



**ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΕΛΟΠΟΝΝΗΣΟΥ**
University of the Peloponnese

**ΣΧΟΛΗ ΟΙΚΟΝΟΜΙΑΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
Π.Μ.Σ. ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΥΠΟΛΟΓΙΣΤΩΝ**

Διπλωματική Εργασία

A SOLID Grasp of C# Design: Principles, Refactoring, and a Case-Driven Approach

Angeliki Freskou

Reg. no.: 2022201802032

Supervisor: Costas Vasilakis

Tripolis, October 2025

Copyright

Copyright © Angeliki Freskou, 2025. All rights reserved.

No part of this thesis may be reproduced, stored, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

Optional University permission clause:

Permission is granted to the University of the Peloponnese to reproduce and distribute this thesis, in whole or in part, for non-commercial, research and educational purposes, provided that the author and source are acknowledged.

Acknowledgements

This work could not have been completed without the guidance and support of many people. Sincere thanks are extended to **Professor Vassilakis** for his confidence in this endeavor and for the clear direction and constructive criticism that shaped the research into its final form. His trust in the project and steady mentorship were invaluable at every stage.

Appreciation is also expressed to the **staff of the Department of Informatics and Telecommunications at the University of the Peloponnese** for the knowledge, resources, and academic environment that enabled steady growth as an engineer and researcher.

Finally, heartfelt thanks go to **family and friends** for their patience, motivation, and encouragement throughout this journey. In particular, deepest gratitude is owed to **my husband, Grigoris Dimitroulakos**, whose unwavering support and understanding sustained me during the most demanding periods. His belief in me provided constant strength and renewed determination to reach this goal.

Abstract

The thesis investigates how the combined application of the SOLID principles and the GRASP patterns can be used to produce robust C# software designs that remain maintainable, testable, and extensible as systems evolve. A unified, principle-to-practice framework is presented in which SOLID secures class- and component-level integrity while GRASP structures responsibilities and collaborations at the architectural level. The approach is operationalized in the .NET ecosystem through idiomatic refactorings—interface extraction, role-focused interface segregation, strategy-based composition, and dependency inversion realized via dependency injection—so that variation can be absorbed at stable seams owned by high-level policy. To move beyond stylistic guidance, the work employs metric-guided evaluation and a cohesive C# case study representative of an e-commerce workflow. Changes are validated using static and change-based indicators (coupling/cohesion proxies, instability indices, files-touched per feature), along with test metrics (coverage and mutation score). The results indicate that dependency boundaries defined and verified at the policy layer localize change, reduce “shotgun surgery,” and enable faster, more reliable testing through mocks and contract tests; extension by addition is favored over risky edits, and architectural refactoring becomes safer. Contributions include: (i) a mapped correspondence between SOLID and GRASP that clarifies when and how to apply each; (ii) a refactoring playbook tied to expected metric movement; (iii) case studies demonstrating end-to-end impact in C#; and (iv) pedagogical scaffolding suitable for studio-style instruction. Limitations are noted regarding external validity across real-time or resource-constrained domains and the use of proxy metrics for maintainability. Future work is outlined for principle-aware analyzers, boundary verification in continuous integration, broader multi-team replication, and curriculum-ready case libraries, with the aim of turning design principles into repeatable, measurable engineering practice.

Περίληψη

Η παρούσα διπλωματική εργασία διερευνά πώς ο συνδυασμός των αρχών SOLID και των προτύπων GRASP μπορεί να χρησιμοποιηθεί για την παραγωγή ανθεκτικών σχεδίων λογισμικού σε C#, τα οποία παραμένουν συντηρήσιμα, ελέγξιμα και επεκτάσιμα καθώς τα συστήματα εξελίσσονται. Παρουσιάζεται ένα ενοποιημένο πλαίσιο από τη θεωρία στην πράξη, στο οποίο οι αρχές SOLID διασφαλίζουν την ακεραιότητα σε επίπεδο κλάσης και συνιστώσας, ενώ τα πρότυπα GRASP οργανώνουν τις ευθύνες και τις συνεργασίες σε αρχιτεκτονικό επίπεδο.

Η προσέγγιση υλοποιείται στο οικοσύστημα .NET μέσω ιδιοτυπικών αναδομήσεων (refactorings), όπως εξαγωγή διεπαφών, διαχωρισμός διεπαφών βάσει ρόλων, σύνθεση μέσω προτύπων στρατηγικής και αντιστροφή εξάρτησης μέσω dependency injection, έτσι ώστε η ποικιλία να απορροφάται σε σταθερά σημεία ελέγχου που ανήκουν στην ανώτερη πολιτική του συστήματος.

Για να υπερβεί το επίπεδο της απλής στυλιστικής καθοδήγησης, η εργασία εφαρμόζει αξιολόγηση με βάση μετρικές και παρουσιάζει μια συνοχή μελέτη περίπτωσης σε C#, αντιπροσωπευτική ενός ηλεκτρονικού εμπορικού σεναρίου (e-commerce workflow). Οι αλλαγές επικυρώνονται μέσω στατικών και δυναμικών δεικτών (πληρεξούσιες μετρικές σύζευξης/συνοχής, δείκτες αστάθειας, αριθμός αρχείων που τροποποιούνται ανά λειτουργία), καθώς και μετρικών δοκιμών (κάλυψη κώδικα και βαθμός μετάλλαξης).

Τα αποτελέσματα δείχνουν ότι τα όρια εξάρτησης, όπως ορίζονται και επαληθεύονται στο επίπεδο πολιτικής, τοπικοποιούν τις αλλαγές, μειώνουν το φαινόμενο της “χειρουργικής με καραμπίνα” (*shotgun surgery*) και επιτρέπουν ταχύτερο και πιο αξιόπιστο έλεγχο μέσω mocks και δοκιμών συμβολαίου (contract tests). η επέκταση μέσω προσθήκης προτιμάται έναντι επικίνδυνων τροποποιήσεων, και η αρχιτεκτονική αναδόμηση καθίσταται ασφαλέστερη.

Οι συνεισφορές περιλαμβάνουν:

- (i) έναν αντιστοιχισμένο χάρτη μεταξύ SOLID και GRASP που αποσαφηνίζει πότε και πώς εφαρμόζεται κάθε αρχή·
- (ii) ένα εγχειρίδιο αναδομήσεων συνδεδεμένο με την αναμενόμενη μεταβολή μετρικών
- (iii) μελέτες περίπτωσης που αποδεικνύουν τον ολιστικό αντίκτυπο σε C# και

- (iv) διδακτικό υλικό κατάλληλο για διδασκαλία τύπου εργαστηρίου (*studio-style instruction*).

Αναγνωρίζονται περιορισμοί όσον αφορά την εξωτερική εγκυρότητα σε τομείς πραγματικού χρόνου ή με περιορισμένους πόρους, καθώς και στη χρήση έμμεσων μετρικών για τη συντηρησιμότητα. Τέλος, προτείνεται μελλοντική εργασία για αναλυτές ευαισθητοποιημένους στις αρχές σχεδίασης, επαλήθευση ορίων σε συνεχή ολοκλήρωση (*continuous integration*), ευρύτερη επαναληψιμότητα σε πολυομαδικά περιβάλλοντα και εκπαιδευτικές βιβλιοθήκες μελετών περίπτωσης, με στόχο τη μετατροπή των αρχών σχεδίασης σε επαναλήψιμη, μετρήσιμη μηχανική πρακτική.

Table of contents

Acknowledgements	1-4
Abstract	1-5
Περίληψη	1-6
Table of contents	1-8
Table of Figures	1-14
Index of Tables	1-22
Chapter 1 Introduction	1-23
1.1 Background and Motivation	1-23
1.2 Research Problem and Rationale	1-25
1.3 Research Objectives	1-26
1.3.1 Theoretical Objectives	1-26
1.3.2 Analytical Objectives	1-26
1.3.3 Practical Objectives	1-26
1.4 Contribution of the Thesis	1-27
1.5 Thesis Structure	1-27
1.6 Significance of the Work	1-29
Chapter 2 Literature Review	2-30
2.1 SOLID Principles	2-30
2.2 GRASP Patterns	2-31
2.3 Synergy of SOLID Principles and GRASP Principles	2-32
2.3.1 Academic Research	2-33
2.3.2 Educational Practices	2-34
2.3.3 Industrial Applications	2-34
2.3.4 Tooling and Case Studies	2-35
Chapter 3 Foundational Mechanisms Behind Dependencies	3-38
3.1 Type of Dependency	3-38

3.1.1	Inheritance (Is-a Relationship)	3-39
3.1.2	Association (Has-a Relationship)	3-41
3.1.3	Composition (Whole-Part Ownership)	3-43
3.1.4	Delegation (Object-to-Object Indirection)	3-46
3.1.5	Dependency Through Parameters (Transient Association)	3-48
3.1.6	Dynamic and Reflective Dependencies	3-49
3.1.7	Summary of Dependency Types	3-52
3.2	Abstractness of the Dependency Reference	3-53
3.2.1	Dependency on a Concrete Class	3-53
3.2.2	Dependency on an Abstract Class	3-55
3.2.3	Dependency on an Interface	3-57
3.2.4	Dependency via Delegates (Function Pointers/Callbacks)	3-60
3.2.5	Dynamic Typing / Reflection as Abstraction	3-62
3.2.6	Summary of dependencies' Abstraction Levels	3-64
3.3	Creation Timing and Instantiation Mechanisms	3-66
3.3.1	Direct Instantiation with new	3-66
3.3.2	Constructor Injection	3-68
3.3.3	Factory Pattern (and Static Factory Methods)	3-70
3.3.4	Service Locator	3-72
3.3.5	Configuration and Container-Based Binding	3-74
3.3.6	Reflection-Based Activation (Activator, etc.)	3-77
3.3.7	Summary of creation mechanisms	3-78
3.4	Span and Depth of Dependency Chains	3-79
3.4.1	Horizontal Dependency Paths	3-79
3.4.2	Vertical Dependency Chains	3-82
3.4.3	Impact on Modularity, Cohesion, and SRP	3-86
3.5	Unifying the Four Axes for SOLID/GRASP Design	3-88
Chapter 4	<i>Single Responsibility Principle</i>	4-94
4.1	SRP Violation Example	4-95
4.2	SRP to GRASP Mapping	4-102
4.3	The Objective vs. Subjective Nature of SRP	4-104
4.3.1	The LCOM metric	4-106
4.3.2	The TCC/LCC metric	4-108
4.3.3	Tackling the problem of false positive cohesion	4-111

4.3.4	The RFC metric	4-112
4.3.5	Version-control change-frequency analytics	4-116
4.3.6	Semantic judgement: what counts as a “responsibility”	4-117
4.3.7	Domain-specific trade-offs	4-117
4.3.8	Best-practice trio: combine metrics, build consensus, honour context	4-118
4.3.9	A Real-World Workflow for Cohesion and Hotspot Metrics	4-119
4.4	Bringing It All Together: Operationalising SOLID and GRASP through SRP	4-120
Chapter 5	<i>Open Closed Principle</i>	5-122
5.1	Relationship Single Responsibility Principle	5-122
5.2	Why OCP still matters	5-123
5.3	OCP Violation Example	5-124
5.4	Open/Closed Principle ↔ GRASP Mapping	5-130
5.4.1	Protected Variations Pattern	5-131
5.4.2	Polymorphism Pattern	5-132
5.4.3	Indirection Pattern	5-134
5.4.4	Pure Fabrication	5-137
5.4.5	Synthesis	5-140
5.5	The Objective vs Subjective Nature of the Open/Closed Principle	5-141
5.5.1	Change-Proneness & Rigidity Metrics	5-141
5.5.2	Afferent/Efferent Coupling & Instability	5-146
5.5.3	Abstractness (A) and the Stable-Abstractions Principle (SAP) Zone	5-147
5.5.4	Static Analysis Rules & Code-Smell Heuristics	5-148
5.5.5	Domain-Driven Trade-offs	5-148
5.5.6	Synthesis—Objective and Subjective in Dialogue	5-149
5.6	Refactoring Playbook — From Rigid to Open	5-150
5.6.1	Extract Superclass / Interface	5-152
5.6.2	Introduce Strategy / Policy Injection.	5-153
5.6.3	Factory & Registration (GRASP <i>Creator</i>)	5-153
5.6.4	Template Method versus Hooks	5-154
5.6.5	Packaging & DI Container Setup	5-155
5.6.6	Bringing It All Together – Operationalising SOLID + GRASP through the Open/Closed Principle	5-158
5.7	Synthesis	5-164

Chapter 6	<i>Liskov Substitution Principle (LSP)</i>	6-166
6.1	Definition and Theoretical Foundations of LSP	6-166
6.2	LSP as a Pillar of Polymorphism and Protected Variations	6-169
6.3	Recognizing LSP Violations: Common Examples and Anti-Patterns	6-171
6.3.1	Strengthening Preconditions	6-172
6.3.2	Weakening Postconditions	6-174
6.3.3	Violating Invariants	6-176
6.3.4	Changing Expected Behavior	6-177
6.3.5	Throwing Unexpected Exceptions	6-179
6.3.6	Narrowing Acceptable Input or Output Types	6-181
6.3.7	Ignoring or Misusing the Base Contract	6-194
6.3.8	Conflating Capabilities in a Single Type → Forced Downcasts at Call Sites	6-196
6.3.9	Synthesis	6-198
6.4	The Objective vs. Subjective Nature of LSP Compliance	6-203
6.4.1	Objective Criteria: Design By Contract	6-203
6.4.2	Subjective Judgment: Design Intent and Domain Constraints	6-224
6.5	Refactoring Playbook – From LSP Violations to Safe Design	6-226
6.5.1	Why another taxonomy?	6-226
6.5.2	Examples, Explanations, and Justifications	6-226
Chapter 7	<i>Interface Segregation Principle</i>	7-266
7.1	Interface Surface Area and ISP: What Clients Really Depend On	7-267
7.2	ISP Violation Example	7-271
7.3	Integrating ISP, LSP, and SRP: Clear, Practical Guidance	7-275
7.3.1	Orthogonality of ISP and LSP: Client-Specific Interfaces vs Substitutability Contracts	7-276
7.3.2	From Reason-to-Change to Reason-to-Depend: The Alignment of SRP and ISP	7-281
7.3.3	From Ideas to Practice: A Simple, Repeatable Method	7-284
7.4	Mapping ISP to GRASP Principles	7-287
7.5	The Objective vs. Subjective Nature of ISP	7-289
7.5.1	Objective Symptoms of ISP Violations	7-289
7.5.2	Subjective Design Considerations	7-312
7.6	Refactoring Playbook – From Fat Interfaces to Focused Interfaces	7-314
7.6.1	Step 1: Scoping the Analysis to Implementers	7-314

7.6.2	Step 2: Create New Client-Specific Interfaces	7-319
7.6.3	Step 3: Refactor Implementing Classes	7-323
7.6.4	Step 4: Update interface references in clients.	7-328
7.6.5	Step 5: Deprecate or Remove the Fat Interface	7-334
7.6.6	Step 6: Validate behavior and performance	7-339
7.6.7	Step 7: Communicate and document	7-344
7.6.8	Synthesis	7-350
Chapter 8	<i>Dependency Inversion</i>	8-355
8.1	Importance of DIP in Modern C# and .NET Design	8-356
8.2	Dependency Injection: A Mechanism to Achieve DIP	8-357
8.3	DIP Violation Example and Refactoring	8-359
8.4	DIP and GRASP: Indirection and Protected Variations	8-363
8.5	Objective and Subjective Impacts of DIP (Metrics and Design Reasoning)	8-365
8.5.1	Objective Criteria	8-367
8.5.2	Subjective Criteria	8-374
8.6	Refactoring Playbook: Implementing DIP Step-by-Step	8-391
8.6.1	Step-by-Step DIP Refactoring Process	8-391
8.6.2	Example: Decoupling an Email Notification Service	8-393
8.6.3	Example: Refactoring an Order Processing Workflow	8-395
8.6.4	Synthesis: DIP's Role in Maintainability, Testability, and SOLID Synergy	8-399
Chapter 9	<i>Conclusions, Limitations, and Future Work</i>	9-404
9.1	Conclusions	9-404
9.1.1	Contributions	9-405
9.1.2	Practical Implications	9-406
9.2	Limitations	9-406
9.2.1	Scope and External Validity	9-407
9.2.2	Measurement and Construct Validity	9-407
9.2.3	Threats to Internal Validity	9-408
9.2.4	Performance and Operational Considerations	9-408
9.3	Future Work	9-408
9.3.1	Automated Guidance and Tooling	9-409
9.3.2	Expanded Empirical Studies	9-409

9.3.3	Design Playbooks and Education	9-410
9.3.4	Integrations with Testing and Operations	9-411
9.3.5	Research on Trade-off Modeling	9-411
9.4	Practical Guidelines (A Consolidated Checklist)	9-411
9.5	Closing Reflection	9-412
<i>Appendix A</i>		9-414
<i>References</i>		9-416

Table of Figures

Figure 1. A simple inheritance example in C#.....	3-39
Figure 2. An association where <i>Student</i> uses (has a reference to) <i>Teacher</i>	3-41
Figure 3. Composition example – Car composes an Engine	3-44
Figure 4. Classes <i>ReportGenerator</i> and <i>InvoiceGenerator</i> delegate printing to <i>Printer</i> class	3-46
Figure 5. <i>EmailSender</i> Transient dependency on <i>SmtplibClient</i>	3-49
Figure 6. Reflective Dependence on <i>IPaymentProcessor</i> complaint class	3-50
Figure 7. Plugin dependence is resolved at runtime.....	3-51
Figure 8. Hard-coded dependency of <i>DataExporter</i> on <i>FileLogger</i> class.....	3-54
Figure 9. Abstraction-based dependency of <i>DataExporter2</i> on <i>LoggerBase</i> abstract class	3-55
Figure 10. Abstraction-based dependency of <i>DataExporter3</i> on <i>ILogger</i> interface.....	3-58
Figure 11. Abstraction-based dependency of <i>DataProcessor</i> on <i>Func<int, bool></i> filter delegate	3-60
Figure 12. Dynamic Dependence Example.....	3-62
Figure 13. Hardcoded instantiation at the source site of dependence.....	3-66
Figure 14. Dependences of <i>ReportService2</i> class are provided through constructor injection	3-68
Figure 15. Sample code segment in constructor root supplying dependencies to application object classes	3-70
Figure 16. Delegating dependences instantiation to <i>ConfigRepositoryFactory</i>	3-71
Figure 17. Instantiate dependence target using <i>ServiceLocator</i> class.....	3-73
Figure 18. Instantiate dependences at startup in constructor root using configuration files	3-75
Figure 19. Instantiate dependence target using Reflection.....	3-77
Figure 20. A coordinator (<i>WorkflowManager</i>) invokes two sibling components (<i>ComponentA</i> , <i>ComponentB</i>) at the same abstraction level— <i>DoTaskA()</i> then <i>DoTaskB()</i> —forming a side-by-side (non-nested) call sequence where each call is an independent sub-operation of the workflow	3-80
Figure 21. A client invokes two sibling operations on the same service.....	3-80
Figure 22. A 3-level deep inter-object sequence— <i>UIHandler.ButtonClick()</i> → <i>OrderService.ProcessOrder()</i> → <i>OrderRepository.StoreOrder()</i> —illustrating a top-down, nested flow typical of UI → Service → Repository architectures	3-82
Figure 23. A depth-3 intra-object call chain— <i>Grandparent()</i> → <i>Parent()</i> → <i>Child()</i> —showing vertical depth created by methods delegating to lower-level helpers inside the same class.....	3-82
Figure 24. Example of code violating SRP principle.....	4-97
Figure 25. Refactored Code of the <i>TradeProcessor</i> class.....	4-101
Figure 26. Trivial example for illustrating LCOM evaluation.....	4-106
Figure 27 Method Field Matrix for <i>Invoice</i> class engine	4-107
Figure 28 Unordered pair of methods for evaluating LCOM1 metric.....	4-107
Figure 29. Method -Field Matrix rendered to a graph with 2 strongly connected components	4-108
Figure 30. <i>DeductionCalculator</i> class before refactoring	4-110

Figure 31 Method-Field matrix for <i>DeductionCalculator</i> class.....	4-111
Figure 32 Transitive closure of distinct methods that <i>ProcessTrades</i> can trigger	4-113
Figure 33 RFC metric value per class after refactoring the <i>TradeProcessor</i> class.....	4-115
Figure 34. <i>ReportGenerator</i> class violating OCP principle.....	5-125
Figure 35. <i>ReportGenerator</i> class after augmenting it with the <i>Profit-And-Loss</i> report.....	5-126
Figure 36. Refactored version of <i>ReportGenerator</i> class complaint to OCP	5-128
Figure 37. <i>CurrencyService</i> class hardcodes access to exchange range provider violating OCP	5-132
Figure 38 Refactored version of <i>CurrencyService</i> satisfying OCP.....	5-132
Figure 39. <i>DocumentExporter</i> class.....	5-133
Figure 40. Refactored <i>DocumentExporter</i> class.....	5-133
Figure 41 <i>CheckoutService</i> class processes orders through interaction with a hardcoded URL	5-136
Figure 42 Refactor <i>CheckoutService</i> class directs payment processing to the injected <i>IPaymentGateway</i> compatible objects	5-137
Figure 43. Customer domain class burdened with persistence logic.....	5-138
Figure 44. Refactored Customer class disconnected from persistence logic.....	5-139
Figure 45. Command to provide a list of files that have been committed ordered by the number of commits.....	5-142
Figure 46. Resulted report from git log in Figure 25.....	5-142
Figure 47. Time windowed report of most frequently modified files.....	5-143
Figure 48. Powershell script to identify “co-change” neighbours for one target file.....	5-143
Figure 49. Generated Report after executing the script of Figure 28.....	5-145
Figure 50. Generated Report findings from <i>rigidity.ps1</i>	5-146
Figure 51. <i>ReportGenerator</i> class generates reports in three different formats.....	5-151
Figure 52. Extract <i>IReport</i> interface	5-152
Figure 53. Introduce Strategy / Policy Injection.....	5-153
Figure 54. Increase of complexity in the generation of proper Report object	5-154
Figure 55. Factory & Registration (<i>GRASP Creator</i>).....	5-154
Figure 56. Apply Template method pattern.....	5-155
Figure 57. ASP .NET Core DI registrations of the application classes	5-157
Figure 58. Reports Controller class	5-157
Figure 59. Container code to create the object of classes corresponding to the keys	5-158
Figure 60. Creating mock objects.....	5-158
Figure 61 how the refactor transformed the codebase	5-159
Figure 62. LSP strengthening preconditions violation example using a customer manager.....	6-174
Figure 63. LSP weakening postconditions violation example using a book lending manager	6-175
Figure 64. The classic example of the Rectangle–Square problem	6-176
Figure 65. Tax Calculator Example.....	6-178
Figure 66. Logger Example.....	6-179

Figure 67. Opposite Logger Scenario	6-180
Figure 68. Overridden Print method Narrowing the intended by the base class input type.....	6-181
Figure 69. Fix A of LSP violation of StringPrinter class.....	6-182
Figure 70. Fix B of LSP violation of StringPrinter class.....	6-183
Figure 71. Example of covariance using the covariant IEnumerable<T> interface.....	6-184
Figure 72. Example of contravariance using the contravariant Action<T> delegate.....	6-184
Figure 73. Example of Invariance using the IList<T> interface.....	6-184
Figure 74. Contravariant input flexibility in C# without changing method-parameter types on an override.....	6-189
Figure 75. How variance was handled before introduction of variance in C#.....	6-191
Figure 76. Example of a game character builder with covariant return types.....	6-193
Figure 77. Graph Node Hierarchy with LSP violation.....	6-195
Figure 78. Conflating capabilities in subtypes.....	6-197
Figure 79. Identifying LSP violations PseudoAlgorithm	6-202
Figure 80. Runtime-enforced base.....	6-206
Figure 81. Sample Main for illustrating Runtime LSP compliance validation.....	6-208
Figure 82. Execution Results.....	6-209
Figure 83. Weapons.cs file.....	6-211
Figure 84. xUnit Test Suite for Weapon.cs file that objectively verifies LSP compliance.....	6-214
Figure 85. Test Explore screenshot showing.....	6-215
Figure 86. Workspace setup.....	6-216
Figure 87. Minimal Graph Model Implementation.....	6-218
Figure 88. GraphModel LSP compliance verification xUnit Test suite	6-220
Figure 89. Stryker configuration file GraphDomain.Tests/stryker-config.json	6-221
Figure 90. Stryker execution report.....	6-222
Figure 91. Stryker html generated report	6-223
Figure 92. Relax the subtype to the base contract.....	6-228
Figure 93. Refactor by sealing the contract in the base	6-229
Figure 94. Additional measures applied to prohibit accidental strengthening of preconditions	6-232
Figure 95. xUnit validation	6-233
Figure 96. Visual Studio Project additional settings	6-233
Figure 97. Running xUnit tests results.....	6-234
Figure 98. Subtype is refactored to satisfy base promise.....	6-236
Figure 99. Refactoring the base class to determine the admissible state changes and verification of postconditions	6-239
Figure 100. Tests that verify postconditions for various subclasses.....	6-240
Figure 101. Separate Square and Rectangle hierarchy	6-242
Figure 102. Keep Square, Rectangle in the same hierarchy but enforce invariants using factories/builders	6-244

Figure 103. Separating logging sinks from logging policies using strategy.....	6-246
Figure 104. Extract a strategy; make the base a stable façade	6-250
Figure 105. Make behavior data-driven in a single sealed type.....	6-252
Figure 106. Broaden behavior within the existing hierarchy	6-255
Figure 107. Type the capability correctly with generics and variance	6-257
Figure 108. Split functionality into two interfaces.....	6-259
Figure 109. Demote “leaf” to state	6-261
Figure 110. Make failure part of the contract.....	6-263
Figure 111. An oversized interface (IAllInOnePrinter) forcing a class to throw NotSupportedException, thus violating ISP. The BudgetPrinter1000 does not need Fax() but must implement it.....	7-272
Figure 112. The printer interfaces are segregated. The multi-purpose interface is split into IPrinter, IScanner, and IFax. Classes implement combinations as required: BudgetPrinter1000 exposes no fax method, whereas ProPrinter3000 implements all three interfaces.	7-273
Figure 113. Minimal temperature-probe contract	7-278
Figure 114. Compliant Celsius thermometer	7-278
Figure 115. Non-compliant Kelvin thermometer	7-279
Figure 116. Client calibrated to Celsius reveals the violation.....	7-279
Figure 117. Broad “all-in-one” interface with a fully correct implementer.....	7-280
Figure 118. Thin client forced to depend on a fat interface.....	7-280
Figure 119. Role interface restores a minimal dependency surface	7-281
Figure 120. Regular expression to identify NotImplemented and NotSupportedException generation in Visual Studio Environment.....	7-290
Figure 121. Regular expression to identify NotImplemented and NotSupportedException exception generation in Powershell	7-290
Figure 122. One-pass ISP screening pipeline. Shortlist “large” interfaces by member count.....	7-296
Figure 123. Per-member breadth pseudocode for ISP detection.....	7-300
Figure 124. Applying per-member breadth to the “All-in-One” example	7-301
Figure 125. Post-refactor roles and wiring. The former “fat” interface is split into IPrinter, IScanner, and IFax.. ..	7-304
Figure 126. Implementer-cohesion diagnosis for ISP.....	7-306
Figure 127. Stepwise pseudo-algorithm that operationalises interface-centric hotspot detection.....	7-309
Figure 128. An example of a “fat” interface, IAllInOnePrinter, which aggregates multiple capabilities. The BudgetPrinter1000 class is forced to implement the Fax method, which it does not support, resulting in a NotSupportedException.....	7-315
Figure 129. An additional class, FaxOnlyKiosk, which only supports faxing, is forced to implement the unsupported Print and Scan methods from the IAllInOnePrinter interface by throwing exceptions	7-316

Figure 130 A matrix illustrating which implementing classes provide meaningful support for each member of the IAllInOnePrinter interface. A checkmark (✓) indicates support, while a cross (✗) indicates a non-functional implementation that throws an exception.....	7-317
Figure 131. Introduction of interfaces to codify usage clusters and implementation asymmetries	7-320
Figure 132. A composite interface, IMultiFunctionPrinter, is created by inheriting from smaller role interfaces (IPrinter, IScanner). An Adapter class is then implemented to provide a unified view for clients that require both capabilities, while delegating calls to the underlying role implementations.....	7-322
Figure 133. The BudgetPrinter1000 class is shown before refactoring, where it implements the large IAllInOnePrinter interface and is forced to provide an implementation for the Fax method, which it does not support, by throwing an exception.	7-324
Figure 134. The three new, segregated interfaces (IPrinter, IScanner, IFax) created by splitting the original fat interface based on distinct capabilities.	7-325
Figure 135. The BudgetPrinter1000 and ProPrinter3000 classes are shown after refactoring. Each now implements only the specific role interfaces corresponding to its actual capabilities, eliminating the need for exception-throwing stubs.	7-325
Figure 136. An example of a defensive guard within a method implementation. This check for _faxEnabled is a symptom of the class being forced to implement a method for a capability it may not support.	7-326
Figure 137. A comparison of unit tests before and after refactoring. The "before" test uses the fat IAllInOnePrinter interface, while the "after" test uses the new, focused IPrinter and IScanner role interfaces to verify the same observable behavior.	7-326
Figure 138. A before-and-after comparison showing a client (FaxJobService) being refactored. Initially dependent on the large IAllInOnePrinter interface, it is updated to depend only on the minimal IFax role interface that it actually uses.....	7-329
Figure 139. The CopyWorkflow class demonstrates handling multiple capabilities by explicitly depending on two separate, focused role interfaces (IScanner and IPrinter) via its constructor.....	7-330
Figure 140. A composite interface IMultiFunctionPrinter is created by inheriting from smaller IPrinter and IScanner roles. A class like ProPrinter3000 can then implement this composite along with other roles like IFax.	7-330
Figure 141. A before-and-after comparison of dependency injection container registrations. The single registration for the fat IAllInOnePrinter is replaced by multiple, specific registrations for the new role interfaces (IPrinter, IScanner).....	7-331
Figure 142. An example of role-specific factories. Instead of a single factory for all devices, separate factory interfaces (IFaxFactory, IScanFactory) are defined to create objects that fulfill specific roles.	7-331
Figure 143. A forward adapter, AllInOneAdapter, which implements the old IAllInOnePrinter interface but internally delegates its method calls to the new, smaller role interfaces (IPrinter, IScanner, IFax). This allows for a gradual migration of clients.....	7-332

Figure 144. A reverse adapter, <i>FaxOnlyView</i> , which implements the new, narrow <i>IFax</i> interface by wrapping and delegating to an existing object that still implements the old, fat <i>IAllInOnePrinter</i> interface.	7-332
Figure 145. The use of the <i>[Obsolete]</i> attribute in C# to mark an interface for deprecation. The first example shows an advisory warning message, while the second example shows how to configure the attribute to produce a compile-time error.	7-335
Figure 146. A transitional compatibility adapter, <i>AllInOneAdapter</i> , marked as obsolete. It implements the old fat interface by composing and delegating to the new, focused role interfaces, providing a backward-compatibility layer during migration.	7-336
Figure 147. The <i>FaxJobService</i> client before refactoring, showing its dependency on the large, multi-purpose <i>IAllInOnePrinter</i> interface.	7-345
Figure 148. The <i>FaxJobService</i> client after refactoring, now depending on the minimal and specific <i>IFax</i> role interface.	7-346
Figure 149. The <i>CopyWorkflow</i> class before refactoring, using a single dependency on the monolithic <i>IAllInOnePrinter</i> to perform both scanning and printing.	7-346
Figure 150. The <i>CopyWorkflow</i> class after refactoring, explicitly composing the <i>IScanner</i> and <i>IPrinter</i> roles by depending on both interfaces.	7-346
Figure 151. A dependency injection registration before refactoring, binding the single fat interface <i>IAllInOnePrinter</i> to a concrete implementation.	7-347
Figure 152. Dependency injection registrations after refactoring, showing separate bindings for each role interface (<i>IPrinter</i> , <i>IScanner</i> , <i>IFax</i>) to their respective concrete implementations.	7-347
Figure 153. A transitional compatibility shim, <i>AllInOneShim</i> , which implements the deprecated <i>IAllInOnePrinter</i> interface by composing and delegating to the new role interfaces, allowing for a phased migration.	7-348
Figure 154. A composite convenience interface, <i>IMultiFunctionPrinter</i> , and its corresponding adapter. This pattern allows clients that legitimately need multiple roles to depend on a single, unified interface without violating ISP for other clients.	7-349
Figure 155. A high-level module, <i>OrderService</i> , directly instantiates and depends on a low-level concrete class, <i>MySQLDatabase</i> , thereby violating the DIP.	8-360
Figure 156. An abstraction, <i>IOrderRepository</i> , is introduced to define the contract for saving an order, decoupling high-level logic from specific persistence implementations.	8-361
Figure 157. The <i>OrderService</i> class is refactored to depend on the <i>IOrderRepository</i> abstraction, which is provided via constructor injection, adhering to DIP.	8-361
Figure 158. The concrete <i>MySQLDatabase</i> class is modified to implement the <i>IOrderRepository</i> interface, making it a substitutable low-level detail.	8-362
Figure 159. The application's composition root creates a concrete <i>MySQLDatabase</i> instance and injects it into the <i>OrderService</i> , completing the dependency inversion.	8-362

Figure 160. The formula for calculating the DIP Compliance Index (DIP-CI), which quantifies a class's adherence to the Dependency Inversion Principle by applying weighted penalties for violations and bonuses for best practices.	8-369
Figure 161. A well-designed OrderService class that adheres to DIP by depending only on abstractions (IOrderRepository, ILogger) which are supplied via constructor injection.	8-370
Figure 162. A ReportService class demonstrating multiple DIP violations, including direct instantiation of concrete infrastructure (SqlConnection, HttpClient) and use of a service locator.	8-372
Figure 163. A DIP-compliant PriceEngine that uses an injected delegate (Func<Cart, IPricing>) as a factory to create short-lived collaborators based on incoming data.	8-372
Figure 164. A CheckoutService that depends on a stable IPaymentGateway abstraction, allowing different payment providers to be used without modifying the service itself, thus protecting it from variations.	8-376
Figure 165. A FraudEngine that accepts a scoring rule as a Func<> delegate, allowing the fraud detection logic to be changed without modifying the engine.	8-376
Figure 166. An image processing pipeline that uses an IFilter interface for applying transformations, but the virtual dispatch inside the hot loop (ProcessSlow) can be a performance bottleneck.	8-378
Figure 167. A refactored, performance-optimized image pipeline where the choice of filter is made outside the hot loop (ProcessFast), and a direct delegate is passed in to be invoked, avoiding virtual dispatch overhead..	8-379
Figure 168. A GeometryKernel with static methods for stable, well-known geometric calculations. In this context, using concrete implementations is preferable to introducing interfaces, as the algorithms are not expected to change.	8-380
Figure 169. An example of a performance-sensitive "hot path" where an IAggregator interface is used within a loop, causing a virtual dispatch on every iteration.	8-382
Figure 170. A refactored version of the aggregator where the dependency inversion "seam" is moved outside the hot loop. A concrete delegate is passed into the loop, allowing the JIT compiler to potentially inline the call for better performance.	8-382
Figure 171. An example of unintentional closure allocation, where a lambda captures the local variable d, forcing the compiler to create a new object on the heap to store its state.	8-384
Figure 172. A refactored version that avoids heap allocation from closures by using a struct to pass state and employing a static local function, which cannot capture variables.	8-384
Figure 173. An example of boxing, where passing an int (a value type) to a method expecting an object causes an allocation on the heap for each call inside the loop.	8-385
Figure 174. The refactored version using a generic interface IConsumer<T>, which eliminates boxing by allowing the method to accept the specific value type int directly.	8-385
Figure 175. An inefficient implementation where a dependency (IRateProvider) is resolved from a service locator inside a loop, incurring lookup overhead on every iteration.	8-386
Figure 176. The optimized version where the IRateProvider dependency is injected once outside the loop, avoiding repeated service location overhead.	8-386

Figure 177. A performance-poor example using reflection (<i>GetMethod</i> and <i>Invoke</i>) inside a loop, resulting in significant overhead from metadata lookups and argument boxing on each iteration.....	8-387
Figure 178. An optimized approach where reflection is used only once outside the loop to create a strongly-typed delegate, which is then invoked directly within the loop, eliminating repeated reflection overhead.	8-387
Figure 179. A simple benchmarking harness to measure the execution time of an action, including a warm-up phase to ensure the JIT compiler has optimized the code before measurement begins.	8-388
Figure 180. The initial design of an <i>EmailNotification</i> class, demonstrating a violation of the <i>Dependency Inversion Principle</i> through the direct instantiation and tight coupling to the concrete <i>SmtplibClient</i> class.	8-393
Figure 181. The refactored, <i>DIP</i> -compliant design where the high-level <i>EmailNotification</i> module depends on the <i>ISender</i> abstraction, which is implemented by the low-level <i>SmtplibEmailSender</i> detail and supplied via constructor injection.	8-394
Figure 182. An example of a complex <i>DIP</i> violation, where the high-level <i>OrderProcessor</i> module is tightly coupled to multiple low-level details (<i>SqlOrderRepository</i> , <i>StripePaymentGateway</i> , <i>SmtplibEmailService</i> , <i>FileLogger</i>) through direct instantiation in its constructor.	8-396
Figure 183. The refactored, <i>DIP</i> -compliant <i>OrderProcessor</i> workflow. High-level policy is decoupled from low-level details by introducing abstractions for each dependency, which are then supplied to the <i>OrderProcessor</i> via constructor injection.	8-397

Index of Tables

<i>Table 1. Summary of dependencies and their characteristics.....</i>	<i>3-53</i>
<i>Table 2. Dependency abstraction levels and their effects.....</i>	<i>3-65</i>
<i>Table 3. Creation and binding mechanisms for dependencies.....</i>	<i>3-78</i>
<i>Table 4. Intuitive Map of SRP to GRASP Patterns</i>	<i>4-104</i>
<i>Table 5. GRASP Patterns Mapped to the Open/Closed Principle</i>	<i>5-131</i>
<i>Table 6. Classic Abstract Factory vs DI Containers.....</i>	<i>5-156</i>
<i>Table 7. Mainstream languages that explicitly support covariant return types in method overrides.....</i>	<i>6-187</i>
<i>Table 8. LSP violations symptoms</i>	<i>6-200</i>

Chapter 1

Introduction

1.1 Background and Motivation

Modern software systems are embedded in nearly every facet of contemporary society. From e-commerce platforms powering global trade to mobile applications shaping individual daily habits, and from mission-critical enterprise resource planning (ERP) systems to the cloud-native services that orchestrate international communication, software today serves as the foundational infrastructure of economic and social activity. This ubiquity, however, is accompanied by an inherent complexity: as systems evolve to meet rapidly shifting requirements, developers are confronted with challenges of scale, maintainability, and long-term adaptability.

The central paradox of software engineering is that software must both **change continuously** and **remain stable**. Customers and organizations demand new features, integrations, and compliance with shifting legal or security landscapes. Yet at the same time, stakeholders expect that existing functionality—already validated and deployed—will remain dependable. The tension between **changeability** and **reliability** is the enduring challenge of software design. Without clear architectural guidance, codebases often devolve into brittle collections of patches. Technical debt accumulates; onboarding of new team members slows; testing costs rise; and, ultimately, organizations lose confidence in their ability to evolve their systems safely.

To counteract this tendency, the software engineering community has, over decades, distilled experiential wisdom into **design principles**. These principles are not recipes or rigid blueprints, but rather heuristics and guidelines that capture recurring insights about what makes code resilient to change. Among the most widely recognized families of such principles are **SOLID** and **GRASP**.

- **The SOLID principles**, popularized by Robert C. Martin in the early 2000s, crystallize five core guidelines for object-oriented software:
 - **Single Responsibility Principle (SRP)**: a class should have only one reason to change.

- o **Open/Closed Principle (OCP):** entities should be open for extension but closed for modification.
- o **Liskov Substitution Principle (LSP):** subtypes must be substitutable for their base types.
- o **Interface Segregation Principle (ISP):** clients should not be forced to depend on methods they do not use.
- o **Dependency Inversion Principle (DIP):** high-level modules should depend on abstractions, not on details.
- **The GRASP patterns** (General Responsibility Assignment Software Patterns), introduced by Craig Larman in the late 1990s, complement SOLID by offering heuristics for assigning responsibilities in a way that balances cohesion and coupling across the system: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Indirection, Pure Fabrication, and Protected Variations.

While SOLID emphasizes **class-level design integrity**, GRASP emphasizes **responsibility assignment at the system level**. Taken together, they provide a complementary methodology: SOLID ensures that individual modules remain cohesive and safe to evolve, while GRASP ensures that the interactions between modules remain intelligible, decoupled, and balanced.

The motivation for combining these two families is twofold. First, software teams in industry frequently struggle when principles are applied in isolation. A system that is “SOLID-compliant” in terms of class design may still suffer from poorly distributed responsibilities, while a GRASP-compliant responsibility assignment may falter if the resulting classes do not respect SRP, OCP, or LSP. Second, the empirical evidence from both academia and industry suggests that systems designed with attention to **both modularity and responsibility assignment** exhibit measurably better maintainability, testability, and evolvability. This dual perspective therefore offers practitioners not only theoretical elegance but also pragmatic resilience.

The context of this thesis is the **C# and .NET ecosystem**, which provides an ideal arena for exploring these principles. C# offers rich support for object orientation, generics, interfaces, reflection, and language-integrated query (LINQ), while the .NET ecosystem provides powerful frameworks for dependency injection, test automation, and architectural layering. Furthermore, .NET is widely used in enterprise settings, where long-lived systems must evolve safely over years

or even decades. By grounding the discussion in idiomatic C# examples, this work ensures that the principles are not abstract slogans but are concretely applicable to modern industrial practice. Finally, the relevance of this work extends beyond industry. In the **academic domain**, teaching design principles is essential for cultivating students’ ability to think critically about architecture rather than only about coding syntax. Experience shows that graduates who can reason about maintainability, testability, and extensibility are better prepared for professional practice. By providing C#-based demonstrations of SOLID and GRASP in action, this thesis aims to contribute also to pedagogy, offering students concrete bridges between design theory and coding practice.

1.2 Research Problem and Rationale

Although both SOLID and GRASP have been widely taught and cited in literature, their combined application has rarely been systematized in a way that provides actionable guidance. Instead, practitioners often encounter fragmented advice: tutorials on SOLID without reference to responsibility assignment, or GRASP discussions without consideration of how to enforce design contracts. This fragmentation leads to two problems.

First, teams may implement principles superficially. For example, a developer might create multiple interfaces to “satisfy” ISP but fail to map those interfaces to actual responsibilities in the domain, creating artificial fragmentation without genuine decoupling. Similarly, a team may adopt GRASP’s Information Expert but overlook that the resulting class has accumulated too many reasons to change, violating SRP. Without integration, principles risk being applied mechanically rather than thoughtfully.

Second, tooling support remains asymmetric. Static analysis tools such as SonarQube or NDepend can flag probable SOLID violations—large classes, deep inheritance hierarchies, unused interface members—but there is little automated support for detecting GRASP misapplications. This gap can leave responsibility assignment largely in the realm of subjective judgement. A unified framework that clarifies **how SOLID and GRASP interrelate** could therefore empower both better human decision-making and more targeted tool support.

The rationale of this thesis is that by **mapping each SOLID principle to relevant GRASP patterns**, a systematic methodology emerges. This methodology provides not only diagnostic power—helping teams recognize when a design is drifting—but also prescriptive guidance,

suggesting concrete refactoring strategies grounded in both families of principles. By embedding these mappings in C# examples, the thesis demonstrates how violations can be recognized, how refactorings can be conducted incrementally, and how the resulting design can be explained both in theoretical and practical terms.

1.3 Research Objectives

The objectives of this thesis can be articulated across three complementary dimensions: theoretical, analytical, and practical.

1.3.1 Theoretical Objectives

1. **Synthesize foundations** by surveying the origins and rationales of both SOLID and GRASP, highlighting their conceptual alignments and differences.
2. **Clarify design intent** by examining how principles have been historically interpreted in both academic and industrial contexts.
3. **Extend understanding** by arguing for the synergy of the two families, showing that SOLID refines local correctness while GRASP ensures global balance.

1.3.2 Analytical Objectives

1. **Identify anti-patterns** in C# code that correspond to violations of SOLID and GRASP principles.
2. **Measure violations** using static analysis metrics such as LCOM, RFC, coupling indices, and change frequency analytics.
3. **Evaluate trade-offs** by distinguishing between objective indicators (metrics, tool warnings) and subjective factors (domain semantics, performance constraints).

1.3.3 Practical Objectives

1. **Demonstrate refactorings** by providing step-by-step C# examples that transform flawed code into principle-compliant design.
2. **Map refactorings to GRASP** so that each code change can be justified not merely in terms of syntax but also in terms of responsibility assignment.

3. **Validate through a case study** by refactoring a small e-commerce module iteratively, applying multiple principles in concert.
4. **Explore advanced principles**—Interface Segregation, Dependency Inversion, and Dependency Injection—to illustrate how they secure large-scale maintainability and testability.

By fulfilling these objectives, the thesis aims to deliver both **scholarly insight** and **practical utility**.

1.4 Contribution of the Thesis

This work contributes in several ways:

- **An integrated framework:** a systematic mapping of SOLID principles to GRASP patterns, showing how class-level correctness and system-level responsibility assignment reinforce each other.
- **Refactoring playbooks:** step-wise examples in idiomatic C#, each demonstrating how to resolve specific violations and how to justify the changes with design principles.
- **Metric-guided analysis:** demonstrations of how code metrics and version-control analytics can be used to diagnose violations objectively, while still allowing for subjective judgement.
- **Case study evidence:** a realistic e-commerce module that demonstrates the iterative application of principles, making the benefits concrete in terms of reduced complexity, improved testability, and enhanced extensibility.
- **Pedagogical value:** an accessible yet rigorous treatment of design principles tailored for final-year Informatics students and practitioners alike.

1.5 Thesis Structure

The remainder of the thesis is organized to progress from foundations to principle-specific analyses and, finally, to synthesis and outlook.

Chapter 2 presents the literature review, where the SOLID principles and the GRASP patterns are surveyed and their emerging synergy across academic, educational, and industrial settings is summarized. Attention is drawn to the differing levels of tool support and to the role of empirical evidence in motivating an integrated stance.

Chapter 3 establishes the **foundational mechanisms behind dependencies**, which serve as the connective tissue of object-oriented systems. Dependencies are examined along four orthogonal axes so that later principle chapters can refer to a common vocabulary and set of trade-offs. First, the **type of dependency** is catalogued (inheritance/generalization, association/aggregation, composition, delegation, parameter-level/usage-only, and dynamic/reflective links). Second, the **abstractness of the dependency reference** is analyzed (concrete classes, abstract classes, interfaces, delegates/callbacks, and dynamic typing or reflection), emphasizing how higher abstraction levels alter substitutability and coupling. Third, the **creation timing and instantiation mechanisms** are compared (direct new, constructor injection, factory and static factory methods, service locator, container-based binding/configuration, and reflection-based activation), with attention to how each choice shifts binding time, flexibility, and operational risk. Fourth, the **span and depth of dependency chains** are characterized (horizontal collaborations among peers versus vertical call chains across layers), together with their implications for modularity, cohesion, and SRP. A unifying synthesis closes the chapter by relating these axes to GRASP (Low Coupling, Indirection, Protected Variations) and to SOLID (especially OCP and DIP), so that subsequent chapters can ground refactorings in dependency mechanics rather than slogans.

Chapter 4 addresses the **Single Responsibility Principle (SRP)**, relating responsibility cohesion to GRASP's Information Expert and Pure Fabrication, and balancing objective indicators (e.g., cohesion and usage metrics) with necessary semantic judgement; a practical workflow for evidence-guided cohesion refactoring is provided.

Chapter 5 develops the **Open/Closed Principle (OCP)**, mapping openness to GRASP strategies (Protected Variations, Polymorphism, Indirection, Pure Fabrication), pairing change-proneness and instability measures with a refactoring playbook that realizes extension by addition rather than risky edits.

Chapter 6 treats the **Liskov Substitution Principle (LSP)**, focusing on behavioral contracts and substitutability hazards, and shows how precise specifications and tests preserve trust in extension hierarchies.

Chapter 7 examines the **Interface Segregation Principle (ISP)**, arguing for client-specific contracts to reduce coupling, aligning with GRASP’s Low Coupling and High Cohesion, and providing a stepwise method from “fat” to focused interfaces.

Chapter 8 presents the **Dependency Inversion Principle (DIP)** and its realization via **Dependency Injection**, showing how abstractions owned by policy localize change and enable reliable testing; links to GRASP’s Indirection and Protected Variations are made explicit.

Chapter 9 offers **Conclusions, Limitations, and Future Work**, synthesizing the contributions, reflecting on scope and validity, and outlining avenues for tooling, empirical replication, and pedagogy.

1.6 Significance of the Work

The significance of this thesis lies not only in its theoretical synthesis but also in its pragmatic orientation. By bridging SOLID and GRASP within a unified framework, it responds to a gap in both scholarship and practice. By grounding the discussion in C#, it ensures relevance to a major industrial ecosystem. By coupling analysis with refactoring playbooks, it transforms principles into actionable routines. Finally, by including metrics and case study validation, it demonstrates that design principles are not abstract ideals but measurable and beneficial practices.

In an era where software complexity continues to rise and development teams are under relentless pressure to deliver rapidly while maintaining quality, principles that foster **robust, evolvable, and understandable design** are not luxuries—they are necessities. This thesis therefore positions SOLID and GRASP not as competing alternatives but as complementary facets of a comprehensive design methodology that can equip students, practitioners, and researchers alike to meet the challenges of contemporary software engineering.

The remainder of the document is not available herein, due to the embargo period specified in the thesis release form.