

The impact of using a contract-driven, test-interceptor based software development approach*

Justus Posthuma
Stellenbosch University
Stellenbosch, Cape Town
justus.posthuma@gmail.com

Fritz Solms
Stellenbosch University
Stellenbosch, Cape Town
fritz@solms.co.za

Bruce W. Watson
Stellenbosch University
Stellenbosch, Cape Town
bruce@bruce-watson.com

ABSTRACT

A contract-driven development approach requires the formalization of component requirements in the form of a component contract. The Use Case, Responsibility Driven Analysis and Design (URDAD) methodology is based on the contract-driven development approach and uses contracts to capture user requirements and perform a technology-neutral design across layers of granularity. This is achieved by taking use-case based functional requirements through an iterative design process and generating various levels of granularity iteratively.

In this project, the component contracts that were captured by utilizing the URDAD approach are used to generate test interceptors which validate whether, in the context of rendering component services, the component contracts are satisfied. To achieve this, Java classes and interfaces are annotated with pre- and postconditions to represent the contracts in code. Annotation processors are then used to automatically generate test-interceptor classes by processing the annotations. The test-interceptor classes encapsulate test-logic and are interface-compatible with their underlying component counterparts. This enable test-interceptors to intercept service requests to the underlying counterpart components in order to verify contract adherence. The generated test interceptors can be used for unit testing as well as real-time component monitoring. This development approach, utilized within the URDAD methodology would then result in unit and integration tests across levels of granularity.

Empirical data from actual software development projects will be used to assess the impact of introducing such a development approach in real software development projects. In particular, the study assesses the impact on the quality attributes of the software development process, as well as the qualities of the software produced by the process.

Process qualities measured include development productivity (the rate at which software is produced), correctness (the rate at which the produced software meets the clients requirements) and the certifiability of the software development process (which certifiability requirements are fully or partially addressed by the URDAD development approach). Software qualities measured include reusability

(empirical and qualitative), simplicity (the inverse of the complexity measure) and bug density (number of defects in a module).

The study aims to show conclusively how the approach impacts the creation of correct software which meets the client requirements, how productivity is affected and if the approach enhances or hinders certifiability. The study also aims to determine if test-interceptors are a viable mechanism to produce high-quality tests that contribute to the creation of correct software. Furthermore, the study aims to determine if the software produced by applying this approach yield improved reusability or not, if the software becomes more or less complex and if more or less bugs are induced.

CCS CONCEPTS

• Software and its engineering → Software development methods; Reusability; Requirements analysis; Software implementation planning; Error handling and recovery;

KEYWORDS

Contract Driven Development, Test-Interceptors, URDAD

ACM Reference Format:

Justus Posthuma, Fritz Solms, and Bruce W. Watson. 2018. The impact of using a contract-driven, test-interceptor based software development approach. In *Proceedings of Technology for Change (SAICSIT 2018)*. ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Contract Driven Development is a design methodology that aims to produce correct software through formalized requirements in the form of component contracts which include the specification of preconditions, postconditions and invariants, also referred to as contracts [6]. The benefits of Contract Driven Development have been demonstrated [4] and it consequently forms the basis of the Use Case, Responsibility Driven Analysis and Design (URDAD) methodology [9, 10]. The URDAD methodology uses contracts to capture user requirements and perform a technology-neutral design across layers of granularity [10]. These component contracts can be used to generate annotated interfaces (for example Java interfaces with pre- and post-condition annotations) which represent the contracts in code. Using these encoded component contracts, test-interceptors are generated. The test-interceptors contain test logic and are used for unit and integration tests, as well as for self-testing components. In the case of unit tests, the unit test developer only needs to specify the test data and call the relevant component via its test-interceptor.

*Masters project

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored.

For all other uses, contact the owner/author(s).

SAICSIT 2018, September 2018, Port Elizabeth, South Africa

© 2018 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06.

https://doi.org/10.475/123_4

This study assesses the impact on the quality attributes of the software development process, as well as the qualities of the software produced by the process. Process qualities measured include development productivity (the rate at which software is produced), correctness (the rate at which the produced software meets the clients requirements) and the certifiability of the software development process. Software qualities measured include reusability, simplicity (the inverse of the complexity measure) and bug density.

2 THE QUALITIES OF THE SOFTWARE DEVELOPMENT PROCESS

2.1 Development Productivity

There are a number of varying metrics that are used to determine productivity. In the context of this project, the following metrics are considered:

- * The rate at which software is produced.
- * Costs to implement.
- * Time to implement.

2.2 Correctness of the produced software

This refers to the rate at which the produced software meets the client requirements and entails producing correct results and handling exceptions properly. It can be measured by counting defects over a period of time with bug-tracking software.

2.3 Certifiability

Will the utilization of the process yield results that adhere to established industry standards, for example IEC 61508 (industrial controls), IEC 62034 (medical), ISO 26262 (automotive), IEC 60880 (nuclear energy), DO-178B (airborne) or EN 50128 (rail transportation)? The industry standard that is relevant to the test environment must be examined to determine which requirements are partially or fully addressed by utilizing a contract-driven development approach.

2.4 Test Quality

This entails the quality of the testing itself, which include the number of tests and the level of detail of tests that can be automatically generated. A high number of test failures may be indicative of good test quality: by quantitatively assessing the number of errors which were not captured by the tests and removing the percentage which is due to incorrect requirements, we are left with only those failures which were in the code but not exposed by tests.

3 THE QUALITIES OF THE SOFTWARE PRODUCED BY UTILIZING THE PROCESS

3.1 Reusability

There are two basic types of methods to measure reusability: **empirical** and **qualitative**. Empirical methods depend on objective data which can be calculated by a tool or analyst automatically and cheaply. Qualitative methods rely on a subjective value as to how well the software adheres to certain guidelines or principles and requires manual effort. [8]. Metrics such as program size, program structure (low coupling), documentation (indicated subjectively on

a scale of 1 to 10), programming language and reuse experience can be used to quantify reusability.

3.2 Simplicity

This quality is the inverse of the complexity measure. Although many methods exist to measure software complexity, most use the following metrics: **Cyclomatic Complexity** which measures how much control flow exists in a program, **Halstead Volume** which measures how much information is in the source code (how many variables are used and how often they are used), and **Maintainability Index** which formulates an overall score of how maintainable a program is [2, 3].

3.3 Bug Density

Bug density is usually defined as defects per thousand lines of code (KLOC) and measured by dividing the size of the module by the number of confirmed defects.

4 IMPLEMENTATION AND MEASUREMENTS

The Use-Case, Responsibility Driven Analysis and Design (URDAD) process focuses on identifying and assigning responsibilities in the early stages of the design. For each of these responsibilities, there should be a *contract* which all service providers (components) that realizes the responsibility, must adhere to. The functional aspects of a contract are defined by an interface with pre- and post-conditions on the services and, if required, invariance constraints on the service provider. [9]

Java interfaces are annotated with pre- and postconditions (and invariants) which represent component contracts. Current annotation frameworks are assessed and selected based on the functionality they provide in order to develop annotation processors that can automatically generate test-interceptor classes by using the annotated pre- and postconditions. Either existing frameworks can be extended, or a new contract annotation framework can be defined if no existing frameworks offer the required functionality.

The test-interceptor classes encapsulate test-logic and are interface-compatible with their underlying component counterparts. This enable test-interceptors to intercept service requests to the underlying counterpart components in order to verify contract adherence by:

- (1) Assessing the truth value of each precondition on service request.
- (2) Delegating the request to the underlying component for processing.
- (3) Upon return the test interceptor verifies:
 - (a) if an exception was generated by the underlying component, verify that the associated precondition did not hold via a contract violation exception which is an error.
 - (b) if the service was provided and returned (no exception) verify that all preconditions were met.
 - (c) if the service was provided (no exceptions) verify that all post-conditions hold true.

The following code illustrates a Java interface with a method that is annotated with preconditions and postconditions:

```

1 public interface Discount_interface
2 {
3     @Precondition(constraint = "isIDValid(ID) == true",
4         raises = InvalidArgumentException.class)
5     @Precondition(constraint = "isSupportedPartner(
6         RewardsPartner) == true", raises =
7         InvalidArgumentException.class)
8     @Precondition(constraint = "itemPrice > 0", raises =
9         InvalidStateException.class)
10
11     @Postcondition(constraint = "returnValue > 0")
12     @Postcondition(constraint = "getCounter() == //
13         getInstanceCounter() + 1//")
14     public double getDiscountPercentage(String ID, String
15         RewardsPartner, double itemPrice) throws Exception;
16 }
    
```

The interceptor class that is generated from the annotations, look as follows:

```

1 public class Discount_TestInterceptor implements
2     Discount_interface
3 {
4     private Discount_interface counterpart = null;
5
6     public Discount_TestInterceptor(Discount_interface
7         counterpart)
8     {
9         this.counterpart = counterpart;
10    }
11
12    @Override
13    public double getDiscountPercentage(String ID, String
14        RewardsPartner, double itemPrice) throws Exception
15    {
16        /*
17         * Evaluate the preconditions
18         */
19        boolean _pre_1 = isIDValid(ID) == true;
20        boolean _pre_2 = isSupportedPartner(RewardsPartner)
21            == true;
22        boolean _pre_3 = itemPrice > 0;
23
24        /*
25         * Evaluate the pre-assessments
26         */
27        int _preAs_1 = getInstanceCounter() + 1;
28
29        double returnValue = 0;
30        try
31        {
32            /*
33             * Call the wrapped method of the counterpart
34             * component to be tested
35             */
36            returnValue = counterpart.getDiscountPercentage(ID,
37                RewardsPartner, itemPrice);
38
39            /*
40             * If method was implemented correctly, it would have
41             * thrown exceptions if any of the preconditions were
42             * false
43             */
44
45            if (!_pre_1)
46            {
47                /*
48                 * The method above did not throw an exception but
49                 * precondition 1 is false. This means there is a
50                 * problem with the implementation of the method -
51                 * it does not properly enforce precondition 1
52                 */
53                throw new PreconditionNotEnforcedException("
54                    isIDValid(ID) == true", "getDiscountPercentage()");
55            }
56            if (!_pre_2)
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
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
    
```

```

51 {
52     /*
53      * The method above did not throw an exception but
54      * precondition 2 is false. This means there is a
55      * problem with the implementation of the method -
56      * it does not properly enforce precondition 2
57      */
58     throw new PreconditionNotEnforcedException("
59         isSupportedPartner(RewardsPartner) == true", "
60         getDiscountPercentage()");
61 }
62 if (!_pre_3)
63 {
64     /*
65      * The method above did not throw an exception but
66      * precondition 3 is false. This means there is a
67      * problem with the implementation of the method -
68      * it does not properly enforce precondition 3
69      */
70     throw new PreconditionNotEnforcedException("
71         isSupportedPartner(RewardsPartner) == true", "
72         getDiscountPercentage()");
73 }
74 /*
75  * Evaluate post conditions and their pre-assessments
76  */
77 if (!(returnValue > 0))
78 {
79     throw new PostconditionNotMetException("
80         returnValue > 0", "getDiscountPercentage()");
81 }
82 if (!(getCounter() == _preAs_1))
83 {
84     throw new PostconditionNotMetException("
85         getCounter() == //getInstanceCounter() + 1//", "
86         getDiscountPercentage()");
87 }
88 /*
89  * The wrapped method threw an exception
90  */
91 catch (InvalidArgumentException iae)
92 {
93     if (!_pre_1)
94     {
95         /*
96          * Precondition 1 is false, so the method was
97          * supposed to throw this exception. This
98          * means it implemented the check for this
99          * precondition correctly. Rethrow the valid
100          * exception.
101          */
102         throw iae;
103     }
104     if (!_pre_2)
105     {
106         /*
107          * Precondition 2 is false, so the method was
108          * supposed to throw this exception. This
109          * means it implemented the check for this
110          * precondition correctly. Rethrow the valid
111          * exception.
112          */
113         throw iae;
114     }
115 }
116 /*
117  * If this point is reached, it means that
118  * the preconditions are valid, but the
119  * method still threw an exception. So
120  * there is a problem with the implementation
121  * of the method.
122  */
123 }
    
```

```

121     throw new
122     PreconditionsHoldButServiceRefusedException("
123     InvalidArgumentException", "getDiscountPercentage()")
124     };
125     }
126     catch (InvalidStateException ise)
127     {
128         if (!_pre_3)
129         {
130             /*
131             Precondition 3 is false, so the method was
132             supposed to throw this exception. This
133             means it implemented the check for this
134             precondition correctly. Rethrow the valid
135             exception.
136             */
137             throw ise;
138         }
139         /*
140         If this point is reached, it means that the
141         preconditions are valid but the method still
142         threw an exception. So there is a problem with
143         the implementation of the method
144         */
145         throw new
146         PreconditionsHoldButServiceRefusedException("
147         InvalidStateException", "getDiscountPercentage()");
148     }
149     /*
150     All preconditions and postconditions are true
151     */
152     return returnValue;
153 }

```

The test-interceptors can now be used in the following scenarios:

- (1) In a live system, components can be wrapped by test-interceptors which assess on the live system whether the wrapped component functions in accordance with the contract.
- (2) Unit tests would dependency inject mock objects for any dependencies of the component under test and specify test data based on an analysis of equivalence partitions and boundary values.
- (3) Integration tests would dependency inject actual objects used for their dependencies and potentially use the same test data as the unit tests to the component.

This development approach, utilized within the URDAD methodology would then result in unit and integration tests across levels of granularity.

4.1 Empirical measurements of the qualities of the software development process

Development productivity can potentially be measured by comparing planned productivity to actual productivity when using Design By Contract with Test Interceptors. Project management tools usually contain enough information to be able to calculate time taken, resources assigned to activities and time taken to complete activities.

Bug-tracking tools (for example BugZilla) can potentially be used to measure turn-around time as well as frequency and quantity of defects over certain time periods. Tools like this can also be used to determine bug density in components.

An industry standard (for example IEC 61508) can be examined to determine if the introduction of a contracts based approach enhances certifiability, and which aspects may actually impede it.

Automated testing tools like JUnit can be used to create tests to quantitatively determine the errors that were not captured by the auto-generated interceptors. This collection of errors will then consist of errors due to incorrect requirements, and errors in the code.

4.2 Empirical measurements of the qualities of the software produced

To empirically measure code reuse, a static code analyzer like PMD could potentially be used. It includes built-in rule-sets (for example to detect duplicate code) and also allows the creation of custom rule sets.

A number of automated tools exist to measure cyclomatic complexity (SourceMeter, Sonarqube), Halstead Volume (Halstead Metrics Tool, JHawk) and the maintainability index (Testwell CMTJava, JHawk) which could potentially be used to measure code simplicity.

5 RELATED WORK

Nebut et al describes a requirement-based testing technique that leverages use-cases in order to generate functional test objectives [7]. Their technique uses declarative formalisms in the form of pre- and post-conditions (contracts) to express dependencies between use-cases, from which a labeled transition system is built to capture all possible valid sequences of use-cases from an initial configuration. The main disadvantage of their technique is that it does not cover integration or unit testing of specific aspects if they are not described at a very high level. They use the pre- and post-conditions of the use-cases to express the system properties that make it applicable, and the properties acquired by the system after its application. The proposed project differs from this approach in that it uses pre- and postconditions throughout levels of granularity in order to determine contract adherence of components, not use-cases. It also has a strong focus on unit and integration testing on all levels. The URDAD methodology with test-interceptors could potentially be used to extend this approach by being applied to the development process used to create the components that would realize the different use-cases.

Jazequel et al have demonstrated that interfaces between software modules should be governed by precise specifications [4], and concludes that Design by Contract is imperative for effective component reuse. This view supports the utilization of Contract Driven Development which this project is based on.

The Use-case, Responsibility Driven Analysis and Design (URDAD) methodology has been formulated to encourage sound design principles and is architecture and technology neutral [9, 10]. This makes it suitable for any test environment.

Belhaouari et al created an experimental platform known as Tamago-Test for software analysis and automated testing [1]. They focus on the automation of test-case generation from specifications written as Design by Contract and rely on First-order logic assertions to express contracts between components in the generation of test-cases. Their approach is very similar to the Contract Driven Design with Test Interceptors approach, but differs in that they use an abstract language with logic assertions to specify the contracts instead of annotations. This could potentially be a more powerful mechanism to specify contracts, but a disadvantage might be that it becomes

much more complex.

Leitner et al recognizes that unit testing is a resource-intensive and time-consuming activity. They created a tool called Cdd which is integrated into EiffelStudio (an Eiffel IDE) that uses contracts in code to extract test cases and run them as background activities during development. This is achieved by exploiting actions that developers perform anyway while writing code [5]. Their approach is interesting in that it uses failures (purposely induced or by mistake) in order to extract test-cases and so doing, build up a comprehensive unit testing suite.

6 CONCLUSION

The study aims to show conclusively how the approach impacts the creation of correct software which meets the client requirements, how productivity is affected and if the approach enhances or hinders certifiability. The study also aims to determine if test-interceptors are a viable mechanism to produce high-quality tests that contribute to the creation of correct software. Furthermore, the study aims to determine if the software produced by applying this approach yield improved reusability or not, if the software becomes more or less complex and if more or less bugs are induced.

REFERENCES

- [1] Hakim Belhaouari and Frederic Peschanski. 2008. Automated Generation of Test Cases from Contract-Oriented Specifications: A CSP-Based Approach. In *HASE '08: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE Computer Society, Washington, DC, USA, 219–228. <https://doi.org/10.1109/HASE.2008.15>
- [2] Jonathau Bloom. 2018. Five Reasons You MUST Measure Software Complexity. (May 2018). <https://www.castsoftware.com/blog/five-reasons-to-measure-software-complexity> [Online; accessed 10. May 2018].
- [3] M Clark, B Salesky, and C Urmsou. 2008. Measuring Software Complexity to Target Risky Modules in Autonomous Vehicle Systems. In *AU-VSI North America Conference (2008-06-11)*. <http://www.mccabe.com/pdf/MeasuringSoftwareComplexityUAV.pdf> [Online; accessed 10. May 2018].
- [4] J.-M. Jazequel and B. Meyer. 1997. Design by contract: the lessons of Ariane. *Computer* 30, 1 (Jan. 1997), 129–130. <https://doi.org/10.1109/2.562936>
- [5] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. 2007. Contract Driven Development = Test Driven Development - Writing Test Cases. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 425–434. <https://doi.org/10.1145/1287624.1287685>
- [6] Bertrand Meyer. 1992. Applying "Design by Contract". *Interactive Software Engineering (1992)*.
- [7] Clémentine Nebul, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. 2003. Requirements by Contracts allow Automated System Testing. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 85.
- [8] Jeffrey Poulin. 2001. Measuring Software Reusability. (01 2001).
- [9] Fritz Solms. [n. d.]. *URDAD for System Design*.
- [10] Fritz Solms and Dawid Loubser. 2010. URDAD as a semi-formal approach to analysis and design. In *Innovatons Syst Softw Eng*.

