes to internal implementation without affecting clients

2.2 Inheritance

2.2.1 Definition

Creating new classes based on existing classes, inheriting their properties and methods. Promotes code reuse and establishes "is-a" relationships.

2.2.2 Example: Processor Inheritance Hierarchy

```

1 # Base class
2 class BaseProcessor(ABC):
3     def __init__(self, name: str):
4         self.name = name
5         self.logger = logging.getLogger(f"__name__.{name}")
6
7     def log_info(self, message: str) -> None:
8         self.logger.info(f"[{self.name}] {message}")
9
10 # Derived class
11 class DataLoader(BaseProcessor):
12     def __init__(self, strategy: DataLoadStrategy):
13         super().__init__("DataLoader") # Call parent
14         self.strategy = strategy
15
16     def load_market_data(self, ticker: str, benchmark: str,
17                          start_date: str, end_date: str):
18         self.log_info("Loading market data") # Inherited
19         return self.strategy.load_data(ticker, benchmark,
20                                         start_date, end_date)
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
50
51
52
53
54
55
56
57
58
59
5
60
61
62
63
64
65
66
67
68
69
6
70
71
72
73
74
75
76
77
78
79
7
80
81
82
83
84
85
86
87
88
89
8
90
91
92
93
94
95
96
97
98
99
9
100
101
102
103
104
105
106
107
108
109
10
110
111
112
113
114
115
116
117
118
119
11
120
121
122
123
124
125
126
127
128
129
12
130
131
132
133
134
135
136
137
138
13
139
140
141
142
143
144
145
146
147
14
148
149
150
151
152
153
154
155
156
15
157
158
159
160
161
162
163
164
165
16
166
167
168
169
170
171
172
173
174
17
175
176
177
178
179
180
181
182
183
18
184
185
186
187
188
189
18
189
190
191
192
193
194
195
196
19
197
198
199
19
199
200
201
202
203
204
205
206
20
207
208
209
20
209
210
211
212
213
214
215
21
216
217
218
219
21
219
220
221
222
223
224
225
226
22
227
228
229
22
229
230
231
232
233
234
235
23
236
237
238
239
23
239
240
241
242
243
244
245
24
246
247
248
249
24
249
250
251
252
253
254
255
25
256
257
258
259
25
259
260
261
262
263
264
265
26
266
267
268
269
26
269
270
271
272
273
274
275
27
276
277
278
279
27
279
280
281
282
283
284
285
28
286
287
288
289
28
289
290
291
292
293
294
295
29
296
297
298
299
29
299
300
301
302
303
304
305
30
306
307
308
309
30
309
310
311
312
313
314
315
31
316
317
318
319
31
319
320
321
322
323
324
325
32
326
327
328
329
32
329
330
331
332
333
334
335
33
336
337
338
339
33
339
340
341
342
343
344
345
34
346
347
348
349
34
349
350
351
352
353
354
355
35
356
357
358
359
35
359
360
361
362
363
364
365
36
366
367
368
369
36
369
370
371
372
373
374
375
37
376
377
378
379
37
379
380
381
382
383
384
385
38
386
387
388
389
38
389
390
391
392
393
394
395
39
396
397
398
399
39
399
400
401
402
403
404
405
40
406
407
408
409
40
409
410
411
412
413
414
415
41
416
417
418
419
41
419
420
421
422
423
424
425
42
426
427
428
429
42
429
430
431
432
433
434
435
43
436
437
438
439
43
439
440
441
442
443
444
445
44
446
447
448
449
44
449
450
451
452
453
454
455
45
456
457
458
459
45
459
460
461
462
463
464
465
46
466
467
468
469
46
469
470
471
472
473
474
475
47
476
477
478
479
47
479
480
481
482
483
484
485
48
486
487
488
489
48
489
490
491
492
493
494
495
49
496
497
498
499
49
499
500
501
502
503
504
505
50
506
507
508
509
50
509
510
511
512
513
514
515
51
516
517
518
519
51
519
520
521
522
523
524
525
52
526
527
528
529
52
529
530
531
532
533
534
535
53
536
537
538
539
53
539
540
541
542
543
544
545
54
546
547
548
549
54
549
550
551
552
553
554
555
55
556
557
558
559
55
559
560
561
562
563
564
565
56
566
567
568
569
56
569
570
571
572
573
574
575
57
576
577
578
579
57
579
580
581
582
583
584
585
58
586
587
588
589
58
589
590
591
592
593
594
595
59
596
597
598
599
59
599
600
601
602
603
604
605
60
606
607
608
609
60
609
610
611
612
613
614
615
61
616
617
618
619
61
619
620
621
622
623
624
625
62
626
627
628
629
62
629
630
631
632
633
634
635
63
636
637
638
639
63
639
640
641
642
643
644
645
64
646
647
648
649
64
649
650
651
652
653
654
655
65
656
657
658
659
65
659
660
661
662
663
664
665
66
666
667
668
669
66
669
670
671
672
673
674
675
67
676
677
678
679
67
679
680
681
682
683
684
685
68
686
687
688
689
68
689
690
691
692
693
694
695
69
696
697
698
699
69
699
700
701
702
703
704
705
70
706
707
708
709
70
709
710
711
712
713
714
715
71
716
717
718
719
71
719
720
721
722
723
724
725
72
726
727
728
729
72
729
730
731
732
733
734
735
73
736
737
738
739
73
739
740
741
742
743
744
745
74
746
747
748
749
74
749
750
751
752
753
754
755
75
756
757
758
759
75
759
760
761
762
763
764
765
76
766
767
768
769
76
769
770
771
772
773
774
775
77
776
777
778
779
77
779
780
781
782
783
784
785
78
786
787
788
789
78
789
790
791
792
793
794
795
79
796
797
798
799
79
799
800
801
802
803
804
805
80
806
807
808
809
80
809
810
811
812
813
814
815
81
816
817
818
819
81
819
820
821
822
823
824
825
82
826
827
828
829
82
829
830
831
832
833
834
835
83
836
837
838
839
83
839
840
841
842
843
844
845
84
846
847
848
849
84
849
850
851
852
853
854
855
85
856
857
858
859
85
859
860
861
862
863
864
865
86
866
867
868
869
86
869
870
871
872
873
874
875
87
876
877
878
879
87
879
880
881
882
883
884
885
88
886
887
888
889
88
889
890
891
892
893
894
895
89
896
897
898
899
89
899
900
901
902
903
904
905
90
906
907
908
909
90
909
910
911
912
913
914
915
91
916
917
918
919
91
919
920
921
922
923
924
925
92
926
927
928
929
92
929
930
931
932
933
934
935
93
936
937
938
939
93
939
940
941
942
943
944
945
94
946
947
948
949
94
949
950
951
952
953
954
955
95
956
957
958
959
95
959
960
961
962
963
964
965
96
966
967
968
969
96
969
970
971
972
973
974
975
97
976
977
978
979
97
979
980
981
982
983
984
985
98
986
987
988
989
98
989
990
991
992
993
994
995
99
996
997
998
999
99
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
100
1009
1010
1011
1012
1013
1014
1015
101
1016
1017
1018
1019
101
1019
1020
1021
1022
1023
1024
1025
102
1026
1027
1028
1029
102
1029
1030
1031
1032
1033
1034
1035
103
1036
1037
1038
1039
103
1039
1040
1041
1042
1043
1044
1045
104
1046
1047
1048
1049
104
1049
1050
1051
1052
1053
1054
1055
105
1056
1057
1058
1059
105
1059
1060
1061
1062
1063
1064
1065
106
1066
1067
1068
1069
106
1069
1070
1071
1072
1073
1074
1075
107
1076
1077
1078
1079
107
1079
1080
1081
1082
1083
1084
1085
108
1086
1087
1088
1089
108
1089
1090
1091
1092
1093
1094
1095
109
1096
1097
1098
1099
109
1099
1100
1101
1102
1103
1104
1105
110
1106
1107
1108
1109
110
1109
1110
1111
1112
1113
1114
1115
111
1116
1117
1118
1119
111
1119
1120
1121
1122
1123
1124
1125
112
1126
1127
1128
1129
112
1129
1130
1131
1132
1133
1134
1135
113
1136
1137
1138
1139
113
1139
1140
1141
1142
1143
1144
1145
114
1146
1147
1148
1149
114
1149
1150
1151
1152
1153
1154
1155
115
1156
1157
1158
1159
115
1159
1160
1161
1162
1163
1164
1165
116
1166
1167
1168
1169
116
1169
1170
1171
1172
1173
1174
1175
117
1176
1177
1178
1179
117
1179
1180
1181
1182
1183
1184
1185
118
1186
1187
1188
1189
118
1189
1190
1191
1192
1193
1194
1195
119
1196
1197
1198
1199
119
1199
1200
1201
1202
1203
1204
1205
120
1206
1207
1208
1209
120
1209
1210
1211
1212
1213
1214
1215
121
1216
1217
1218
1219
121
1219
1220
1221
1222
1223
1224
1225
122
1226
1227
1228
1229
122
1229
1230
1231
1232
1233
1234
1235
123
1236
1237
1238
1239
123
1239
1240
1241
1242
1243
1244
1245
124
1246
1247
1248
1249
124
1249
1250
1251
1252
1253
1254
1255
125
1256
1257
1258
1259
125
1259
1260
1261
1262
1263
1264
1265
126
1266
1267
1268
1269
126
1269
1270
1271
1272
1273
1274
1275
127
1276
1277
1278
1279
127
1279
1280
1281
1282
1283
1284
1285
128
1286
1287
1288
1289
128
1289
1290
1291
1292
1293
1294
1295
129
1296
1297
1298
1299
129
1299
1300
1301
1302
1303
1304
1305
130
1306
1307
1308
1309
130
1309
1310
1311
1312
1313
1314
1315
131
1316
1317
1318
1319
131
1319
1320
1321
1322
1323
1324
1325
132
1326
1327
1328
1329
132
1329
1330
1331
1332
1333
1334
1335
133
1336
1337
1338
1339
133
1339
1340
1341
1342
1343
1344
1345
134
1346
1347
1348
1349
134
1349
1350
1351
1352
1353
1354
1355
135
1356
1357
1358
1359
135
1359
1360
1361
1362
1363
1364
1365
136
1366
1367
1368
1369
136
1369
1370
1371
1372
1373
1374
1375
137
1376
1377
1378
1379
137
1379
1380
1381
1382
1383
1384
1385
138
1386
1387
1388
1389
138
1389
1390
1391
1392
1393
1394
1395
139
1396
1397
1398
1399
139
1399
1400
1401
1402
1403
1404
1405
140
1406
1407
1408
1409
140
1409
1410
1411
1412
1413
1414
1415
141
1416
1417
1418
1419
141
1419
1420
1421
1422
1423
1424
1425
142
1426
1427
1428
1429
142
1429
1430
1431
1432
1433
1434
1435
143
1436
1437
1438
1439
143
1439
1440
1441
1442
1443
1444
1445
144
1446
1447
1448
1449
144
1449
1450
1451
1452
1453
1454
1455
145
1456
1457
1458
1459
145
1459
1460
1461
1462
1463
1464
1465
146
1466
1467
1468
1469
146
1469
1470
1471
1472
1473
1474
1475
147
1476
1477
1478
1479
147
1479
1480
1481
1482
1483
1484
1485
148
1486
1487
1488
1489
148
1489
1490
1491
1492
1493
1494
1495
149
1496
1497
1498
1499
149
1499
1500
1501
1502
1503
1504
1505
150
1506
1507
1508
1509
150
1509
1510
1511
1512
1513
1514
1515
151
1516
1517
1518
1519
151
1519
1520
1521
1522
1523
1524
1525
152
1526
1527
1528
1529
152
1529
1530
1531
1532
1533
1534
1535
153
1536
1537
1538
1539
153
1539
1540
1541
1542
1543
1544
1545
154
1546
1547
1548
1549
154
1549
1550
1551
1552
1553
1554
1555
155
1556
1557
1558
1559
155
1559
1560
1561
1562
1563
1564
1565
156
1566
1567
1568
1569
156
1569
1570
1571
1572
1573
1574
1575
157
1576
1577
1578
1579
157
1579
1580
1581
1582
1583
1584
1585
158
1586
1587
1588
1589
158
1589
1590
1591
1592
1593
1594
1595
159
1596
1597
1598
1599
159
1599
1600
1601
1602
1603
1604
1605
160
1606
1607
1608
1609
160
1609
1610
1611
1612
1613
1614
1615
161
1616
1617
1618
1619
161
1619
1620
1621
1622
1623
1624
1625
162
1626
1627
1628
1629
162
1629
1630
1631
1632
1633
1634
1635
163
1636
1637
1638
1639
163
1639
1640
1641
1642
1643
1644
1645
164
164
```

2.3 Polymorphism

2.3.1 Definition

The ability of objects of different types to be treated as instances of the same type through a common interface. Enables runtime method resolution and flexible, extensible code.

2.3.2 Example: Progress Observer Polymorphism

```

1 # Common interface
2 class IProgressObserver(ABC):
3     @abstractmethod
4     def update_progress(self, step: str,
5                         progress: float, message: str) -> None:
6         pass
7
8 # Different implementations
9 class ConsoleProgressObserver(IProgressObserver):
10    def update_progress(self, step: str,
11                        progress: float, message: str) -> None:
12        print(f"[{step}] {progress}% - {message}")
13
14 class StreamlitProgressObserver(IProgressObserver):
15    def update_progress(self, step: str,
16                        progress: float, message: str) -> None:
17        if self.progress_bar:
18            self.progress_bar.progress(progress / 100.0)
19
20 # Polymorphic usage
21 observers = [ConsoleProgressObserver(),
22               StreamlitProgressObserver()]
23 for observer in observers: # Same interface, different behavior
24     observer.update_progress("analysis", 50.0, "Processing")

```

2.3.3 Benefits

- **Flexibility:** Easy to swap implementations without changing client code
- **Extensibility:** New implementations can be added without modifying existing code
- **Runtime behavior:** Behavior is determined at runtime, not compile time
- **Code reuse:** Client code can work with multiple implementations

2.4 Abstraction

2.4.1 Definition

Hiding complex implementation details and showing only essential features. Provides clear contracts and reduces cognitive load.

2.4.2 Example: Abstract Base Classes

```
1 # Abstract base class - defines contract without implementation
2 class AnomalyDetectionStrategy(ABC):
3     @abstractmethod
4         def detect(self, data: pd.DataFrame,
5                    **kwargs) -> pd.DataFrame:
6             """Detect anomalies using the specific strategy"""
7             pass
8
9     @abstractmethod
10    def get_parameters(self) -> Dict[str, Any]:
11        """Get the current parameters"""
12        pass
13
14 # Concrete implementation
15 class ZScoreStrategy(AnomalyDetectionStrategy):
16     def detect(self, data: pd.DataFrame,
17                **kwargs) -> pd.DataFrame:
18         # Complex Z-score calculation logic
19         events = []
20         for idx, row in data.iterrows():
21             if abs(row['Z_score']) > self.z_threshold:
22                 events.append({
23                     'Date': idx,
24                     'Event_Type': 'Positive Outlier'
25                     if row['Z_score'] > 0
26                     else 'Negative Outlier',
27                 })
28         return pd.DataFrame(events)
```

2.4.3 Benefits

- **Simplifies complex systems:** Clients only see what they need
- **Reduces cognitive load:** Users don't need to understand implementation details
- **Provides clear contracts:** Interfaces define what methods must be implemented
- **Enables multiple implementations:** Different classes can implement the same interface

Chapter 3

SOLID Principles Implementation

3.1 Single Responsibility Principle (SRP)

3.1.1 Definition

A class should have only one reason to change. Each class should have a single, well-defined responsibility.

3.1.2 Implementation

```
1 # Single responsibility: Data loading only
2 class DataLoader(BaseProcessor):
3     def load_market_data(self, ticker: str, benchmark: str,
4                          start_date: str, end_date: str):
5         return self.strategy.load_data(ticker, benchmark,
6                                         start_date, end_date)
7
8 # Single responsibility: Data persistence only
9 class BaseDataRepository(IDataRepository):
10    def save_events(self, events: pd.DataFrame,
11                    filename: str) -> None:
12        filepath = f"{self.base_path}/{filename}"
13        events.to_parquet(filepath)
14
15    def load_events(self, filename: str) -> pd.DataFrame:
16        filepath = f"{self.base_path}/{filename}"
17        return pd.read_parquet(filepath)
```

3.1.3 Focused Classes

Each processor class has a single, clear responsibility:

- **DataLoader:** Only responsible for loading market data
- **AnomalyDetector:** Only responsible for detecting anomalies
- **NewsRetriever:** Only responsible for retrieving news
- **EmbeddingGenerator:** Only responsible for generating embeddings

- **SimilarityAnalyzer:** Only responsible for similarity analysis
- **AIEExplainer:** Only responsible for generating AI explanations

3.2 Open/Closed Principle (OCP)

3.2.1 Definition

Software entities should be open for extension but closed for modification. New functionality should be added through extension, not by modifying existing code.

3.2.2 Example: Strategy Pattern for Extension

```

1 # Base class - closed for modification
2 class AnomalyDetectionStrategy(ABC):
3     @abstractmethod
4     def detect(self, data: pd.DataFrame,
5                **kwargs) -> pd.DataFrame:
6         pass
7
8 # Existing implementation - no modification needed
9 class ZScoreStrategy(AnomalyDetectionStrategy):
10    def detect(self, data: pd.DataFrame,
11               **kwargs) -> pd.DataFrame:
12        # Z-score implementation
13        pass
14
15 # New implementation - extension without modification
16 class NewAnomalyStrategy(AnomalyDetectionStrategy):
17     def detect(self, data: pd.DataFrame,
18                **kwargs) -> pd.DataFrame:
19        # New anomaly detection algorithm
20        pass
21
22 # Client code works with all strategies
23 detector = AnomalyDetector(NewAnomalyStrategy())

```

3.3 Liskov Substitution Principle (LSP)

3.3.1 Definition

Objects of a superclass should be replaceable with objects of a subclass without breaking functionality. Derived classes must honor the contracts of their base classes.

3.3.2 Example: Strategy Substitution

```

1 # Base class contract
2 class AnomalyDetectionStrategy(ABC):
3     @abstractmethod
4     def detect(self, data: pd.DataFrame,
5                **kwargs) -> pd.DataFrame:

```

```

6     pass
7
8 # Subclass implementations honor the contract
9 class ZScoreStrategy(AnomalyDetectionStrategy):
10    def detect(self, data: pd.DataFrame,
11               **kwargs) -> pd.DataFrame:
12        return events_dataframe
13
14 class IsolationForestStrategy(AnomalyDetectionStrategy):
15    def detect(self, data: pd.DataFrame,
16               **kwargs) -> pd.DataFrame:
17        return events_dataframe
18
19 # LSP compliance - both can be used interchangeably
20 detector1 = AnomalyDetector(ZScoreStrategy())
21 detector2 = AnomalyDetector(IsolationForestStrategy())
22
23 # Both work the same way
24 events1 = detector1.detect(data)
25 events2 = detector2.detect(data)

```

3.4 Interface Segregation Principle (ISP)

3.4.1 Definition

Clients should not be forced to depend on interfaces they don't use. Create specific, focused interfaces rather than large, general-purpose ones.

3.4.2 Example: Segregated Interfaces

```

1 # Good: Segregated interfaces (used in project)
2 class IDataLoader(ABC):
3     @abstractmethod
4     def load_data(self): pass
5
6 class IRepository(ABC):
7     @abstractmethod
8     def save_data(self): pass
9
10 class IProgressObserver(ABC):
11     @abstractmethod
12     def update_progress(self, step: str,
13                          progress: float, message: str) -> None:
14         pass
15
16 class IAnomalyDetector(ABC):
17     @abstractmethod
18     def detect(self, data: pd.DataFrame) -> pd.DataFrame:
19         pass

```

3.5 Dependency Inversion Principle (DIP)

3.5.1 Definition

High-level modules should not depend on low-level modules. Both should depend on abstractions. Depend on abstractions, not concretions.

3.5.2 Example: Dependency Injection

```
1 # Good: Depends on abstractions (used in project)
2 class QuantitativeAnalysisPipeline(BaseAnalysisPipeline):
3     def __init__(self, config: Dict[str, Any]):
4         # Dependencies created through factories (abstractions)
5         self.data_loader = DataLoader(YFinanceDataStrategy())
6         self.anomaly_detector = AnomalyDetector(ZScoreStrategy())
7         # Or through factory methods
8         self.news_retriever = NewsRetrieverFactory.create_retriever(
9             'finnhub', api_key)
```

Chapter 4

Design Patterns Implementation

4.1 Strategy Pattern

4.1.1 Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Allows runtime selection of algorithms.

4.1.2 Implementation Areas

The project extensively uses Strategy pattern for:

- Data loading (YFinanceDataStrategy, AlphaVantageDataStrategy)
- Anomaly detection (ZScoreStrategy, IsolationForestStrategy)
- News retrieval (FinnhubNewsStrategy, YahooNewsStrategy)
- Embedding generation (SentenceTransformerStrategy, OpenAIEmbeddingStrategy)
- Similarity analysis (FAISSStrategy, CosineSimilarityStrategy)
- AI explanations (GroqExplainerStrategy, OpenAIExplainerStrategy)

4.1.3 Example Code

```
1 # Strategy interface
2 class AnomalyDetectionStrategy(ABC):
3     @abstractmethod
4     def detect(self, data: pd.DataFrame,
5                **kwargs) -> pd.DataFrame:
6         pass
7
8 # Concrete strategies
9 class ZScoreStrategy(AnomalyDetectionStrategy):
10    def detect(self, data: pd.DataFrame,
11               **kwargs) -> pd.DataFrame:
12        events = []
13        for idx, row in data.iterrows():
14            if abs(row['Z_score']) > self.z_threshold:
```

```

15         events.append({
16             'Date': idx,
17             'Event_Type': 'Positive Outlier',
18             if row['Z_score'] > 0
19             else 'Negative Outlier',
20         })
21     return pd.DataFrame(events)
22
23 # Context class
24 class AnomalyDetector:
25     def __init__(self, strategy: AnomalyDetectionStrategy):
26         self.strategy = strategy
27
28     def detect(self, data: pd.DataFrame,
29                **kwargs) -> pd.DataFrame:
30         return self.strategy.detect(data, **kwargs)

```

4.2 Factory Pattern

4.2.1 Definition

Create objects without specifying their exact class. Encapsulates object creation logic.

4.2.2 Example: News Retriever Factory

```

1 class NewsRetrieverFactory:
2     @staticmethod
3     def create_retriever(service: str,
4                           api_key: str = None) -> NewsRetriever:
5         if service.lower() == 'finnhub':
6             strategy = FinnhubNewsStrategy(api_key)
7         elif service.lower() == 'yahoo':
8             strategy = YahooNewsStrategy()
9         else:
10             raise ValueError(f"Unknown service: {service}")
11
12     return NewsRetriever(strategy)
13
14 # Usage
15 retriever = NewsRetrieverFactory.create_retriever(
16     'finnhub', api_key='your_key')

```

4.3 Observer Pattern

4.3.1 Definition

Define a one-to-many dependency between objects so that when one object changes state, all dependents are notified.

4.3.2 Example: Progress Observer

```

1 # Observer interface
2 class IProgressObserver(ABC):
3     @abstractmethod
4     def update_progress(self, step: str, progress: float,
5                         message: str) -> None:
6         pass
7
8 # Subject (Observable)
9 class BaseProcessor(ABC):
10    def __init__(self, name: str):
11        self._observers: List[IProgressObserver] = []
12
13    def add_observer(self,
14                      observer: IProgressObserver) -> None:
15        self._observers.append(observer)
16
17    def notify_progress(self, step: str, progress: float,
18                        message: str) -> None:
19        for observer in self._observers:
20            observer.update_progress(step, progress, message)
21
22 # Usage
23 processor = DataLoader(YFinanceDataStrategy())
24 processor.add_observer(ConsoleProgressObserver())
25 processor.notify_progress("data_loading", 50.0,
26                           "Loading market data...")

```

4.4 Template Method Pattern

4.4.1 Definition

Define the skeleton of an algorithm in a base class, letting subclasses override specific steps.

4.4.2 Example: Pipeline Template Method

```

1 # Abstract base class with template method
2 class BaseAnalysisPipeline(ABC):
3     def run(self, **kwargs) -> Dict[str, Any]:
4         """Template method - defines algorithm structure"""
5         self.logger.info(f"Starting {self.name} pipeline")
6         self.notify_progress("start", 0.0,
7                             f"Starting {self.name}")
8
9         # Template method steps
10        self._preprocess(**kwargs)
11        self._analyze(**kwargs)
12        self._postprocess(**kwargs)
13
14        self.notify_progress("complete", 100.0,
15                            f"{self.name} completed")
16        return self._results
17
18    @abstractmethod

```

```

19     def _preprocess(self, **kwargs) -> None:
20         pass
21
22     @abstractmethod
23     def _analyze(self, **kwargs) -> None:
24         pass
25
26     @abstractmethod
27     def _postprocess(self, **kwargs) -> None:
28         pass

```

4.5 Builder Pattern

4.5.1 Definition

Construct complex objects step by step. Provides flexibility in object construction and improves readability.

4.5.2 Example: Pipeline Builder

```

1 class PipelineBuilder:
2     def __init__(self):
3         self.config = {}
4         self.observers = []
5
6     def with_ticker(self, ticker: str) -> 'PipelineBuilder':
7         self.config['ticker'] = ticker
8         return self
9
10    def with_benchmark(self,
11                        benchmark: str) -> 'PipelineBuilder':
12        self.config['benchmark'] = benchmark
13        return self
14
15    def build(self) -> QuantitativeAnalysisPipeline:
16        return PipelineFactory.create_quantitative_pipeline(
17            self.config, self.observers)
18
19 # Usage - fluent interface
20 pipeline = (PipelineBuilder()
21             .with_ticker('TSLA')
22             .with_benchmark('SPY')
23             .with_anomaly_detection(z_threshold=3.0)
24             .build())

```

4.6 Repository Pattern

4.6.1 Definition

Encapsulate data access logic and provide a uniform interface. Separates data access logic from business logic.

4.6.2 Example: Data Repository

```
1 # Repository interface
2 class IRepository(ABC):
3     @abstractmethod
4         def save_events(self, events: pd.DataFrame,
5                         filename: str) -> None:
6             pass
7
8     @abstractmethod
9         def load_events(self, filename: str) -> pd.DataFrame:
10            pass
11
12 # Concrete implementation
13 class BaseDataRepository(IRepository):
14     def __init__(self, base_path: str = "results"):
15         self.base_path = base_path
16         os.makedirs(base_path, exist_ok=True)
17
18     def save_events(self, events: pd.DataFrame,
19                     filename: str) -> None:
20         filepath = os.path.join(self.base_path, filename)
21         events.to_parquet(filepath)
```

Chapter 5

Architecture Overview

5.1 Layered Architecture

The project follows a layered architecture with clear separation of concerns:

Layer 1: Presentation Layer UI Components (Streamlit app, console interface), User interaction handling, Progress display and feedback

Layer 2: Business Logic Layer Pipeline orchestration (Template Method), Component coordination, Workflow management

Layer 3: Processing Layer Data processors (Strategy Pattern), Algorithm implementations, Business logic execution

Layer 4: Data Access Layer Data repositories (Repository Pattern), Data persistence, Data retrieval

Layer 5: Infrastructure Layer External API integrations, File system operations, Logging and error handling

5.2 UML Diagrams

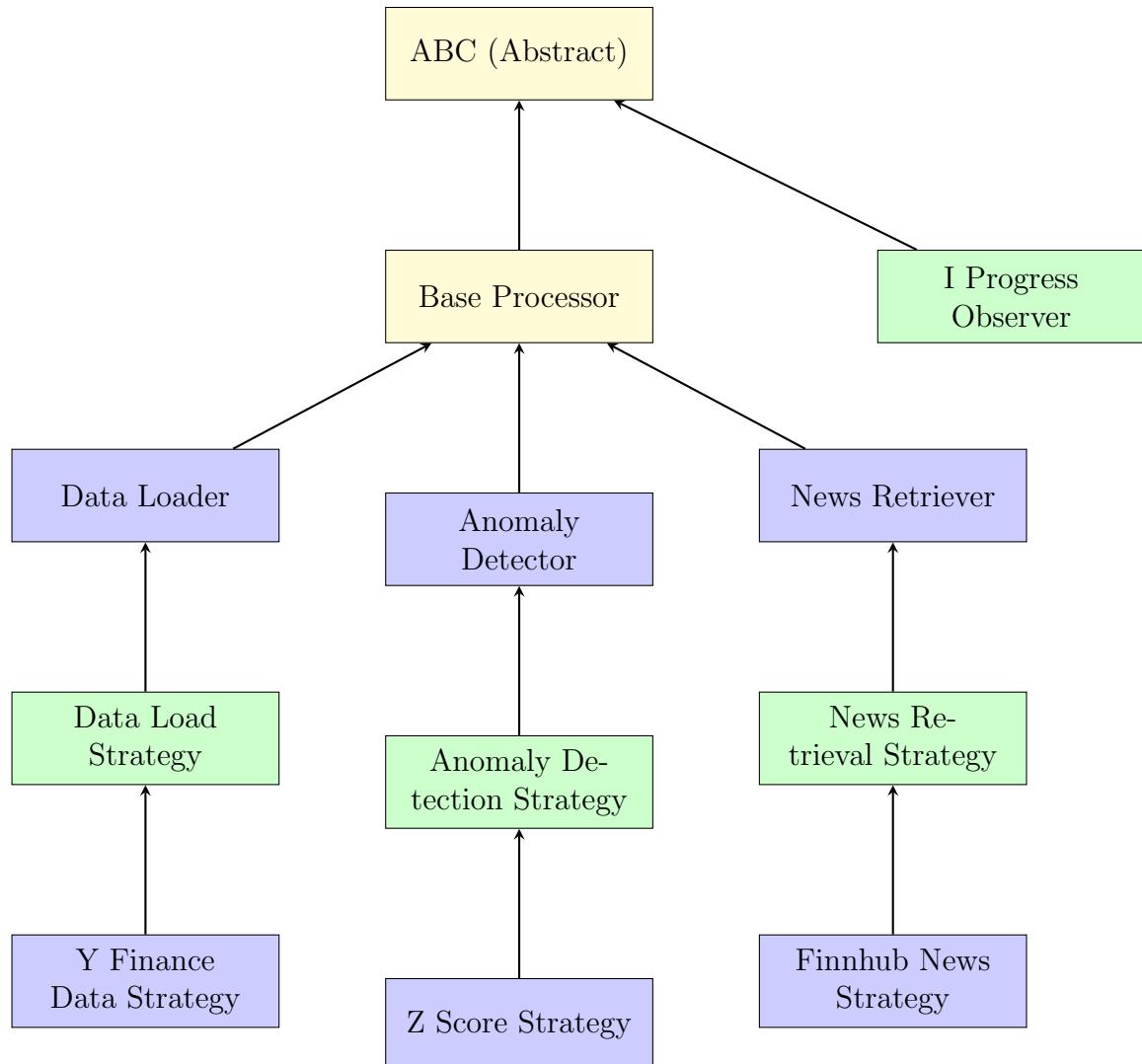


Figure 5.1: Core Processor Hierarchy - Inheritance and Strategy Pattern

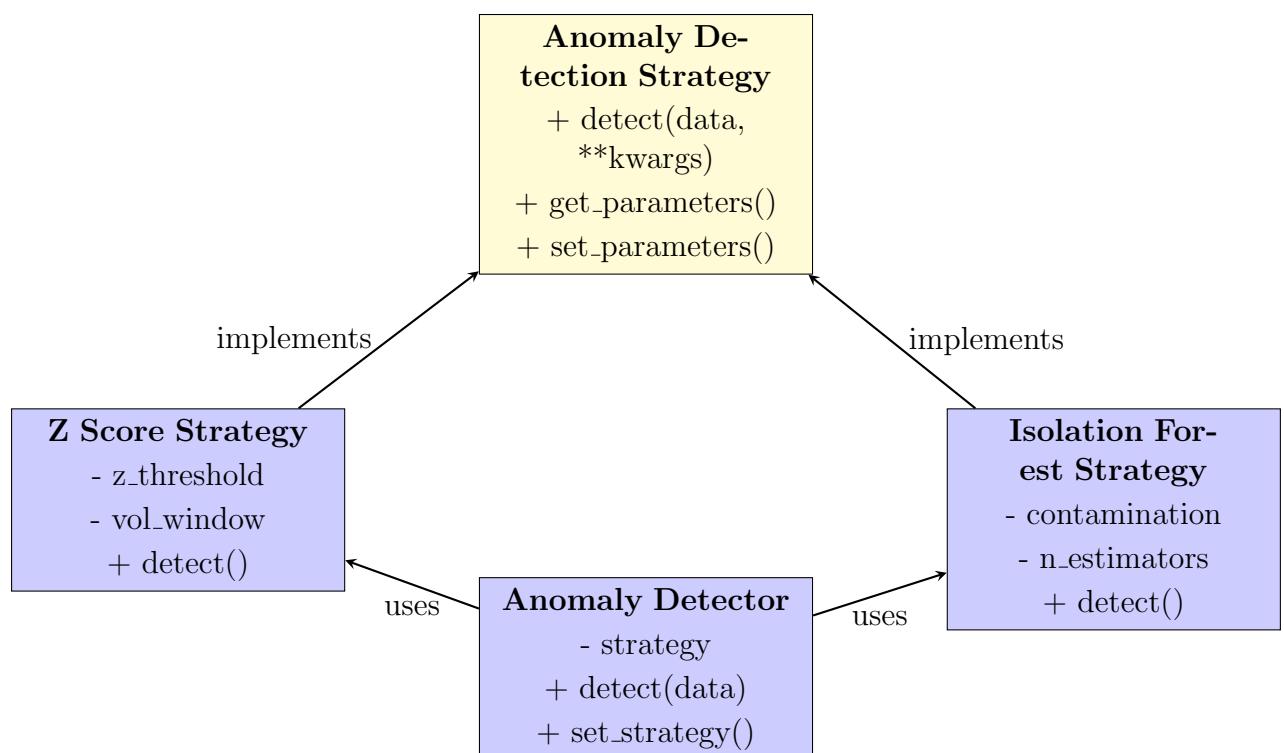


Figure 5.2: Strategy Pattern for Anomaly Detection

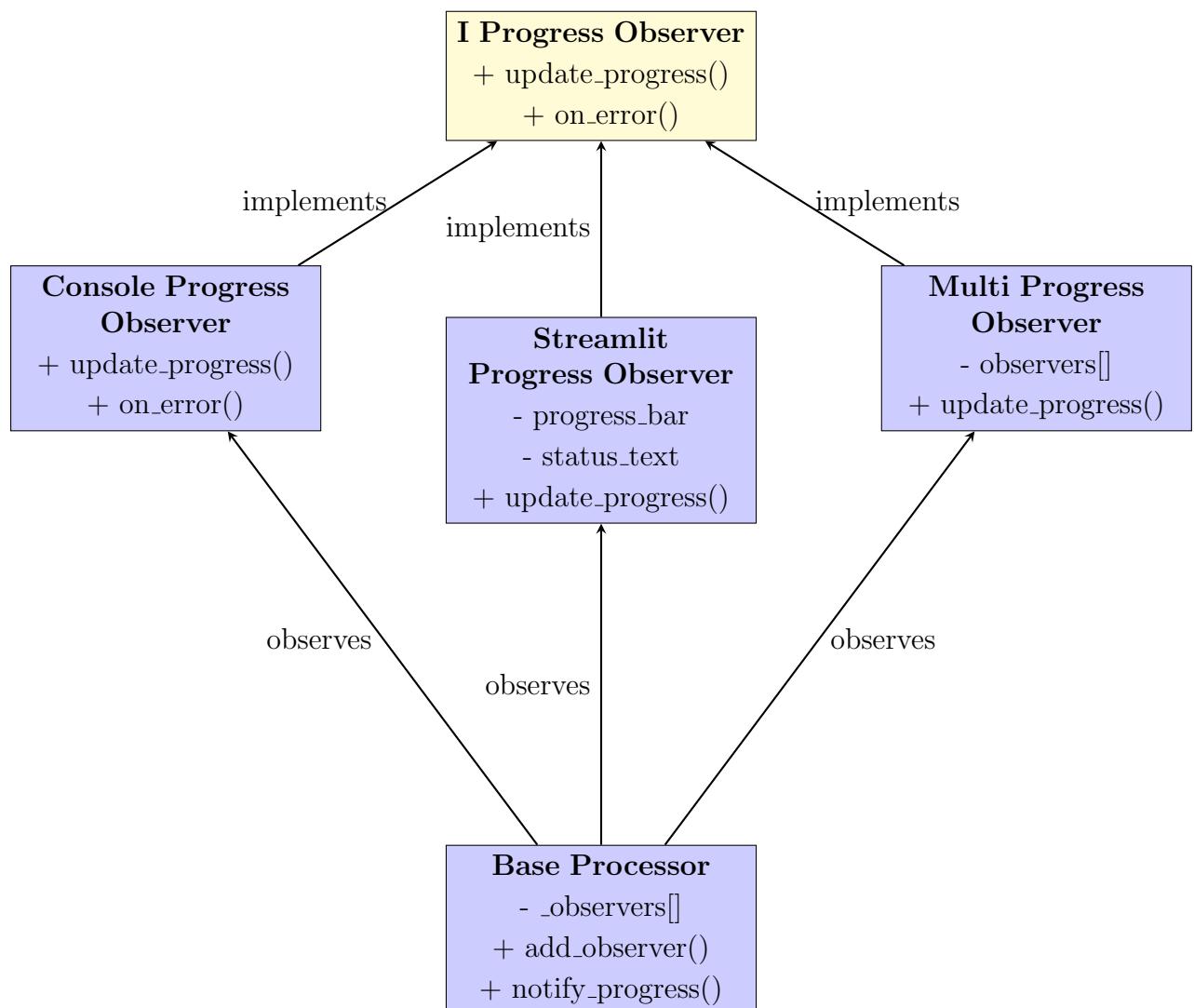


Figure 5.3: Observer Pattern for Progress Tracking

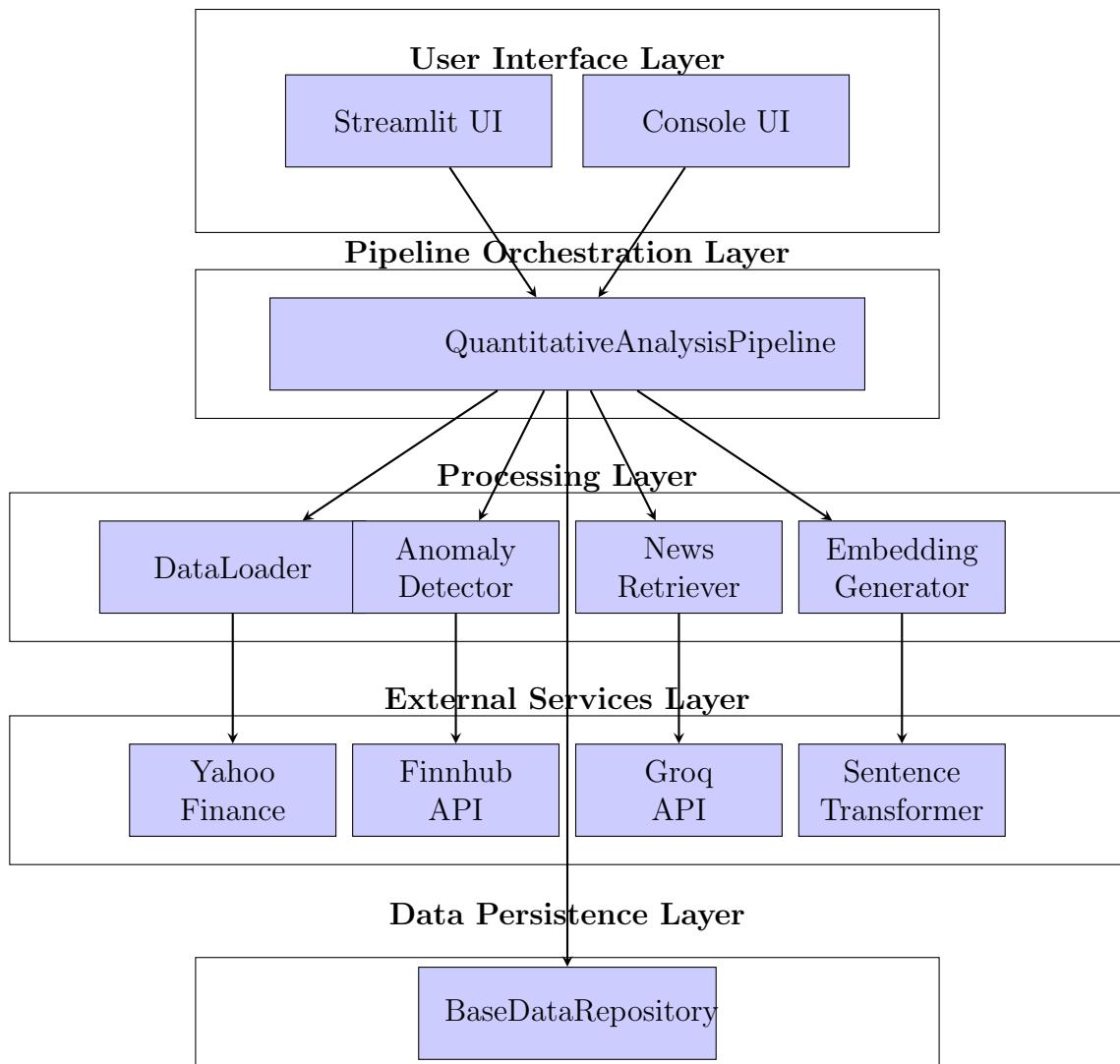


Figure 5.4: System Component Architecture

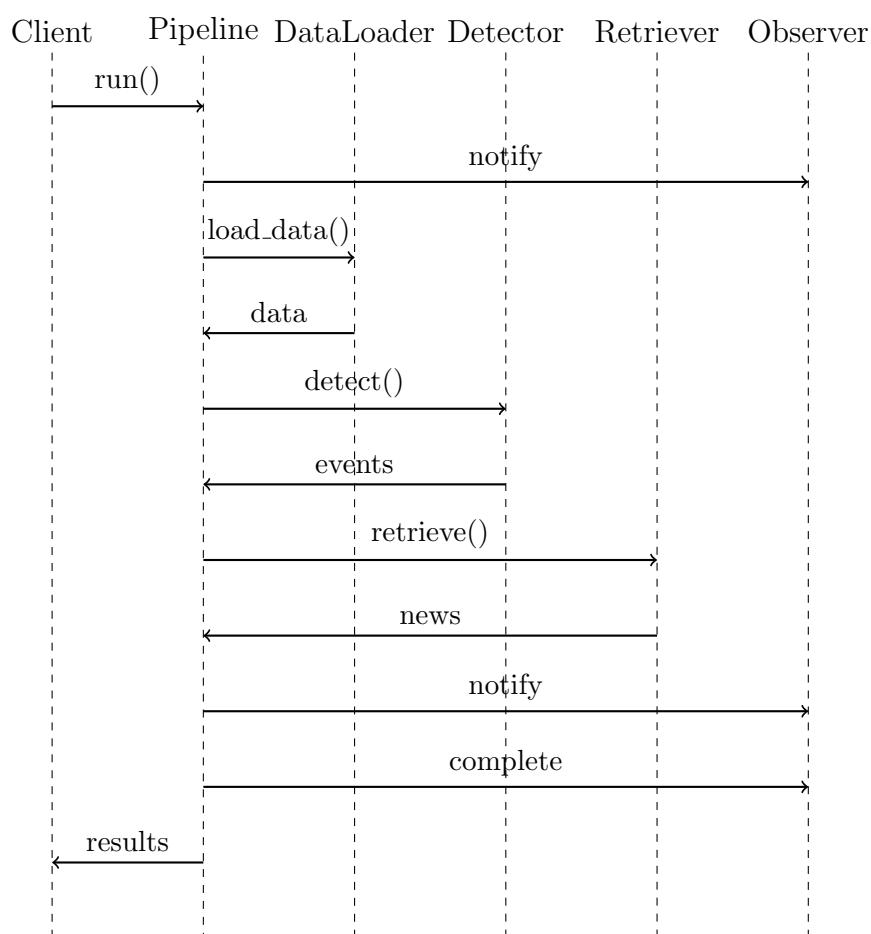


Figure 5.5: Sequence Diagram - Pipeline Execution Flow

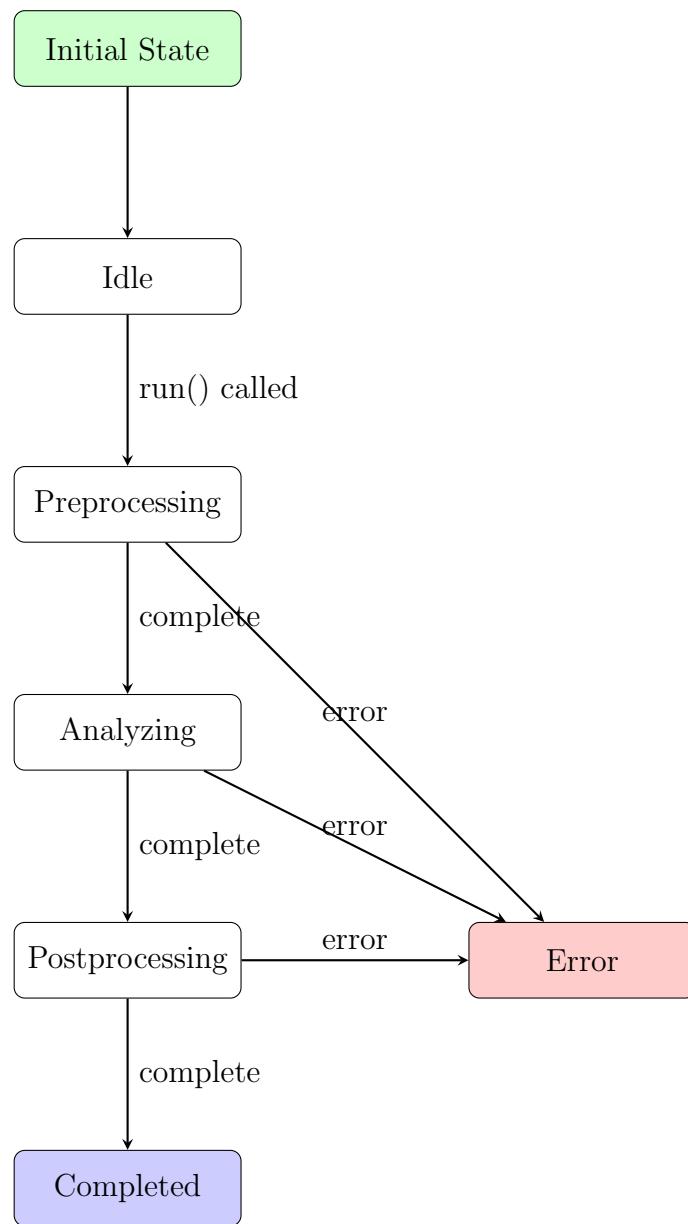
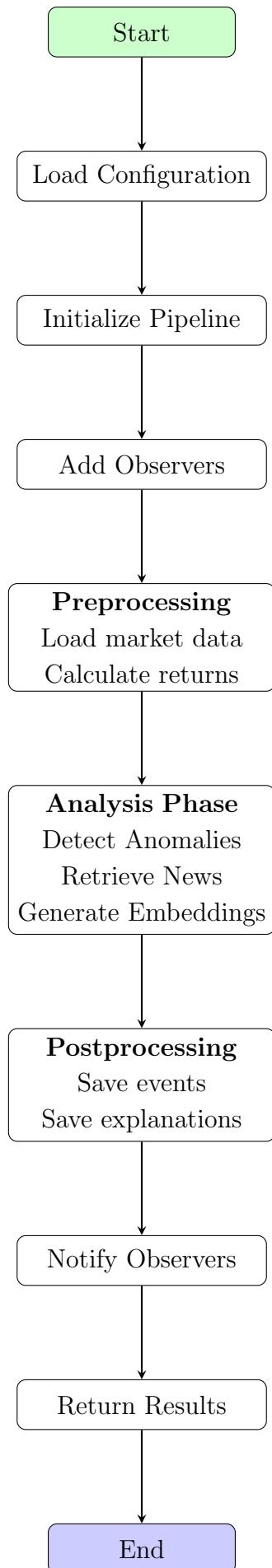


Figure 5.6: State Diagram - Pipeline Execution States



Chapter 6

Code Examples and Demonstrations

6.1 Complete Pipeline Execution

```
1 from pipeline.pipeline_factory import PipelineBuilder
2 from pipeline.progress_observer import ConsoleProgressObserver
3
4 # Using Builder pattern
5 pipeline = (PipelineBuilder()
6     .with_ticker('TSLA')
7     .with_benchmark('SPY')
8     .with_date_range('2024-01-01', '2024-12-31')
9     .with_anomaly_detection(z_threshold=3.0, vol_window=15)
10    .with_news_service('finnhub', api_key='your_key')
11    .with_ai_service('groq', api_key='your_key')
12    .with_console_observer()
13    .build())
14
15 # Run analysis
16 results = pipeline.run()
```

6.2 Custom Strategy Implementation

```
1 from processors.anomaly_detector import (
2     AnomalyDetectionStrategy, AnomalyDetector
3 )
4
5 class CustomAnomalyStrategy(AnomalyDetectionStrategy):
6     def detect(self, data: pd.DataFrame,
7                **kwargs) -> pd.DataFrame:
8         # Custom anomaly detection logic
9         events = []
10        # ...
11        return pd.DataFrame(events)
12
13    def get_parameters(self) -> Dict[str, Any]:
14        return {'custom_param': 'value'}
15
16    def set_parameters(self, **kwargs) -> None:
17        # Set custom parameters
```

```
18     pass
19
20 # Use custom strategy
21 detector = AnomalyDetector(CustomAnomalyStrategy())
22 events = detector.detect(data)
```

6.3 Multiple Observers

```
1 from pipeline.progress_observer import (
2     ConsoleProgressObserver,
3     StreamlitProgressObserver,
4     MultiProgressObserver
5 )
6
7 # Create multiple observers
8 console_observer = ConsoleProgressObserver()
9 streamlit_observer = StreamlitProgressObserver(
10     progress_bar, status_text
11 )
12
13 # Create multi-observer
14 multi_observer = MultiProgressObserver([
15     console_observer,
16     streamlit_observer
17 ])
18
19 # Add to pipeline
20 pipeline.add_observer(multi_observer)
```

Chapter 7

Benefits and Best Practices

7.1 Benefits of OOP in This Project

1. **Maintainability:** Easy to modify and extend individual components
2. **Reusability:** Components can be reused in different contexts
3. **Testability:** Each component can be tested in isolation
4. **Flexibility:** Easy to swap implementations using interfaces
5. **Scalability:** New features can be added without modifying existing code
6. **Readability:** Code structure mirrors real-world concepts
7. **Type Safety:** Type hints and interfaces ensure correct usage
8. **Documentation:** Clear interfaces serve as documentation

7.2 Best Practices Demonstrated

1. **Use Interfaces for Contracts:** Define clear contracts between components
2. **Favor Composition over Inheritance:** Use object composition for flexibility
3. **Single Responsibility:** Each class has one clear purpose
4. **Dependency Injection:** Inject dependencies rather than creating them
5. **Error Handling:** Comprehensive exception hierarchy
6. **Logging:** Proper logging throughout the application
7. **Type Hints:** Use type hints for better code documentation
8. **Documentation:** Clear documentation for all public methods
9. **Testing:** Design for testability
10. **SOLID Principles:** All SOLID principles properly applied

7.3 Design Pattern Benefits Summary

| Pattern | Benefits |
|-----------------|---|
| Strategy | Runtime algorithm selection, easy to add new algorithms, separates logic from client code |
| Factory | Encapsulates object creation, provides flexibility, reduces coupling |
| Observer | Loose coupling, supports multiple observers, event-driven architecture |
| Template Method | Defines common algorithm structure, allows customization, promotes code reuse |
| Builder | Step-by-step construction, fluent interface, flexible configuration |
| Repository | Abstracts data access, uniform interface, easy to swap storage |

Table 7.1: Design Pattern Benefits Summary

Chapter 8

Conclusion

This project successfully demonstrates:

8.1 Comprehensive OOP Implementation

- ✓ **Encapsulation:** Data and methods bundled with controlled access
- ✓ **Inheritance:** Code reuse through class hierarchies
- ✓ **Polymorphism:** Runtime behavior selection through interfaces
- ✓ **Abstraction:** Complex implementation hidden behind clear contracts

8.2 SOLID Principles

- ✓ **Single Responsibility Principle:** Each class has one responsibility
- ✓ **Open/Closed Principle:** Open for extension, closed for modification
- ✓ **Liskov Substitution Principle:** Subclasses honor base class contracts
- ✓ **Interface Segregation Principle:** Focused, specific interfaces
- ✓ **Dependency Inversion Principle:** Depends on abstractions, not concretions

8.3 Design Patterns

- ✓ **Strategy Pattern:** 6+ implementations across different domains
- ✓ **Factory Pattern:** 4+ factories for object creation
- ✓ **Observer Pattern:** Comprehensive progress tracking system
- ✓ **Template Method Pattern:** Structured pipeline execution
- ✓ **Builder Pattern:** Fluent interface for configuration
- ✓ **Repository Pattern:** Clean data persistence layer

8.4 Professional Software Engineering

- ✓ Clean, maintainable, and extensible code
- ✓ Well-structured, layered architecture
- ✓ Industry-standard coding practices
- ✓ Comprehensive error handling and logging

The project serves as an excellent example of how Object-Oriented Programming principles and design patterns can be applied to create robust, maintainable, and extensible software systems. The implementation demonstrates deep understanding of OOP concepts and their practical application in solving real-world problems.
